2016-09

"Why does MPTCP have to make things so complicated?": cross-path NIDS evasion and countermeas

Foster, Henry August

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

"WHY DOES MPTCP HAVE TO MAKE THINGS SO COMPLICATED?": CROSS-PATH NIDS EVASION AND COUNTERMEASURES

by

Henry August Foster

September 2016

Thesis Advisor:	Geoffrey Xie
Second Reader:	Robert Beverly

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2016	Master's Thesis 06-29-2014 to 09-23-2016

4. TITLE AND SUBTITLE

"WHY DOES MPTCP HAVE TO MAKE THINGS SO COMPLICATED?": CROSS-PATH NIDS EVASION AND COUNTERMEASURES

5. FUNDING NUMBERS

6. AUTHOR(S)

Henry August Foster

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)*

A recent enhancement to Transmission Control Protocol (TCP) is Multipath TCP (MPTCP), a new transport layer protocol that enhances TCP to be capable of communicating over multiple paths by establishing several "subflow" connections between endpoints. Each subflow behaves in the same way that a traditional, single-path, TCP connection would. Previous work has demonstrated that adversaries can perform cross-path data fragmentation to evade Network Intrusion Detection Systems (NIDS) when the NIDS is unable to integrate related subflows into a single MPTCP data stream. We present a general solution to enable current penetration testing tools to perform MPTCP cross-path fragmentation attacks. On the defensive side, we demonstrate that existing transport layer proxies can be used in conjunction with an MPTCP kernel to transparently convert a multipath connection into a single-path connection that can be analyzed by a NIDS. We also investigate extending Snort to perform MPTCP stream reassembly and create a prototype Snort plugin for accomplishing this functionality.

14. SUBJECT TERMS

MPTCP, multipath TCP, Intrusion Detection, Cross-path fragmentation, networking, IDS, session-splicing, Snort, proxy

15. NUMBER OF PAGES 103

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

"WHY DOES MPTCP HAVE TO MAKE THINGS SO COMPLICATED?": CROSS-PATH NIDS EVASION AND COUNTERMEASURES

Henry August Foster,
Civilian, Scholarship for Service Program
B.A., University of California, Berkeley, 2012

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2016

Approved by: Geoffrey Xie
Thesis Advisor

Robert Beverly
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A recent enhancement to Transmission Control Protocol (TCP) is Multipath TCP (MPTCP), a new transport layer protocol that enhances TCP to be capable of communicating over multiple paths by establishing several "subflow" connections between endpoints. Each subflow behaves in the same way that a traditional, single-path, TCP connection would. Previous work has demonstrated that adversaries can perform cross-path data fragmentation to evade Network Intrusion Detection Systems (NIDS) when the NIDS is unable to integrate related subflows into a single MPTCP data stream. We present a general solution to enable current penetration testing tools to perform MPTCP cross-path fragmentation attacks. On the defensive side, we demonstrate that existing transport layer proxies can be used in conjunction with an MPTCP kernel to transparently convert a multipath connection into a single-path connection that can be analyzed by a NIDS. We also investigate extending Snort to perform MPTCP stream reassembly and create a prototype Snort plugin for accomplishing this functionality.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API	Application Programming Interface
C2	Command and Control
DSS	Data Sequence Signal
HIDS	Host-based Intrusion Detection System
HMAC	Hash Message Authentication Code
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPC	Inter-Process Communication
IPv4	Internet Protocol version 4
MitM	Man-in-the-Middle
MPTCP	Multipath TCP
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIDS	Network Intrusion Detection System
NIPS	Network Intrusion Prevention System
NSM	Network Security Monitoring
PDU	Protocol Data Unit
RTT	Round Trip Time

TCP Transmission Control Protocol

UDP User Datagram Protocol

VPN Virtual Private Network

Acknowledgments

I would like to thank:

Geoff, my advisor, for his unrelenting support and guidance while I completed this thesis.

Rob, my second reader, for providing excellent feedback and suggestions on how to improve this work.

My friends and family for their support and understanding while I disappeared for two years to stare at a computer screen.

Dan and Warren, fellow students who are pursuing related research (mostly for putting up with me when I ramble on about tangentially relevant things during meetings).

The other denizens of the Networking Lab for creating a fun environment to work in.

My peers in the the Scholarship for Service program at NPS. Over the last two years, you have become my close friends. Many of you deserve to be singled out here by name, but I think those of you I would list already know who you are. To avoid accidentally leaving any of you out, I will leave it at that.

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

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

With the adoption of Internet-enabled technologies, the demand for increased bandwidth and reliability of communications over the Internet grows. One recent protocol that promises to provide increased bandwidth and reliability over existing networking infrastructure is Multipath TCP (MPTCP) [1]. MPTCP is an enhancement to Transmission Control Protocol (TCP) [2]. MPTCP enables applications that could previously only establish communication channels over single network paths to communicate over multiple network paths. MPTCP is an enhancement to TCP that provides numerous benefits by allowing endpoints to multiplex a single stream of data over multiple network paths. Each such path is described as a "subflow." The benefits associated with MPTCP include increased bandwidth via load-balancing as well as robustness and reliability through the use of redundant connections between endpoints. This functionality is implemented in the transport layer of the TCP/Internet Protocol (IP) stack, which allows existing software to take advantage of these enhancements without being altered or recompiled.

A Network Intrusion Detection System (NIDS) is a device that analyzes network traffic looking for patterns that match known malicious traffic. However, the current generation of NIDS devices expect applications only to communicate over one path, and are therefore ill-equipped to recognize malicious traffic when attackers use MPTCP to fragment transmissions over multiple network paths [3]–[5]. Many NIDSs function by analyzing network traffic with a set of signatures designed to match known malicious traffic. As attackers employ various techniques that allow them to evade detection by NIDSs, NIDS technologies have improved to counter these evasions. One of these evasion techniques involved fragmenting a malicious payload into several small TCP segments so that no single segment will match a rule meant to detect the malicious payload in question. Current NIDS systems attempt to reassemble TCP data streams to prevent this evasion from being effective. However, to a NIDS that is not MPTCP-aware, an MPTCP subflow appears to be a traditional (single-path) TCP connection. Therefore, if an attacker can split malicious traffic across multiple subflows such that each subflow's stream of data does not match a signature, a NIDS can be evaded even if the MPTCP data stream would match an existing

signature after it is reassembled at its endpoint.

1.1 Research Questions

In 2014, at the Black Hat Las Vegas conference, researchers presented a brief that highlighted the lack of network analysis tools that supported MPTCP [3]. This concern was validated by Afzal and Lindskog who found that many Snort [6] rules can be evaded with cross-path fragmentation over multiple MPTCP subflows [4]. Their findings were based on parsing Snort rules and crafting network traffic designed to trigger these rules; they did not perform the actual attacks that the rules had been created to detect. Another commonly used NIDS, Bro [7], lacked support for MPTCP. Baugnies extended Bro to parse MPTCP options and made progress in enhancing Bro to be capable of reassembling MPTCP data streams [5].

This thesis seeks to address these related research questions:

1. How could attackers go about performing MPTCP cross-path fragmentation with actual attacks? That is, what methodology would be required to reproduce the results of Afzal and Lindskog with real attacks instead of simulating them with traffic generated from existing rules?
2. How can adversaries be prevented from using MPTCP cross-path fragmentation attacks to evade NIDS?
3. How can Snort be enhanced to perform MPTCP stream reassembly?

1.2 Summary of Contributions

By addressing the research questions in Section 1.1, this thesis makes the following contributions:

- A methodology is presented for integrating existing penetration testing systems (such as the Kali Linux distribution) into MPTCP scenarios without requiring any modification to the systems.
- Two methods are explored to strengthen Snort to mitigate NIDS evasion risks associated with MPTCP deployment: The first leverages existing MPTCP code bases, and the second investigates the feasibility of adding MPTCP awareness to Snort 3 [8] through a dynamic plugin.

1.3 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 describes the operation of MPTCP, and discusses prior research on the security concerns associated with MPTCP and the attempts that have been made at resolving them. Chapter 3 focuses on how adversaries can perform MPTCP cross-path data fragmentation based evasions, and presents a tool that enables existing penetration testing systems to evade MPTCP-unaware NIDS. Chapter 4 investigates methods that defenders may employ to counteract cross-path fragmentation evasions; specifically, transport layer proxies are used to convert multipath traffic into single-path traffic, and a dynamic plugin is created to allow Snort access to reassembled MPTCP data streams. Chapter 5 summarizes this thesis and highlights some areas for future research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter seeks to familiarize the reader with MPTCP and some of the security concerns associated with the protocol. Additional discussion concerns how attackers have historically used fragmentation to bypass Network Security Monitoring (NSM) tools, how defenders responded, and finally how defenders are currently ill-equipped to contend with fragmentation attacks launched over MPTCP. Those who are interested in a hands-on introduction to MPTCP should peruse the tutorial provided in Appendix B.1.

2.1 Introduction to MPTCP

The purpose of this section is to describe MPTCP, the benefits provided by the protocol, and to define terms that are frequently used in this thesis. Discussion of the security risks that accompany MPTCP adoption can be found in Section 2.2.

2.1.1 What Is MPTCP?

MPTCP is an extension to the TCP protocol that allows multiplexing of data across multiple network paths. An MPTCP connection communicating on multiple paths does so over multiple "subflows." A subflow operates in the same way that a traditional, single-path, TCP connection does. However, it is distinguishable from traditional TCP in that segments carry an MPTCP option field in the TCP header. The MPTCP option field contains the control information necessary for both endpoints to coordinate the larger MPTCP connection. Figure 2.1 shows traditional TCP operation compared to MPTCP operation with multiple subflows. Note that one of the arrows representing an MPTCP subflow is drawn in a different direction; after the initial subflow of an MPTCP connection is established, either endpoint may initiate new subflows. Each of these subflows is capable of carrying either a portion of or the entire MPTCP data-stream (if no additional subflows are added).

2.1.2 Definition of Terms

For clarity, we provide the following definitions of frequently used terms:

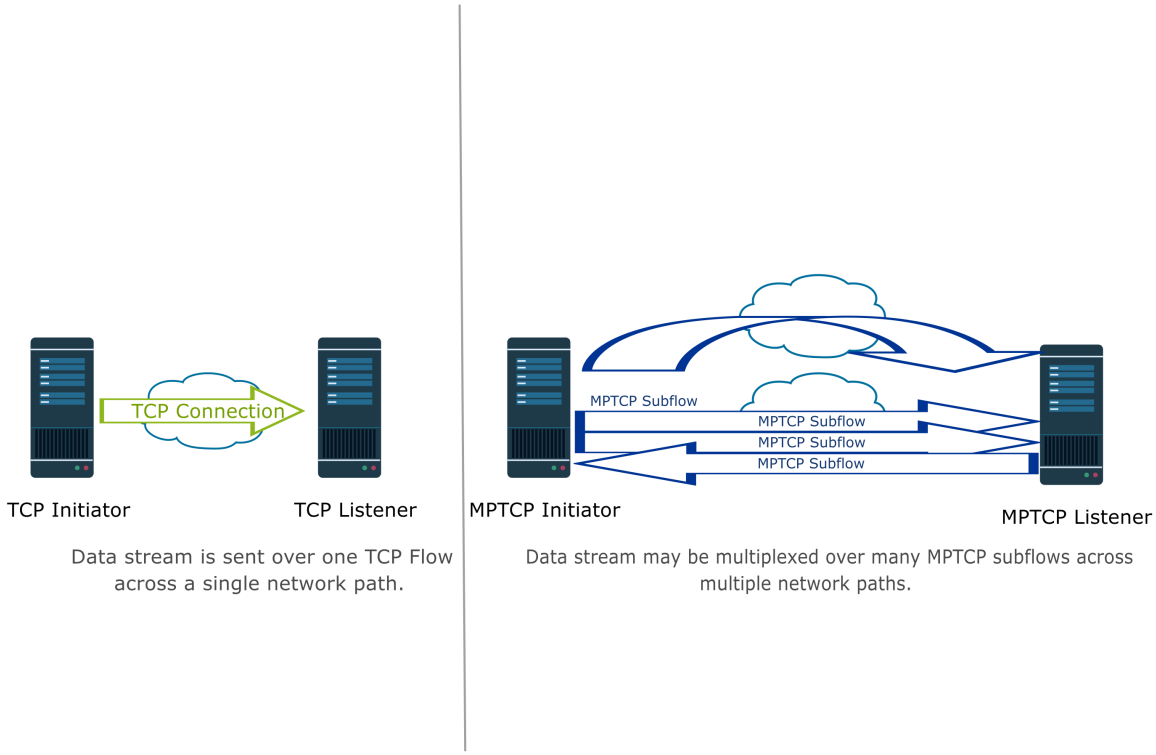

Figure 2.1. Traditional TCP vs. MPTCP Operation.

Subflow

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" [1].

It is important to note that "each MPTCP subflow looks like a regular TCP flow whose segments carry a new TCP option type" [1]. This means that to network analysis tools that do not specifically look for the MPTCP option, subflows are indistinguishable from traditional TCP connections.

NIDS

A NIDS is a device "which monitors network traffic for particular network segments or devices and analyzes the network and application protocol activity to identify suspicious activity" [9].

When discussing NIDS in this paper, we are referring to signature-based NIDS. Specifically, "a signature is a pattern that corresponds to a known threat. Signature-based detection is the process of comparing signatures against observed events to identify possible incidents" [9].

MPTCP vs. traditional TCP vs. TCP

When we refer to an MPTCP connection, we mean a connection that has at least one subflow that has completed an MP_CAPABLE handshake [1]. Since it is possible to have an MPTCP connection with only one subflow, we attempt to make it clear when we are discussing MPTCP connections that are comprised of multiple subflows.

When we refer to "traditional" TCP we are referring to single-path TCP connections, for which the TCP headers do not carry MPTCP options. An example of an exception to this is when an MPTCP option is sent in the initial SYN of a TCP 3-Way handshake but is not present in the returning SYN/ACK. For an MPTCP connection to be created, a specific subtype of the MPTCP option field must be present in all three packets of the TCP 3-Way handshake for the initial subflow. If this condition is not met, the connection will fall-back and be a traditional TCP connection.

"TCP" may refer to either MPTCP or traditional TCP.

Middlebox

We use the RFC 3234 definition of "middlebox," where a middlebox is "any intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host" [10].

Cross-path fragmentation

"Cross-path fragmentation" refers to a stream of data being fragmented across multiple paths. In the case of MPTCP, this means that portions of an MPTCP data stream will be fragmented over multiple subflows.

TCP Session-splicing

"TCP session-splicing" is largely synonymous with TCP fragmentation, however with the emphasis that the fragmentation was performed maliciously to make the transmitted payload

more evasive [11].

In this thesis, "MPTCP session-splicing" can be considered to be synonymous with "MPTCP cross-path fragmentation."

2.1.3 Summary of MPTCP Implementations

The following lists at least some of the MPTCP implementations that exist or that are being worked on.

1. Linux kernel implementation [12].[1]
2. Apple iOS implementation [14].
3. FreeBSD implementation [15].
4. A Solaris implementation is reportedly in the works [16].
5. Android implementation [17].[2]

2.1.4 Benefits of MPTCP

The benefits provided by MPTCP include robustness, load balancing, and full network utilization [19].

- Robustness: MPTCP is capable of establishing and maintaining many network paths between endpoints at once. If one of these paths fails or drops in quality, MPTCP will respond by preferring other available paths. In the case of MPTCP, one failed subflow will not interrupt the transfer of the entire data stream.
- Load Balancing: If MPTCP detects that its preferred path is congested, it will begin sending additional traffic over other established network paths. This increases bandwidth over TCP through the intelligent rebalancing of load.
- Utilization of unused network capacity: An organization may have underutilized areas of their network, especially if they added additional network infrastructure for redundancy in the event that a main data pipeline fails. MPTCP is capable of

[1] The Linux kernel implementation is considered, at least by some, to be the reference MPTCP implementation [13].

[2] The most recent Android device that a published MPTCP kernel has been created for is the Nexus 5. At the time of writing, the Nexus 5 device is multiple generations old, and no longer receives Android version updates [18].

directing traffic over these little used paths and increasing the return on investment for this additional infrastructure. [19]

2.2 Related Work

2.2.1 Security Concerns over MPTCP

When the MPTCP protocol was being designed, a threat analysis was conducted to identify potential security issues with the protocol; RFC 6181 presents the results of this analysis [20]. The findings of RFC 6181 went on to inform the decisions made while solidifying the protocols design, which is presented in RFC 6824 [1]. An additional, residual threat analysis was conducted for the RFC 6824 definition of MPTCP; the results of this analysis are contained in RFC 7430 [21]. While the threats discussed in RFC 7430 are certainly important to MPTCP security, they are largely outside the scope of this thesis, which focuses on how adversaries can use MPTCP to evade NSM tools. The threats presented in RFC 7430 can be considered attacks on the MPTCP protocol, we distinguish between these and application level attacks that take advantage of MPTCP to become more difficult to detect.

In the Black Hat Las Vegas 2014 briefing titled "Multipath TCP: Breaking Today's Networks with Tomorrow's Protocol," Pearce and Thomas discussed security difficulties that accompany MPTCP adoption [3]. The chief difficulty that they emphasize is that many network security tools that defenders rely upon do not support MPTCP analysis. Generally, MPTCP-unaware tools will consider MPTCP subflows to be traditional single-path TCP connections. A major consequence of this behavior is that if an MPTCP-unaware tool attempts to reassemble a TCP connection's data stream to perform deep packet inspection, it will miss important parts of the traffic if those segments were transmitted on a different subflow. A fragmentation tool was released to accompany the briefing that fragmented a Hypertext Transfer Protocol (HTTP) GET request over multiple subflows [22]. This "cross-path fragmentation" technique can be used to evade MPTCP-unaware NIDS when an adversary attacks an MPTCP-capable target.

The importance of the cross-path fragmentation problem was validated by Afzal and Lindskog who demonstrated that many widely deployed Snort rules written to inspect TCP traffic could be evaded by fragmenting traffic across multiple MPTCP subflows [4]. To do this,

they parsed rules from the Snort rule set that inspected TCP payload and crafted payload designed to trigger these rules and generate alerts. The number of alerts triggered when the generated payload was sent over multiple MPTCP paths was significantly lower than the number of alerts generated when the payload was sent over traditional single-path TCP.

