
# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# THESIS

**A COOPERATIVE IDS APPROACH AGAINST MPTCP ATTACKS**

by

Warren L. Barksdale III

June 2017

| | |
|---|---|
| Thesis Advisor: | Geoffrey G. Xie |
| Second Reader: | John Fulp |

**Approved for public release. Distribution is unlimited.**

THIS PAGE INTENTIONALLY LEFT BLANK

# REPORT DOCUMENTATION PAGE

*Form Approved OMB No. 0704–0188*

*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 | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | June 2017 | Master's Thesis 09-20-2015 to 06-30-2017 |

**4. TITLE AND SUBTITLE**
A COOPERATIVE IDS APPROACH AGAINST MPTCP ATTACKS

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Warren L. Barksdale III

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Naval Postgraduate School
Monterey, CA 93943

**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 document 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 Protocol Number: N/A.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release. Distribution is unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(maximum 200 words)*

Recent thesis work by a Naval Postgraduate School graduate has proven that intrusion detection systems (IDS) can be defeated by leveraging Multipath Transmission Control Protocol (MPTCP). Furthermore, the ability to enhance a single IDS to better detect and defend against attacks leveraging MPTCP was presented. However, large organizations and entities have multiple IDSs that may not communicate or share connection information. We assume an attacker will launch an attack that leverages MPTCP's ability to connect a source and destination over multiple paths, and that the paths intentionally traverse through different IDSs on the target's network. We validate related work regarding enhancing an IDS to reconstruct MPTCP subflows and detect malicious content. Next, we build physical testbeds in order to present a methodology that allows distributed IDSs (DIDS) to cooperate in a manner that permits effective detection of such attacks.

**14. SUBJECT TERMS**
MPTCP, IDS, Distributed IDS, IPS

**15. NUMBER OF PAGES** 77

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UU |

**NSN 7540-01-280-5500**

**Standard Form 298 (Rev. 2–89)**
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release. Distribution is unlimited.**

**A COOPERATIVE IDS APPROACH AGAINST MPTCP ATTACKS**

Warren L. Barksdale III
First Lieutenant, United States Army
B.S., Hampton University, 2015

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**June 2017**

Approved by: Geoffrey G. Xie
Thesis Advisor

John Fulp
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Recent thesis work by a Naval Postgraduate School graduate has proven that intrusion detection systems (IDS) can be defeated by leveraging Multipath Transmission Control Protocol (MPTCP). Furthermore, the ability to enhance a single IDS to better detect and defend against attacks leveraging MPTCP was presented. However, large organizations and entities have multiple IDSs that may not communicate or share connection information. We assume an attacker will launch an attack that leverages MPTCP's ability to connect a source and destination over multiple paths, and that the paths intentionally traverse through different IDSs on the target's network. We validate related work regarding enhancing an IDS to reconstruct MPTCP subflows and detect malicious content. Next, we build physical testbeds in order to present a methodology that allows distributed IDSs (DIDS) to cooperate in a manner that permits effective detection of such attacks.

THIS PAGE INTENTIONALLY LEFT BLANK

# Table of Contents

**1 Introduction** **1**
1.1 Research Questions . . . . . 2
1.2 Summary of Contributions . . . . . 3
1.3 Thesis Structure . . . . . 3

**2 Background and Related Work** **5**
2.1 What Is an IDS?. . . . . 5
2.2 What Is Multipath TCP? . . . . . 5
2.3 What Is a Distributed IDS? . . . . . 12
2.4 Related Work . . . . . 14

**3 Single Intrusion Detection System (IDS) Experiments** **17**
3.1 Scope of Tests . . . . . 17
3.2 Building Testbeds . . . . . 17
3.3 Results . . . . . 32
3.4 Chapter Summary . . . . . 36

**4 Multi-IDS Experiments** **37**
4.1 Designing and Implementing IDS Cooperation . . . . . 37
4.2 Validation . . . . . 38
4.3 Results . . . . . 42
4.4 Chapter Summary . . . . . 45

**5 Conclusion** **47**
5.1 Future Work . . . . . 48

**Appendix: Snort3 Install Guide, Bashrc Tutorial, and "mp_tracker.py" Modifications** **49**
A.1 SNORT 3.0 Install Guide . . . . . 49

A.2 Custom Commands in Ubuntu 14.04 . . . . . 54
A.3 “mp_tracker.py” Modification . . . . . 55

**List of References** **57**

**Initial Distribution List** **59**

# List of Figures


Figure 2.1 Diagram of MPTCP. Source: [1] . . . . . 6

Figure 2.2 IDS Communication and Aggregation . . . . . 13

Figure 2.3 No IDS Communication and Aggregation . . . . . 14

Figure 3.1 Testbed 1 . . . . . 20

Figure 3.2 Testbed 2 . . . . . 25

Figure 3.3 Testbed 3 . . . . . 29

Figure 3.4 Subflow Red and Blue . . . . . 33

Figure 3.5 Attacker's Content Reassembled and Presented On Victims Terminal 34

Figure 4.1 Testbed 4 . . . . . 41

Figure 4.2 MPTCP Attack Launched and Spliced . . . . . 43

Figure 4.3 Snort3 Receiving Traffic and Generating Alerts . . . . . 43

Figure 4.4 Victim Receiving Traffic . . . . . 44

Figure 4.5 Subflow 0; 10.100.0.2 -> 10.100.3.2 . . . . . 44

Figure 4.6 Subflow 1; 10.100.1.2 -> 10.100.4.2 . . . . . 45

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Tables

Table 2.1 MPTCP Option Subtypes. Source: [2]. . . . . 9

Table 4.1 Testbed 4 Routing Tables . . . . . 39

THIS PAGE INTENTIONALLY LEFT BLANK

# List of Acronyms and Abbreviations

| | |
|---|---|
| **ACK** | Acknowledgement |
| **AIMD** | Additive Increase Multiplicative Decrease |
| **BS** | Base Station |
| **CWND** | Congestion Window |
| **DIDS** | Distributed IDS |
| **DOD** | Department of Defense |
| **DSS** | Data Sequence Signal |
| **FIN** | Finish |
| **GUI** | Graphical User Interface |
| **HMAC** | Hash-based Authentication Code |
| **IDMEF** | Intrusion Detection Message Exchange Format |
| **IDS** | Intrusion Detection System |
| **IDWG** | Intrusion Detection Exchange Format Working Group |
| **IETF** | Internet Engineering Task Force |
| **IP** | Internet Protocol |
| **IPS** | Intrusion Prevention System |
| **MP** | Multipath |
| **MPTCP** | Multipath Transmission Control Protocol (TCP) |
| **NIDS** | Network IDS |
| **NPS** | Naval Postgraduate School |

| | |
|---|---|
| **OS** | Operating System |
| **RLB** | Randomized Load Balancing |
| **RST** | Reset |
| **SRTT** | Smoothed Round Trip Time |
| **SYN** | Synchronization |
| **TCP** | Transmission Control Protocol |
| **TWH** | Three Way Handshake |
| **UDP** | User Datagram Protocol |
| **USA** | U.S. Army |
| **USN** | U.S. Navy |
| **USG** | United States Government |
| **VM** | Virtual Machine |
| **WSN** | Wireless Sensor Network |

# Acknowledgments

I would like to thank:
Katherine, my wife, for her constant love, support, and understanding. Especially with handling our three children for entire days at a time while I was doing school work.

Geoff, my thesis advisor, for his steady support and mentorship while I was still learning from the ground up what exactly this magic, the Internet, is and how I could make a contribution to it and the rest of the information technology community.

Dan and Henry, my friends and colleagues. Without their advice and endless patience I would have been lost.

John, my second reader, for agreeing to be my second reader despite already having numerous other thesis students that required his undivided attention.

My fellow brothers and sisters at arms in my cohort. Sharing their experiences and wisdom about academia and service in the military has provided me a greater understanding of what to expect in other academic and military areas. Lastly, explaining how they solved and overcame various challenges allowed me to think from other perspectives about similar challenges.

My father, Warren Barksdale Jr., without his guidance about the potential work and rewards presented in the information technology field I would not have begun the journey that has led me here.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 1:
# Introduction

The world has seen significant technological advancement within the past two decades. With every technology comes two opposing types of effort, those efforts focused on exploiting the technology's vulnerabilities for various motives and those efforts focused on securing the technology's vulnerabilities from exploitation. The technology addressed in this thesis is Multipath Transmission Control Protocol (TCP) (MPTCP). MPTCP is an improvement over standard TCP in numerous ways according to the Internet Engineering Task Force (IETF)'s TCP Extensions for Multipath Operation with Multiple Addresses [2].

According to Paasch et al.,

> Multipath TCP is a solution that allows to simultaneously use multiple interfaces/Internet Protocol (IP)-addresses for a single data-stream, while still presenting a standard TCP socket API to the application. Multipath TCP thus allows to increase the download-speed by aggregating the bandwidth of each interface. Additionally, MPTCP allows mobile hosts to handover traffic from Wi-Fi to 3G, without disrupting the application [3].

With mobile devices acquiring more capabilities through new and improved sensors, the demand for power efficiency and availability increases as well. MPTCP is able to help mobile devices save power by making connection hand-offs smoother, as in reducing the amount of time a device has to actively probe for a new mobile base station, and utilizing Wi-Fi and 3G/4G simultaneously to offload as much traffic from one of those networks as possible, thereby increasing speeds [4].

This thesis is about how to enhance an Intrusion Detection System (IDS), such as Snort [5], to detect the new exploits that leverage MPTCP. There exist attacks, leveraging standard TCP, and techniques that send traffic via different routes to the target in order to evade an IDS. However, they may now potentially be even more effective because MPTCP has not been widely encountered and prepared for. For example, a prevention/detection engine may not be alerted to content that triggers a static or dynamic rule designed to detect specific

strings. Assume the rule is to drop all packets that contain the string "malicious." With MPTCP an adversary is able to fragment that string and send the fragments past an IDS via the subflows of the main connection. A single connection can have many subflows allowing the adversary to send a third of the string through one of three subflows. According to RFC 6824, a subflow is "a flow of TCP segments operating over an individual path, which forms part of a larger MPTCP connection. A subflow is started and terminated similar to a regular TCP connection " [2]. This means that the detection/prevention engine will only check the rules against one third of the string at a time instead of the whole string. It will not raise an alarm because "mal" does not equal "malicious", "ici" does not equal "malicious", and "ous" does not equal "malicious". However, the full message "malicious" still arrives at its destination. This is also known as a cross-path fragmentation.

A prior Naval Postgraduate School (NPS) M.S. thesis investigated this type of attack [6]. The thesis developed and implemented an MPTCP attack that could evade a single IDS, and enhanced a single IDS that can detect an MPTCP attack. From the results of that thesis we were able to enhance and implement multiple IDSs to detect MPTCP attacks that attempt to evade IDSs through cross-path fragmentation.

## 1.1 Research Questions

Organizations that have multiple access points to the Internet often have an IDS or Intrusion Prevention System (IPS) monitoring traffic for malicious content at each of the access points. Furthermore, these systems are, more than likely, unaware of MPTCP based attacks. This vulnerability makes them susceptible to attacks leveraging MPTCP.

This thesis addresses the following research questions:

1. How can multiple IDS devices for the same network coordinate and aggregate MPTCP flows amongst each other to identify malicious MPTCP flows with high accuracy?
2. How can we implement multi-IDS coordination and aggregation if the IDSs are hosted in one or more virtual machines (VM) in the cloud?

## 1.2 Summary of Contributions

While addressing the research questions in Section 1.1, this thesis presents the results from performing the following tasks:

- Validating previous thesis experiments [6] in order to evaluate the hypotheses on a more realistic testbed environment.
- Developing a methodology to incorporate multiple IDSs, real and virtual, to be able to detect cross-path fragmentation MPTCP attacks, and to conduct offensive MPTCP experiments to better secure vulnerabilities.

## 1.3 Thesis Structure

The remaining chapters of the thesis are organized as follows. Chapter 2 consists of the background and related work that pertains to MPTCP. Chapter 3 is comprised of validating previous thesis work performed by Henry Foster in Chapters 3 and 4 of his thesis [6]. Chapter 4 consists of the extended experimentation similar to Foster's Chapter 3 experiments but with the addition of additional IDSs. Chapter 5 concludes this thesis by reviewing key points and suggesting areas of future research and development.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 2:
# Background and Related Work

This chapter covers the background of this thesis, such as addressing how MPTCP functions compared to standard (regular) TCP, how a MPTCP connection operates, what an IDS and Distributed IDS (DIDS) are, and related work involving MPTCP. In order to make the explanation of the various subjects less confusing we also define a few of the most frequently used terms in the thesis.

## 2.1 What Is an IDS?

Intrusion detection attempts to identify penetrations into a system. Similar to intrusion prevention, intrusion detection monitors the events occurring on a computer system or network and generates alerts when known signatures or anomalies are observed but does not prevent the intrusion. These alerts can notify the system administrators or security officers. Essentially, the IDS detects attempts and successes of breaching the network, and through the detection of these successes and attempts the network administrator is able to harden the network against repeat attacks and other intrusion attempts. For this thesis we will be relying and focusing on IDSs that employ the signature based detection technique. For example, the packets from an IP address will be analyzed and compared to known attacks in order to ensure that it is not an attack that has already been seen elsewhere or previously on the same network. The weakness of the signature based detection is that attack signatures that have not been observed will circumvent the IDS analysis and will execute unnoticed.

