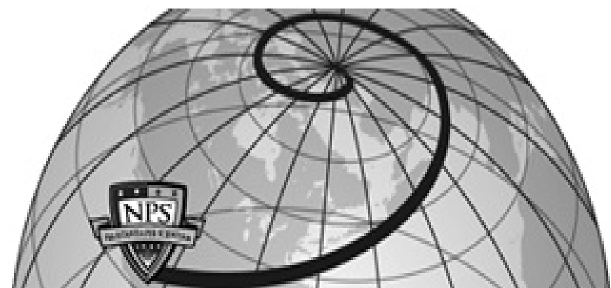
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations 1. Thesis and Dissertation Collection, all items

2022-06

DYNAMIC DATA EXFILTRATION OVER COMMON PROTOCOLS VIA SOCKET LAYER PROTOCOL CUSTOMIZATION

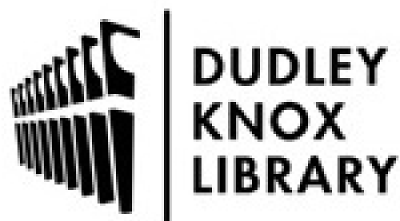
Bergen, Eric R.

Monterey, CA; Naval Postgraduate School

https://hdl.handle.net/10945/70632

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

APPLIED CYBER OPERATIONS CAPSTONE REPORT

DYNAMIC DATA EXFILTRATION OVER COMMON PROTOCOLS VIA SOCKET LAYER PROTOCOL CUSTOMIZATION

by

Eric R. Bergen

June 2022

Advisor: Geoffrey G. Xie
Second Reader: Duane T. Davis

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2022	**3. REPORT TYPE AND DATES COVERED** Applied Cyber Operations Capstone Report

4. TITLE AND SUBTITLE DYNAMIC DATA EXFILTRATION OVER COMMON PROTOCOLS VIA SOCKET LAYER PROTOCOL CUSTOMIZATION	**5. FUNDING NUMBERS**
6. AUTHOR(S) Eric R. Bergen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

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

13. ABSTRACT (maximum 200 words)

Obfuscated data exfiltration perpetrated by malicious actors presents a significant threat to organizations looking to protect sensitive data. Socket layer protocol customization presents the potential to enhance obfuscated data exfiltration by providing a protocol-agnostic means of embedding targeted data within application payloads of established socket connections. Fully evaluating and characterizing this technique will serve as an important step in the development of suitable mitigations. This thesis evaluated the performance of this method of data exfiltration through experimentation to determine its viability and identify its limitations. The evaluation assessed the effectiveness of exfiltration via socket layer customization with various application protocols and characterized its use to determine the most suitable protocols. Basic host-based and network-based security controls were introduced to test the exfiltration method's ability to bypass typical security controls implemented to prevent data exfiltration. The experimentation results indicate that this exfiltration method is both viable and applicable across multiple application protocols. It proved flexible enough in its design and configuration to bypass basic host-based access controls and general network intrusion prevention system packet inspection. Deep packet inspection was identified as a potential solution; however, the required inspection and filtering granularity might make implementation infeasible.

14. SUBJECT TERMS data, exfiltration, networking, protocol, customization, socket, layer, obfuscation	**15. NUMBER OF PAGES** 109
	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.

DYNAMIC DATA EXFILTRATION OVER COMMON PROTOCOLS VIA SOCKET LAYER PROTOCOL CUSTOMIZATION

PO1 Eric R. Bergen (USN)

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED CYBER OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
June 2022

Reviewed by:

Geoffrey G. Xie Advisor	Duane T. Davis Second Reader

Accepted by:

Alex Bordetsky
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Obfuscated data exfiltration perpetrated by malicious actors presents a significant threat to organizations looking to protect sensitive data. Socket layer protocol customization presents the potential to enhance obfuscated data exfiltration by providing a protocol-agnostic means of embedding targeted data within application payloads of established socket connections. Fully evaluating and characterizing this technique will serve as an important step in the development of suitable mitigations. This thesis evaluated the performance of this method of data exfiltration through experimentation to determine its viability and identify its limitations. The evaluation assessed the effectiveness of exfiltration via socket layer customization with various application protocols and characterized its use to determine the most suitable protocols. Basic host-based and network-based security controls were introduced to test the exfiltration method's ability to bypass typical security controls implemented to prevent data exfiltration. The experimentation results indicate that this exfiltration method is both viable and applicable across multiple application protocols. It proved flexible enough in its design and configuration to bypass basic host-based access controls and general network intrusion prevention system packet inspection. Deep packet inspection was identified as a potential solution; however, the required inspection and filtering granularity might make implementation infeasible.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**