Neither of the aforementioned demonstrations of cross-path fragmentation conducted any live attacks. In the first example, mptcp-scapy [23] was used to send a simple HTTP get request that was fragmented over multiple subflows. In the second, actual attacks were not performed, rather traffic specifically designed to generate alerts was sent to test Snort's resilience to evasions based on MPTCP cross-path fragmentation. In Chapter 3 we present a technique for conducting live attacks over MPTCP connections configured to be evasive to NIDS.

2.2.2 Risk Management Strategies for MPTCP

An obvious way to avoid the risks associated with MPTCP is to prevent its operation on one's network. This can be accomplished by stripping the MPTCP option from packets during a TCP handshake. The first subflow of an MPTCP connection carries an MP_CAPABLE option subtype in the handshake and future subflows carry the MP_JOIN option subtype. MPTCP is designed to fallback to traditional TCP if a middlebox interferes with the exchange of MPTCP control information during a subflow's initiation [1]. It has been shown that MP_CAPABLE and other option stripping occurs on Internet paths. [24]. Also, Apple warns iOS users that "many commercial routers replace unknown TCP options with NOOP data" [14]. For instance, Cisco ASA firewalls are capable of overwriting MPTCP Options with NOOPs and do so by default [25]. A SANS whitepaper lists example iptables rules for blocking packets containing MPTCP options [26]. Approaches for preventing MPTCP and forcing a fallback to traditional TCP or limiting MPTCP connections to a single subflow are further discussed in Section 4.1.1.

Another potential way to prevent MPTCP is by configuration management of endpoints. The Linux kernel implementation has a kernel parameter `net.mptcp.mptcp_enabled` that specifies whether or not MPTCP will be used. While there is no guarantee that all implementations will offer the same capability to turn off MPTCP easily, if it is present, then that endpoint may be prevented from participating in MPTCP connections.

While preventing MPTCP will protect one from the threats mentioned in Section 2.2.1, it is undesirable because one also forfeits the many benefits that accompany MPTCP adoption. Therefore, there is a need to upgrade NIDS to be able to analyze MPTCP traffic.[3]

Baugnies enhanced Bro [7] to support the generation of MPTCP events [5]. This allows Bro to be used to detect attacks on the MPTCP protocol (the kind of attacks discussed in RFC 7430). For instance, a non-MPTCP version of Bro would be unable to distinguish between a TCP SYN flood and an MP_JOIN flood attack. Baugnies also made progress extending Bro to reassemble MPTCP streams, which would allow Bro to resist the MPTCP cross-path fragmentation based evasions. However, changes that were made to support reassembly had a negative effect on Bro's ability to detect attacks on the MPTCP protocol.[4] Our decision to investigate MPTCP stream reassembly with Snort may have been a stroke of luck and allowed us to avoid many of the complications involved with making Bro MPTCP-aware. Snort 2 and, even more so, Snort 3 are designed to be pluggable [8], this allows a great deal of MPTCP functionality can be added without modifying the core internals of the NIDS. While we do not attempt to enhance Snort with the ability to detect RFC 7430 style attacks in this thesis, we do discuss how this can be accomplished by creating dynamic rules in Section 4.3. In Section 4.2 we use a Snort 3 dynamic inspector plugin to pass reassembled MPTCP streams directly to Snort's detection engine. The actual stream reassembly is performed outside of Snort, however, so our implementation was not as ambitious as the one attempted by Baugnies.

In addition to validating MPTCP cross-path fragmentation evasions for the Snort rule set, Afzal and Lindskog also present an approach for MPTCP reassembly [4]. An MPTCP-Linker tool is used to reassemble MPTCP streams into TCP streams (the payload remains the same) which are written to pcap files. These pcap files may then be analyzed by Snort, which is then able to inspect the contents of the data streams. Our experimentation with MPTCP reassembly is similar but differs in that it is much more closely integrated into Snort. Our implementation in Section 4.2 uses Snort 3 as both the capture tool, and we use a dynamic inspector plugin to submit reassembled directly to Snort's detection engine (avoiding the step where the reassembled streams are converted to TCP and written to pcap

[3] We propose criteria for what should constitute MPTCP-awareness for NIDS in Section 4.3.

[4] This was due to modifying Bro to treat multiple subflows as a single connection, which had the unintended effect of breaking Bro's ability to raise events for MPTCP-events that occurred on some subflows.

files). Only the reassembly is performed in a separate process than Snort 3.

It has been suggested that MPTCP-TCP proxies should be used to encourage MPTCP adoption [27]. Afzal, Lindskog, and Lidén discuss that an additional benefit of these proxies would be that they convert MPTCP traffic to standard TCP traffic; in the process, they defragment multipath data streams converting them to single path streams that may be inspected by NIDS [28]. We also experiment with using transport layer proxies to defragment cross-path fragmented MPTCP data streams in Section 4.1. Unlike, the proxy developed in [4], which was written in Python and re-implements MPTCP logic in user space, ours uses existing tools to create TCP port-forwardings on top of Linux MPTCP kernels. This has the benefit of leveraging an existing mature MPTCP code base to perform the defragmentation. The downside is that when our port-forwarder proxy initiates a connection to an endpoint on the internal (monitored) network, it will attempt to make this an MPTCP connection. This will not affect the ability of a NIDS to analyze this traffic so long as this connection is limited to a single subflow. Furthermore, steps to prevent MPTCP operation on the internal network could be taken to force a fallback to traditional TCP, which would assure that an MPTCP-unaware NIDS could investigate the data stream. We believe that the related approach of placing application layer proxies, such as Squid [29], on top of MPTCP kernels may offer notable benefits. Many organizations have already deployed reverse HTTP proxies; it is possible that they could deploy MPTCP for their web applications simply by installing an MPTCP enabled kernel on these devices. Further investigation of this topic is suggested as future work in Section 4.3. At least one commercial application proxy with MPTCP support already exists [30].

An issue that is related to performing MPTCP stream reassembly is subflow association. In order to correctly reassemble an MPTCP data stream, it is necessary first to identify which subflows belong to an MPTCP connection. The obvious way to do this is to compare the value of a token exchanged between MPTCP endpoints when new subflows are being added to an MPTCP connection with keys that were exchanged during the establishment of the initial subflow (the tokens are derived from the keys). However, this will not work if the information exchanged during subflow establishment was not captured. Zhang et al., present an algorithm for subflow association that first attempts to use the token method, and if this fails, will instead attempt to perform associations between subflows based on the MPTCP data sequencing information contained in the MPTCP DSS option subtype [31].

The reassembly work we present in Chapter 4 only uses the token method of subflow association; it is left for future work to enhance our MPTCP reassembler with this full algorithm.

Another issue that can complicate MPTCP stream reassembly occurs when the devices meant to perform the reassembly are deployed along multiple network paths such that no single device sees all the traffic that traverses these paths. If an MPTCP data stream is fragmented along these paths, then the reassembly devices must coordinate with one another to rebuild the entire stream. At the 2015 IEEE INFOCOM conference, Ma, Le, and Russo presented a sophisticated approach to this problem in their paper, "Detecting distributed signature-based intrusion: The case of multi-path routing attacks" [32]. The authors presented a distributed string matching algorithm (based on the Aho-Corasick matching algorithm) wherein multiple sensors can share state information, allowing them to match strings striped across different sensors. The authors conducted an experiment where an implementation of their algorithm was deployed to two computers acting as network monitors along two paths. By employing the author's algorithm, neither monitor required a copy of the entire stream of data to perform detection. Sharing the current state of the string matching engine was sufficient to recognize strings even when fragmented over multiple network paths. The monitors were able to detect "malicious" traffic[5] that was fragmented over two network paths using MPTCP. While we do not use this algorithm in our investigation, it is a promising solution for how a distributed series of NIDS can coordinate their MPTCP reassembly efforts while minimizing the amount of bandwidth consumed by this coordination. We leave this coordination problem for future work, but discuss some different approaches for NIDS coordination in Section 4.3.

2.3 MPTCP Operation

This section describes the normal operation of MPTCP. For more detailed information about how MPTCP works, one should consult the sources which were used in the writing of this section. IETF RFC 6824 [1] provides a detailed discussion of the operation of the protocol. For a more approachable description with a discussion of design considerations and difficulties experienced by the creators of the Linux kernel implementation [12], consult

[5]The authors simulated malicious traffic by designating randomly generated strings as dirty words they attempt to detect.

the paper "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP" [19].

2.3.1 Anatomy of an MPTCP Connection

The easiest way to describe MPTCP is to say that it allows multiplexing of data over multiple TCP connections, or "subflows." Additionally, it is implemented in such a way that it is indistinguishable from TCP by higher layer application protocols.[6]. For descriptive purposes, it might be useful to consider an MPTCP connection as having four phases:

1. Initiation of the MPTCP connection.
2. Creation of additional subflows.
3. Transmission of a stream of data (over the existing subflows).
4. Termination of the MPTCP connection.

Note that the middle two phases are more conceptual and do not necessarily proceed in any order. That is, data transmission may begin before the addition of new subflows, or additional subflows may be added after the data transmission begins. A description of a typical MPTCP connection between two MPTCP capable endpoints follows.

The client initiates a TCP connection with a service. In the ensuing 3-Way handshake, the endpoints exchange an MP_CAPABLE message in the TCP Options field of the TCP header (if one of the endpoints failed to send the MP_CAPABLE message, the connection would fall back to traditional TCP operation).

After the connection is initiated, MPTCP will attempt to create new subflows for that connection. The MPTCP component that determines which subflows to create is the "Path Manager." Different path management schemes will create different subflows. For instance, the "fullmesh" path manager included with the Linux kernel implementation of MPTCP will attempt to create subflows between every possible pair of IP addresses belonging to each endpoint [12]. However, use of a different path manager will cause MPTCP to

[6] The benefit of implementing MPTCP in operating system kernels, or in a way that replaces traditional TCP implementations, is that all software that uses TCP will seamlessly begin taking advantage of MPTCP without needing to be modified or recompiled. That said, if an Application Programming Interface (API) were provided to allow applications to control MPTCP operation, it would likely be beneficial [33], [34]. MPTCP could also be implemented in user-space as well. In fact, there is discussion over whether it would be beneficial practice to begin implementing networking stacks in user space [35]

behave differently. The “ndiffports” path manager of the Linux implementation will create a configurable number of subflows over the two interfaces used in the creation of the initial subflow.

Therefore, one of the endpoints, if it possesses multiple IP addresses, might initiate more subflows to the other endpoint from these interfaces. During the 3-Way handshake of these new connections, MP_JOIN messages are exchanged, causing these subflows to join the larger MPTCP connection. Sometimes firewalls and other middleboxes may prevent some of the join messages from reaching their destination; this prevents the new subflow from being added to the MPTCP connection. To avoid this problem, the endpoints can exchange ADD_ADDR messages over the initial subflow. This message contains additional addresses that belong to the sending endpoint so that the receiver of the ADD_ADDR message can attempt to create a subflow to this address. Frequently, a middlebox may interfere with MP_JOIN messages in one direction, but if the subflow is initiated by the other endpoint, the addition of the new subflow will be successful. Also, it should be noted that there is no restriction that new subflows must involve new network layer addresses, as is the case with the ndiffports path manager. Therefore, MPTCP may attempt to create new subflows between the same IP addresses but on different TCP ports. There is no set consensus on what path manager provides optimal operation of MPTCP. Therefore, it is up to the user to configure their MPTCP installation to use the path manager most beneficial to their environment.

At some point, the stream of data will begin flowing over the MPTCP connection. As data is exchanged between endpoints, Data Sequence Signal (DSS) messages in the MPTCP option map the contents of each subflow stream onto a portion of the larger MPTCP stream. DATA_ACKs, a type of Data Sequence Signal (DSS) message, are sent to confirm receipt of data into the MPTCP stream after it has been received by an endpoint. While the DSS messages coordinate data transfer at the MPTCP stream level, each subflow uses standard TCP sequence and acknowledgment numbers to coordinate delivery of data being sent on that subflow. While the MPTCP path manager governs the creation of subflows, two additional components govern the subset of active subflows that are used to transmit data on, the scheduler and the chosen Congestion Control Algorithm. Different MPTCP schedulers and congestion control mechanisms will result in different behavior. For instance, a scheduler could send data upon the lowest latency link until a certain congestion control

algorithm deems that this network path is saturated, then the scheduler would begin to send additional data over the next best subflow. This means that many of the subflows created by the path manager may not be used to send any of the data stream for the entirety of the MPTCP connection.

When all data has been sent, the MPTCP connection will need to be closed. This occurs by endpoints exchanging DATA_FINs, another type of DSS message, and DATA_ACK'ing these DATA_FINs. Finally, each subflow that is part of the closing MPTCP connection will close in the normal way of TCP; TCP FIN flags are exchanged and acknowledged with TCP ACK flags.

2.3.2 MPTCP Option Subtypes

Proper MPTCP functioning relies on control information exchanged in the MPTCP option, which is transmitted in the Options field of the TCP header. Within the MPTCP option, the following MPTCP option subtypes are used in the basic operation (handshake, data transmission, and teardown) of MPTCP. This subsection is included to provide more depth into the operation of the protocol than was included in the previous story about an MPTCP connection. Note that detail is spared here. For a complete description of the protocol refer to RFC 6824 [1].

- MP_CAPABLE: This option subtype is exchanged in the TCP three-way handshake of the initial subflow. If it is not received by each endpoint for each part of the handshake, the connection falls back to become a traditional TCP connection. Each endpoint randomly generates a 64-bit number that becomes a key for authenticating additional subflows. These keys are exchanged in the MP_CAPABLE handshake.
- MP_JOIN: This option subtype is included in the TCP three-way handshake of additional subflows. These subflows are authenticated before becoming part of the larger MPTCP connection by both sides sending a Hash Message Authentication Codes (HMACs) generated from the keys exchanged in the MP_CAPABLE handshake and nonces exchanged in this MP_JOIN handshake.
- DSS: This option subtype, "Data Sequence Signal," is a primary workhorse of MPTCP. It is responsible for transmitting information used to map the stream of data sent on a subflow into the larger MPTCP data stream, to acknowledge receipt of

data into the MPTCP data stream, and to signal that there is no more data to be sent across any subflow and begin closing the MPTCP connection.

Data sequence mapping is accomplished by sending, a relative subflow sequence number (this is a relative TCP sequence number), an absolute data sequence number (this is a separate sequence number for an entire MPTCP connection's unidirectional data stream), and a data-level length. The next data-level length number of bytes of the subflow's stream belong to the MPTCP data stream starting at the sent data sequence number.

MPTCP stream data acknowledgments are signaled with a DATA_ACK DSS flag. This should not be confused with TCP ACKs. TCP ACKs exist in the standard TCP header, while DATA_ACKs exist in the MPTCP options with the DSS subtype.

The DSS option subtype is also used to carry MPTCP checksums. This isn't just to assure that the same data arrives as was sent, but is also useful for detecting when a middlebox is tampering with a subflow's stream to the extent that the data sequence mapping is no longer valid. Because of this worry, a failed DSS checksum will result in the termination of the subflow via the MP_FAIL option subtype. Use of MPTCP checksums occurs only if it was enabled during the MP_CAPABLE handshake.

Finally, DATA_FIN is a DSS flag used to signal that no more data will be sent over any subflow. This initiates the graceful closure of all subflows and the larger MPTCP connection. For the MPTCP connection to be closed gracefully, both endpoints need to send and receive a DATA_FIN and receive a DATA_ACK in response to their sent DATA_FINs.

- ADD_ADDR: This option subtype allows one endpoint to advertise new addresses with which the other endpoint can initiate an MP_JOIN handshake. This is useful in the case that the MPTCP initiator, the "client," is behind a Network Address Translation (NAT) or a firewall and the MPTCP receiver, the "server," is multi-homed. In this case, the server is not able to initiate a new subflow with an MP_JOIN subtype as the client's firewall will block the first initiating SYN sent by the server.
- REMOVE_ADDR: This option subtype is sent by an endpoint to signal the other that a previously available addresses is no longer valid. MPTCP will seek to validate that this is true, and proceed to send TCP RST flags on existing subflows corresponding to this address so that middleboxes can observe that the connection has ceased.
- MP_PRIO: This is used for one endpoint to change the priority of an existing subflow.

This allows the sending endpoint to request that the receiver prefer sending data over certain subflows and avoid sending it over others. For instance, an implementation of MPTCP for mobile devices might seek to lower the priority of subflows that traverse a cellular network, where bandwidth is usually more costly.

- MP_FAIL: This option subtype is sent when a DSS checksum fails and the data sequence mapping (allows TCP sequence numbers to be used to assemble correctly-ordered data in the MPTCP stream) can no longer be used. In this case, the receiver of the failed checksum will send a segment with a TCP RST flag and MP_FAIL on the offending subflow to destroy it.
- MP_FASTCLOSE: This is the MPTCP level equivalent of a TCP segment with a RST flag. MP_FASTCLOSE signals that an entire MPTCP connection is abruptly terminating. This will prompt endpoints to send TCP RST segments along all subflows belonging to the MPTCP connection. To prevent abuse, MP_FASTCLOSE carries the receiver's key (originally exchanged during the MP_CAPABLE handshake).

2.4 History of Fragmentation Attacks

2.4.1 IP Fragmentation Attacks

One way NIDS can operate by comparing networking headers and payloads against signatures (signature-based detection) of known attacks. Attackers seek to evade IDS by obfuscating their network traffic in such a way that it will no longer match the signatures programmed into an IDS.

Fragmentation is one category of techniques that an attacker can leverage to evade IDS. This involves splitting the payload of an attack into several smaller Protocol Data Units (PDUs). A historical example of this is using the fragmentation features of Internet Protocol version 4 (IPv4) to hide attacks from defenders. Here, the fragmentation occurs at the Internet layer of the TCP/IP stack. An attacker can split a TCP or User Datagram Protocol (UDP) payload into several IP packets. The Intrusion Detection System (IDS) checks each of these packets individually as they enter the defender's network. However, no single packet contains enough information to match one of the rules configured into the IDS.

For countering IPv4 fragmentation, a relatively simple solution presented itself to defenders:

just block fragmented packets from entering the network. While this might result in dropping some legitimate traffic, its effect should be ultimately negligible as higher-layer protocols or applications reduce the size of payloads sent in each packet so that it falls within the Maximum Transmission Unit (MTU) of the network path the data is taking.