## 2.2 What Is Multipath TCP?

MPTCP is an emerging technology that enhances standard TCP by distributing and sending data through multiple IP-addresses, and their respective interfaces, at the same time. Applications treat MPTCP as standard TCP. However, it spreads data across a user defined or default amount of sub-flows. According to Paasch and S. Barré, “the benefits of this include better resource utilization, better throughput and smoother reaction to failures” [7].

Essentially, MPTCP sends TCP packets over multiple data paths to the destination to take advantage of the available throughput capacity and the inherent resiliency of the different paths. MPTCP flows are designed to co-exist with regular TCP flows in a friendly manner, i.e., the aggregate throughput of a MPTCP flow is similar to a regular TCP link that shares the choke point link of the sub-flow with the best throughput. The scheduler for MPTCP uses the fastest subflow to send data. According to Yang et al.,

> MPTCP employs an Additive Increase Multiplicative Decrease (AIMD) coupled congestion control mechanism. Each subflow continually increases its Congestion Window (CWND) even to a point that exceeds the available path capacity (defined as the maximum number of packets in flight of this subflow) before detecting a loss. If the number of outstanding packets of a subflow has reached its available path capacity, sending more packets through the subflow will cause congestion loss [8].

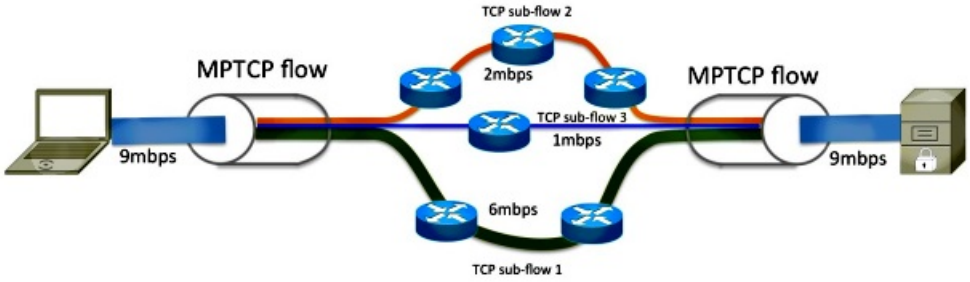


Image displays how an MPTCP connection logically functions with three subflows.

Figure 2.1: Diagram of MPTCP. Source: [1]

### 2.2.1 Definition of Terms

In order to ensure that we are communicating the content of the thesis well, we are providing the definitions of frequently used terms.

**Cross-path fragmentation**: is observed/experienced when a stream of data is fragmented and sent through different paths to the intended destination. This may occur more often with the use of MPTCP, because for each established link there maybe multiple subflows that carry fragments of a data stream.

**MPTCP vs TCP**: Through out the thesis we refer to MPTCP connections. By MPTCP connection we are referring to a connection where at least one subflow has completed a Multipath (MP)_CAPABLE handshake. An MP_CAPABLE handshake is similar to how standard TCP performs the three way handshake, except the Synchronization (SYN), SYN/Acknowledgement (ACK), and ACK packets all carry an additional flag, MP_CAPABLE. If any of the three packets (SYN, SYN/ACK, and ACK), do not contain the MP_CAPABLE flag, the connection reverts back to standard TCP. The MP_CAPABLE handshake has multiple purposes, but the primary purpose of the MP_CAPABLE handshake is to verify whether or not the listening host (the receiver) supports MPTCP. The secondary purpose is to exchange information between the two hosts in order to authenticate and establish any additional subflows.

### 2.2.2 Platforms and Systems Able to Implement MPTCP

The following list identifies the various platforms/systems that MPTCP has been implemented on and is being developed to apply to.

1. **Datacenters**: MPTCP has been proposed as an alternative approach to Randomized Load Balancing (RLB). RLB is used "to randomly choose a path for each flow from among the possible parallel paths" [9]. The intent is to leverage MPTCP's linked congestion controller in each MPTCP end system. The congestion controller is able to respond and operate on very short timescale to advance its traffic from paths where congestion has been detected to path it has determined are less congested [9].
2. **Linux kernel**: MPTCP has been widely implemented and experimented with via the Linux kernel. The experiments in this thesis used the Linux kernel implementation as well. MPTCP is being implemented by the IP Networking Lab and is being hosted on multipath-tcp.org for users, testers, and developers [7]. Furthermore, the Linux kernel implementation has allowed further experimentation with MPTCP being implemented in mobile devices. Specifically, the Nokia N950 that was used to measure the energy consumption during downloads from an MPTCP enabled web server using either regular TCP over Wi-Fi or 3G, or by using MPTCP over both Wi-Fi and 3G [4].
3. **Apple Operating System (OS) 7**: iOS can support MPTCP and allow the device to maintain a TCP connection in reserve over the user's cellular data.

4. **Android**: It has been proven with measurements performed with smartphones that popular applications function correctly through a SOCKS [10] proxy and MPTCP enabled smartphones.
5. **NS-3**: Coudron and Secci assert that "ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use." The development of MPTCP for a simulator will have a vast multitude of applications through allowing users to model and simulate their designs before beginning the process of actually building and implementing said design. Another reason a MPTCP simulator is desired is because "experimenting with MPTCP in the real world can be complex depending on the scenario. Mobility is a major use case and usually requires access to cellular (4G) and Wi-Fi interfaces. Not only does it have a cost, but 4G is not ubiquitous and experiments involving wireless channels are time consuming because of the variability and care their setup require" [11].

### 2.2.3 Goals and Expected Benefits of MPTCP

1. **High Redundancy**: MPTCP is able to utilize multiple connection sources on multiple interfaces. If a connection is to unexpectedly terminate, a download in progress would not be aborted; instead, it would continue by using the remaining available connections.
2. **Seamless Mobility**: In standard TCP, having to change our IP address generally means that our TCP session has been terminated and the device needs to establish a new session in order to continue certain service (i.e. GPS, video, audio, etc.). With MPTCP a device could dynamically establish additional connections before a session is terminated, and in the event a session terminates, the device will be able to retransmit the data that was lost via the remaining active sessions.
3. **Increased Confidentiality**: An attacker that may be observing traffic to and from a MPTCP enabled devices may have a more difficult time understanding the multiple flows and their respective subflows (we are not implying security through obscurity).
4. **Guaranteed Delivery**: Although standard TCP guarantees delivery, MPTCP's ability to send duplicated packets across multiple links can increase the probability that packets will traverse an uncongested path. However, this may incur a slight overhead.

### 2.2.4 Operation of MPTCP Connections

According to MPTCP RFC6824, in order to ensure maximum flexibility of MPTCP, two key constraints were imposed.

- It must be backwards-compatible with current, regular TCP, to increase its chances of deployment.
- It can be assumed that one or both hosts are multihomed and multiaddressed [2].

We reference MPTCP RFC6824 for the following terms:

- Path: A sequence of links between a sender and a receiver, defined in this context by a 4-tuple of source and destination address/ port pairs.
- MPTCP Connection: A set of one or more subflows, over which an application can communicate between two hosts. There is a one-to- one mapping between a connection and an application socket.
- Data-level: The payload data is nominally transferred over a connection, which in turn is transported over subflows. Thus, the term "data-level" is synonymous with "connection level", in contrast to "subflow-level", which refers to properties of an individual subflow.
- Token: A locally unique identifier given to a multipath connection by a host. May also be referred to as a "Connection ID".
- Host: An end host operating an MPTCP implementation, and either initiating or accepting an MPTCP connection [2].

Table 2.1: MPTCP Option Subtypes. Source: [2].

| Value(hex) | Symbol | Name | Reference |
|---|---|---|---|
| 0x0 | MP_CAPABLE | Multipath Capable | Section 3.1 |
| 0x1 | MP_JOIN | Join Connection | Section 3.2 |
| 0x2 | Data Sequence Signal (DSS) | Data Sequence Signal* | Section 3.3 |
| 0x3 | ADD_ADDR | Add Address | Section 3.4.1 |
| 0x4 | REMOVE_ADDR | Remove Address | Section 3.4.2 |
| 0x5 | MP_PRIO | Change Subflow Priority | Section 3.3.8 |
| 0x6 | MP_FAIL | Fallback | Section 3.6 |
| 0x7 | MP_FASTCLOSE | Fast Close | Section 3.5 |

*(Data ACK and data sequence mapping)

MPTCP connections are similar to TCP connections in the manner that they provide a bidirectional bytestream connection. This connection is initiated by some host (e.g. Host A) and established with another host (e.g. Host B) via a Three Way Handshake (TWH), and similarly terminated when either host sends a Finish (FIN) or Reset (RST) packet to the other host. However, during a MPTCP TWH each packet of the TWH carries the MP_CAPABLE option, see Table 2.1. There are many uses for this option, the first of which is to determine whether or not the receiving host is MPTCP capable or not. More importantly, the option allows the hosts to exchange additional information about other available paths (in this case IP addresses and interfaces). This would allow the addition of other connections (subflows) to be included by the master flow.

If and when a host has multiple paths with multiple IP addresses, any machine/device without MPTCP enabled will view each subflow as an individual standard TCP connection. A user is able to determine the number of subflows created by a MPTCP connection. Regardless of a machine being MPTCP enabled, each subflow undergoes their own TWH immediately following the master/main TWH. Lastly, if and when a new subflow may need to be added/associated with an existing MPTCP connection, it undergoes its own TWH, but this TWH must contain the MP_JOIN option in each packet. In order to prevent multiple subflows from multiple MPTCP connections from being incorrectly associated, a token is generated from the key that is obtained by the master subflow during the first TWH, for each subflow. The MPTCP connection to be joined is inferred from the token generated from the MP_CAPABLE handshake, and is authenticated via the Hash-based Authentication Code (HMAC). In order to calculate the HMAC it requires the keys from the MP_CAPABLE handshake and the nonces, identified as sender's random number, that were in the initial MP_JOIN messages. A host is informed about additional IP addresses via the MPTCP option, ADD_ADDR (add address), and informed about the removal of an address, REMOVE_ADDR.

In order to close a MPTCP connection, host A sends a "Data FIN" as part of the DSS. The connection termination is initiated with a TCP_FIN packet. Only after all data has been received by the participating parties, the TCP_FIN gets acknowledged with a DATA_ACK As described earlier, this is very similar to the standard TCP connection termination initiated with a FIN signal to a host. Upon all of the packets on the connection being received, a DATA_ACK is used to finally acknowledge the DATA_FIN.

### 2.2.5 MPTCP Security Considerations

This section addresses a few security considerations. MPTCP RFC6181 [12] identifies a few of the threats. The first being "Flooding (or bombing) attacks" [12]. This type of attack assumes that an attacker knows the IP address of their intended victim. The setup for this scenario, as depicted in MPTCP RFC6181, is an attacker associated with the IP address "IPA", a machine/server that generates large amounts of traffic with the IP address "IPS", and the target with the IP address "IPT". The exploit is executed via the following procedure:

1. First the attacker connects to the server via a MPTCP connection, and initiates the download of a significant amount of data.
2. Once the download has been initiated the address "IPT" is added by the attacker as a potential address to be used for the download. This step depends on whether the address management is explicit or implicit.
3. The attacker makes the connection between IPA and IPS appear congested. The intent of this step is to cause the server to send more traffic via the IPS-IPT link instead of the IPA-IPS link.

The next type of attacks are hijacking oriented attacks. The setup for this scenario is where there are two nodes, Node 1 connected to Node 2 with IP addresses IP1 and IP2, respectively, and an Attacker with the address IPA connected to Node 2. For this scenario the connection is using one address for each endpoint. RFC6181 [12] states, the attack is launched by the addition of the attacker's IP-address, "IPA", as a potential Node 1 address. For the connection to be successfully hijacked, the attacker must know the 4-tuple, also known as the socket pair, that is associated with that connection. It is assumed that an attacker would be able to derive the server's socket pair. A socket pair is the connection's source IP and port and destination IP and port. However, discovering the port of the client may prove to be more difficult but not impossible. The attacker could intercept packets from the client in order to learn the port. Once the attacker learns the socket pair and sends the ADD_ADDR message that adds the address "IPA" as a potential address to be accepted and used by the MPTCP connection, he/she will be able to send packets effectively from Node 2.

In order to address these security concerns, the MPTCP working group concluded:

- To better defend against flooding attacks, MPTCP should incorporate a reach-ability check via a random nonce before a new address is added.
- Stated in RFC6181, "The default security mechanisms for MPTCP should be to exchange a key in cleartext in the establishment of the first subflow and then secure following address additions by using a keyed HMAC using the exchanged key" [12].
- A pre-shared key should be supported by the MPTCP security mechanism that is accepted and used as part of the HMAC.
- A procedure that would prevent replay attacks using messages from another connection should be available, such as a sequence number protected by the HMAC.
- MPTCP should be able to incorporate a variety of security measures that can be implemented as needed.

## 2.3 What Is a Distributed IDS?

There already exists issues with coordinating and implementing DIDS on a large network to detect various attacks. DIDS is defined as "multiple Intrusion Detection Systems (IDS) [spread] over a large network, all of which communicate with each other, or with a central server that facilitates advanced network monitoring, incident analysis, and instant attack data." [13]. The motivation for the development and improvement of DIDS is addressed by the SANS Institute paper, "Distributed Intrusion Detection Systems: An Introduction and Review". It lists the motivation as:

1. The existence of single intruder attacks that cannot be detected based on the observations of only a single site.
2. Coordinated attacks involving multiple attackers that require global scope for assessment.
3. Normal variations in system behavior and changes in attack behavior that cause false detection and identification.
4. Detection of attack intention and trending is needed for prevention.
5. Advances in automated and autonomous attacks, i.e. rapidly spreading worms, require rapid assessment and mitigation.
6. The sheer volume of attack notifications received by ISPs and host owners can become overwhelming [13].