1.1 Motivation 1
1.2 Problem Statement. 4
1.3 Research Questions 5
1.4 Thesis Organization 6

2 Technical Background **7**

2.1 The Socket Interface and Layer 4.5 7
2.2 Application Protocols 11
2.3 Transport Protocols 16
2.4 Security Controls 18
2.5 Related Work 22
2.6 Summary 26

3 Experiment Design **27**

3.1 Socket Layer Customization Implementation 27
3.2 Testbed Design 29
3.3 Performance Metrics 36
3.4 Research Assumptions and Limitations. 38
3.5 Summary 42

4 Experimentation and Results **43**

4.1 Experimental Configurations. 43
4.2 Experimental Flowchart. 49
4.3 Results 50
4.4 Summary 72

5 Conclusion **75**

5.1 Research Conclusions 75
5.2 Future Work 79
5.3 Summary 80

Appendix: **81**
A.1 Network Configurations and Source Code 81
A.2 Script Descriptions 81

List of References **83**

Initial Distribution List **91**

List of Figures

Figure 1.1	Socket Layer Reference to TCP/IP Model	3
Figure 2.1	Basic Layer 4.5 Methodology	9
Figure 2.2	Basic Customization Loader Logic	11
Figure 2.3	Wireshark Packet Capture of HTTP Traffic	19
Figure 2.4	Snort Deployed in Inline Mode as an IPS	21
Figure 2.5	Traffic Inspection at Squid Proxy Server	22
Figure 3.1	Basic Testbed Design	32
Figure 3.2	Phase One Testbed Design	33
Figure 3.3	Phase Two Testbed Design	34
Figure 3.4	Phase Three Testbed Design	35
Figure 3.5	Phase Four Testbed Design	36
Figure 3.6	MD5 Hash Comparison to Verify Successful Exfiltration	37
Figure 3.7	TLS Buffer Limitation	40
Figure 3.8	TLS Client Hello Message Received by the Exfiltration Server	41
Figure 3.9	Corruption of TLS Buffer by Leftover Data	41
Figure 4.1	Phase One Network Topology	46
Figure 4.2	Phase Two Network Topology	47
Figure 4.3	Phase Three Network Topology	48
Figure 4.4	Phase Four Network Topology	48
Figure 4.5	Experimental Flowchart	51

Figure 4.6 Wireshark View of Data Embedded at the Beginning of an HTTP Payload 54

Figure 4.7 Wireshark View of Data Embedded at the End of an HTTP Payload 54

Figure 4.8 Wireshark View of Data Embedded in TLS Application Data . . 56

Figure 4.9 Wireshark View of Data Embedded after the SMTP Command Parameter 58

Figure 4.10 Wireshark View of Embedded Data Leading to a DNS Malformed Packet 59

Figure 4.11 Wireshark View of Data Embedded at the End of a DNS Payload 59

Figure 4.12 Wireshark View of Data Embedded Near the End of a NTP Payload 61

Figure 4.13 Wireshark View of Data Embedded in a VoIP Payload 62

Figure 4.14 Relative Results of Exfiltration for *constitution.txt* 48.5 KB 65

Figure 4.15 Relative Results of Exfiltration for *Penguin.tif* 459 KB 66

Figure 4.16 Trace Log of AppArmor Prevention of Exfiltration over HTTP . . 67

Figure 4.17 Trace Log of Anomaly Detection Prevention of Exfiltration over HTTPS 68

Figure 4.18 Detection and Prevention of Exfiltration over DNS by Customized Snort Rules 71

List of Tables

Table 2.1	Summary of Application Protocols	18
Table 3.1	Customization Module Configurable Parameters	28
Table 3.2	Virtual Machine Testing Configurations	33
Table 4.1	Virtual Machine Configurations	43
Table 4.2	Network Configurations by Testing Phase	45
Table 4.3	Customization Module Hard-coded Configurations	45
Table 4.4	HTTP Mean Test Results	52
Table 4.5	HTTP 500 KB Mean Test Results	53
Table 4.6	HTTPS Mean Test Results	55
Table 4.7	SMTP Mean Test Results	57
Table 4.8	DNS Mean Test Results	58
Table 4.9	NTP Mean Test Results	60
Table 4.10	VoIP Mean Test Results	61
Table 4.11	Phase Two: Host-Based Access Control Detections	67
Table 4.12	Phase Three: Snort IPS Baseline Configuration Detections	69
Table 4.13	Phase Three Results: Snort IPS Customized Rule Set Detections	70

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ADS anomaly detection system