It is worth noting that fragmentation can be used by attacker's at any layer of the networking stack when the receiver of the traffic will still reassemble the payload correctly. Fragmenting traffic at a lower layer will assist in making all higher layer traffic more evasive. That is, IPv4 fragmentation will also result in the fragmentation of TCP, UDP, data. If one can successfully perform IPv4 fragmentation, not only will it evade rules that check only Layer 3 headers, but also Layer 4 content. Of course, IPv4 fragmentation is now easy to prevent, and it is considered a best practice to do so [36]–[38].

2.4.2 TCP Session-Splicing

While IPv4 fragmentation attacks are no longer likely to be successful, fragmentation at a higher layer can still be very powerful [38]. Doing so with TCP is a technique that has been referred to as "session-splicing" [11].

Generally, when an application seeks to send data over a TCP socket, a buffer is passed to a system's networking stack. The networking stack (usually implemented in the kernel) handles splitting large quantities of data into segments small enough to traverse the network path to the intended destination. This is to say that fragmentation is a crucial function performed by TCP, and, therefore, that one cannot simply block TCP fragmentation at a network perimeter. Splitting payloads into several segments is so integral to the operation of TCP, that suggesting preventing it from doing this is hardly intelligible. Attackers can take advantage of TCP fragmentation ("session-splicing") and be confident that the same mitigation that prevents IPv4 fragmentation will no longer be available to defenders.

Performing TCP session-splicing is relatively simple; instead of handing a large buffer to the networking stack in one system call on the attacker's system, the attacker splits the large buffer into several smaller ones and makes several calls. By using buffers as small as 1-byte and adding a small (fraction of a second) delay between calls to send data, the attacker can effectively split a large payload into several very small segments. This data is reassembled at the destination machine and treated by the destination application as if it had been all

contained in one PDU.

While this has a higher likelihood of being effective than IPv4 fragmentation, there are still important mitigations that NIDS uses:

- Signatures can be written which detect session-splicing: these signatures watch for a number of abnormally small TCP segments occurring a connection. [11]
- A "modern" NIDS will perform TCP stream reassembly. Here, the payload for either a part of or an entire TCP connection is saved as it traverses the IDS. The IDS reassembles these segment payloads to see a larger or the entire data being sent in a TCP connection. This larger, reassembled, segment data can then be analyzed to see if it matches any signatures.

This does not necessarily mean that session-splicing is useless against IDSs that implement these mitigations [38]. With respect to the first mitigation, an attacker can increase and vary the size of the segments sent. The goal of the attacker here is to send segments sufficiently large to avoid triggering a rule meant to detect fragmentation while still small enough to avoid rules designed to catch the actual attack being carried in the payload. Regarding the second, stream reassembly is a both computationally and memory intensive operation. It is memory intensive since the IDS would ideally save all the payload data for the entirety of all TCP connections. If there is a lot of TCP traffic on the network being watched, this becomes untenable very quickly. Compromises can be made by either retaining only part of every connection or only performing stream reassembly on certain connections–both suggest ways that an attacker could sneak in traffic.

CHAPTER 3:
Offensive Experiments

This chapter evaluates how MPTCP can be leveraged by attackers to evade MPTCP-unaware NIDSs that were never designed to reassemble the data stream of multi-subflow MPTCP connections. The implementation of a port-forwarding MPTCP session-splicer is discussed. This approach allows penetration testing and red teaming tools to use MPTCP cross-path fragmentation to evade NIDS without having to modify the tool's source code.

We present the design of a Python program called "splicer.py." When splicer.py is used on top of an MPTCP-enabled kernel and placed in-line between a penetration testing system (e.g. Kali Linux) and a target, splicer.py can perform MPTCP session-splicing (cross-path fragmentation attacks) allowing testers to evade MPTCP-unaware NIDSs.

Early work in MPTCP cross-path fragmentation used [23] to fragment a static payload over multiple subflows [22]. Afzal and Lindskog continued this work by showing that an existing Snort rule set could be evaded with these techniques [4]. The researches crafted payloads specifically designed to generate alerts by parsing the Snort rule set, and then compared the number of alerts generated by Snort when it observed these payloads being sent over traditional TCP with the number of alerts generated by sending the payloads with MPTCP and varying degrees of cross-path fragmentation. Both works demonstrated the validity of cross-path NIDS evasions by splicing a pre-defined payload; neither work attempted to interact with a target service to cause a malicious effect. This chapter presents a simple methodology that can be used to perform MPTCP cross-path fragmentation on arbitrary TCP payload while allowing full TCP interaction between endpoints; this should provide general support for MPTCP based penetration testing.

3.1 Design of Experiments

3.1.1 Scope

We focus on the case where the attacker initiates a TCP connection to the target. Specifically, we focus on server-side exploits and the ability of current NIDSs to detect malicious

MPTCP traffic. While MPTCP may have an interesting impact on client-side attacks, this investigation is left for future research.

Two general goals we had when developing an offensive methodology were ease of implementation and compatibility with existing offensive tools. When implementing a technique that uses MPTCP to evade network detection systems, the better we can meet these goals, the more impactful any demonstration of the technique should be. That is, a technique that is both relatively easy to implement and integrable with existing tools demonstrates a lower barrier-of-entry for attackers to desiring to use MPTCP evasions. Showing there is a low barrier to entry to perform these evasions underscores the need for defensive solutions. However, it should be noted that the greatest current obstacle for adversaries seeking to utilize these evasions is the low deployment of MPTCP [39]; it is difficult to abuse a protocol to hide from a NIDS if one cannot find targets using the protocol.

To these ends, our solution involves:

- A Linux machine running an MPTCP kernel configured for our purposes.
- Software that combines the functionality of a port-forwarder and a TCP session splicer. This runs on the Linux machine.
- A machine loaded with attack tools (this may be the Linux machine assuming the needed tools run on top of the MPTCP kernel smoothly).

In order to successfully exploit a service on a target machine while using session-splicing multipath evasion, the target must meet the following criteria:

1. The target is running a vulnerable service that uses TCP.
2. The target machine is capable of using MPTCP either via running an MPTCP kernel or an MPTCP Proxy [40], [41].
3. The target will accept new subflows (MP_JOINs) initiated by clients.
4. We must have a working exploit to use against the server.

3.1.2 Splicer

Splicer.py Description

Splicer.py is a python script that integrates the functionality of a port-forwarder and a TCP session-splicer. 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. Figure 3.1 shows how splicer.py operates when it is splicing and forwarding traffic.

What distinguishes splicer.py from other port-forwarders is that it performs TCP session splicing on data forwarded to the target machine. By default, Splicer will send payload intended for the target 1-byte at a time at 0.1 second intervals. Time intervals of 0.1 seconds were chosen because it is long enough to allow each spliced segment to be acknowledged before attempting to send the next one (the Round Trip Times (RTTs) for our lab environment were well below 200 milliseconds), which avoids multiple bytes accumulating in the socket's send buffer. If multiple bytes accumulate in the send buffer, then they will be all sent as one segment, reducing the overall fragmentation. Granted, this is an imperfect way to ensure that the spliced segment size is respected. Splicer.py could be improved by modifying its source code to use the TCP_NODELAY socket option, which should allow much faster transmission of spliced segments. Supporting a time delay is still useful, however, since long delays have been used to evade some attempts at TCP session reassembly [11].

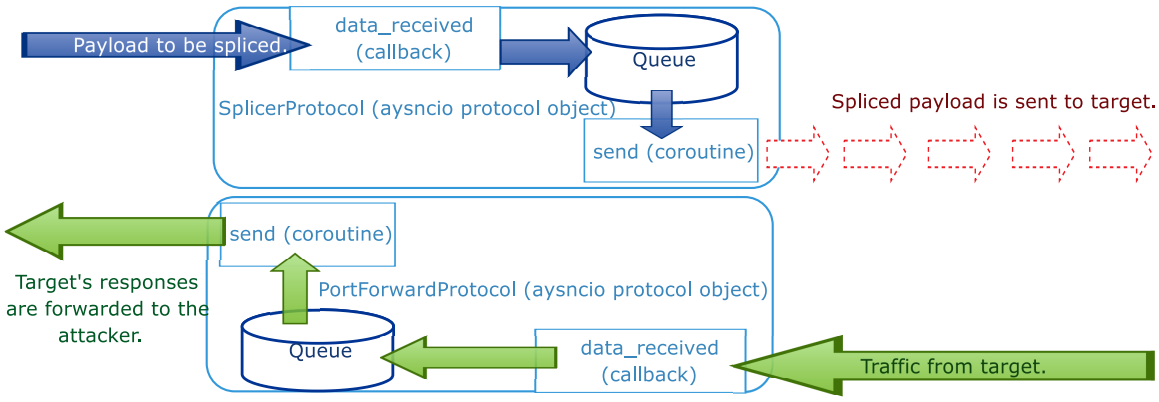

Figure 3.1. splicer.py Operation

Splicer should be able to forward any exploit that targets an application layer service that uses TCP. After configuring the socket mappings in splicer, a tester need only configure their exploit to target the incoming connections socket on the machine Splicer is running on. This may be `localhost`. However, it is not required to be, so long as the attacker's exploitation machine can communicate with the machine Splicer is running on and Splicer, in turn, can communicate with the target.

Kernel Configuration for MPTCP Session-Splicing

To allow Splicer to perform MPTCP session-splicing, it needs to be running on a machine with an MPTCP networking stack. Currently, this means a Linux machine with an MPTCP kernel installed. Also, it is necessary to configure the kernel to use the "roundrobin" scheduler; this option has been in the MPTCP kernel source since at least version 0.89 [42]. However, it must be included at compilation. The following additional configurations are also desirable, though in some instances may not be strictly necessary to evade network intrusion systems.

- MPTCP should be configured to use the `roundrobin` scheduler. This scheduler will send each successive segment on a different subflow (assuming there is more than one subflow).
- If the Splicer machine possesses a single IP Address that is routable to the target, the "ndiffports" path manager should be used.
  Additionally the `/sys/module/mptcp_ndiffports/parameters/num_subflows` file should specify a number greater than 1. This will ensure that upon connection to the victim, MPTCP will attempt to create at least one additional subflow that can be used to offload packets onto.
- If the Splicer machine is multi-homed and owns multiple IP Addresses that can be routed to the victim, then the `fullmesh` path manager is desirable. In this case, MPTCP will attempt to create a subflow between every possible interface pair. This potentially offers more evasiveness than `ndiffports` as there is a much better chance that the alternate subflows will take a different physical path through the network.

Interaction of the Splicer With Different Types of Payloads

Splicer.py will not modify or inspect any of the payload that it forwards; this means that it will forward the binary data for any attack. However, employing splicer.py involves altering the routing topology of a scenario, which is important to keep in mind when choosing different types of attacks. A discussion of the considerations that should be taken into account for three common types of shellcode follows as an example. The gist of this is that splicer.py can forward any binary stream, but testers and attackers should be aware that they are pushing their traffic through a port-forwarder and how this may affect or interfere the tools and procedures that they employ.

While shellcode can be written to cause a wide variety of effects on a system, we choose to consider three common types and considerations that should be taken into account when they are used in conjunction with splicer.py.

1. Bind payloads typically open a port on a machine and wait for incoming connections. For instance, after exploitation of a service on a victim results in the opening of a port and when an attacker initiates a connection with that port, a shell process is launched that takes input and directs output to the socket.
2. Callback payloads will cause the victim machine to create a new connection calling back to the attacker. This is useful if a victim machine is behind a firewall or NAT which prevents inbound TCP connections.
3. The third type of payload does not require the creation of a new connection. This type of payload allows interaction with the victim over the connection that delivered the payload.

With respect to Bind payloads, an attacker will need to remember to connect to the victim server and not the splicer (unless a port-mapping between the bound port on the splicing machine and the target has been created). This is not surprising in any way. However, it is worth noting it explicitly since some automated tools may attempt to follow-up an exploit which used a Bind payload, by connecting to the same IP address the payload was launched at. In this case, the attacker would fail to create a Command and Control (C2) channel unless the attacker had created a new mapping to the target on the splicing machine. This would allow the attacker to conduct C2 over a spliced channel. This may even make the attacker's C2 more evasive. In fact, the effect techniques like this have on the detectability

of attacker C2 may be an interesting area for future work.

Callback payloads should be configured to callback to the attacker's machine, and not the splicer machine, assuming that they are different. Again, this is unsurprising but is worth noting since some tools and scripts may not allow this fine a grain of configuration. A port forwarding from the splicer's target-facing interface to the attacker could be accomplished using a tool like socat. Also, splicer.py could be extended with an option to open callback ports.

In situations where the third type of payload is acceptable, it is likely the best choice as no tools will need to be required with different IP addresses.

Currently, the splicer, by itself, only performs standard TCP session splicing and relies on a properly configured kernel to perform splicing over MPTCP. A user-space MPTCP implementation using raw sockets could be integrated into splicer, however, it would require a significant amount of work to create such functionality. This would allow splicer to perform MPTCP cross-path fragmentation when running on a non-MPTCP kernel. An alternate, and likely less difficult approach would be to modify an existing user-space MPTCP proxy to perform cross-path fragmentation. One such possible proxy is [41].

3.1.3 Design of Scenario 1: Attacking with Normal TCP Operation

Purpose

Test Snort's ability to detect TCP content when attackers allow TCP to operate normally.

Topology

Figure 3.2 shows the network topology used in this experiment.

Description

The following Virtual Machines are networked together in a single broadcast and collision domain (each machine is able to view all traffic on the network).

- An attacker (Kali 2016 Virtual Machine) will send traffic to a victim.

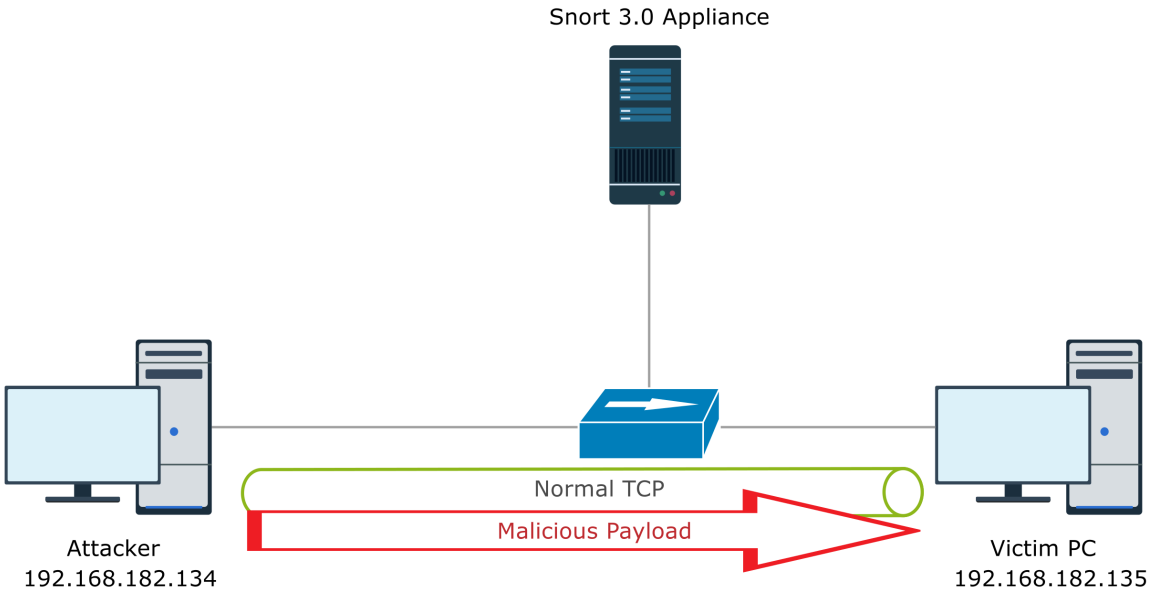

Figure 3.2. Scenario 1 Topology

- 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 Snort 3)

Two tests will be performed. First, we attempt to detect the simple string `"FIREFIREFIRE"` when it is transmitted from the attacker to the victim. Next, we attempt to exploit a service that executes received data by sending it shellcode.[7]

Procedure for Test 1

1. The victim will use netcat to listen on TCP Port 7878 for incoming connections:

   ```
   nc -lkp 7878
   ```

2. The IDS machine will begin monitoring traffic on the network looking for the string with the following rule:

[7]The string `"FIREFIREFIRE"` could be replaced with any sequence of bytes that a Snort rule has been written to detect. `"FIREFIREFIRE"` just happens to be an entertaining sequence of characters that can be easily entered from a keyboard into a netcat session. The second tests performed for each scenario in Chapter 3 replace `"FIREFIREFIRE"` with actual shellcode that might accompany a real exploit.

```
alert ip any any -> any any (content: "FIREFIREFIRE"; msg:
↪ "FIREx3 detected!"; sid:100000001;)
```

3. The attacker will connect to the victim with the netcat utility and send the string.
4. The IDS box will be checked to see if it detected the content.

Procedure for Test 2

1. The vulnerable service will be started on the victim.
2. The IDS machine will begin monitoring traffic with a rule to match the shellcode used.[8]
3. The attacker will send a Metasploit generated payload to the victim.

```
(msfvenom -p linux/x64/shell_find_port CPORT=4445 -f raw ;
↪ \textbackslash\newline cat -) | nc -p 4445 192.168.182.135
↪ 7879
```

4. The attacker will verify that a shell was spawned and can be interacted with.
5. The IDS box will be checked to see if it detected the content.

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. Some of the predicted characteristics of the attacker's traffic are shown in Table 3.1.

[8] Shellcode from the Metasploit Framework is used; specifically the linux/x64/shell_find_port payload. The snort rule to detect this payload can be found in the Snort rules appendix.

# of TCP Connections (subflows):	1
# of Segments to Transmit Payload for Test 1:	1
Detectable w/o TCP Stream Reassembly?:	Yes
Detectable w/o MPTCP Stream Reassembly?:	Yes

Table 3.1. Hypothesized Characteristics of Scenario 1

3.1.4 Design of Scenario 2: Attacking with TCP Session-Splicing

Purpose

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

Topology

Figure 3.3 shows the network topology used in this experiment.

Description

The following Virtual Machines are networked together. The splicer virtual machine, the victim and the IDS share a broadcast and collision domain which allows the IDS to observe traffic between the splicer and the victim.

- 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 Snort 3)
- A splicer box (64-bit Ubuntu 14.04 running Python 3.5 with splicery.py) running normal TCP.

Two tests will be performed. First, we attempt to detect the simple string `"FIREFIREFIRE"` when it is transmitted in single-byte packets from the splicer to the victim. Second, we attempt to exploit a service that executes received data by sending it shellcode.