The benefit of DIDS is that aggregated attack information leads to a higher possibility of a quicker more effective response to intrusion attempts. However, the use of DIDS may complicate the detection of MPTCP based attacks if an effective communication method between the IDSs is not implemented. For instance, if one IDS does not forward/report MPTCP subflows to the other IDSs or a central system that aggregates the reports, an attack may go unnoticed. See figures 2.2 and 2.3.

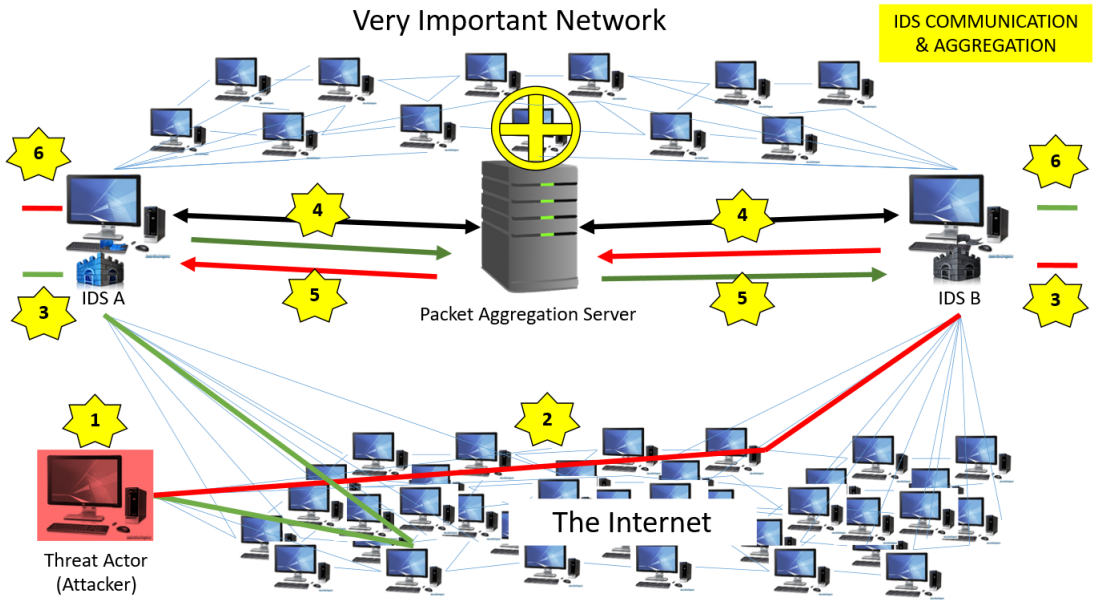


Figure 2.2: IDS Communication and Aggregation

The following list describes the order of events that is expected to occur in Figure 2.2.

1. A threat actor launches some attack using MPTCP, the red and green lines are the different subflows of the same attack.
2. The subflows travel via the Internet to the IDSs respectively.
3. The respective IDSs record the subflows
4. The IDSs report the subflows to a central server for analysis.
5. The server detects the malicious subflows and forwards the information to the IDSs respectively.
6. The IDSs do not let the subflows enter the “Very Important Network”

The end result is that the attacker does not reach his/her target.

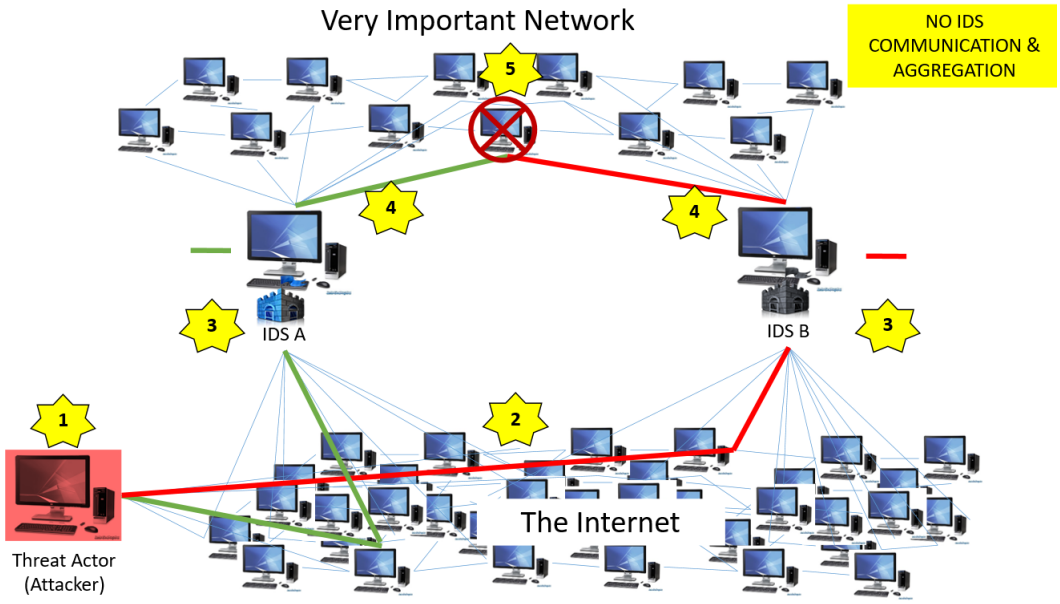


Figure 2.3: No IDS Communication and Aggregation

The following list describes the order of events that is expected to occur in Figure 2.3.

1. A threat actor launches some attack using MPTCP, the red and green lines are the different subflows of the same attack.
2. The subflows travel via the Internet to the IDSs respectively.
3. The respective IDSs record the subflows.
4. The IDSs do not detect malicious signatures and allow the subflows to enter the network.
5. The subflows join on the target and execute.

The end result is the malicious subflows were not detected and the attacker reached his/her target.

## 2.4 Related Work

This section presents work that has incorporated MPTCP and work that has explored DIDS.

### 2.4.1 Henry Foster's Thesis: Cross-Path Network IDS (NIDS) Evasion and Countermeasures

Henry Foster's thesis was oriented towards enhancing a single IDS to be able to detect cross-path fragmentation attacks that leveraged MPTCP. His thesis addressed and developed a methodology that presents how an IDS would detect cross-path fragmentation attacks, defend and mitigate cross-path fragmentation attacks, and how an IDS such as Snort might be enhanced to perform MPTCP stream reassembly. His experiments entailed both offensive and defensive scenarios. For instance, how MPTCP could defeat the Snort IDS and similarly how the Snort IDS enhancements would detect the MPTCP attack.

The discoveries from the results of the experiments yielded two defensive countermeasures. The countermeasures are either implementing a MPTCP proxy or a security testing tool known as Trudy. The former option would accept all incoming MPTCP subflows and connections, using the "socat" command [14], and essentially convert them to a single stream/flow similar to standard TCP, thus allowing Snort to detect the attack. The latter option functions as a TCP Main-in-the-Middle (MitM) [6]. The conclusion of Foster's thesis was that it is possible to evade a single MPTCP unaware IDS, and it is also possible to mitigate and defend against MPTCP cross-path fragmentation attacks via enhancing Snort with one of the two countermeasures mentioned in his thesis.

### 2.4.2 Distributed IDS Case Study

The "Distributed Intrusion Detection System for Wireless Sensor Networks" paper presents the use of DIDS to better secure Wireless Sensor Network (WSN)s [15]. Because WSNs are exposed to a variety of attacks, the best solution to prevent and detect the attacks is the use of IDSs. However, the aforementioned paper states, "IDS based security mechanisms suggested for other network models such as ad hoc networks, cannot directly be used in WSNs" [15]. Furthermore, according to the same paper, researchers have continuously proposed a variety of IDSs, resulting in classifying the approaches into two categories, "distributed and stand alone" [15]. Furthermore, Sharma et al. states that, the "distributed IDS can be further classified into hierarchical, mobile agent based and hybrid IDS" [15].

Sharma et al. states the definitions of the two categories and their subcategories as:

1. Stand-alone: In this category each node operates as independent IDS and is respon-

sible for detecting attacks only for itself. Such an IDS does not share any information or cooperate with other systems. This architecture implies that all the nodes of the network are capable of running IDS.

2. Distributed: Here, all nodes still are running their own IDS, but the IDS cooperate in order to create a global intrusion detection mechanism.
3. Hierarchical: In this case the network is divided into clusters with cluster-head nodes. These nodes are responsible for routing within the cluster and accept all the accusation messages from the other cluster members indicating something malicious. Additionally, the cluster-head nodes may also detect attacks against the other cluster-head nodes of the network, as they constitute the backbone of the routing infrastructure.
4. Mobile agent based: Mobile agent based IDSs use pieces of mobile code charged with a specific mission and sent to other nodes. Mobile agents are intelligent program threads that function continuously and are able to learn, communicate and migrate themselves from host to host to gather information and perhaps perform specific tasks on behalf of a user. Mobile agents are dispatched to hosts where they activate the sensor there, process collected data, and send it to the main station, which signals the agents to either stop collecting data or continue, with possible changes to the collection frequency and context [15].

Lastly, a section of this paper [15] addresses three ways a DIDS may be configured. The first configuration is known as "Purely Distributed". In this configuration an IDS is installed on every node, and each will check the neighboring nodes' behavior for actions and events that are not considered normal for that node. The second configuration is "Purely Centralized". For this configuration the Base Station (BS) has an IDS agent installed that needs to run an "additional special routing protocol that gathers or collects information from nodes to analyse the behaviour of sensor nodes collectively" [15]. Lastly, the third configuration is "Distributed Centralized". This configuration is oriented towards reducing power consumption and control overhead. In order to achieve this the IDS agent is installed on monitor nodes only, and these nodes are dual-hatted (two main functions). One function is to it act as a standard node, and the second function is to perform intrusion detection [15].

# CHAPTER 3:
# Single IDS Experiments

In this chapter we will be covering the experiments conducted in previous thesis work by Henry Foster [6]. We will replicate the tests via following the same procedure for each test as described in the prior thesis [6] but on a more realistic testbed consisting of physical network devices and computer hosts. This is necessary in order to ensure a solid foundation for our current and future assumptions.

## 3.1 Scope of Tests

In recent and related work pertaining to MPTCP offense and defense, most experimentation has been performed with Virtual Machine (VM)s. Even though VMs allow great flexibility and convenience, various factors that play a role in the results of experiments and other testing potentially cause the final results to be slightly, if not more, askew. Because of this, all of the following testbeds, except where one machine is a VM in two testbeds (Testbed 2 and Testbed3), are physically constructed, connected, and configured to run the tests. We will be testing:

- Snort3 Detection Capability
- Offensive MPTCP Applications
- Counter-measures for Offensive MPTCP Attacks

With these tests we validate previous thesis work and establish a foundation for further testing and development of MPTCP capabilities and defensive countermeasures for potential MPTCP attacks.

## 3.2 Building Testbeds

The inspiration for some of the testbeds for specific test are from Henry Foster's thesis experiments which also relate to MPTCP offensive measures and defensive countermeasures. The standard set of tools for each machine is the basic install of:

- Wireshark

- MPTCP Kernel
- Kali Linux VM to act as the Attacker in all tests.

The OS utilized in every computer, except the "Attacker", is Ubuntu 14.04, also referred to as "TrustyTahr". We decided this was our best option for all of the experiments because it has already been used in other related work [6], and it is currently supported by Christoph Paasch, Sébastien Barré, and other developers at multipath-tcp.org [7]. We were able to install the MPTCP Kernel in each machine, except the "Attacker". We installed this kernel on each machine for simplicity in the event we wanted or needed to change the machine's kernel from MPTCP capable to non-MPTCP capable. However, in one of our tests we had to manually configure and install the MPTCP kernel in order to perform certain tests. Lastly, the only other modification made to a few of the testbeds is that the NIDS is also used as a bridge between machines.

### 3.2.1 Testbed 1

**Purpose**

We built Testbed 1 to validate and expand on Henry's Scenario 1 tests. Henry's Scenario 1 purpose is: "Test Snort's ability to detect TCP content when attackers allow TCP to operate normally." [6]. For this scenario Henry networked:

- An attacker (Kali 2016 Virtual Machine) will send traffic to a victim.
- The victim (64-bit Ubuntu 14.04 Desktop Virtual Machine) will listen on TCP port 7878 (Test 1) and will be running a vulnerable service on port 7879.
- An IDS box (64-bit Ubuntu 14.04 Desktop Virtual Machine running Snort3)

The first test for his scenario was to detect the string "FIREFIREFIRE" when it is sent from the attacker to the victim. The second test was to attempt to exploit a service that executed received data by sending it shellcode. [6]. Henry made the hypothesis "Snort will generate an alert for both tests. The victim will successfully receive the string sent in Test 1. The attacker will be able to spawn a shell on the victim and interact with the victim computer in Test 2" [6]. The results were as he predicted and the result are as follows:

```
Test 1
```

```
07/15-02:22:46.581952 [**] [1:100000001:0] ‘‘FIREx3 detected!’’ [**]
[Priority: 0] {TCP} 192.168.182.134:44838 -> 192.168.182.135:7878}

Test 2
07/15-06:30:12.120354 [**] [1:100000002:0] ‘‘MSF!
linux/x64/shell_find_port’’ [**] [Priority: 0] {TCP}
192.168.182.130:4445 -> 192.168.182.135:7878
```

### Topology

Figure 3.1 displays the networking topology used in this experiment.

### Testbed 1 Description

In order to expand on this scenario, we begin testing with MPTCP enabled on all of the machines displayed in Figure 3.1. Testbed 1 consists of six machines labeled:

- Red one: An Attacker (Kali Linux v2016.2 VM) to send attack traffic.
- Red two: A NIDS and bridge (64-bit Ubuntu installed on a Windows 7 Pro) to monitor and forward traffic between the network. Running the following software:
  - Snort version 3.0.0-a4
  - Wireshark version 1.10.6
- Red three: A victim (64-bit Ubuntu installed on a Windows 7 Pro) will listen on port 30333 (Test 1) and running a vulnerable service on port TBD (Test 2). Running the following software:
  - Apache2 version 2.4.7
  - Wireshark version 1.10.6
- Red four: An observer (64-bit Ubuntu installed on a Windows 7 Pro) for future experimentation and to observe traffic to the victim. Running the following software:
  - Wireshark version 1.10.6
- Purple one: A switch to allow more connection to other machines in the future.
- Green one: A hub to forward traffic to all connected machines.

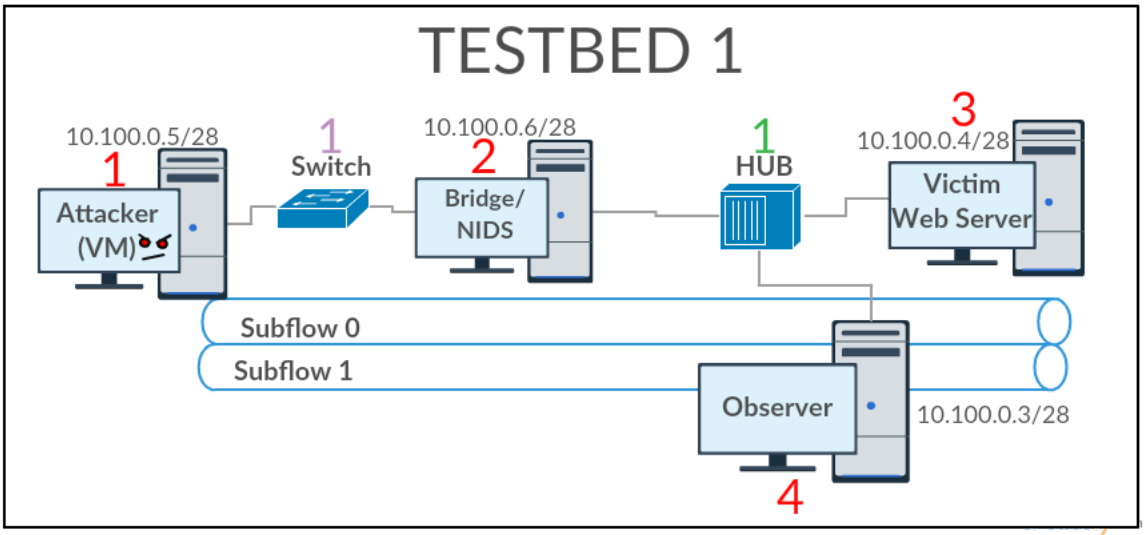


Figure 3.1: Testbed 1

Similar to Henry Foster's Scenario one tests, the tests that we will run on Testbed 1 are:

- Test 1: Testing Snort's ability to detect malicious content, a packet containing the string "SHUTDOWNALLMACHINES".
- Test 2: Snort's ability to detect a known attack, executing shellcode.

**Procedure for Test 1**

1. Start Snort3 Monitoring

   **Snort options used:**

   (a) -A alert_fast: logs alerts to the console.

   (b) -R <rule file>: include a specific rule set in the default policy

   (c) -c <conf>: Use this configuration

   (d) -i <interface>: interface(s) to capture on

   (e) --plugin-path <plug-in path>: where to find plug-ins

   **$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -r /home/testuser/Desktop/TB1/TB1_Test1/TB1_test_1a_from_victim.pcapng -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules**

2. Start netcat service on Victim

   **nc -lkp 50000**

3. Start Wireshark on victim and bridge machines
   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker
   **nc 10.100.0.4 50000**
5. Send malicious string
   **Type and send "SHUTDOWNALLMACHINES" in the netcat terminal**
6. End Snort monitoring
7. End wireshark captures
8. Analyze Results

**Procedure for Test 2**

1. Start Snort3 Monitoring
   **Snort options used:**
   (a) -A alert_fast: logs alerts to the console.
   (b) -R <rule file>: include a specific rule set in the default policy
   (c) -c <conf>: Use this configuration
   (d) -i <interface>: interface(s) to capture on
   (e) --plugin-path <plug-in path>: where to find plug-ins
   **$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -r /home/testuser/Desktop/TB1/TB1_Test1/TB1_test_1a_from_victim.pcapng -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules**
2. Start netcat service on Victim
   **nc -lkp 50000**

3. Start Wireshark on victim and bridge machines
   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker
   **nc 10.100.0.4 50000**
5. Send malicious payload that exploits vulnerable service

```
(msfvenom -p linux/x64/shell_find_port CPORT=4445 -f raw ; echo
,! ; cat -) | nc -p 4445 10.10.10.11 7878
```

**Using metasploit framework (MSF), send exploit via nc to the victim**

6. End Snort monitoring
7. End wireshark captures
8. Analyze Results

**Hypothesis**

If the traffic is not fragmented between different paths/subflows, then Snort3 will successfully generate an alert for both tests. The victim will receive the string sent in test 1. The attacker will be able to create a shell on the victim machine and interact with it in test 2.

### 3.2.2 Testbed 2

**Purpose**

We built Testbed 2 to validate Henry's Scenario 3 tests. Henry's Scenario 3 purpose is:

"Test Snort's ability to reassemble MPTCP data streams and to perform detection on the content of these streams when attackers attempt evasion with TCP session-splicing." [6].

For Henry's Scenario 3 he networked:

- An attacker (Kali 2016 Virtual Machine) will send traffic to a victim.
- The victim (64-bit Ubuntu 14.04 Desktop Virtual Machine) will listen on TCP port 7878 (Test 1) and be running a vulnerable service on port 7879.
- An IDS box (64-bit Ubuntu 14.04 Desktop Virtual Machine running Snort3).
- A splicer box (64-bit Ubuntu 14.04 running Python 3.5 with splicery.py) running Multipath TCP [6].

In Henry's Scenario 3, the splicer VM, the victim, and the IDS share a broadcast and collision domain which allows the IDS to observe traffic between the splicer and the victim. [6]. He then performed two tests, the first to attempt to detect the string "FIREFIREFIRE" when it was transmitted in single-bye packets from the splicer to the victim along multiple subflows. The second test to attempt to exploit a service that executes received data by sending it in shell code. His results for test 1 and two (respectively):

> No Snort alerts were generated by the first test. As can be seen from the Wireshark analysis of the two streams generated by Test 1, 'FIREFIREFIRE' was present in neither subflow stream. However, it was present in the larger MPTCP Data Stream. No Snort alerts were generated by the second test. The attacker was able to interact with a shell on the target. [6].

**Topology**

Figure 3.2 displays the networking topology used in this experiment

**Splicer.py Description**

The Splicer python script was developed by Henry Foster. It combines the functionality of a port-forwarder and a TCP session-splicer. The high level description Henry provided was:

> It is configured with the information to create a mapping between two socket addresses; it binds to and listens for incoming connections on one, and connects to the other when an incoming connection occurs. Splicer then begins to forward all the traffic received on each connection to the other connection [6].

The expected performance of the splicer is that is should be able to forward any exploit that targets the application layer service that uses TCP.

**Configuring Linux Kernel for MPTCP Session Splicing**

The only kernel we had to configure manually with Henry Foster's guidance, was the Splicer machine's, because we had to ensure the round-robin MPTCP scheduler was included in the build so that when we ran the splicer.py program it would alternate which subflows the exploit payload would transmit on. In order to configure the Splicer machine to function the way we needed, we performed the following using the guide:

1. Install dependencies:
   *sudo apt-get install git build-essential fakeroot*
   *kernel-package libqt4-dev pkg-config*

2. Obtain MPTCP source code:
   *git clone https://github.com/multipath-tcp/mptcp.git*
3. After successfully downloading the source code enter the directory and check out the desired branch, in our case mptcp_v0.90
   *cd mptcp*
   *git checkout mptcp_v0.90*
4. Launch the xconfig window:
   *make xconfig*
5. When the Graphical User Interface (GUI) appears in order to enable an option double click. Enable the round-robin scheduler option. Found under, "Networking Options –> MPTCP:advanced scheduler control –> MPTCP Round-Robin" and also set the default MPTCP Scheduler to "Round-Robin".
6. Save the configuration by hitting the keys Ctrl+S or clicking the save icon. Close the window.
7. Remove the stale parameters from the previous builds before starting the new compilation.
   *make-kpkg clean*
8. Finally, build the package:
   *fakeroot make-kpkg -j 8 --initrd --revision=0.90.mptcp.complete kernel_image*
   This command will create a .deb package that will be used to install the new linux-image with the dpkg utility.
   NOTE: If this does not succeed because of a "Permission Denied" error, use 'sudo' instead of 'fakeroot'.
9. Install the new kernel:
   *cd ..;sudo dpkg -i linux-image-3.18.34_0.90.mptcp.complete_amd64.deb*
   NOTE: If the you used your own settings, revision name, mptcp branch, the name of the file will be different but still begin with 'linux-image'.

**Testbed 2 Description**

The testbed consists of:

- Red one: An Attacker (Kali Linux 2016 VM) to send attack traffic to the Splicer.
- Red two: The Splicer (64-bit Ubuntu Desktop installed on a Windows 7 Pro) runs

Splicer.py [6] program on all the traffic sent to it by the Attacker. Running the following software:

  - Virtualbox version 5.18
  - Python 3.5

- Red three: A NIDS and bridge (64-bit Ubuntu installed on a Windows 7 Pro) to monitor and forward traffic between the network. Running the following software:
  - Snort version 3.0.0-a4
  - Wireshark version 1.10.6
- Red four: A victim (64-bit Ubuntu installed on a Windows 7 Pro) will listen on port 30333 (Test 1) and be running a vulnerable service on port TBD (Test 2). Running the following software:
  - Apache2 version 2.4.7
  - Wireshark version 1.10.6
- Red five: An observer (64-bit Ubuntu installed on a Windows 7 Pro) for future experimentation and to observe traffic to the victim. Running the following software:
  - Wireshark version 1.10.6
- Purple one: A switch to allow more connections to other machines in the future.
- Green one: A hub to forward traffic to all connected machines.

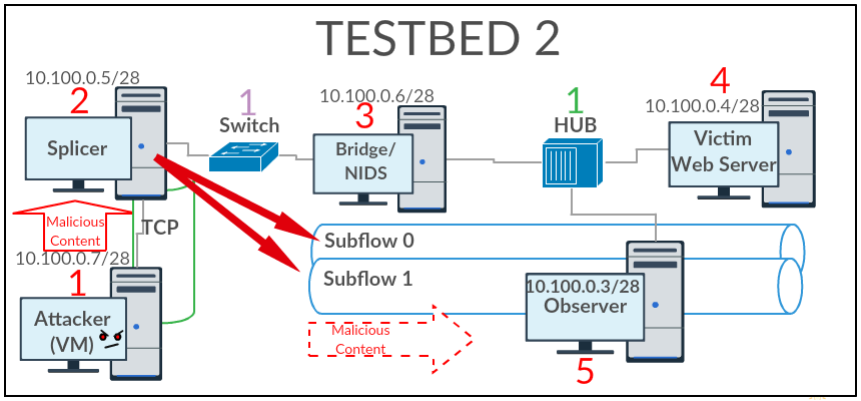


Figure 3.2: Testbed 2

We will be performing the same two tests we previously performed on Testbed 1 in order to validate the Scenario 3 tests, Henry performed: test 1 to test Snort's ability to detect malicious content; test 2 to test Snort's ability to detect a known attack.

**Procedure for Test 1**

1. Start Snort3 Monitoring

   **Snort options used:**

   (a) -A alert_fast: logs alerts to the console.

   (b) -R <rule file>: include a specific rule set in the default policy

   (c) -c <conf>: Use this configuration

   (d) -i <interface>: interface(s) to capture on

   (e) --plugin-path <plug-in path>: where to find plug-ins

   **$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -r /home/testuser/Desktop/TB1/TB1_Test1/TB1_test_1a_from_victim.pcapng -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules**
2. Start netcat service on Victim

   **nc -lkp 50000**

3. Start Wireshark on victim and bridge machines

   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker

   **nc 10.100.0.4 50000**
5. Start Splicer.py and connect attacker to bridge

   **splicer.py 0.0.0.0 10.100.0.4**
6. Send malicious string

   **Type and send "SHUTDOWNALLMACHINES" in the netcat terminal**
7. End Snort monitoring
8. End wireshark captures
9. Analyze Results

**Procedure for Test 2**

1. Start Snort3 Monitoring

   **Snort options used:**

   (a) -A alert_fast: logs alerts to the console.