Procedure for Test 1

1. The victim will use netcat to listen on TCP Port 7878 for incoming connections:

   ```
   nc -lkp 7878
   ```

2. The IDS machine will begin monitoring traffic on the network looking for the string with the following rule:

   ```
   alert ip any any -> any any (content: "FIREFIREFIRE"; msg:
   ↪ "FIREx3 detected!"; sid:100000001;)
   ```

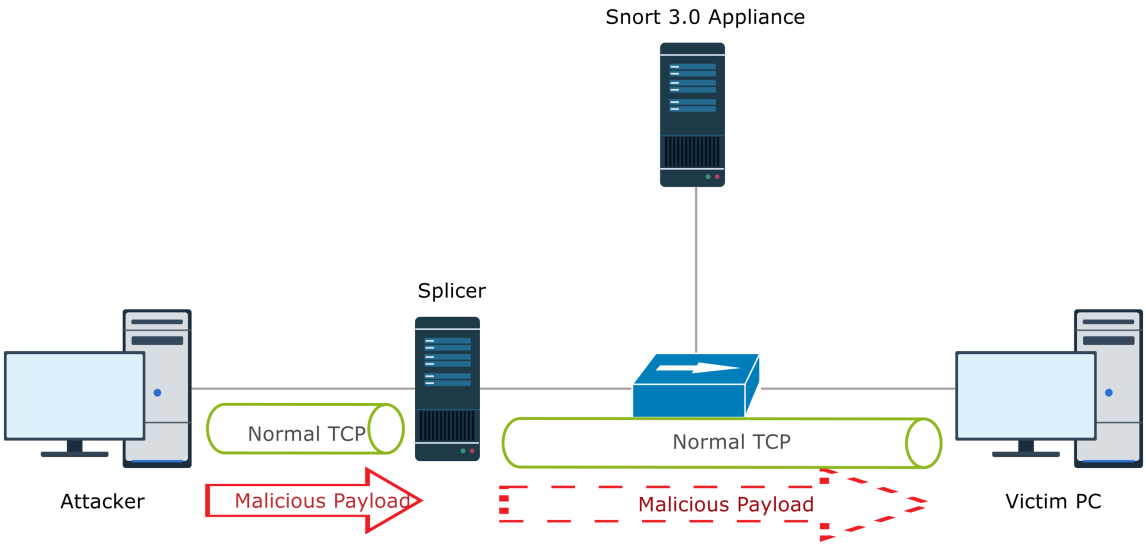

Figure 3.3. Scenario 2 Topology

3. A port binding will be configured on the splicer as follows: incoming TCP connections on port 7878 will be spliced and forwarded to port 7878 on the victim.

   ```
   python3.5 splicer.py 0.0.0.0:7878 192.168.182.135:7878
   ```

4. The attacker will connect to the splicer with the netcat utility and send the string.
5. The victim will be checked to see if it received the transmission.
6. The IDS box will be checked to see if it detected the content.

Procedure for Test 2

1. The vulnerable service will be started on the victim.
2. The IDS machine will begin monitoring traffic with a rule to match the shellcode used.
3. A port binding will be configured on the splicer as follows: incoming TCP connections on port 7878 will be spliced and forwarded to port 7878 on the victim.

   ```
   python3.5 splicer.py 0.0.0.0:7878 192.168.182.135:7878
   ```

4. The attacker will send a Metasploit generated payload to the splicer.

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

5. The attacker will verify that a shell was spawned and can be interacted with.

6. The IDS box will be checked to see if it detected the content.

Hypothesis

Snort will generate an alert for both tests. Some of the predicted characteristics of the attacker's traffic are shown in Table 3.2.

# of TCP Connections (subflows):	1
# of Segments to Transmit Payload for Test 1:	12
Detectable w/o TCP Stream Reassembly?:	No
Detectable w/o MPTCP Stream Reassembly?:	Yes

Table 3.2. Hypothesized Characteristics of Scenario 2

3.1.5 Design of Scenario 3: Attacking with MPTCP Session Splicing

Purpose

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.

Topology

Figure 3.4 shows the network topology used in this experiment.

Description

The following Virtual Machines are networked together. The splicer virtual machine, the victim and the IDS share a broadcast and collision domain which allows the IDS to observe traffic between the splicer and the victim.

- 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 Snort 3)
- A splicer box (64-bit Ubuntu 14.04 running Python 3.5 with splicery.py) running Multipath TCP.

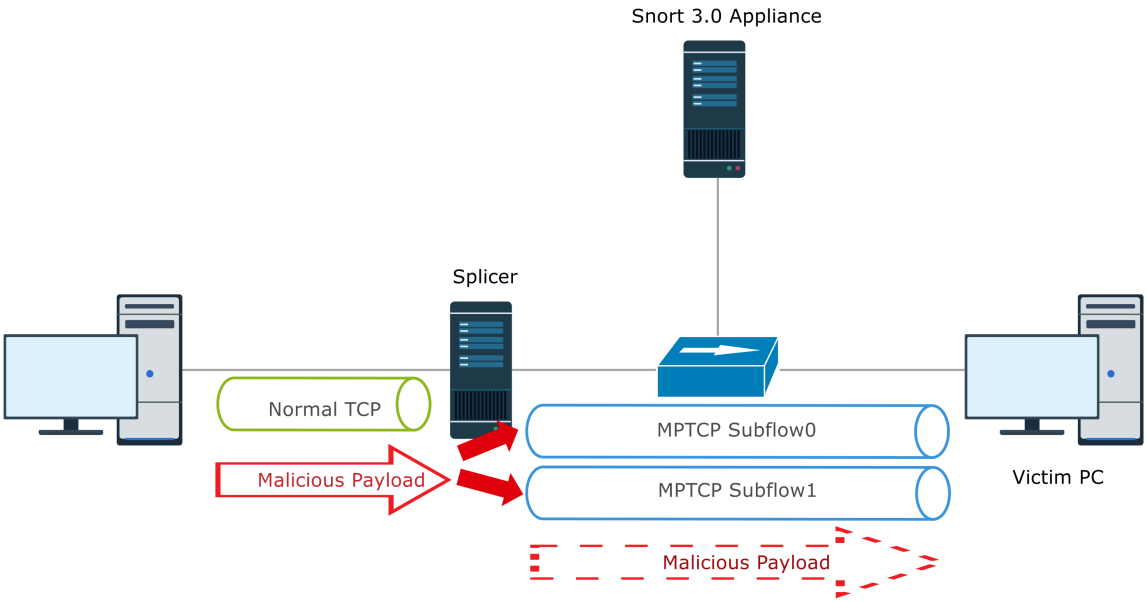

Figure 3.4. Scenario 3 Topology

Two tests will be performed. First, we attempt to detect the simple string "FIREFIREFIRE" when it is transmitted in single-byte packets from the splicer to the victim along multiple subflows. Second, we attempt to exploit a service that executes received data by sending it shellcode.

Procedure for Test 1

1. The victim will use netcat to listen on TCP Port 7878 for incoming connections:

   ```
   nc -lkp 7878
   ```

2. The IDS machine will begin monitoring traffic on the network looking for the string with the following rule:

   ```
   alert ip any any -> any any (content: "FIREFIREFIRE"; msg:
   ↪ "FIREx3 detected!"; sid:100000001;)
   ```

3. The splicer will be running MPTCP with the roundrobin scheduler and ndiffports path manager. The num_subflows parameter of ndiffports is set to 2.
4. A port binding will be configured on the splicer as follows: incoming TCP connections on port 7878 will be spliced and forwarded to port 7878 on the victim.

   ```
   python3.5 splicer.py 0.0.0.0:7878 192.168.182.135:7878
   ```

5. The attacker will connect to the splicer with the netcat utility and send the string.
6. The victim will be checked to see if it received the transmission.
7. The IDS box will be checked to see if it detected the content.

Procedure for Test 2

1. The vulnerable service will be started on the victim.
2. The IDS machine will begin monitoring traffic with a rule to match the shellcode used.
3. The splicer will be running MPTCP with the roundrobin scheduler and ndiffports path manager. The num_subflows parameter of ndiffports is set to 2.
4. A port binding will be configured on the splicer as follows: incoming TCP connections on port 7878 will be spliced and forwarded to port 7878 on the victim.

   ```
   python3.5 splicer.py 0.0.0.0:7878 192.168.182.135:7878
   ```

5. The attacker will send a Metasploit generated payload to the splicer.

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

6. The attacker will verify that a shell was spawned and can be interacted with.
7. The IDS box will be checked to see if it detected the content.

Hypothesis

Snort will fail to generate an alert for both tests. Some of the predicted characteristics of the attacker's traffic are shown in Table 3.3.

# of TCP Connections (subflows):	2
# of Segments to Transmit Payload for Test 1:	12
Detectable w/o TCP Stream Reassembly?:	No
Detectable w/o MPTCP Stream Reassembly?:	No

Table 3.3. Hypothesized Characteristics of Scenario 3

3.2 Results

For each of the tests, Snort 3 configured with a rule to detect a portion of the traffic sent from the attacking host by matching content transferred to the target machine in TCP payload. Regarding Scenario 1, Snort does not need to perform any reassembly of payload data because the offending payloads for both tests are small enough to fit into one segment. Regarding Scenario 2, Snort needs to perform traditional TCP stream reassembly since splicer is used to transmit the offending content in 1-byte chunks. Regarding Scenario 3, Snort needs to perform reassembly of an MPTCP data stream that has been fragmented over multiple subflows. Failure of Snort to generate alerts for the tests performed in this scenario validates splicer.py's ability to obfuscate TCP content when targeting an MPTCP-enabled host on a network with an MPTCP-unaware NIDS.

3.2.1 Scenario 1 Results

Test1

A snort alert was generated when the string was transmitted by the attacker machine to the victim machine. For instance:

```
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

A Snort alert was generated by the shellcode and the attacker was able to interact with a shell on the target.

```
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
```

3.2.2 Scenario 2 Results

Test 1

A Snort alert was generated when the string was transmitted by the attacker machine to the victim machine. For instance:

```
07/15-03:01:54.741825 [**] [1:100000001:0] "FIREx3 detected!" [**]
 ↪  [Priority: 0] {TCP} 192.168.182.134:55364 ->
 ↪  192.168.182.135:7878}
```

Test 2

A Snort alert was generated by the shellcode and the attacker was able to interact with a shell on the target.