   (b) -R <rule file>: include a specific rule set in the default policy

   (c) -c <conf>: Use this configuration

   (d) -i <interface>: interface(s) to capture on

(e) --plugin-path <plug-in path>: where to find plug-ins

**$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -r /home/testuser/Desktop/TB1/TB1_Test1/TB1_test_1a_from_victim.pcapng -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules**

2. Start netcat service on Victim
   **nc -lkp 50000**

3. Start Wireshark on victim and bridge machines
   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker
   **nc 10.100.0.4 50000**
5. Start Splicer.py and connect attacker to bridge
   **splicer.py 0.0.0.0 10.100.0.4**
6. Send malicious payload that exploits vulnerable service
   **Using metasploit framework (MSF), send exploit via nc to the victim**
7. End Snort monitoring
8. End wireshark captures
9. Analyze Results

**Hypothesis**

If the splicer.py is used to fragment attack traffic, then Snort3 will not generate an alert for either test. However, the victim will receive the string sent in test 1 and the attacker will be able to create a shell on the victim machine and interact with it in test 2.

### 3.2.3 Testbed 3

**Purpose**

We build Testbed 3 to validate Henry's suggested defensive measure against Cross-path MPTCP Attacks. The idea is to have a machine, Linux Router, running the socat command with a specific configuration in order to take the multiple ingress subflows and turn them into a single standard TCP egress flow towards its intended target after passing through the IDS box. The specific command used is of the format:

**socat TCP-LISTEN:<LISTEN PORT>,fork TCP:<TARGET IP>:<TARGET PORT>**

Henry's results from leveraging socat to implement an MPTCP proxy resulted in Snort3 generating an alert even after the attack traffic was spliced. The alert Snort3 generated was [6]:

```
09/08-00:56:37.215615 [**] [1:100000001:0] "FIREx3 detected!" [**]
[Priority: 0] {TCP} 192.168.41.254:33377 -> 192.168.41.1:30333
```

### Topology

Figure 3.3 displays the networking topology used in this experiment.

### SOCAT Description

According to Soact's manual,

> Socat is a command line based utility that establishes two bidirectional byte streams and transfers data between them. Because the streams can be constructed from a large set of different types of data sinks and sources (see address types), and because lots of address options may be applied to the streams, socat can be used for many different purposes.

In our case we are using socat to accept the multiple subflows of the MPTCP connection and send them out of another socket that is configured for standard TCP, effectively recombining the subflows and allowing the malicious content to be detected.

### Testbed 3 Description

The testbed consists of:

- Red one: An Attacker (Kali Linux 2016 VM) to send attack traffic to the Splicer.
- Red two: The Splicer (64-bit Ubuntu Desktop installed on a Windows 7 Pro) runs Splicer.py [6] program on all the traffic sent to it by the Attacker. Running the following software:
  - Virtualbox version 5.18

  - Python 3.5
- Red three: A NIDS and bridge (64-bit Ubuntu installed on a Windows 7 Pro) to monitor and forward traffic between the network. Running the following software:
  - Snort version 3.0.0-a4
  - Wireshark version 1.10.6
- Red four: A victim (64-bit Ubuntu installed on a Windows 7 Pro) will listen on port 30333 (Test 1) and run a vulnerable service on port TBD (Test 2). Running the following software:
  - Apache2 version 2.4.7
  - Wireshark version 1.10.6
- Red five: An observer (64-bit Ubuntu installed on a Windows 7 Pro) for future experimentation and to observe traffic to the victim. Running the following software:
  - Wireshark version 1.10.6
- Purple one: A switch to allow more connections to other machines in the future.
- Green one: A hub to forward traffic to all connected machines.
- Grey one: A Router (64-bit Ubuntu installed on a Windows 7 Pro configured as a router). Running the following software:
  - Wireshark version 1.10.6
  - quagga version 0.99.22.4
  - socat version 1.7.2.3

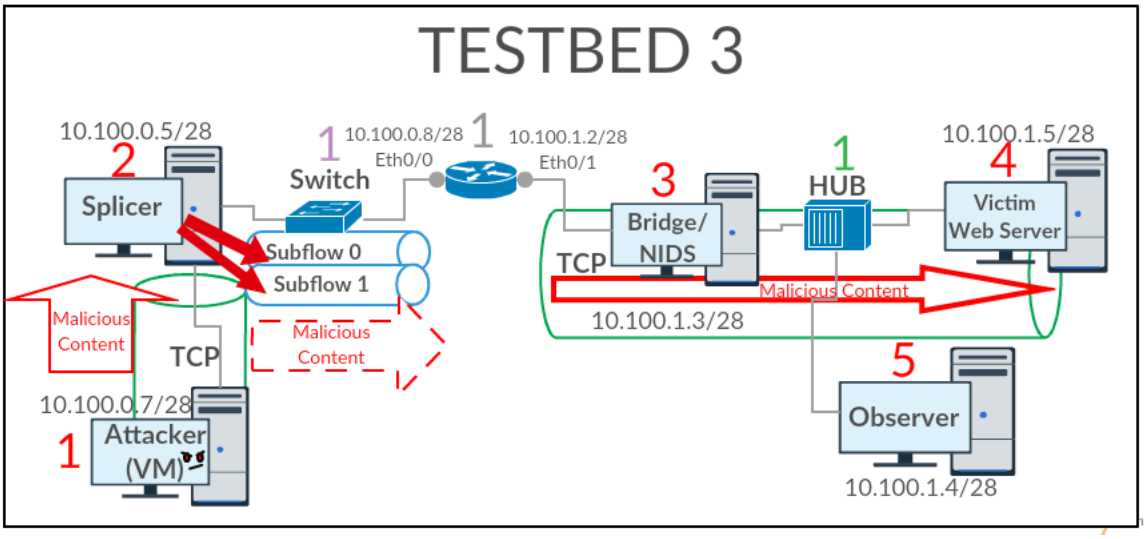


Figure 3.3: Testbed 3

We will be performing the same two tests we previously performed on Testbed 1 and Testbed 2, in order to validate the defensive countermeasure tests, Henry performed: test 1 to test Snort's ability to detect malicious content; test 2 to test Snort's ability to detect a known attack.

**Procedure for Test 1**

1. Start Snort3 Monitoring
   **Snort options used:**
   (a) -A alert_fast: logs alerts to the console.
   (b) -R <rule file>: include a specific rule set in the default policy
   (c) -c <conf>: Use this configuration
   (d) -i <interface>: interface(s) to capture on
   (e) --plugin-path <plug-in path>: where to find plug-ins
   **$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -r /home/testuser/Desktop/TB1/TB1_Test1/TB1_test_1a_from_victim.pcapng -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules**
2. Start netcat service on Victim
   **nc -lkp 50000**
3. Start Wireshark on victim and bridge machines
   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker
   **nc 10.100.0.4 50000**
5. Start Splicer.py and connect attacker to bridge
   **splicer.py 0.0.0.0 10.100.0.4**
6. Start socat
   **socat TCP-LISTEN:50000, fork TCP:10.100.1.5:50000**
7. Send malicious string
   **Type and send "SHUTDOWNALLMACHINES" in the netcat terminal**
8. End Snort monitoring
9. End wireshark captures
10. Analyze Results

**Procedure for Test 2**

1. Start Snort3 Monitoring
   **Snort options used:**
   (a) -A alert_fast: logs alerts to the console.
   (b) -R <rule file>: include a specific rule set in the default policy
   (c) -c <conf>: Use this configuration
   (d) -i <interface>: interface(s) to capture on
   (e) --plugin-path <plug-in path>: where to find plug-ins
   **$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -r /home/testuser/Desktop/TB1/TB1_Test1/TB1_test_1a_from_victim.pcapng -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules**
2. Start netcat service on Victim
   **nc -lkp 50000**
3. Start Wireshark on victim and bridge machines
   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker
   **nc 10.100.0.4 50000**
5. Start Splicer.py and connect attacker to bridge
   **splicer.py 0.0.0.0 10.100.0.4**
6. Start socat
   **socat TCP-LISTEN:50000, fork TCP:10.100.1.5:50000**
7. Send malicious payload that exploits vulnerable service
   **Using metasploit framework (MSF), send exploit via nc to the victim**
8. End Snort monitoring
9. End wireshark captures
10. Analyze Results

**Hypothesis**

If socat is used while the splicer.py program is being used, Snort3 will generate an alert for each test. However, the victim will receive the string sent in test 1 and the attacker will be able to create a shell on the victim machine and interact with it in test 2.

## 3.3 Results

We configured Snort3 for content based detection, therefore Testbed 1 and Testbed 3 were able to be detected. However, Testbed 2 was not able to be detected because the payload/content was fragmented among the different subflows. Each testbed was monitored with the same rule we wrote for Snort3. For each testbed's test 1 and test 2, respectively, the rule was:

```
alert tcp any any -> any any (content:"SHUTDOWNALLMACHINES"; msg:"SKYNET
attack detected!!"; sid:10000001;)
```

```
alert tcp any any -> any any (content:"\xffH1\xdb\xb3\x14H)\xdcH\x8d\x14$H\x8
dt$"; msg:"MSF! linux/x64/shell\_find\_port"; sid:10000003;)
```

### 3.3.1 Testbed 1 Results

**Test 1**

A Snort alert was generated when the packet containing the content "SHUTDOWNALL-MACHINES" was transmitted from the attacker to the victim machine. It yielded the following alert:

```
02/19/-15:03:28.273234 [**] [1:10000001:0]"SKYNET attack detected!!"
[**] [Priority: 0] {TCP} 10.100.0.8:40068 -> 10.100.1.5:50000
```

**Test 2**

A Snort alert was generated when the packets containing the signature content of an msf exploit was detected. The exploit was able to create a shell that was able to access the victim machine's directories. The alert generated by Snort3 was:

```
02/21-12:02:47.931875 [**] [1:10000003:0] "MSF! linux/x64/shell_
find_port" [**] [Priority: 0] {TCP} 10.100.0.7:4445 -> 10.100.0.4:7878
```

### 3.3.2 Testbed 2 Results

#### Test 1

Snort3 failed to generate an alert for this Testbed's test 1, as can be seen from the Wireshark analysis (see Figure 3.4 and Figure 3.5). The content "SHUTDOWNALLMACHINES" was not present in either subflow but still appeared on the victim's terminal

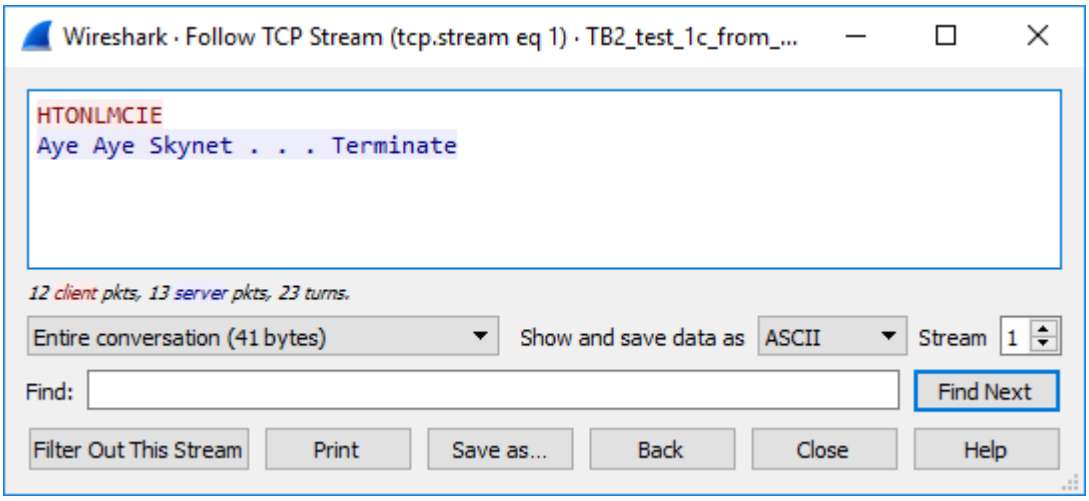


Figure 3.4: Subflow Red and Blue

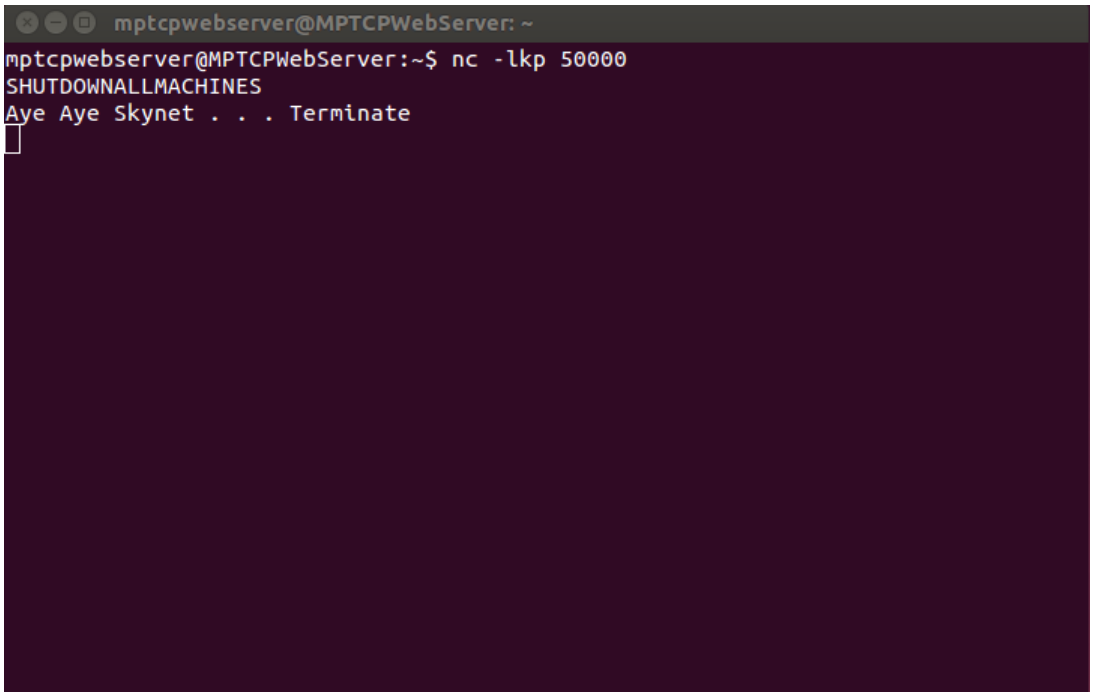


Figure 3.5: Attacker's Content Reassembled and Presented On Victims Terminal

**Test 2**

Snort3 failed to generate an alert and the attacker was able to interact with a shell of the victim's.