```
07/27-08:27:34.011309 [**] [1:100000002:0] "MSF!
 ↪  linux/x64/shell\_find\_port" [**] [Priority: 0] {TCP}
 ↪  192.168.182.134:4445 -> 192.168.182.135:7878
```

3.2.3 Scenario 3 Results

Test 1

No Snort alerts were generated by the first test. As can be seen from the Wireshark analysis (Figures 3.5 and 3.6) 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.

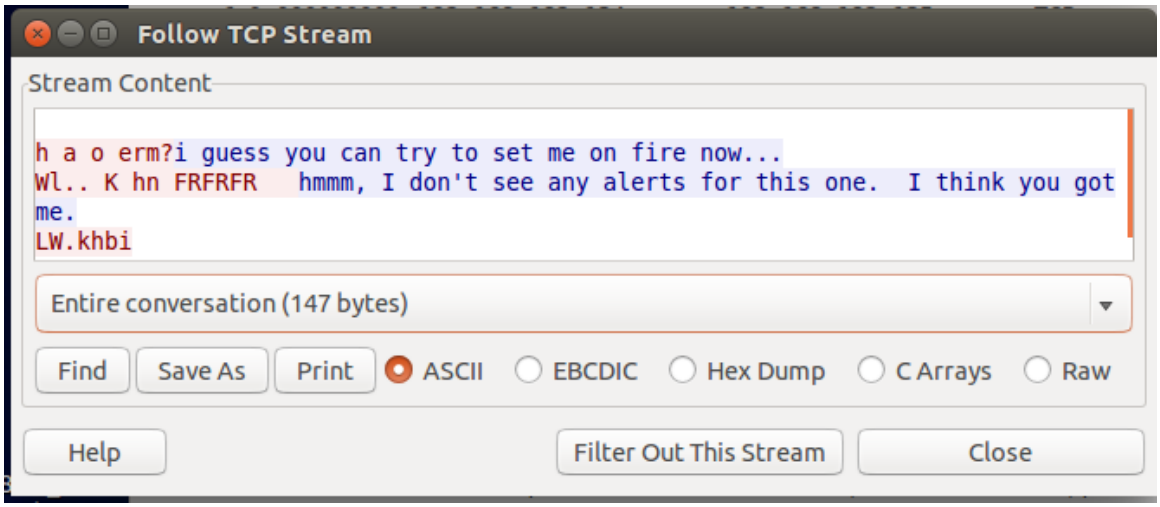

Figure 3.5. Wireshark Display of First Subflow Stream From Scenario 3 Test 1

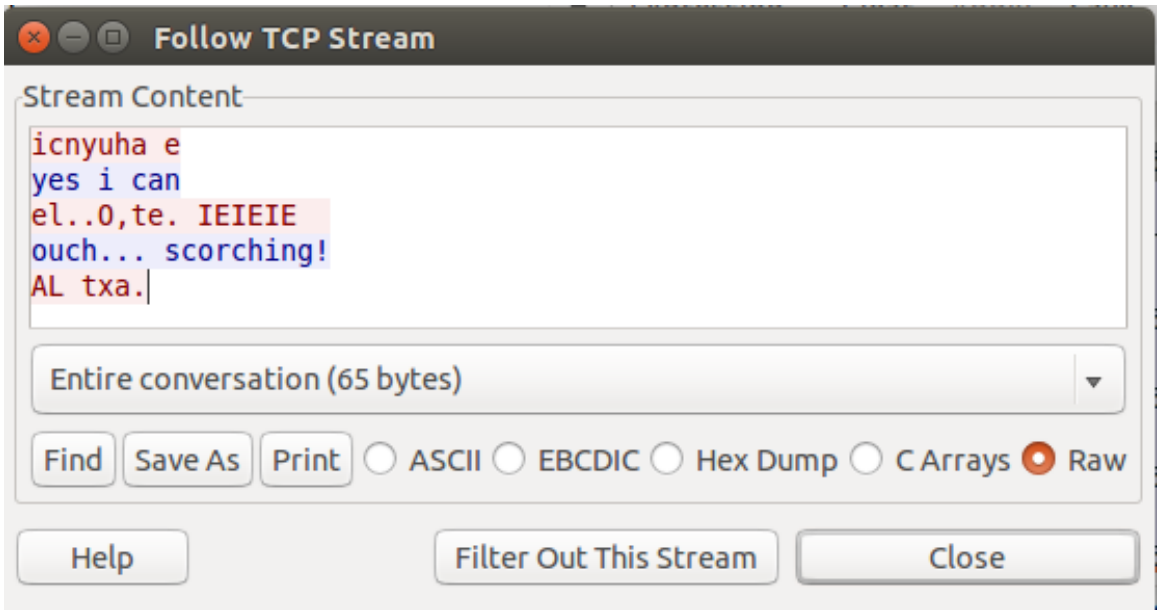

Figure 3.6. Wireshark Display of Second Subflow Stream From Scenario 3 Test 1

Test 2

No snort alerts were generated by the second test. The attacker was able to interact with a shell on the target.

3.3 Discussion

This chapter demonstrated that MPTCP cross-path fragmentation is an effective technique for evading an MPTCP-unaware NIDS like Snort 3. Snort 3 did not generate an alert for content that matched a loaded rule when that content was fragmented across multiple MPTCP subflows. Some avenues for future work are discussed in this section.

Improvements to splicer.py

While splicer.py functions sufficiently well to perform MPTCP evasions, there are simple ways that it could be improved. For instance, additional configuration options to make the program more customizable could be added. Currently, the program will fragment payload into 1-byte segments which it sends every 0.1 seconds. Also, support for allowing multiple source to target mappings at a time could be added. This change should not pose too much difficulty as splicer.py was created with Python's asynchronous I/O library, asyncio. There are likely many different features that would be useful to add. For instance, experiments in this chapter use a find port shellcode that requires specifying the client port of the outbound connection; when sent through splicer.py, the connection received by the server would have a different source port than the shellcode was programmed to find. To allow this shellcode to work through splicer, the –preserve-client-port command line switch was added. Splicer.py would benefit from further testing with different kinds of traffic to discover the enhancements that would make it more useful. Splicer.py could also be made easier to use via a graphical user interface or a web interface. Currently, the best way to use splicer.py is to create a virtual machine, install Linux, and an MPTCP kernel compiled to include the roundrobin scheduler. A great deal of time and effort could be saved if a pre-built VM was available. A solution like Vagrant [43] could be used to simplify the procedure for running splicer. This is the same as an approach the Linux Kernel development team uses for deploying MPTCP on machines if changing the kernel is not desired or possible [44].

Client-Side Attack Research

This chapter focuses on evading NIDS while performing server-side exploitation. It would be worthwhile to investigate if MPTCP can be leveraged to evade NIDS while conducting client-side attacks. For instance, whether cross-path fragmentation could be used to hide a malicious binary that a target downloads, or if it is capable of hiding malicious javascript

on a website that a target was social engineered into visiting could be investigated.

Making Post-Exploitation Network Traffic More Evasive

Another area of research worth pursuing is if MPTCP can be used to make it more difficult for network defenders to detect post-exploitation activities, like data exfiltration or command and control traffic (for example, a malicious implant sending a beacon to a control server).

CHAPTER 4:
Defensive Countermeasures

This chapter investigates two countermeasures that can be employed to prevent the evasions shown in Chapter 3.

The first countermeasure involves using a transport layer proxy to convert MPTCP streams with multiple subflows into a single-path stream that can be reassembled by existing NIDS. This scheme has been suggested by Afzal, Lindskog, and Lidén, who developed an MPTCP-TCP proxy that is written in Python and operates in user space [28]. Alternatively, we present a way to proxy MPTCP connections that use multiple subflows into a single-path connection using TCP port-forwarding on top of MPTCP kernels. This solution has the benefit of leveraging a mature MPTCP implementation to combine subflows. To show that the port-forwarding scheme works, we use socat [45]. Trudy [46] is used to demonstrate that the port-forwarding approach can be used to create a transparent proxy.

The second countermeasure investigates enhancing Snort 3 with the capability to reassemble cross-path fragmented MPTCP streams as it encounters them. To this end, we create a reassembly program, mpr_server.py and a dynamic Snort 3 plugin, mpr_server.py. The mpr_server.py program provides MPTCP reassembly services to the dynamic inspector. Working in conjunction, the two provide Snort 3's detection engine with reassembled MPTCP data streams which it can inspect against its loaded rule set. Another approach to reassembling MPTCP streams for Snort involved generating TCP pcap files that contained the same payload as observed MPTCP streams; these pcap files were then fed to Snort [4]. Our investigation continues this work by making the reassembled MPTCP stream directly available to Snort 3 with a dynamic inspector plugin. Similar work was also performed with the Bro NIDS. However, full MPTCP reassembly was not implemented. Additionally, difficulties encountered by the author suggest that it may be more complicated to implement MPTCP reassembly in Bro than it is in Snort.

4.1 Cross-Path Defragmentation with Transport Layer Proxies

The MPTCP proxying scheme allows multipath communication on exterior networks while making traffic on internal networks single-path. Because the internal traffic occurs over either a single subflow or a traditional TCP connection, its data stream may be inspected by MPTCP-unaware NIDS. This scheme is shown in Figure 4.1.[9]

4.1.1 Description of Using Port-Forwarding as an MPTCP Proxy

Concerning what software should be used to proxy MPTCP traffic, there are some options. These options include using simple port-forwarding on top of an MPTCP enabled kernel, an application specific proxy, or a proxy specially designed to convert MPTCP to TCP.

We choose to use simple port-forwarding. The proxy acts as an MPTCP receiver, when incoming connections are accepted from an MPTCP client, it creates a single-path connection to the host where the service the client is attempting to connect to resides. The port-forwarder then receives information directly from the client and passes it to the server over the new TCP connection. Because the port-forwarder operates in user space, the MPTCP stack implemented in the kernel will have already reassembled the data stream before it is made available to the forwarding application.

An application specific proxy (e.g., an HTTP Proxy) would likely defragment MPTCP streams in the same manner as a simple port-forwarder would. This would occur if the proxy was implemented in user space and created a second, single-path connection after receiving the traffic from the initiator. The downside of this approach is that it will only function for the application layer protocol for which the proxy was designed. By using a port-forwarder instead, it is possible to proxy all MPTCP traffic regardless of the application layer protocol.

An MPTCP specific proxy could be used as well but creating our own would require a substantial effort. There is an implementation available of an MPTCP proxy [40]. However, it has not been updated in some time and by using a simple port-forwarder we can take

[9]MPTCP traffic on the internal network will not pose a problem if each connection is restricted to a single subflow. To the NIDS it will appear the same as traditional TCP with an unusual TCP option field present.

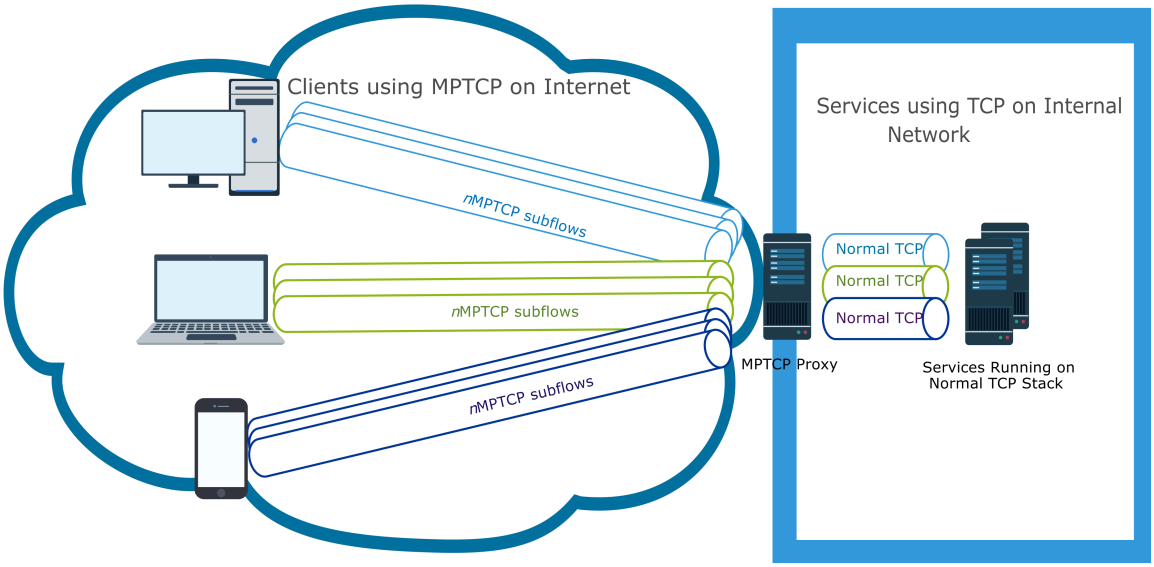

Figure 4.1. MPTCP to TCP Conversion by Proxy

advantage of an up-to-date kernel MPTCP implementation.

Preventing MPTCP Traffic on the Internal Network

Since the simple port-forwarding scheme described in Section 4.1.1 sits in user space and leverages MPTCP capabilities in the kernel, the behavior of MPTCP occurs transparently to the forwarding application. This means that after the proxy receives an MPTCP connection on its external interface, it will initiate an MPTCP connection when it connects to the internal host the service is running on. This is undesirable as it may break our scheme of using MPTCP-unaware NIDS to investigate the traffic if the new MPTCP connection uses multiple paths.

However, enforcing that only traditional TCP traffic exists on the internal network is not strictly necessary to allow proper NIDS operation. It is more important that the data be sent along a single path; this single path could be a traditional TCP connection or a single-subflow MPTCP connection.

A normal installation and configuration of MPTCP will tend to send data sequentially and prefer using few subflows over many. In the Linux kernel implementation [12] the default behavior of the Path Manager is only to accept additional subflows but will not initiate creation new subflows with the MP_JOIN handshake. Chapter 3 describes techniques

and configurations that an attacker would need to go through to make MPTCP evasive. The port-forwarder would not be configured in this manner. Therefore, it is unlikely that MPTCP-unaware NIDS would be impeded even if the proxy were allowed to initiate a connection with an MP_CAPABLE handshake on the internal network. However, this does not guarantee that traffic will not span multiple subflows (at least not without controlling the configuration of the internal hosts). Therefore it is worth discussing techniques to ensure single-path communication on the internal network:

- Use only non-MPTCP endpoints to host services that the NIDS is monitoring. This solution may seem trivial, since if there were no MPTCP-capable hosts within a network, the network administrators would likely not be concerned with NIDS evasion conducted via MPTCP fragmentation. However, we believe some organizations may opt to utilize MPTCP-TCP proxies in this fashion so that they may reap the benefits of MPTCP over lower-quality public networks while utilizing traditional TCP on high-quality internal networks.
- Block MPTCP by configuring a firewall to drop packets with the MPTCP option. A SANS whitepaper provides example iptables rules to accomplish this [26]. RFC 6824 specifies that when multiple MP_CAPABLE SYNs fail, MPTCP implementations should fallback and attempt a traditional TCP handshake [1]. That is, MPTCP should fallback to traditional TCP even if entire packets of the MPTCP handshake are dropped. However, the number of MP_CAPABLE attempts and the length of the timeout is left up to "user-policy." Therefore, while effective for preventing MPTCP on a local network, this approach may lead to unacceptable delay and performance degradation depending on the MPTCP implementation being used and its configuration.
- Instead of blocking all packets that carry an MPTCP option, it is likely wiser only to drop MPTCP option carrying packets with the MP_JOIN subtype [1]. This will restrict MPTCP communication to only one subflow, and MPTCP-unaware NIDS should not have an issue reassembling the data stream since it will not be fragmented over multiple subflows. The occasion that this will not allow reassembly of a full TCP stream is if a multi-homed device joins a new subflow on an unmonitored network to an MPTCP connection that was initiated on the monitored network. For example an employee could bring an MPTCP capable mobile device to work and

connect it to a monitored wireless network. An app on the mobile device then initiates an MPTCP connection to a service over the monitored wireless network. This connection becomes established because MP_CAPABLE MPTCP sub options are allowed. The mobile device then initiates a subflow using a cellular network; this subflow is joined to the MPTCP connection. A NIDS monitoring traffic from the wireless network will not be able to observe any data sent on the cellular network. This exception is less of an issue if the wireless network is segmented away from the rest of the organization's assets and treated as an untrusted network. Additionally, an organization could enforce a policy that prohibited unauthorized multi-homing of network assets. Finally, if the mobile device is owned by the organization, all network traffic could be forced through a monitored Virtual Private Network (VPN).

- Intercept MPTCP packets and erase the MPTCP option field in transit. There are existing firewalls that do this [24]. For instance, the proxy could be placed in front of a Cisco ASA firewall, which, unless configured not to, will rewrite MPTCP options to NOOP options [25]. Even if the packet arrives at an MPTCP capable host, the host will not receive an MP_CAPABLE SYN and the connection will fallback to traditional TCP. Like dropping packets carrying MPTCP options, this will prevent MPTCP on the internal network. However, it will avoid the performance hit that would occur if MPTCP waited for multiple SYN packets to be unacknowledged before switching to traditional TCP.

An attractive solution would be to disable MPTCP on the proxy's internal interface. It is unclear if this can be easily accomplished. The online documentation for the Linux kernel implementation describes how an MPTCP compatible version of the iproute [47] utility could be used to disable MPTCP on specific interfaces [48]. Unfortunately for our interests, this only affects MPTCP's operation in joining additional subflows; TCP connections initiated from the "disabled" interface will not be affected. That is, the first packet of a new TCP connection will carry the MP_CAPABLE option. This functions as intended by the developers and is clearly mentioned in the documentation.

For our experiments, we do not go so far as to prevent MPTCP traffic on our internal network. Instead, we configure all these hosts to utilize the default MPTCP path manager of the Linux kernel implementation. The default path manager will accept subflows but will not create more. Since all MPTCP connections in the internal network are created with

the default path manager, they are limited to a single subflow and may be investigated by MPTCP-unaware NIDSs.

4.1.2 Using socat Port Forwarding as a MPTCP Proxy

Port Forwarding with socat

The first procedure we present that uses an MPTCP proxy for cross-path defragmentation is performed using socat. A networking utility, socat, allows for the creation of a port-forwarding tunnel that funnels incoming TCP connections toward a port on a target machine. The command-line invocation we use to achieve the behavior described in Section 4.1.1, is as follows:

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

A characteristic of this scheme is that port-forwardings must be configured independently for every service upon which we want to perform reassembly on. Also, the TCP initiator must attempt to connect to the proxy's IP address instead of the intended service. This likely is not much of an issue when attempting to reassemble incoming traffic directed at services running in the internal network. However, this scheme becomes trickier if we attempt to reassemble streams that are initiated by internal clients (this is largely beyond the scope of this thesis, as we focus on detecting server-side exploitation). There are tradeoffs here that are akin to those when choosing between an overt application-layer proxy and a transparent one. We experiment with transparent MPTCP proxy functionality in 4.1.3.

Experiment Network Topology and Procedure

Figure 4.2 shows the scheme we use to counter the evasion that was successful in Section 3.1.5.

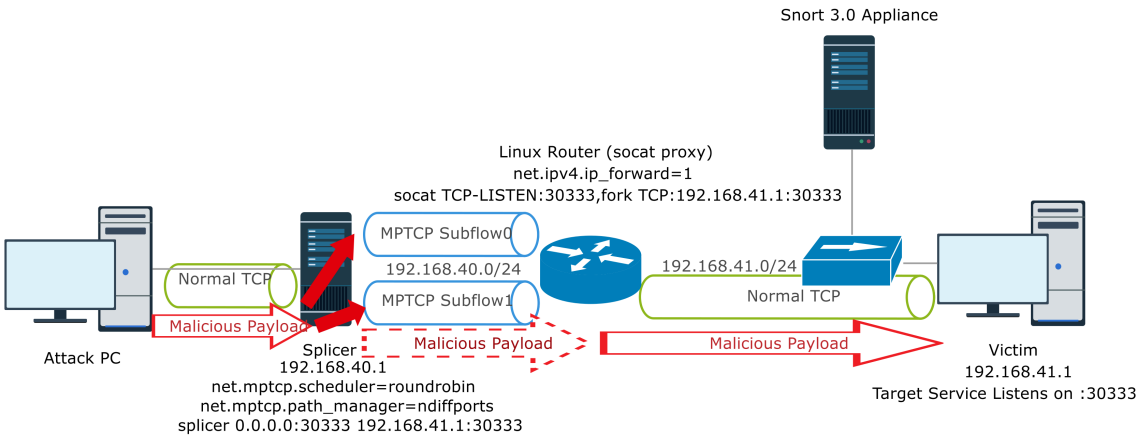

Figure 4.2. Cross-Path Defragmentation of MPTCP Traffic with socat

The same test used in Section 3.1.5 was reproduced with the addition of a middlebox (the Linux router shown in the figure) acting as a TCP proxy. The Attack PC sent traffic with the string "FIREFIREFIRE" to the TCP service host (the victim). Snort 3 was configured with a rule to detect this string. After sending the traffic, we observed that Snort 3 generated an alert for the string (this is in contrast to the in the Section 3.1.5 experiment where no proxy was present to perform cross-path defragmentation and no alert was generated).

The procedure used to conduct this experiment is listed here:

1. Start splicer.py with 192.168.40.254:30333 as the target socket:

   ```
   python3 splicer.py 0.0.0.0:30333 192.168.40.254:30333
   ```

2. Create a port-forwarding on the Linux router with socat to function as a multipath to single-path proxy:

   ```
   socat TCP-LISTEN:30333; fork TCP:192.168.41.1:30333
   ```

3. Start a netcat listener on the victim host:

   ```
   nc -lkp 30333 -v
   ```

4. From the attack host, connect to the splicer and send a the payload that Snort 3 has been configured to watch for:

   ```
   (echo "sometext   FIREFIREFIRE  ok"; cat -) | nc 192.168.40.1
   ↪  30333 -v
   ```

Hypothesis

We hypothesize that our scheme will be successful and that our Snort 3 machine will generate an alert even though the attacker employed the evasions described in Chapter 3.

Results

Snort 3 did generate an alert, therefore the hypothesis was validated.

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

4.1.3 Trudy as a Transparent TCP Proxy

Trudy is security testing tool that functions as a TCP Man-in-the-Middle (MitM) [46], [49]. When placed on a network path, Trudy is capable of transparently intercepting and modifying all TCP traffic on that path. For our purposes, modification of the payload is not necessary. Like socat, Trudy operates in user space using the kernel to present it with a single stream of payload.

The reason to explore using Trudy instead of socat (see Section 4.1.2) is that Trudy will function as a transparent proxy for all TCP traffic. This provides two benefits over our use of socat:

- Independent port mappings do not need to be configured, instead iptables rules are used to direct incoming traffic to Trudy.
- Trudy functions as a transparent proxy in our scenario. Unlike the behavior described in Section 4.1.2, TCP initiators will not need to connect directly to an IP address owned by the proxy. Instead, they will attempt to connect to the IP address of the intended TCP receiver.

Description of Trudy's Operation

Trudy operates by listening for incoming TCP connections on a configured port (6666 by default). To intercept TCP traffic for other ports, Trudy relies on iptables rules to be

configured. In our testing, we use a blanket iptables rule to redirect all TCP traffic to Trudy– for all destination network addresses. After an incoming TCP connection is redirected to Trudy by iptables, Trudy accepts the connection and begins receiving payload sent from the TCP initiator. Trudy then "retrieves" the original destination address from the kernel and creates a new TCP connection to the TCP initiator's intended destination. Trudy will forward payload it receives from one of these connections to the other, allowing for bi-directional communication between the TCP initiator and the service it intended to connect to.

By running Trudy on top of an MPTCP kernel, the codebase of an existing MPTCP implementation can be used to defragment incoming MPTCP data streams. Like socat, Trudy lives in user space, relies on the kernel to present it with a single-stream of data (Trudy is not aware whether the data it receives was fragmented across multiple paths or not). Because Trudy establishes a new TCP connection, which it forwards traffic intended for the TCP receiver on, all that is required is to restrict this connection to a single subflow or force it to fallback to traditional TCP (as described in Section 4.1.1) for MPTCP-unaware NIDS to be able to inspect the data stream.

Experiment Network Topology and Procedure

Figure 4.3 shows the scheme we use to counter the evasion that was successful in Section 3.1.5 with Trudy. The routing topology is identical to Section 4.1.2 (socat), however, in this case, Trudy is used as a transparent proxy, while socat was used as a non-transparent proxy.

The same test used in Sections 3.1.5 and 4.1.2 will be reproduced. The attacking host will send traffic with the string "FIREFIREFIRE" to the victim. We observe whether Snort 3 generates an alert after being configured with a content matching rule to detect this string. Like in Section 4.1.2, the Linux router serves as the gateway to the internal network that is being monitored by a NIDS. In this case, however, the attacker will configure splicer.py with the IP addresses of the victim (when socat was used, splicer.py needed to be configured with IP address of the Linux router host that socat was running on).

1. An iptables rule is configured on the Linux router to redirect all TCP traffic to Trudy.

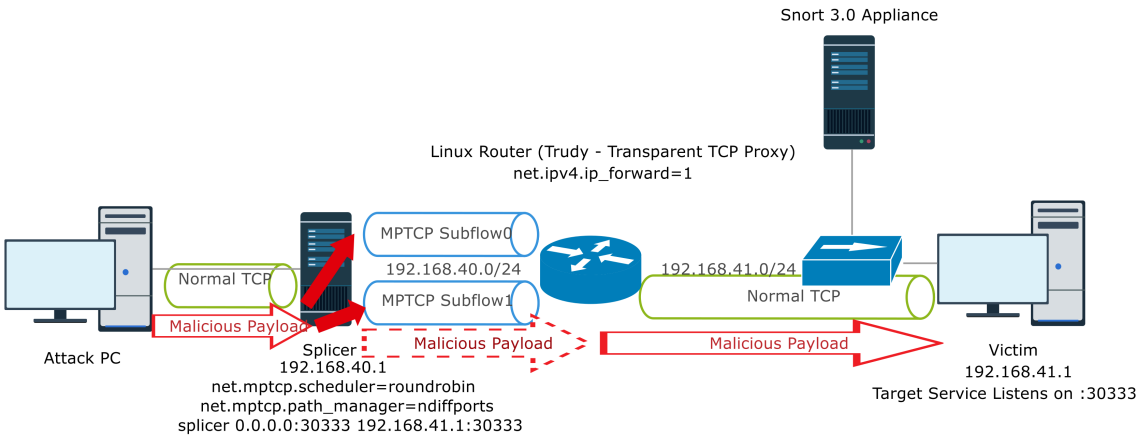

Figure 4.3. Cross-Path Defragmentation of MPTCP Traffic with Trudy

```
iptables -t nat -A PREROUTING -i eth1 -p tcp -m tcp -j REDIRECT
↪  --to-ports 6666
```

2. Trudy is started on the Linux router: `sudo trudy`
3. The splicer.py program is started and configured with the following command:

   ```
   python3 splicer.py 0.0.0.0:30333 192.168.41.1
   ```

4. A netcat listener is started on the victim host:

   ```
   nc -lkp 30333 -v
   ```

5. The attacker connects to the splicer and sends a the payload that Snort 3 has been configured to watch for:

   ```
   (echo "sometext   FIREFIREFIRE  ok"; cat -) | nc 192.168.40.1
   ↪  30333 -v
   ```

Hypothesis

We hypothesize that our scheme will be successful and that our Snort 3 machine will generate an alert even though the attacker employed the evasions described in Chapter 3.

Result

Snort 3 did generate an alert, therefore the hypothesis was validated.