### 3.3.3 Testbed 3 Results

**Test 1**

Snort3 was successfully able to generate an alert after the content was reassembled and analyzed by the detection engine. The alert generated was:

```
02/19-15:03:28.176318 [**] [1:10000001:0] "SKYNET attack detected!!"
[**] [Priority: 0] {TCP} 10.100.1.2:35597 -> 10.100.1.5:50000
```

**Test 2**

Snort3 was successfully able to generate an alert after the signature of the content was reassembled and analyzed by the detection engine. The alert generated was:

```
02/21-16:08:31.231551 [**] [1:10000003:0] "MSF! linux/x64/shell_find_port"
[**] [Priority: 0] {TCP} 10.100.0.8:44728 -> 10.100.1.5:50000
```

### 3.3.4 Challenges

For these testbeds and tests the challenges varied between each. Furthermore, each machine and test development had its own challenges. The most challenging was the Snort3 machine.

- **Attacker**: Runs Kali Linux. The logic behind using Kali Linux as the attacker for these experiments is because it comes with a suit of tools that are oriented towards:
  - Penetration Testing
  - Security Research
  - Reverse Engineering

  Kali Linux contains a vast array of penetration testing tools. Using the tools that are provided within Kali, we are able to discover a vulnerability and exploit it.
- **Bridge/NIDS**: The github guide for installing Snort3 is slightly confusing. After a few hours we were able to figure it out and get it running properly. Lastly, we were stuck for a few weeks on figuring out how to get Snort3 to analyze the packets after reassembling the fragmented packets in Testbed 3.
- **Victim**: Is the intended target for all of the attacks being launched from the Attacker, Kali Linux VM. The only challenges for this machine was syntax.
- **Observer**: An experimental add for future procedures.
- **Router**: The challenge for this machine was installing and configuring quagga. Quagga is a package with a few dependencies that enables a machine to emulate the functions of a router, allowing us to test Snort3 on routed traffic.
- **Splicer**: The challenge with this machine was making sure we were not running too many process. While we thought we had enough memory and storage, the machine would sometimes crash (blue/black screen). Furthermore, some changes we made were not permanent. So when the machine would crash we would forget to re-implement the changes and it would interfere with testing and the results.

## 3.4 Chapter Summary

The rationale for repeating Henry Foster's tests was to validate his hypotheses for the expected behavior of Snort3 and MPTCP. This foundation allows us to expand on his experiments and create new trials to be conducted on new testbeds that are configured differently. The results from repeating his tests on replicated versions of his testbeds have revealed that a determined and competent adversary is able to create tools that can leverage the capabilities that MPTCP offers to successfully launch a cross-path fragmentation attacks. Furthermore, they have revealed that the Snort IDS can be enhanced to defend against these attacks only if the attack signature is known, and the specific IDS that observes all subflows containing the fragmented signature has the capability of reassembling the cross-path fragmented signature.

# CHAPTER 4:
# Multi-IDS Experiments

In this chapter, we will be experimenting with adding an additional IDS machine. The motivation for this is that a majority of companies, organizations, and other enterprises incorporate multiple IDSs. Multiple IDSs are deployed to reduce the work load of all the IDSs and better monitor traffic for malicious content.

## 4.1 Designing and Implementing IDS Cooperation

This section details a design that allows multiple IDSs to cooperate in order to detect spliced subflows that are from the same source and destined for the same target, but taking different paths.

### 4.1.1 Designation of Primary and Secondary IDSs

There are several potential strategies to enable multi-IDS cooperation, such as, installing Snort3 on every IDS and having each IDS communicate with every other IDS to obtain a full picture of traffic entering the enterprise. While every IDS would be able to reassemble the cross-path fragmented traffic, doing so would incur a significant amount of communication overhead. Another strategy would be having at least two IDSs that monitor all traffic on the network. This strategy would be great for small networks that are not constantly busy. However, this becomes a constraint if the network eventually becomes large and consistently active over long periods of time. We have chosen an approach that designates one of the IDSs as primary and all others as secondary. A secondary IDS must detect and forward a copy of all MPTCP traffic to the primary IDS, which is responsible for assembling subflows and checking for MPTCP exploits. The rationale for primary and secondary IDSs is to reduce the potential overhead generated by having too many enhanced IDSs, while still having the ability to adjust to the size of the organization's network and effectively monitor and share subflow information for cross-path fragmented malicious signatures. This ensures that malicious traffic that may have been split between multiple sublfows and multiple network paths will be collected and reassembled by an IDS that is able to run rules against the reassembled traffic. Lastly the primary IDSs are running Snort3 and the secondary IDSs

are running Snort 2.9.6.0. This is because Snort3 requires more effort, time, and resources to install, configure, and run, and if we ran Snort3 for every IDS network performance might be adversely affected. Aside from the different Snort version, the secondary IDSs will be configured the same as the primary IDSs.

**MPTCP Traffic Redirection Methodology**

The secondary IDS will serve as a MPTCP traffic detector and re-director. The MPTCP traffic will be detected by checking each packet's MPTCP Capable Flag and the port the packet arrived on. If the flag or port, used for MPTCP, is detected the packet will be forwarded to the primary IDS. Furthermore, the primary and secondary IDSs will serve as a bridges. The reason both IDS machines serve as bridges is to allow the IDS to inspect the traffic as it passes through the bridge. Table 4.1 displays the routing tables required for Testbed 4.

## 4.2 Validation

We validate our proposed solution by building a testbed that is configured with a pair of primary and secondary IDSs that monitor traffic on two separate paths into the victim's network. The primary and secondary IDS approach we implement in Testbed 4 will sufficiently demonstrates that having at least one primary and secondary IDS will be able to detect cross-fragmented malicious content that is deployed on two different paths toward two different IDSs.

### 4.2.1 Scope of Tests

The multiple IDS experiments require different testing than the experiments with a single IDS. We will be performing tests based on the assumption that:

1. The adversary is able to send traffic from their splicer machine to multiple IDSs.
2. The adversary configured their splicing machine to use MPTCP's fullmesh path manager with two subflows per link and its roundrobin scheduler, in order to send spliced attack traffic among each link's two subflows.
3. The organization's IDSs are configured as primary and secondary IDSs; in this case we will be using one primary IDS and one secondary IDS.

Table 4.1: Testbed 4 Routing Tables

| Destination Prefix | Outgoing Interface |
|---|---|
| Splicer | |
| 10.100.0.* /24 | eth0 (0) |
| 10.100.1.* /24 | eth4 (1) |
| Primary Snort | |
| 10.100.0.* /24 | eth0 (0) |
| 10.100.2.* /24 | eth1 (1) |
| 10.100.3.* /24 | eth2 (2) |

| Destination Prefix | Outgoing Interface |
|---|---|
| Secondary Snort | |
| 10.100.1.* /24 | eth0 (0) |
| 10.100.2.* /24 | eth1 (1) |
| 10.100.4.* /24 | eth2 (2) |
| Victim | |
| 10.100.3.* /24 | eth0 (0) |
| 10.100.4.* /24 | eth1 (1) |

We will be performing the same test that was conducted on Testbed 3 test 1, to ensure the secondary IDS's ability to detect and redirect MPTCP traffic, and to ensure the primary IDS's ability to reassemble the spliced traffic among the two links (two subflows). This test is performed after the routing tables have been modified according to Table 4.1.

### 4.2.2 Multi-IDS Configuration Testing

**Purpose**

We designate an IDS with Snort3 installed as primary and another IDS with an older version of Snort as a secondary. The secondary, will detect and forward a copy of all MPTCP traffic to the primary upon detection. In order to implement this new concept we will have to be able to filter/detect MPTCP traffic. In order to detect MPTCP traffic we write a rule that matches the "TCP Option" field of a packet that may contain "MPTCP Capable" flag. Specifically we are looking for the expected hexadecimal value of "0x1e". To ensure we are not observing false positives we will be running Wireshark with the filter: **tcp.option_kind==0x1e**.

### Topology

Figure 4.1 displays the networking topology used in the following experiments.

### Testbed 4 Description

The testbed consists of:

- Red one: An Attacker (Kali Linux 2016 VM) to send attack traffic to the Splicer.
- Red two: The Splicer (64-bit Ubuntu Desktop installed on a Windows 7 Pro) runs Splicer.py [6] program on all the traffic sent to it by the Attacker. Running the following software:
  - Virtualbox version 5.18
  - Python 3.5
- Red three: A NIDS and bridge (64-bit Ubuntu installed on a Windows 7 Pro) to monitor, reassemble, and forward traffic between the networks. Running the following software:
  - Snort version 3.0.0-a4
  - Wireshark version 1.10.6
- Red four: A victim (64-bit Ubuntu installed on a Windows 7 Pro) will listen on port 30333 (Test 1) and run a vulnerable service on port 50000 (Test 2). Running the following software:
  - Apache2 version 2.4.7
  - Wireshark version 1.10.6
- Red five: A secondary NIDS and bridge (64-bit Ubuntu installed on a Windows 7 Pro) to monitor and redirect MPTCP traffic to the primary IDS and forward all other traffic between the networks. Running the following software:
  - Snort version 2.9.6.0
  - Wireshark version 1.10.6
- Red six: Henry Foster's VM (64-bit Ubuntu) to monitor and reassemble cross-fragmented traffic and generate alerts. Running the following software:
  - Snort version 3.0.0-a4
  - Wireshark version 1.10.6
  - Python 3.5

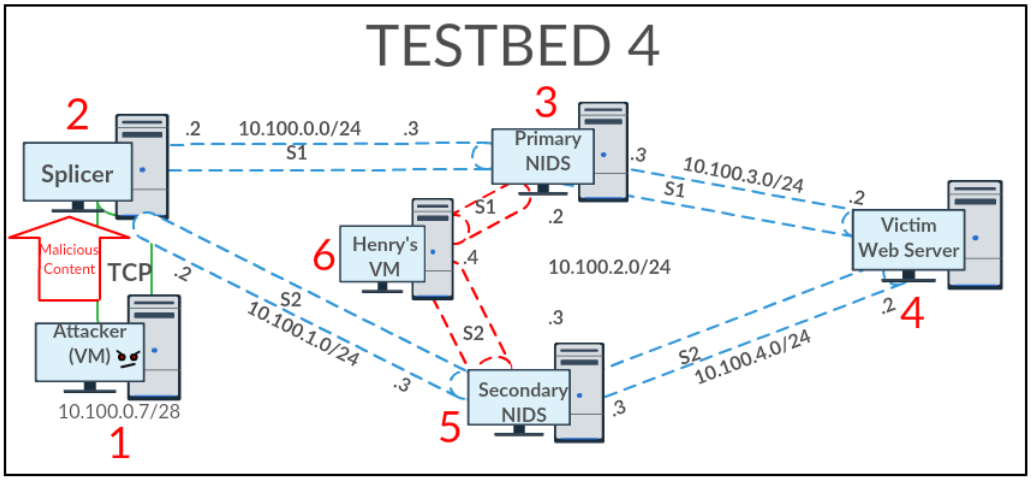


Figure 4.1: Testbed 4

**Procedure for Test 1**

1. Start Snort2 Monitoring
   **$my_path/bin/snort -c /home/testuser/snort_bin/etc/snort/snort.lua --plugin-path /home/testuser/snort_bin/lib -A alert_fast -R /home/testuser/snort_bin/etc/snort/local.rules -i ens33**
2. Start netcat service on Victim
   **nc -lkp 50000**
3. Start Wireshark on victim and IDS machines
   **<in terminal> sudo wireshark**
4. Connect to victim via netcat from the attacker
   **nc 10.100.0.2 50000**
5. Start Splicer.py and connect attacker to victim
   **splicer.py 0.0.0.0 10.100.3.2**
6. Send malicious string
   **Type and send "SHUTDOWNALLMACHINES" in the netcat terminal**
7. End Snort monitoring
8. End wireshark captures
9. Analyze Results

**Hypothesis**

If the secondary IDS is able to detect and redirect all MPTCP traffic, then the primary IDS should be able to reassemble the spliced traffic between the subflows and detect the malicious content.