```
09/08-01:12:22.019058 [**] [1:100000001:0] "FIREx3 detected!" [**]
↪ [Priority: 0] {TCP} 192.168.41.254:33378 -> 192.168.41.1:30333
```

4.2 MPTCP Data Stream Reassembly

This section investigates enhancing Snort so that it is capable of reassembling cross-path fragmented MPTCP streams. Extending Snort in this way allows Snort to analyze cross-path fragmented network traffic without needing to use a middlebox to perform cross-path defragmentation. One advantage over the proxy approach is that MPTCP data stream reassembly would allow Snort to run against captured MPTCP traffic, while, for the proxy solution to be successful, the proxy needs to intercept the connection while it is active.

In this section, we investigate the possibility of making Snort 3 capable of reassembling MPTCP data streams. The approach we take is to create a dynamic Snort 3 plugin that sends the TCP/IP fields necessary for MPTCP reassembly to a reassembly server running in a remote process. The reassembly server is responsible for performing the reassembly of the MPTCP data stream and sends the reassembled stream back to Snort so that Snort can perform detection on it.

4.2.1 Design of Snort 3 Inspector and MPTCP Reassembly Server

Our integration of MPTCP data stream reassembly into Snort 3 consists of two main components, a Snort 3 dynamic inspector, and a reassembly server written in Python 3. The two communicate with Inter-Process Communication (IPC). Figure 4.4 shows our design.

The inspector sends the necessary fields of the IP and TCP headers of every observed TCP packet to the reassembly server. For this task, the necessary header fields are serialized with Protocol Buffers [50] and sent to the reassembly server over a Unix domain socket connection. The mptcp_stream inspector waits for the reassembly server to return the reassembled data stream. Once the mptcp_stream inspector has received the reassembled stream, it crafts a “pseudo-packet” and submits it to Snort’s detection engine for checking against the currently loaded ruleset.

The main goal of this work was to create a prototype implementation and not necessarily a production ready system. Therefore, many of the design choices were made to ease and

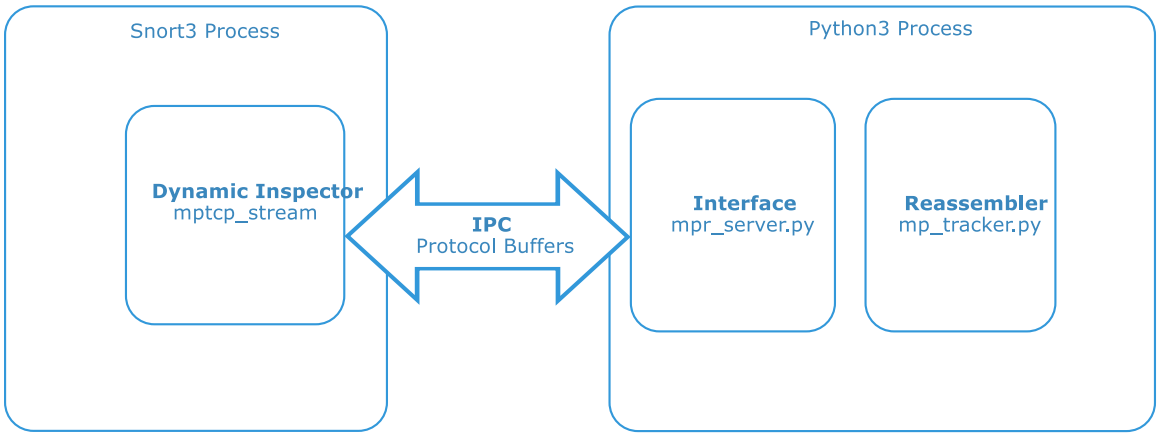

Figure 4.4. Design of Snort 3 Inspector and MPTCP Reassembly Server

simplify the development process. This involved trade-offs, while the chosen design made it easier to experiment and explore the problem space, little attention was paid to performance and stability which are important for production systems. For instance:

- Snort 3 was chosen over Snort 2 because Snort 3 is designed to be more pluggable than Snort 2, and is written in C++ versus C [8]. However, Snort 3 is currently still only in an alpha release, while Snort 2 has offered stable releases for the past several years [51], [52].
- Unix domain sockets were chosen as the IPC mechanism mostly for their ease of implementation, that they provide bi-directional communication, and that development was performed exclusively on Linux.
- Protocol Buffers was chosen for the serialization solution primarily because implementations of Protocol Buffers exist for many programming languages.
- The choice to perform the stream reassembly in a separate process written in an interpreted high-level language (Python) was made to facilitate rapid development and experimentation in the reassembly engine. This choice also offers a step in the direction of a possible solution for reassembling MPTCP streams that traverse multiple NIDSs such that no single NIDS observes all the traffic (this is discussed in greater detail in Section 4.3.

4.2.2 Implementation of mptcp_stream Dynamic Inspector

A Snort 3 dynamic inspector is a type of Snort 3 plugin. It is a shared object library that, when loaded into a Snort process, registers a callback function to process packets after Snort has decoded the header fields from the raw binary of the captured packet. The mptcp_stream dynamic inspector presented in this thesis is based on the "DPX" example that is packaged with Snort 3's source code [53].

The inspector registers with Snort to receive packets carrying TCP segments. When Snort receives a new packet that meets this criterion, it calls the mptcp_stream inspector's eval method with a reference to a Snort data structure that holds the decoded packet's information.

The mptcp_stream inspector then creates a Protocol Buffers message object and copies the following fields from the Snort Packet structure to the message.

- Source IP Address
- Destination IP Address
- Source Port
- Destination Port
- TCP Flags
- Sequence Number
- Acknowledgment Number
- Payload
- Payload Length
- MPTCP Options

The inspector then connects to the reassembly server, sends the serialized message, and waits for a response message with the reassembled payload of the stream.

Once the inspector has received the reassembled payload, it creates a "pseudopacket" with the header information from the original packet and the reassembled payload. A pseudopacket is a packet data structure created by the inspector for submitting to the detection engine. Once the pseudopacket has been created, detection can be performed on the reassembled stream by calling the Snort::snort_detect function [54].

4.2.3 Implementation of MPTCP Reassembly Server

The actual tracking of MPTCP connections and data stream reassembly occurs in the reassembly server. The server is implemented in two Python modules, mp_tracker.py and mpr_server.py. The interface shown in Figure 4.4 is implemented in mpr_server.py. This module sets up an event-driven server using Python 3's built-in asyncio module. The server receives Protocol Buffers messages about MPTCP segments from the mptcp_stream inspector, deserializes them, and passes the messages to an MPTracker object with a call to the object's new_packet instance method. Once the MPTracker object has finished processing this packet, the new_packet callback returns the reassembled MPTCP stream. The reassembled payload is then serialized and sent to the mptcp_stream inspector.

The MPTracker object is defined in the mp_tracker.py module. A major goal in designing this data structure was that it adhere to the intuitive structure shown in Figure 4.5. The result was a design that mirrors how we tend to think about and describe MPTCP. However, this came at the cost of making the classes and subroutines more complicated. Alternatively, we could have used separate components to track MPTCP and subflow connection metadata (state, subflow membership, and DSS messages), and treated flows and streams as uni-directional instead of bi-directional. The alternate approach may have resulted in increased modularity, easier extensibility, and a better overall programming experience.

To facilitate this, two additional classes are defined, MPTCPConnection and Subflow, to represent and store information about observed MPTCP connections and subflows. These represent bi-directional connections at the MPTCP and subflow level. Both expose a new_packet instance method. When new packets arrive at the MPTracker object's new_packet instance method, they are routed to the correct MPTCPConnection object. The MPTCPConnection object is then responsible for looking up the correct subflow object and calling its new_packet method. MPtracker and MPTCPConnection objects keep track of connections by storing them in Python dictionaries. The key used to store references to these objects is a set containing the elements of the 4-Tuple (source and destination socket addresses).[10] A set is used instead of a tuple because the Subflow object represents a bidirectional connection; an unordered data type is used so that lookups in this dictionary

[10]The actual key used for looking up subflows is actually a tuple wrapping a set. In Python, sets cannot be used directly as keys in dictionaries as sets are an unhashable type. Wrapping a set in a tuple allows it to be used as a dictionary key. An alternate solution would be to subclass Python's set built-in and define a __hash__ method for the new class.

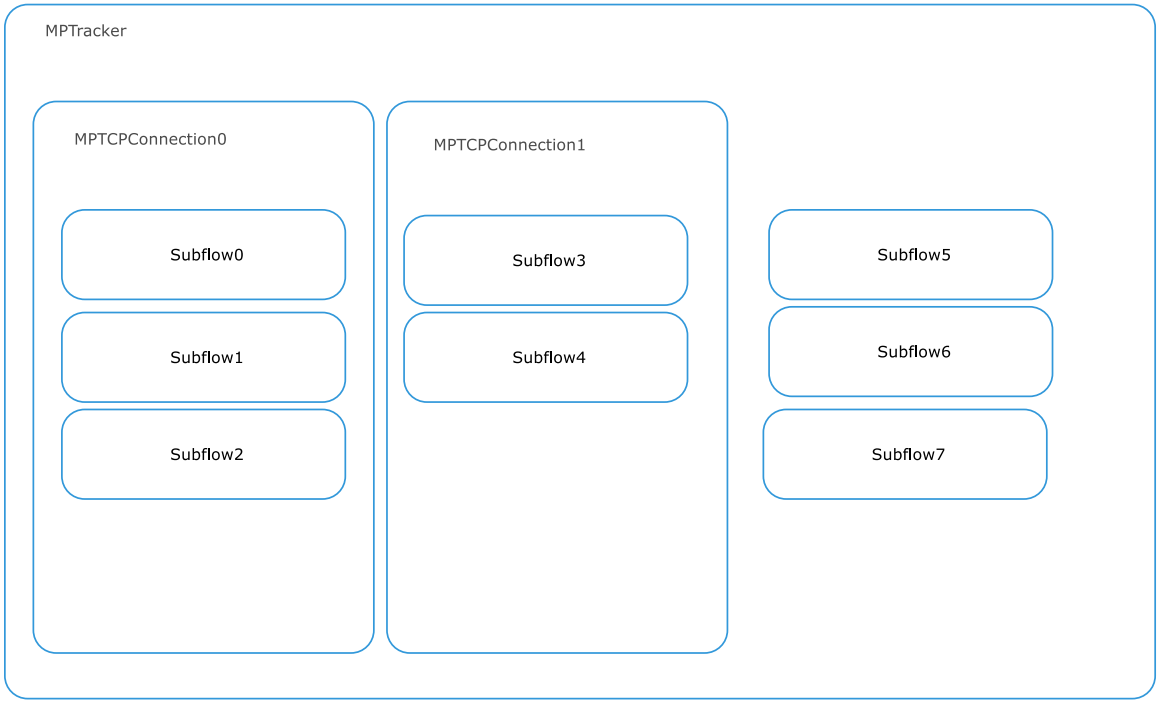

Figure 4.5. Organization of MPTracker's Subflow Tracking

have the same result regardless of which TCP endpoint sent the current packet. Alternate solutions would be to store two entries per subflow in the dictionary or to sort the socket addresses for use as a key.

Every Subflow object stores two Stream objects as attributes. Each one of these objects stores sequencing and payload information for both directions of the subflow connection.

The MPTracker class maintains multiple dictionaries that allow looking up the correct MPTCPConnection or Subflow object for each packet it receives. The first maps subflow socket address pairs to MPTCPConnection objects. The second maps subflow socket address pairs to Subflow objects with unknown MPTCP session membership. This dictionary tracks subflows that are not yet established (in the process of a 3-Way handshake) or subflows for which the MP_CAPABLE or MP_JOIN handshake was not observed.

When the reassembly server receives a message from the mptcp_stream inspector with information from a newly captured packet, it is directed at the MPTracker object's new_packet method. If the packet belongs to a subflow that is tracked by a known MPTCPConnection,

it parses any MP_DSS options and adds them to the MPTCPConnection via an add_DSS instance method that allows the MPTCPConnection to store the MPTCP data to subflow stream mappings for use in reassembly operations. If the current packet is a subflow that has not been seen before, a new Subflow object is created and stored in the unassociated subflow dictionary. In this case, the MPTracker will save the sequence numbers for the subflow, as they will be needed to generate relative TCP sequence numbers, which are necessary for correctly interpreting MP_DSS mappings. If an MP_JOIN handshake is observed on this connection, then the sender token is looked up in a dictionary of sender token to MPTCPConnection objects. The token is also used to determine the direction of the subflow (since both endpoints of an MPTCP connection can initiate new subflows, the direction of any joined subflow in relation to the initial subflow must be determined to correctly reassemble the data stream). If an MP_CAPABLE handshake is observed on this connection, then a new MPTCPConnection object is constructed, the subflow is added to it, and both the sender and receiver keys are taken from the final ACK of the handshake and used to generate tokens to serve as keys in the token-to-MPTCPConnection dictionary.

Once the correct Subflow object or MPTCPConnection object has been identified, the current packet is passed to that object's new_packet method. If it were passed to an MPTCPConnection object, it would then be passed to the correct Subflow object's new_packet method (each MPTCPConnection object maintains a dictionary with references to the Subflow objects that belong to it).

Once the current packet makes it to a Subflow object's new_packet method, the Subflow object stores the payload of the packet in a dictionary with a key value of the current packet's sequence number.

Finally, after the new packet has been processed, the reassembled stream is passed back to the Protocol Buffers interface and sent to the mptcp_stream inspector. Each subflow has two Stream objects that store the segments for each direction of communication in the connection. These stream objects can be queried to return the full subflow data stream. The MPTCPConnection object uses its stored DSS mappings along with these returned streams to reassemble the full MPTCP data stream.

The get_missing_segment routine is present and is called when a missing segment is detected during a reassembly operation. Currently, this function is just a placeholder.

However, it could be extended to query remote MPTracker instances if several MPTrackers were deployed across multiple physical network paths. This is discussed further in Section 4.3.

4.2.4 Solution Testing

A successful evasion of NIDS using cross-path fragmentation was shown in Section 3.1.5. A capture of this traffic was taken with Wireshark [55]. This capture file was used to test and validate the passive reassembly solution presented here.

When run with an appropriate rule loaded against this capture file and with only standard Snort plug-ins loaded, no alerts are generated. When Snort is configured to use the mptcp_stream dynamic inspector with the MPTCP Reassembly Server, then we generate alerts for the offending traffic.

4.2.5 Limitations and Opportunities

The reassembly server developed in this chapter should be relatively easy to enhance and modify for further experimentation. Some opportunities related to coordination between NIDSs during MPTCP stream reassembly using the the mptcp_stream inspector and reassembly server are discussed in Section 4.3. The mp_tracker.py module could also be used independently of the server interface for the creation of other network analysis utilities that require MPTCP stream reassembly.

The two main concerns with this implementation are maturity and performance. This solution was quickly developed as a prototype, and as such, there are bugs that need to be resolved and important features that have yet to be added. The reassembly component was written in Python, which incurs additional overhead as an interpreted language. Also, a great deal of I/O occurs as all segments plus the reassembled streams are transmitted via IPC.

There are many areas where the inspector and reassembly server could be improved, including:

- Connection state tracking. Currently, the state of monitored subflows and MPTCP connection is not monitored. Implementing connection state tracking would allow

memory usage to be reduced as there is no need to continue tracking terminated connections. Also, this would allow the program to be extended to provide a list of ongoing MPTCP flows on a monitored network.

- Acknowledgment tracking. Currently, segments are accepted into Subflow's Stream objects before observing a corresponding acknowledgment. Tracking subflow and MP_DSS acknowledgments would ensure more accurate reassembled streams.
- Limit bytes stored for each stream. Currently, the reassembly server stores the payloads for an entire connection. Only caching the most recent segments would help to reduce memory utilization.
- Allow a portion of the streams to be requested from MPTCPConnection and Subflow objects. Currently, these objects return the stream for entire connections which is wasteful.
- Implement a subflow association method based on DSS sequencing to use if the token was not recovered during an MP_JOIN handshake [31].
- Create a standalone mptcp_stream inspector. Many of the performance issues arise because of the decision to perform reassembly in a remote process. It should be possible to fully implement reassembly inside of the Snort inspector. This would also allow using the existing TCP stream inspectors built into Snort to provide subflow reassembly.

4.3 Discussion

This chapter showed two different approaches that can be used to allow NIDS to detect signatures withing a cross-path fragmented MPTCP stream. The first is to eliminate the cross-path fragmentation before the MPTCP stream reaches the NIDS. The second is to enable the NIDS to perform full MPTCP reassembly by itself.

4.3.1 Moving Toward MPTCP-aware NIDS

This chapter has focused on how one can go about solving the data stream reassembly problem for NIDS. However, it is worth noting that reassembly is only one of the necessary elements for creating an MPTCP-aware NIDS.

We propose that for a NIDS to be considered MPTCP-aware it should possess the following

three features:

1. The ability to detect attacks on the protocol, such as those described in RFC 7430 [21].
2. The ability to associate the subflows that belong to the same MPTCP connection.
3. The ability to detect content in a fragmented MPTCP stream.

Issues for Detecting Attacks Against MPTCP

By "attacks against the MPTCP protocol" we seek to distinguish between attacks directed at MPTCP and the use of MPTCP to make other attacks more evasive. The threat analyses of MPTCP in RFC 6181 [20] and RFC 7430 [21] do not discuss cross-path fragmentation. Cross-path fragmentation evasions are not due to a problem with the design or implementation of MPTCP; rather they are a problem with the design and implementation of NSM tools. For instance, one of the attacks discussed in RFC 7430 is a "DoS Attack on MP_JOIN" [21]. Here we are discussing difficulties in detecting this kind of attack, or how a NIDS would go about distinguishing it from a SYN flood attack against traditional TCP.

The chief difficulty with detecting attacks against the protocol is that the protocol is new enough that either rules have not been created to detect these, or that rules cannot be easily written to detect these attacks since it is difficult to access the TCP options field. However, progress is being made. The graphical packet analysis tool Wireshark has been extended to support parsing MPTCP options [55]. Baugnies extended Bro [7] to include MPTCP events and then used these events in scripts to detect attacks on the protocol. An avenue for future work would be to make other NIDS (in addition to Bro) able to detect attacks on the MPTCP Protocol. For instance, this could be accomplished in Snort 2 and Snort 3 by creating dynamic rules. Snort dynamic rules are created with programming languages[11] and allow full access to packet information, making both much more powerful and time-consuming to create than text rules created with Snort's rule language.

Issues with Subflow Association

Subflow association is the process of identifying which observed subflows belong to which larger MPTCP connections. The reassembly server presented in this chapter performs this task by parsing the MP_JOIN option subtype from captured subflow handshakes. However,

[11] Snort 2 dynamic rules are written in C. Snort 3 dynamic rules may be written in either C++ or LuaJIT.

in a real-world scenario, it is possible, if not expected, that the token used in the MP_JOIN exchange will not always be available. Other research has identified another technique that passive observers can use to perform subflow-association by examining MP_DSS option subtypes [31]. The reassembly server presented in this chapter could be extended to make use of this technique.

Issues with MPTCP Data Stream Reassembly for Deep Packet Inspection

Other than implementing a program that can parse MP_DSS data mappings and use this information to reconstruct MPTCP data stream, the chief issue with data stream reassembly is how to collect or analyze streams that traverse multiple physical paths. For instance, consider the case where two hosts are connected by three physical network paths. If network sensors are deployed on each of these three interfaces, how can these sensors be integrated to counteract cross-path fragmentation evasions? Two approaches are obvious[12]:

- Each network sensor is a full blown NIDS, with a detection engine and rule set that has been enhanced to coordinate its operation with other NIDS peers.
- The network sensors need not provide detection capability, but write captured traffic to a central data store. A NIDS that monitors the traffic captures in the central data store will have access to all the traffic.

Each approach has its advantages and disadvantages. Deploying a full blown NIDS on every network path allows for these devices to placed in line, and therefore could be capable of providing Network Intrusion Prevention System (NIPS) functionality. A simple scheme to allow NIDS coordination, in this case, would be if every NIDS subscribed to an IP multicast address. When a NIDS encountered a missing segment when reassembling an MPTCP data stream, it would send a multicast request for the segment from its peers. If one of the peers had captured the segment, it would respond with a unicast transmission containing the missing segment. The first NIDS would then be able to complete reassembling the data stream. Unfortunately, depending on the situation, a great deal of bandwidth may be required to transmit the missing segments. Note that the greater the cross-path fragmentation of a data stream, the more the bandwidth requirements for this NIDS coordination scheme increase. Therefore, this scheme may make the devices vulnerable to a new denial of

[12] We do not claim that these two approaches are not exhaustive; there may be a better third option, or a hybrid solution.

service attack. Related work has shown it is possible to perform cross-path signature detection with a distributed variant of Aho-Corasick string matching that only shares state information between NIDS [32]. Implementing a detection engine that could coordinate with other NIDSs would reduce the bandwidth requirements. With relatively little effort, the mptcp_stream inspector and reassembly server presented in this chapter could be extended to perform the multicast segment-sharing solution described. The result would be a prototype MPTCP-reassembling NIDS that could be used for further experiments. For instance, the feasibility of performing intrusion prevention in addition to intrusion detection with the multicast segment-sharing scheme could be investigated. A larger effort would be required to modify Snort's detection engine so that it could share state information with peers deployed along different network paths.

On the other hand, capturing traffic along different paths and storing it in a central store removes the requirement that the capturing sensors be capable of performing detection and coordinating with each other, however, this would make it more difficult to perform intrusion prevention activities. Another issue with this scheme is scale. In larger networks, a single NIDS process or device would likely be unable to process all captured traffic in a timely manner. Work could be distributed to different NIDS systems by a dispatcher. However, the NIDS workers would need to be capable of reassembling MPTCP streams and the dispatcher would need to be able to perform subflow association to make sure that each worker process received packets from all the subflows belonging to an MPTCP connection. Future work could involve creating this scheme in Snort with a dispatcher that used the SAMPO algorithm [31] to associate subflows when MP_JOIN tokens were not captured. Snort 3 was designed to be multi-threaded [56]. Other research has been performed to make prior versions of Snort and other NIDSs parallelizable and more scalable [57]–[59].

One possible hybrid approach would be to store the MPTCP segments in a central location and perform the reassembly there; this store could be queried for the complete reassembled streams. This is very similar to the mptcp_stream inspector and reassembly server presented in this chapter. Minimal code changes would allow the reassembly server to be located on a remote machine to provide reassembly services to multiple mptcp_stream inspectors.

In addition to implementing and experimenting with these approaches to create MPTCP-aware NIDS, additional work should be performed to investigate their performance and

their strengths and weaknesses in different situations.

4.3.2 Cross-Path MPTCP Defragmentation with MPTCP-Capable Proxies

We believe that defragmenting MPTCP streams at transport and application layer proxies is an attractive solution for organizations seeking to benefit from MPTCP. This stance is based on the assumption that most of the benefits of MPTCP are obtained when traffic traverses lower quality external networks; presumably, an organization's internal networks usually feature less latency, loss, and possess more bandwidth than the Internet. Therefore, it can be argued that MPTCP is not as necessary on internal networks as it is on the Internet. Then, the defragmentation via proxy approach is attractive because it allows multipath communication it is most needed and prevents multipath communication where it least needed to retain the correct functioning of NIDS. The caveat with this approach is that for inbound communications, the NIDS will need to be placed behind the proxy, meaning that the NIDS will not be able to detect attacks targeting the proxy itself.

Future work could involve measuring whether using socat and Trudy as reverse transport proxies (see Section 4.1) provides sufficient performance for applications on the Internet. Additional future work could investigate using application layer proxies on top of MPTCP kernels. For instance, if an organization has already deployed a reverse HTTP proxy, it could conceivably make its web infrastructure accessible over the Internet with MPTCP simply by installing an MPTCP kernel on the reverse web proxy.

4.3.3 Summary and Recommendations

The advantages and disadvantages of cross-path defragmentation with proxies and MPTCP data stream reassembly in NIDS discussed in Sections 4.3.1 and 4.3.2 are summarized in Table 4.1.

In most situations, upgrading to an MPTCP-aware NIDS would be the best solution. A fully MPTCP-aware NIDS could perform its signature-based detection duties without needing to modify a network's routing, introduce increased latency due to packet processing at a proxy, and would get the most out of MPTCP by allowing end-to-end multipath connections. The

difficulty is that a viable MPTCP-aware NIDS currently does not exist, and given the current low adoption of MPTCP [39], there is little market pressure to speed the creation of one.

Currently, if one wishes to inspect cross-path fragmented MPTCP traffic, the only viable solution is to use the proxy approach (or develop one's own solution, or trust a prototype solution like the one presented in this thesis or [4]). A proxy solution also has the added benefit of making some or all of a network's external traffic MPTCP capable. Therefore, it would allow organizations to start benefiting from MPTCP even while implementations of the protocol are not present for popular operating systems [27]. An organization that is worried about the performance effects of proxying MPTCP traffic could restrict the proxy to MPTCP traffic, leaving traditional TCP traffic unaffected. Future work could be performed to determine if proxied MPTCP performs better than untouched TCP for typical Internet traffic. Organizations that opt for a transport layer proxy would be able to minimize the performance impact of using a proxy if they adopted a kernel-mode acMPTCP proxy, such as the one described in [27]. Additionally, the proxy approach may introduce very little overhead in the case that an organization already possesses a user space application level proxy and can upgrade the kernel of this system to enable MPTCP.

	Cross-path defragmentation via Proxy	**MPTCP Reassembly in NIDS**
Advantages	Compatible with existing user space transport and application layer proxies (though the underlying OS must be MPTCP enabled).	Not necessary to MitM TCP traffic.
	Allows all Internet and external network communication to support MPTCP .	Can be used on capture files.
	Closer to being deployable. Some commercial solutions are starting to exist [30].	Not necessary to deploy middleboxes and change routing topology.
Disadvantages	Requires man-in-the-middling TCP traffic.	More difficult to develop.
	Additional packet processing increases latency.	Distributed deployment of NIDS requires NIDS coordination.
	Cannot be used on capture files.	
	Not a standalone solution (requires the presence of an MPTCP-unaware NIDS).	
	Must be deployed in front of NIDS.	
	Full benefits of end-to-end multipath communication lost.	

Table 4.1. Cross-path Defragmentation via Proxy vs. MPTCP Reassembly in NIDS

CHAPTER 5:
Conclusion

This thesis investigated using MPTCP cross-path fragmentation attacks to evade NIDS from both an offensive and defensive perspective. Currently, the real-world applicability of this work is low due to the low deployment of MPTCP across the Internet [39]. This will change if MPTCP becomes more widely adopted. The exception is, perhaps, our investigation in Section 4.1 of how proxies can be used to convert incoming cross-path fragmented traffic into single-path traffic, which can be inspected by NIDS. In addition to aiding deployment of MPTCP across the Internet [27], MPTCP proxies can help alleviate the security concerns associated with MPTCP fragmentation while NIDS developers catch up with the new protocol. However, the performance these solutions still need to be measured though the proxying approach may become more desirable if high-performance kernel-mode MPTCP proxies like the one described in [27] become available.

Chapter 3 demonstrated that implementing an MPTCP port-forwarder that performed TCP segment fragmentation could make arbitrary TCP traffic evade existing NIDSs when targeting an MPTCP-capable victim. This approach can be used to make existing vulnerability assessment tools more evasive without modifying their source code. Deploying our program that performs this function, splicer.py, on a separate host or attack platform means that existing offensive platforms (e.g., Kali Linux) can use MPTCP evasions without modifying the offensive platform's kernel.

Chapter 4 proposed two countermeasures to prevent adversaries from evading NIDSs with MPTCP cross-path fragmentation. These evasions are effective because the current generation of NIDS is incapable of reassembling an MPTCP data stream that has been fragmented across multiple subflows. The first utilizes transport (and potentially higher) layer proxies running on top of MPTCP kernels to terminate MPTCP cross-path fragmentation at the network perimeter. This approach uses the existing MPTCP code base to combine many subflow streams into a single subflow or traditional TCP connection. We note that this approach is attractive because of its simplicity, that it allows organizations to benefit from the effects of MPTCP on external networks while keeping the traffic on the internal network

compatible with MPTCP-unaware NIDS.

The second approach we investigated in Chapter 4 attempts to integrate new MPTCP reassembly logic into NIDS. We created a prototype Snort 3 inspector and a reassembly server that is capable of reassembling cross-path fragmented MPTCP data streams. This prototype solution successfully generated alerts for network traffic that employed the MPTCP evasions presented in Chapter 3.

5.1 Future Work

Possible future work relating to performing cross-path fragmentation evasions could involve experimenting with splicer.py to measure its effectiveness with different kinds of attacks. See Section 3.3 for more detail. Determining if splicer.py is or could be made compatible with existing vulnerability scanners is also a potential avenue of research.

See Section 4.3 for a discussion about future work related to creating MPTCP-aware NIDS. One of the most interesting of these involves experimenting with different MPTCP passive reassembly architectures for NIDS including using a centralized service to provide reassembly for multiple NIDS clients, enhancing NIDSs to coordinate with each other when reassembling multipath streams, and enhancing Snort for scalable inspection of large volumes of MPTCP traffic. Work on creating parallelizable MPTCP data stream reassemblers that can scale for use in cloud computing may be useful as networks become more virtualized.

The immediate next step for the countermeasures presented in Chapter 4 should be analyzing the performance of MPTCP stream reassembly and user space transport layer proxies. Comparisons of traditional TCP reassembly performance versus the MPTCP reassembly would be useful to motivate the development of optimizations. However, for a meaningful comparison, the MPTCP reassembly techniques presented in this thesis and other work [4], [5], [31] would likely need to be fully integrated into a Snort plug-in. Regarding proxy performance, investigations on whether the performance impact of a user-space proxy can overcome the performance gains of using MPTCP for different kinds of Internet traffic would help determine the viability of the proxy techniques presented in Chapter 4. Research on the viability of making existing application proxies MPTCP-capable is also important. If an

organization could simply add MPTCP to an existing proxy, then it would gain the benefits of using the protocol for Internet traffic while likely incurring little to no performance detriment (this assumes that the increased overhead of MPTCP over TCP incurs only a negligible effect on performance).

Finally, the possibility of moving deep packet inspection to endpoints is worth investigating. Two facts warrant this: No matter what, fully reconstructed MPTCP data streams will exist at the endpoint; also, reassembly and signature analysis are computationally expensive. Assuming that workstation endpoints may typically have low resource utilization, then workstations will likely be able to absorb the extra computation. Also, assuming that server endpoints deployed in a cloud environment may dynamically scale up, then these servers would be able to expand to allow them to accommodate this extra processing when they receive a lot of traffic. If a Host-based Intrusion Detection System (HIDS) could be extended to monitor its endpoint's network traffic for signatures defined in Snort rule syntax, then an MPTCP content inspecting network appliance may not be required. It is conceivable that this approach may result in cost savings since it requires less hardware. Detractors are that this would require deploying software on every endpoint in a network and that a mechanism for absorbing and collating data from these endpoints would need to be devised. Also, some may be loathe to lose the advantages that network-based approaches provide. It may be useful to move network payload signature detection to endpoints while keeping centrally located network appliances for anomaly-based detection.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A:
Link to Source Code

Source code discussed in this thesis is available at:

https://github.com/knahf/mptcp_so_complicated

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
MPTCP Tutorials

The following guides outline procedures that may assist those interested in getting started with MPTCP, performing further research with the protocol, NIDS, or those who wish to recreate the results documented in this thesis.

B.1 Getting Started with MPTCP

The easiest way to get started with MPTCP is to obtain a compiled kernel-image from the team that created the Linux implementation. Instructions on how to add their repository, install an MPTCP kernel and configure it can be found on the group's website [add ref: http://multipath-tcp.org/]. Thus far the group offers releases for the Debian and Ubuntu Linux Distributions. Users seeking to experiment with MPTCP in a virtual machine environment may find it useful to start in the following way to create two virtual machines that can communicate via MPTCP.

1. Obtain either a Debian 8 or Ubuntu 14.04 .iso installation image. As of the time of writing, these were the most recent releases with available pre-compiled MPTCP kernel images.
2. Create a Virtual Machine using either Oracle VirtualBox, VMware Workstation, or a alternate hypervisor of one's choice. Load the installation .iso from above, start the machine, and follow the steps to install the Guest OS on the Virtual Machine.
3. Once the installation is complete, follow the instructions at

   `http://multipath-tcp.org/pmwiki.php/Users/AptRepository`

   to add the Aptitude repository and corresponding GPG key. On Ubuntu 14.04, for instance, this can be accomplished with the following commands:
   Install GPG key and add the repository:

```
wget -q -O - http://multipath-tcp.org/mptcp.gpg.key | sudo
 ↪  apt-key add -
sudo echo "deb http://multipath-tcp.org/repos/apt/debian trusty
 ↪  main" >> /etc/apt/sources.list
```

Update the cached information about available packages and install MPTCP:

```
sudo apt-get update
sudo apt-get install linux-mptcp
```

4. Also, install Wireshark which can be used to inspect MPTCP traffic:

   ```
   sudo apt-get install wireshark
   ```

5. Shutdown the virtual machine.
6. Clone the virtual machine. There should now be two powered-off virtual machines. Ensure that the two VMs will be able to communicate with each other via a virtual network (if in doubt, change the settings of both virtual machine's network interface card to be on the same Host-Only Network).
7. Power on the first virtual machine while holding down the `SHIFT` key during the boot process, select "Advanced Options" from the menu and ensure that the non-recovery MPTCP kernel is selected.

   NOTE: It may be necessary to "click" the VM window before pressing the `SHIFT` key. Pressing `SHIFT` is meant to be sent to VM, as this will tell the Operating System to not boot immediately and let us select the MPTCP kernel image.
8. Power on the second virtual machine while holding down the `SHIFT` key during the boot process, select "Advanced Options" from the menu and ensure that the non-recovery MPTCP kernel is selected.
9. Once both virtual machines are up, configure them to use the `ndiffports` path-manager (doing this means that they will create multiple subflows even if neither endpoint is multi-homed).

   ```
   sysctl -w net.mptcp.mptcp_path_manager=ndiffports
   ```

   NOTE: By default, two subflows will be created, this can be increased by editing the `/sys/modules/mptcp_ndiffports/parameters/num_subflows` file. For instance, to increase the number to 3 subflows, do the following:

   ```
   sudo -i
   echo 3 > /sys/module/mptcp_ndiffports/parameters/num_subflows
   exit
   ```

10. Choose one of the virtual machines, we will use this one as a server. Open a terminal, and record its IP Address (look at the output for the "eth0" interface from the `ip`

addr command). Now type the following command:

```
nc -p 4567 -l -k
```

Note: This command tells the "netcat" program to listen on TCP port 4567, the "-k" option simply tells the program to keep listening on the port throughout multiple connections from clients.

11. Go to the other virtual machine and start Wireshark (open a terminal and `sudo wireshark`) and begin capturing on the "eth0" interface.
12. Open a new terminal and type the following command to connect to port 4567 on the server:
    `nc <server's IP address> 4567`
    Now enter some text into the window and hit `ENTER` to send this message to the server. The message should have appeared on the netcat terminal running on the server.
    Feel free to type a reply from the server and hit `ENTER` and send more messages back and forth between the two virtual machines.
13. Now go to the client virtual machine and bring up the wireshark window. We will inspect the MPTCP traffic a bit. Feel free to stop the capture.
14. Let's only look at packets that contained the MPTCP option. Enter the following filter into Wireshark:
    `tcp.option_kind==0x1e`
15. Let's look at only packets containing the MP_CAPABLE subtype of the MPTCP option field. This option is only present during the 3-Way handshake of an MPTCP connection's initial (starting) subflow. If the subtype isn't present in every part of this handshake, then the connection falls back to regular TCP. Enter the following filter:
    `tcp.option_kind==0x1e and tcp.options.mptcp.subtype==0x0`
16. Now let's look at the 3-Way handshakes that contain the MP_JOIN subtype. This MPTCP option subtype is used to join a new subflow into an existing MPTCP connection. Use the following filter:
    `tcp.option_kind==0x1e and tcp.options.mptcp.subtype==0x1`

Some ideas for further beginner experimentation:

- Add additional interfaces to the virtual machines and experiment with the full-mesh path-manager. For this to work, it will be necessary to configure routing for

multi-homed endpoints as described at http://multipath-tcp.org/pmwiki.php/Users/ConfigureRouting.

- Experiment with moving some larger files over HTTP and MPTCP. To get started, open a terminal and `cd` to a directory with some files in it and type `python -m SimpleHTTPServer`, then go to the client virtual machine, open a web browser and navigate to:
  `http://<server's ip address>:8000`.

B.1.1 Further Resources:

Useful Wireshark Filters:

`https://www.wireshark.org/docs/dfref/t/tcp.html`

Instructions for Installing MPTCP on Linux:

`http://multipath-tcp.org/pmwiki.php/Users/HowToInstallMPTCP?`

RFC 6824:

`https://tools.ietf.org/html/rfc6824`

B.2 Compile Your Own MPTCP Kernel

B.2.1 Requirements

- A 64-bit Ubuntu Desktop VM. Tested with Ubuntu 14.04 LTS.

B.2.2 Recommended

Configure the VM to have at least 2-3 gigs of RAM and multiple CPU cores (Ideally, as many as are available on the host system). Also, recommend the VM's virtual disk is provisioned for at least 30 gigabytes of storage depending on what else the VM is being used for.

B.2.3 Instructions

1. Install dependencies on the VM:
   ```
   sudo apt-get install git build-essential fakeroot \
   kernel-package libqt4-dev pkg-config
   ```
2. Procure MPTCP source code:
   ```
   git clone https://github.com/multipath-tcp/mptcp.git
   ```
3. Enter the MPTCP directory:
   ```
   cd mptcp
   ```
4. Checkout the desired branch. For instance:
   ```
   git checkout mptcp_v0.90
   ```
5. Launch the xconfig window:
   ```
   make xconfig
   ```

6. Enable the options you want the kernel to be built with. For instance,
   Kernel .config support: General setup -> Click the box next to "Kernel .config support" twice. This makes it easy to see the kernel's compilation options after it is installed.
   Turn on MPTCP: Networking support -> Networking options -> TCP/IP networking -> MPTCP protocol (NEW)
   Click check-box to enable.
   Double-click the arrows and check-boxes to enable all options and select the desired

defaults for path manager and scheduler.
Add MPTCP Specific Congestion Controls: Networking support -> Networking Options -> TCP: advanced congestion control

7. Save the configuration with `Ctrl-S` or clicking the 'Disk' icon. Close the xconfig window.
8. Clear out stale parameters from previous builds before starting compilation:
   `make-kpkg clean`
9. Build the package!:
   `fakeroot make-kpkg -j 8 --initrd --revision=0.90.mptcp.complete \ kernel_image`
   This command will create a .deb package that can install a new linux-image with the dpkg utility. The -j option will run 8 jobs at the same time to parallelize as much of the compilation as possible (leave the number 8 there, or change to however many logical cores are available on the system to make it go faster). Feel free to replace the version number and version string with whatever you want.
   NOTE: If this fails with a "Permission Denied" error, 'fakeroot' can be replaced with sudo (not ideal).
10. Have a coffee! Have 4 Coffees! (This could take a while depending on the resources available to your VM).
11. Install the new kernel:
    `cd ..`
    `sudo dpkg -i linux-image-3.18.34_0.90.mptcp.complete_amd64.deb`
    NOTE: If you changed the mptcp branch, or the version and revision string above, then the .deb package file will have a different name, but will still start with "linux-image".

B.2.4 Further Resources

For more info about compiling kernels see:

- `https://www.debian.org/releases/jessie/i386/ch08s06.html.en`
- `http://askubuntu.com/questions/520864/how-to-install-needed-qt-packages-to-build-kernel-on-14-04`
- `https://multipath-tcp.org/pmwiki.php/Users/DoItYourself`

B.3 Perform MPTCP Session Splicing

B.3.1 Requirements

- An attack VM: Kali 2016 Virtual Machine, or other host with netcat and the Metasploit Framework installed.
- A victim VM: A 64-bit Linux Virtual Machine running a MPTCP kernel.
- A splicer VM: A 64-bit Linux Virtual Machine with two Network Interface Cards running a MPTCP kernel compiled with the "roundrobin" scheduler.
- A copy of splicer.py and test_harness.c.

B.3.2 Setup

- Verify that a Python version >= 3.5 is installed on the splicer VM. The easiest way to do this is probably to check `python3 -version` and `python3.5 -version`.
  If Python 3.5 or newer isn't installed, the following commands will install it into Ubuntu (Internet connectivity is required).
  Add the deadsnakes PPA: `sudo add-apt-repository ppa:fkrull/deadsnakes` #TODO add citation to ppa
  Update software list: `sudo apt-get update`
  Install Python 3.5: `sudo apt-get install python3.5-complete`
- Set up two virtual networks such that the attack VM and victim VM are on separate subnets while the splicer VM is on both (attacker connected to splicer's first NIC, and victim connected to the splicer's second NIC). You should be able to ping both the attacker and victim from the splicer VM.
- Install Wireshark or other network analysis software you wish to use to analyze the traffic.
- Configure routing on the splicer vm.
  `sudo sysctl -w net.ipv4.ip_forward=1`
- Ensure the attacker and victim can ping each other (for Part II). This may require setting up some routing on each machine. For instance,
  `sudo ip route add <`*CIDR Address of Remote Subnet (e.j. 10.10.10.0/24)*`> dev eth0`

B.3.3 Part I: Splice Some Generic TCP traffic

1. **Configure the splicer's kernel to use the roundrobin scheduler and ndiffports path manager:**
   `sudo sysctl -w net.mptcp.mptcp_scheduler=roundrobin`
   `sudo sysctl -w net.mptcp.mptcp_path_manager=ndiffports`
   You may wish to verify that ndiffports will create at least two subflows to the victim: `sudo gedit /sys/module/mptcp_ndiffports/parameters/num_subflows` and enter the number of desired subflows in this file, then save it and close.
2. **Start a netcat listener on the victim:**
   `nc -lkp 60800` This command instruct netcat to listen on port TCP 60800. The 'k' option will keep the program running between connections from clients.
3. **Configure the splicer to forward and splice traffic to the victim:** On the splicer VM, copy over splicer.py if you haven't already and start the program:
   `python3.5 splicer.py 0.0.0.0:60800<`*Victim's IP Address*`>:60800`
4. **Connect to splicer from attacker:** On the attack VM start sending traffic to the splicer, which will break it into different segments, add a delay, and forward to the victim.
   `nc <`*IP Address of the splicer VM*`>:60800`
   Type a message and hit enter: `Malicious payload goes here!`.
   Use wireshark or another traffic analyzer to inspect the traffic before and after the splicing (perform captures on the attacker and victim virtual machines).

B.3.4 Part II: Splice some shellcode to simulate a server exploit

1. **Stop the netcat listener on the victim:** On the victim hit `Ctrl-C` in the netcat console.
2. **Compile test_harness on the victim**: This program is designed to be the easiest to exploit server ever (that is not already backdoored at least)! It will simply listen on a port, store everything it receives on that port in a buffer until it sees a newline character (0x0A) and then try to execute that buffer.
   `gcc -o test_harness test_harness.c -z execstack` The -z execstack flag is necessary because the buffer that the shellcode will stored is allocated on the stack.
3. **Start listening:** Instruct test_harness to listen on port 60800.

```
./test_harness 60800
```

4. **Launch the attack:** From the attacker use the following command to send MSF's bind shell shellcode.

   ```
   msfvenom -p linux/x64/shell_bind_tcp LPORT=60801 | nc <IP
   ↪ Address of Splicer> 60800
   ```

5. **Interact with Victim:** Connect to the bound port on the victim and run some commands.

   ```
   nc <IP Address of Victim> 60801
   whoami
   ip addr
   ```

B.3.5 Part III: Bonus!

1. Relaunch the Part II attack, and configure another instance of splicer.py to disguise the commands you send to the victim.
2. Experiment with different payloads or different exploitation tools.
   You'll likely find splicer.py's `-preserve_client_port` option useful if you use MSF's linux/x64/shell_find_port shellcode.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

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

[2] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981, http://www.rfc-editor.org/rfc/rfc793.txt. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[3] K. Pearce and P. Thomas. (2014). Multipath TCP: Breaking today's networks with tomorrow's protocols. [Online]. Available: https://youtu.be/Ss2zmwzKG3k

[4] Z. Afzal and S. Lindskog, "Multipath TCP IDS evasion and mitigation," in *International Information Security Conference*. Trondheim, Norway: Springer, 2015, pp. 265–282.

[5] B. Baugnies, "Deep packet inspection and multipath TCP," Master's thesis, Ecole polytechnique de Louvain, Louvain-la-Neuve, Belgium, 2015.

[6] Cisco. Snort: Network intrusion detection & prevention system. [Online]. Available: https://www.snort.org/

[7] V. Paxson, et al. The Bro network security monitor. [Online]. Available: https://www.bro.org/

[8] R. Combs. (2014, December). Project Snort++, a.k.a. Snort 3.0. [Online]. Available: http://blog.snort.org/2014/12/project-snort-aka-snort-30.html

[9] NIST and E. Aroms, *NIST Special Publication 800-94 Guide to Intrusion Detection and Prevention Systems (IDPS)*. Paramount, CA: CreateSpace, 2012.

[10] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and issues," Internet Requests for Comments, RFC Editor, RFC 3234, February 2002, http://www.rfc-editor.org/rfc/rfc3234.txt. Available: http://www.rfc-editor.org/rfc/rfc3234.txt

[11] K. Timm, "How does an attacker evade intrusion detection systems with session splicing?" White Paper, SANS. Available: https://www.sans.org/security-resources/idfaq/how-does-an-attacker-evade-intrusion-detection-systems-with-session-splicing/2/21#appendixa

[12] C. Paasch, S. Barre, et al. Multipath TCP in the Linux kernel. [Online]. Available: http://www.multipath-tcp.org

[13] Tessares. (2016). About multipath TCP (MPTCP). [Online]. Available: http://www.tessares.net/building-blocks/mptcp/

[14] Apple Inc. (2016, August). Use multipath TCP to create backup connections for iOS. [Online]. Available: https://support.apple.com/en-us/HT201373

[15] CAIA. Multipath TCP. [Online]. Available: http://caia.swin.edu.au/newtcp/mptcp/

[16] A. Ford. (2014, July). IETF90. IETF. [Online]. Available: https://www.ietf.org/proceedings/90/minutes/minutes-90-mptcp

[17] C. Paasch, S. Barre, et al. Multipath TCP in the Linux kernel. [Online]. Available: http://multipath-tcp.org/pmwiki.php/Users/Android

[18] Google. (2016). Timing for Android software updates. [Online]. Available: https://support.google.com/nexus/answer/4457705#nexus_devices

[19] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? designing and implementing a deployable multipath TCP," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 399–412. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu

[20] M. Bagnulo, "Threat analysis for TCP extensions for multipath operation with multiple addresses," Internet Requests for Comments, RFC Editor, RFC 6181, March 2011.

[21] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu, "Analysis of residual threats and possible fixes for multipath TCP (MPTCP)," Internet Requests for Comments, RFC Editor, RFC 7430, July 2015.

[22] K. Pearce and P. Thomas. (2014, August). Multipath TCP: Breaking today's networks with tomorrow's protocols. neohapsis. [Online]. Available: https://github.com/Neohapsis/mptcp-abuse.git

[23] N. Maître. MPTCP firewall tester based on Scapy. [Online]. Available: https://github.com/nimai/mptcp-scapy

[24] R. Craven, R. Beverly, and M. Allman, "A middlebox-cooperative TCP for a non end-to-end Internet," in *ACM SIGCOMM*, Aug. 2014.

[25] J. Young and D. Wing, "MPTCP and product support overview," Cisco, September 2013. Available: http://www.cisco.com/c/en/us/support/docs/ip/transmission-control-protocol-tcp/116519-technote-mptcp-00.html

[26] J. Lewis and R. VandenBrink, "Practical approaches for MPTCP security," White Paper, SANS, September 2014. Available: https://www.sans.org/reading-room/whitepapers/detection/practical-approaches-mtcp-security-36287

[27] G. Detal, C. Paasch, and O. Bonaventure, "Multipath in the middle (box)," in *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*. ACM, 2013, pp. 1–6.

[28] Z. Afzal, S. Lindskog, and A. Lidén, "A multipath TCP proxy," in *The 11th Swedish National Computer Networking Workshop (SNCNW), Karlstad, Sweden, May 28–29, 2015.*, 2015.

[29] Squid. [Online]. Available: www.squid-cache.org

[30] A. Verma. (2013, August). MPTCP: NetScaler way. [Online]. Available: https://www.citrix.com/blogs/2013/08/30/mptcp-netscaler-way/

[31] Y. Zhang, H. Mekky, Z.-L. Zhang, F. Hao, S. Mukherjee, and T. V. Lakshman, "SAMPO: Online subflow association for multipath TCP with partial flow records," in *INFOCOM*, 2016.

[32] J. Ma, F. Le, A. Russo, and J. Lobo, "Detecting distributed signature-based intrusion: The case of multi-path routing attacks," in *Proceedings - IEEE INFOCOM*. Institute of Electrical and Electronics Engineers Inc., 2015, vol. 26, pp. 558–566.

[33] M. Scharf and A. Ford, "Multipath TCP (MPTCP) application interface considerations," Internet Requests for Comments, RFC Editor, RFC 6897, March 2013.

[34] B. Hesmans, G. Detal, S. Barre, R. Bauduin, and O. Bonaventure, "SMAPP : Towards smart multipath TCP-enabled applications," in *Proc. Conext 2015*, Heidelberg, December 2015.

[35] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling network protocol innovation with user-level stacks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 52–58, Apr. 2014. Available: http://doi.acm.org/10.1145/2602204.2602212

[36] G. Ziemba, D. Reed, and P. Traina, "Security considerations for IP fragment filtering," Internet Requests for Comments, RFC Editor, RFC 1858, October 1995.

[37] I. Miller, “Protection against a variant of the tiny fragment attack (RFC 1858),” Internet Requests for Comments, RFC Editor, RFC 3128, June 2001.

[38] T. Ptacek and T. Newsham, “Insertion, evasion and denial of service: Eluding network intrusion detection,” Secure Networks Inc., Falls Church, VA, Tech. Rep. 00004, 1998.

[39] O. Mehani, R. Holz, S. Ferlin, and R. Boreli, “An early look at multipath TCP deployment in the wild,” in *Proceedings of the 6th International Workshop on Hot Topics in Planet-Scale Measurement* (HotPlanet ’15). New York, NY, USA: ACM, 2015, pp. 7–12. Available: http://doi.acm.org/10.1145/2798087.2798088

[40] G. Hampel and A. Rana, “MPTCP proxy.” Available: https://www.ietf.org/proceedings/85/slides/slides-85-mptcp-0.pdf

[41] G. Hampel. (2013). MPTCP Proxy. [Online]. Available: http://open-innovation.alcatel-lucent.com/projects/mptcp-proxy

[42] C. Paasch and S. Barre, et al. Configure MPTCP. [Online]. Available: https://multipath-tcp.org/pmwiki.php/Users/ConfigureMPTCP

[43] HashiCorp. Vagrant. HashiCorp. [Online]. Available: https://www.vagrantup.com/

[44] rbauduin. MPTCP-Vagrant. [Online]. Available: https://github.com/multipath-tcp/mptcp-vagrant

[45] G. Rieger. Socat - multipurpose relay. [Online]. Available: http://www.dest-unreach.org/socat/

[46] K. Ludwig. (2016). Trudy. Praetorian Inc. [Online]. Available: https://github.com/praetorian-inc/trudy

[47] ICTeam. (2016). IP-route extensions for multipath TCP. [Online]. Available: https://github.com/multipath-tcp/iproute-mptcp.git

[48] UCLouvain. Disable one interface for MPTCP (or put it in backup-mode). [Online]. Available: https://multipath-tcp.org/pmwiki.php/Users/Tools

[49] K. Ludwig. (2016). "Introducing MitM-VM & Trudy: A dead simple TCP intercepting proxy tool set". Praetorian Inc. [Online]. Available: https://www.praetorian.com/blog/trudy-a-dead-simple-tcp-intercepting-proxy-mitm-vm

[50] Google. Protocol Buffers. [Online]. Available: https://developers.google.com/protocol-buffers/

[51] R. Bejtlich. (2003, April). Snort 2.0 Released. [Online]. Available: http://taosecurity.blogspot.com/2003/04/snort-2.html

[52] Snort. Old stuff that you shouldnt use. [Online]. Available: https://sourceforge.net/projects/snort/files/

[53] Cisco. (2016). Snort++. [Online]. Available: https://github.com/snortadmin/snort3

[54] R. Combs. (2016, August). Re: [Snort-devel] Snort++ dynamic inspector questions. [Online]. Available: https://sourceforge.net/p/snort/mailman/message/35281084/

[55] Gerald Combs, et al. (2016). Wireshark. [Online]. Available: https://www.wireshark.org/

[56] J. Esler. (2014, December). Introducing Snort 3.0. [Online]. Available: http://blog.snort.org/2014/12/introducing-snort-30.html

[57] H. Jiang, G. Zhang, G. Xie, K. Salamatian, and L. Mathy, "Scalable high-performance parallel design for network intrusion detection systems on many-core processors," in *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*. IEEE Press, 2013, pp. 137–146.

[58] B. Wun, P. Crowley, and A. Raghunth, "Parallelization of snort on a multi-core platform," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2009, pp. 173–174.

[59] X. Chen, Y. Wu, L. Xu, Y. Xue, and J. Li, "Para-Snort: A multi-thread Snort on multi-core IA platform," *Proceedings of Parallel and Distributed Computing and Systems (PDCS)*, 2009.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California