## 4.3 Results

We did not have to alter Snort3's configuration from Testbed 3 in order to detect the attack traffic. However, we had to alter the "mp_tracker.py" program from Foster's thesis work [6] to account for the additional path that the malicious traffic took. Please refer to the "mp_tracker.py Modifications" section of the appendix for further details.

**Test 1**

The malicious traffic was detected and clones were made and sent to the primary IDS. Upon both subflows of malicious traffic arriving at the primary IDS, the traffic was successfully reassembled. An alert was generated for each subflow that contained fragments of the malicious signature:

```
05/31/-18:51:04.041607 [**] [1:10000001:0]"SKYNET attack detected!!"
[**] [Priority: 0] {TCP} 10.100.0.2:56909 -> 10.100.3.2:50000
05/31/-18:51:04.143886 [**] [1:10000001:0]"SKYNET attack detected!!"
[**] [Priority: 0] {TCP} 10.100.1.2:55994 -> 10.100.4.2:50000
```

The order of events is presented by the following figures. Figure 4.2 displays the Attacker VM sending the malicious content to the splicer.py program running on the host OS. Figure 4.3 displays the Wireshark capture of both subflows and the reassembly and detection of the malicious content. Figure 4.4 shows the Victim receiving the content of both subflows. Figures 4.5 and 4.6 display the reassemble stream of each subflow.

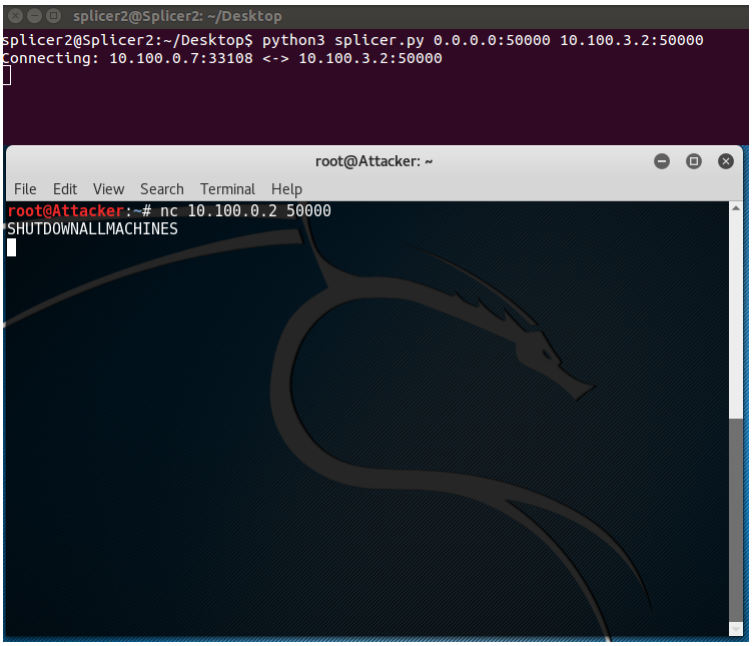


Figure 4.2: MPTCP Attack Launched and Spliced

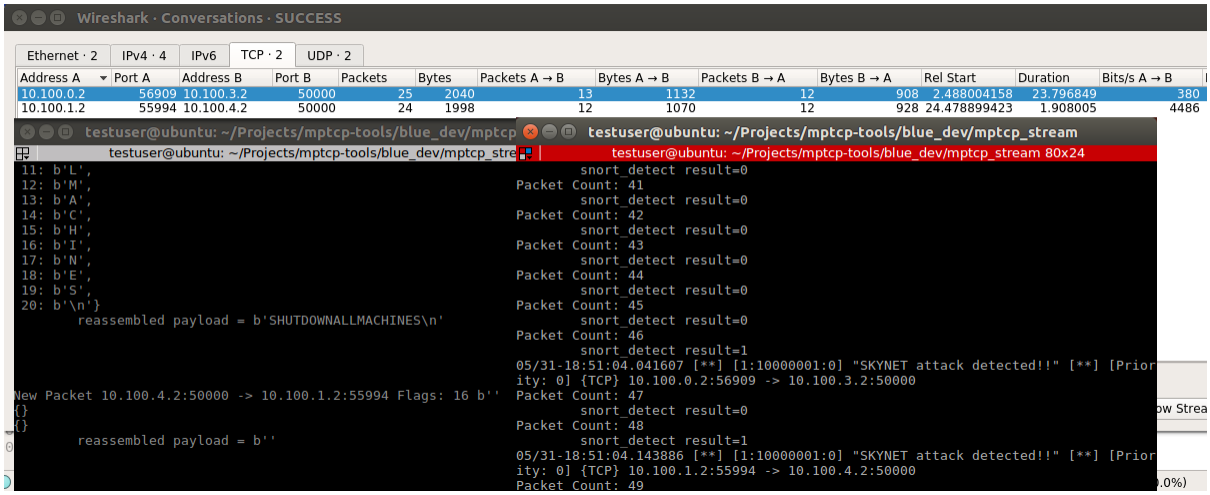


Figure 4.3: Snort3 Receiving Traffic and Generating Alerts

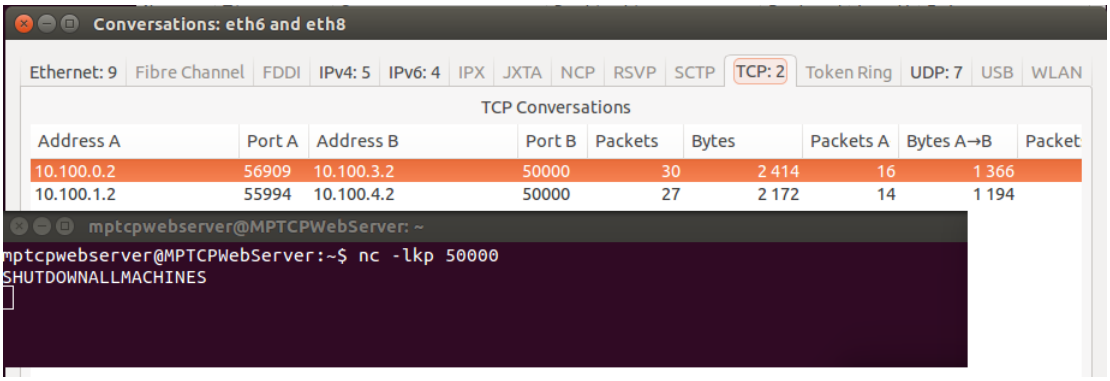


Figure 4.4: Victim Receiving Traffic

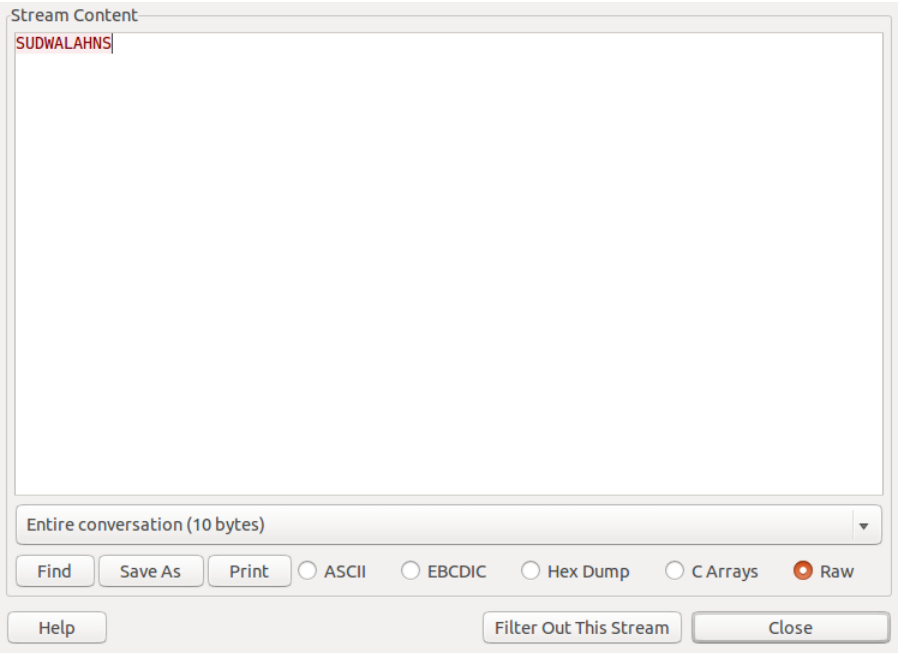


Figure 4.5: Subflow 0; 10.100.0.2 -> 10.100.3.2

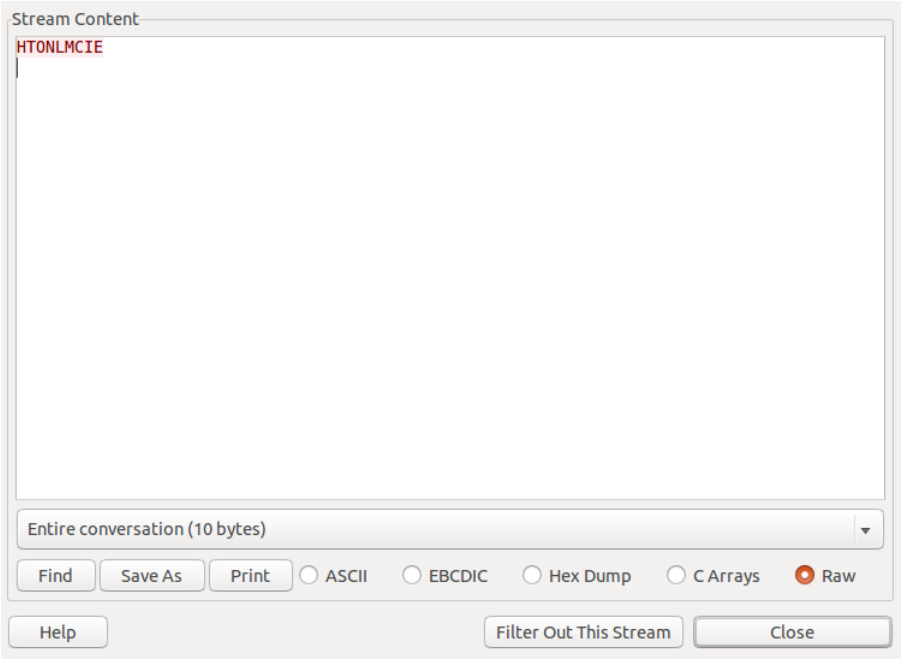


Figure 4.6: Subflow 1; 10.100.1.2 -> 10.100.4.2

## 4.4 Chapter Summary

In this chapter we explained the methodology of redirecting MPTCP capable packets to a primary IDS, and performed a validation test on a testbed with a pair of primary and secondary IDSs monitoring two distinct network entry points to a victim server. The focus of this testing is to be able to detect MPTCP traffic and reassemble cross-fragmented traffic. Figure 4.1 displays the logical map of the testbed we performed the tests on. Our hypothesis was, if the secondary IDS is able to detect and redirect all MPTCP traffic, then the primary IDS should be able to reassemble the spliced traffic between the subflows and detect the malicious content. Our tests have reflected that our hypothesis was correct. The results display that any subflow that contains fragments of the malicious content will be attributed with sending the malicious traffic.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 5:
# Conclusion

In this thesis we investigated methods of configuring and enhancing IDSs to cooperate in a manner that allows them to detect a MPTCP attack using cross-path fragmentation across multiple IDSs. This is an important subject because of the increasing number of large networks that incorporate more than one IDS and the limited deployment experience of MPTCP.

In Chapter 3 we established a foundation to expand and experiment upon by repeating Henry Foster's tests. The results have revealed that the Snort IDS can be enhanced to defend against these attacks only if the attack signature is known, and the specific IDS that observes all subflows containing the fragmented signature has the capability of reassembling the cross-path fragmented signature. We are able to obtain expected results on a more realistic testbed, which shows that the assumptions about MPTCP and Snort3 are credible. With a solid credible foundation we create a new trial and testbed to test our hypothesis about MPTCP and Snort3.

In Chapter 4 we demonstrated the feasibility of adding one or more secondary IDSs to assist a primary IDS with defending a large network with multiple external access links against cross-path MPTCP attacks. The defense is based on a IDS cooperation strategy where each secondary IDS clones and forwards a copy of all MPTCP traffic it sees to the primary IDS. Our methodology for detecting and redirecting traffic is based on the "MPTCP Capable" flag in the TCP option field. We are able to avoid manipulating the source and destination IP addresses of each packet by cloning the packet and only sending the clone to the primary IDS. In Testbed 4 we had to create static routes between the machines because Linux routers were incorporated. The tests we performed followed the procedures similar to Testbed 3. The results of the tests confirmed our hypothesis that if the secondary IDS were to detect and clone all MPTCP traffic, then the primary IDS would be able to reassemble and detect the malicious traffic.

## 5.1 Future Work

Future work is available for researching and developing a message exchange format that accounts for MPTCP. The Intrusion Detection Exchange Format Working Group (IDWG) addresses topics and areas of research around Intrusion Detection Message Exchange Format (IDMEF). RFC4766 defines the requirements expected for IDMEF and conveys the rationale for its importance. One of which is that having a common message exchange format would allow IDS research to "migrate into commercial products more easily." [16]

Snort is just one of many kinds of available IDSs. Additional future work could be oriented toward researching how to have two different kinds of IDSs cooperate in an efficient manner. This may potentially lead back into researching and developing a common IDMEF.

In this thesis we created and used a single methodology for implementing DIDS. Research and development can be oriented toward implementing DIDS with different methodologies which may or may not yield more valuable results. For instance, implementing DIDS on for a wireless network instead of a wired network.

Lastly, research and development can be done toward developing a network model that would simulate the effects of DIDS and MPTCP. This is important because MPTCP can be complex in operational environments. An accurate simulation model would allow a user to examine the behavior of MPTCP in a controlled environment with a reasonable level of deployment cost.

# APPENDIX: Snort3 Install Guide, Bashrc Tutorial, and "mp_tracker.py" Modifications

## A.1 SNORT 3.0 Install Guide

This appendix details the sources and commands used to install Snort 3.0 on Ubuntu 14.04.

### A.1.1 Dependencies

According to Snort3's github instructions if you already have built snort then you should be good to go. If not these are the dependencies required.

- autotools or cmake to build from source
- daq from http://www.snort.org for packet IO
- dnet from https://github.com/dugsong/libdnet.git for network utility functions
- g++ >= 4.8 or other C++11 compiler
- hwloc from https://www.open-mpi.org/projects/hwloc/ for CPU affinity management
- LuaJIT from http://luajit.org for configuration and scripting
- OpenSSL from https://www.openssl.org/source/ for SHA and MD5 file signatures, the protected_content rule option, and SSL service detection
- pcap from http://www.tcpdump.org for tcpdump style logging
- pcre from http://www.pcre.org for regular expression pattern matching
- pkgconfig from https://www.freedesktop.org/wiki/Software/pkg-config/ to locate build dependencies
- zlib from http://www.zlib.net for decompression [17]

### A.1.2 Installing Dependencies

**Create a directory for all the SNORT files** we will be downloading e.g.(snort_files). We will be installing all of the dependencies on 64-bit Ubuntu 14.04. Starting from the top of their list of required dependencies. Not mentioned on the github page, you'll need to install a few other dependencies in order to compile the dependencies, **sudo apt-get install libdumbnet-dev libglib2.0-dev libpcap-dev bison flex./**.

**Cmake**

For this thesis we will use cmake. In order to install cmake run the following command in the terminal.

**sudo apt-get install cmake**

In order to check that it installed successfully run:

**cmake -version**

**Daq**

If the following command does not work, go to the SNORT website and under the downloads page should be a listing for the SNORT 3.0 daq.

**wget https://www.snort.org/downloads/snortplus/daq-2.2.1.tar.gz**

**Dnet (libdnet)**

**sudo apt-get install libdnet**

**g++ >= 4.8**

**sudo apt-get install g++**

**dpkg –list | grep compiler**

**hwloc**

**wget https://www.open-mpi.org/software/hwloc/v1.11/downloads/hwloc-1.11.5.tar.gz**

**LuaJIT**

**wget http://luajit.org/download/LuaJIT-2.0.4.tar.gz**

**OpenSSL**

**wget https://www.openssl.org/source/openssl-1.0.2k.tar.gz**

**Pcap**
**wget http://www.tcpdump.org/release/tcpdump-4.8.1.tar.gz**

**Pcre**
**wget https://ftp.pcre.org/pub/pcre/pcre-8.40.tar.gz**

**Pkgconfig**
**wget https://pkg-config.freedesktop.org/releases/pkg-config-0.29.1.tar.gz**

**zlib**
**wget http://www.zlib.net/zlib-1.2.11.tar.gz**

**SNORT Source Tar ball**
Depending if you are using cmake or autotools you'll need to also download the corresponding tar ball from the snort website if the wget command does not work.

**wget https://www.snort.org/downloads/snortplus/snort-3.0.0-a4-228-cmake.tar.gz**

**Opening Tarballs**
An example of installing a tarball:

tar -xzf archive-name.tar.gz
cd archive-name
./configure
make
sudo make install

**Tarball Exceptions:**
Some of the tar files do not all have the same capabilities. For example after opening the LuaJIT tar file/directory it does not have a ./configure program. Therefore move to the

next command "make" which does what it is supposed to. Then the next command "make install" will also, and should install the software.

### A.1.3 After Installing Dependencies

After installing all dependencies:

1. open whichever snort tarball you chose to download and navigate to its directory.
   **tar -xzf snort-tarball***
   **cd snort-tarball***
2. setup install path
   **export my_path=<the path to the snort directory>**
   if you are in the snort directory run the "pwd" command to get the full path to the directory.
3. Compile and Install:
   **./configure_cmake.sh - -prefix=$my_path**
   **cd build**
   **sudo make -j 8 install**

A note from SNORT's Github "If you are using autotools with a github clone, first do autoreconf -isvf. If you can do src/snort -V you built successfully. If you are familiar with cmake, you can run cmake/ccmake instead of configure_cmake.sh. cmake –help will list any available generators, such as Xcode. Feel free to use one, however help with those will be provided separately."

### A.1.4 Running SNORT

If all went well in the previous section we should be able to run SNORT with the following commands.

```
export LUA_PATH=$my\_path/include/snort/lua/\?.lua\;\;
export SNORT_LUA_PATH=$my\_path/etc/snort
```

The rest of the guide to test if everything is function is found here:

"https://github.com/snortadmin/snort3"

### A.1.5 Potential Errors

Installing everything right the first time is hard. Below is a list of errors I encountered and how I solved them.

1. `CMake Error at /usr/share/cmake-2.8/Modules/FindPackageHandleStandardArgs.cmake:108 (message): Could NOT find OpenSSL, try to set the path to OpenSSL root folder in the system variable OPENSSL_ROOT_DIR (missing: OPENSSL_LIBRARIES OPENSSL_INCLUDE_DIR) Call Stack (most recent call first): /usr/share/cmake-2.8/Modules/FindPackageHandleStandardArgs.cmake:315 (_FPHSA_FAILURE_MESSAGE) /usr/share/cmake-2.8/Modules/FindOpenSSL.cmake:313 (find_package_handle_standard_args) cmake/include_libraries.cmake:8 (find_package) CMakeLists.txt:18 (include)`
   **FIX:**
   **apt-get install libssl-dev**

2. For some reason while performing step 3 in "After installing Dependencies" we are not able to run the command make -j 8 install. Unfortunately, the only fix/alternative for this is to download SNORT 3.0 from github and attempt the method associated with that.
   **FIX:**
   The issue was that when we ran **./configure_cmake.sh –prefix=$my_path** it stored everything in the snort directory instead of snort's subdirectory, namely the "build" directory.

3. `checking for GLIB... no configure: error: Either a previously installed pkg-config or "glib-2.0 >= 2.16" could not be found. Please set GLIB_CFLAGS and GLIB_LIBS to the correct values or pass --with-internal-glib to configure to use the bundled copy.`
   **FIX:**
   **sudo apt-get install libglib2.0-dev**

### A.1.6 SNORT 3.0 Download from Github (Alternative)

Before beginning run the command to install git (sudo apt-get install git).

1. **git clone git://github.com/snortadmin/snort3.git**
2. **cd snort3**
3. **export my_path=/path/to/snorty** #actual path to where you downloaded Snort 3.0
4. **./configure_cmake.sh –prefix=$my_path**
5. **cd build**
6. **sudo make -j 8 install**

## A.2 Custom Commands in Ubuntu 14.04

### A.2.1 What is bashrc?

Essentially the bashrc script runs in every terminal and details the definition of commands that are executed in the terminal(s). We are able to write our own commands to consolidate the amount of time and effort it takes to perform tasks that are not generic to Ubuntu.

### A.2.2 Accessing bashrc

In order to access the bashrc script you will need to run:

**sudo gedit /.bashrc**

This opens the bashrc script in a text editor, allowing you to make changes to it. However, it is strongly recommended to make a original copy of it in case anything should "break".

### A.2.3 Making Aliases (Custom Commands)

In order to create a custom command, also known as an alias, navigate to the bottom of the file and begin a new line with:

**alias <name of your command> = '<the commands to run>'**

Example:

**alias snort3_dir= 'cd /home/linuxrouter1/snort3_files/snort-3.0.0-a4/build'**

**alias snort3='snort3_dir;sudo src/snort'**

**alias snort3_test=‘snort3_dir;src/snort -V’**
**alias snort3_options=‘snort3_dir;cd src;./snort -?’**

### A.2.4 Reloading bashrc

In order for your alteration to take affect you would normally have to log off and log back on to your account. However, the command **source /.bashrc**, will update the bash script. In order to be more convenient you could also write an alias to save you the trouble. Such as:
**alias srcForge=’source /.bashrc’**

### A.2.5 Common Issues

While experimenting with aliases you may encounter errors such as:

1. `bash: alias: srcForge: not found`
   `bash: alias: =: not found`
   `bash: alias: source ~/.bashrc: not found`
   `bash: alias: /home/linuxrouter1/snort3_files/snort-3.0.0-a4: not found`

   **Fix:** Make sure there is not an immediate space on either side of the equal sign, and if you are running multiple commands, make sure there is not a space after the semi-colon and before the next command.

## A.3 “mp_tracker.py” Modification

We had to alter a few lines of code because the unaltered code began content reassembly later than we needed it to, it did not know what to do with the second message in the “MP_JOIN” handshake, and it did not understand how to associate the additional subflow of the connection with the master subflow. Because of this the following lines were altered:

1. Line 87
   From: self.SSN = int().from_bytes(opt_data[data_ptr:data_ptr + 4],
   BYTE_TO_INT_ENDIANESS)
   To: self.SSN = int().from_bytes(opt_data[data_ptr:data_ptr + 4],
   BYTE_TO_INT_ENDIANESS) **- 1**

2. Line 421
   From: elif isinstance(subtype, MPTCPContainers[MP_JOIN]):
   To: elif isinstance(subtype, MPTCPContainers[MP_JOIN]) **and pkt.tcp_flags == TCP_SYN**:
3. Line 427
   From: mp.add_subflow(self.unclaimed_subflows[id])
   To: mp.add_subflow(**subflow**)

# List of References


[1] (2013). MPTCP overview. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/ip/transmission-control-protocol-tcp/116519-technote-mptcp-00.html. Accessed April 09, 2017.

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP extensions for multipath operation with multiple addresses," Internet Requests for Comments, RFC 6824, January 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[3] C. Paasch, G. Detal, S. Barré, F. Duchêne, and O. Bonaventure. (2015). The fastest TCP connection with multipath TCP. [Online]. Available: http://multipath-tcp.org/pmwiki.php?n=Main.50Gbps. Accessed April 7, 2017.

[4] C. Paasch, G. Detal, S. Barré, F. Duchêne, O. Bonaventure, and C. Raiciu. (2012). Exploring mobile/wifi handover with multipath TCP. [Online]. Available: https://inl.info.ucl.ac.be/system/files/cell06-paasch.pdf. Accessed April 7, 2017.

[5] Snort. (2017, January). Getting started. [Online]. Available: https://www.snort.org/. Accessed May 4, 2017.

[6] H. Foster, "Why does MPTCP have to make things so complicated?: Cross-path NIDS evasion and countermeasures," M.S. thesis, Naval Postgraduate School, 2016. [Online]. Available: http://calhoun.nps.edu/handle/10945/50546

[7] C. Paasch and S. Barré. (2007). MPTCP - linux kernel implementation: Homepage. [Online]. Available: http://www.multipath-tcp.org/. Accessed June 9, 2016.

[8] F. Yang, P. Amer, and N. Ekiz. (2013). A scheduler for multipath TCP. [Online]. Available: https://www.sans.org/reading-room/whitepapers/detection/distributed-intrusion-detection-systems-introduction-review-897. Accessed July 02, 2016.

[9] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handly. (2010). Data center networking with multipath TCP. [Online]. Available: http://conferences.sigcomm.org/hotnets/2010/papers/a10-raiciu.pdf. Accessed April 09, 2017.

[10] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "SOCKS protocol version 5," Internet Requests for Comments, RFC 1928, March 1996. [Online]. Available: https://tools.ietf.org/html/rfc1928

[11] M. Coudron and S. Secci. (2017). An implementation of multipath TCP in ns3. [Online]. Available: http://hal.upmc.fr/hal-01382907v2/document. Accessed April 09, 2017.

[12] M. Bagnulo, “Threat analysis for TCP extensions for multipath operation with multiple addresses,” Internet Requests for Comments, RFC 6181, March 2011. [Online]. Available: https://datatracker.ietf.org/doc/rfc6181/

[13] R. Robbins. (2002). Distributed intrusion detection systems: An introduction and review. [Online]. Available: https://www.sans.org/reading-room/whitepapers/detection/distributed-intrusion-detection-systems-introduction-review-897. Accessed June 16, 2016.

[14] G. Rieger. (2010). Socat(1) - linux man page. [Online]. Available: https://linux.die.net/man/1/socat. Accessed Feb 15, 2017.

[15] A. Sharma, S. Saroj, and P. Kumar. (2013). Distributed intrusion detection system for wireless sensor networks. [Online]. Available: http://www.iosrjournals.org/iosr-jce/papers/Vol14-issue1/J01416170.pdf?id=6825. Accessed May 4, 2017.

[16] M. Wood and M. Erlinger, “Intrusion detection message exchange requirements,” Internet Requests for Comments, RFC 4766, March 2007. [Online]. Available: https://www.ietf.org/rfc/rfc4766.txt

[17] Snort. (2017, January). Snort++. [Online]. Available: https://github.com/snortadmin/snort3. Accessed June 1, 2017.

# Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California