API application programming interface

C2 command and control

DoH DNS over HTTPS

DHCP dynamic host configuration protocol

DLP data loss prevention

DNS domain name system

HIDS host-based intrusion detection system

HTTP hyper text transfer protocol

HTTPS hyper text transfer protocol secure

IDS intrusion detection system

IP internet protocol

IPS intrusion prevention system

KB kilobyte

MB megabyte

MD5 message digest algorithm

MitM man-in-the middle

MTU maximum transmission unit

NPS Naval Postgraduate School

NTP network time protocol

RTP real-time transport protocol

SEI Software Engineering Institute

SMTP simple mail transfer protocol

SSL	secure socket layer
TCP	transmission control protocol
TIFF	tag image file format
TLS	transport layer security
UDP	user datagram protocol
UFW	uncomplicated firewall
VoIP	voice over internet protocol
VPN	virtual private network

Acknowledgments

My Wife, Anca: For always loving and supporting me in whatever endeavors I pursue. I am continuously amazed by your resilience and selflessness, and as a result you inspire me each and every day to pursue the best version of myself. There is nobody I would rather go through this journey of life with than you.

Dr. Xie: Thanks for all your help and guidance throughout this process. I can honestly say I enjoyed the journey and learned a lot and that is all a credit to you as an advisor and a mentor. Thanks for everything.

LCDR Lukaszewski: Thanks for your patience and willingness to help me through my many technical issues throughout this process. It is not hyperbole to say this research would be nothing without your work, and it largely enabled me to pursue something I was interested in. Thanks again for everything.

Dr. Davis: For being willing to help me out in a pinch as a second reader and always giving me honest feedback as an academic advisor. I am honestly proud of this product and a large part of that is due to the time you were willing to put in to help me make it better. Thank you.

The Men and Woman of TSE: The deed is all, not the glory. Feed the Wolf.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 Motivation

The MITRE Corporation defines exfiltration as the use of various techniques by malicious actors to steal data from proprietary networks [1]. These exfiltration techniques can vary greatly in complexity. Some adversaries, for instance, will simply attempt to exfiltrate data over an established command and control (C2) channel, which can often be detected. However, more sophisticated techniques attempt to transfer the data covertly to avoid detection by typical security controls. One of the more common ways in which malicious actors accomplish this is through the use of encrypted channels. This kind of exfiltration can be particularly hard to mitigate since the data payloads are encrypted and cannot be inspected effectively for detectable signatures without special security controls. Alternatively, adversaries may opt to use obfuscation, also known as network steganography, to exfiltrate data using common network protocols by embedding data within their packets [2]. Obfuscation via commonly used application protocols and the use of encrypted channels over virtual private network (VPN) tunnels or allowable web services are considered more discreet and sophisticated methods of data exfiltration and have been successful in bypassing typical security controls [3].

Many cybersecurity incidents involve some form of data exfiltration and can be perpetrated by both insider and outsider threats [4]. Obfuscated data exfiltration can be a particularly appealing to malicious actors because of its ability to take advantage of network applications and services utilized for daily operations to blend in with normal network traffic. The SolarWinds incident is a perfect example of how effective this type of network obfuscation can be [5]. As detailed in Robert Chesney's "Cybersecurity Law, Policy, and Institutions," once Russia's Sluzbha Vneshney Razvedk (Foreign Intelligence Service) obtained a foothold into target systems via the SolarWinds Orion update mechanism, they utilized the communication channel between the Orion program and the SolarWinds' servers to establish a covert channel to their own C2 server [5]. Thus, they were able to conduct a host of malicious activities to include data exfiltration before they were ultimately detected by one of their vic-

tims, the cybersecurity firm FireEye. As the SolarWinds incident demonstrates, traditional methods of obfuscated data exfiltration have proven difficult to detect and prevent.

In response, many organizations, particularly those concerned with protecting proprietary or sensitive data, are implementing a variety of more sophisticated security controls designed to detect and ultimately prevent unauthorized traffic from leaving a proprietary network. This can include controls designed to detect and block the more secure and covert forms of data exfiltration that utilize encrypted channels [6]. Additionally, ports and services that are utilized to establish these channels can be filtered, limited, or blocked entirely. Transparent inline proxy solutions, such as secure socket layer (SSL) proxy, can enable any authorized use of encrypted channels for network traffic to be controlled and monitored [7]. In these cases, malicious actors are forced to use obfuscation via common protocols that are allowed within the bounds of the organization's policy [2]. Although this method has proven successful, it can be risky in that it involves having to perform packet modification that can potentially be audited and therefore be detected by a host-based intrusion detection system (HIDS) or a sophisticated anomaly detection system (ADS) [3]. Additionally, intrusion detection systems (IDSs), intrusion prevention systems (IPSs), and inline proxy solutions are becoming increasingly sophisticated and able to perform network data flow analysis to detect anomalies indicating potential data exfiltration [8].

Socket layer protocol customization has been suggested as an effective method to modify packets for data exfiltration that might enable malicious actors to bypass more sophisticated security controls. Understanding its use, its capabilities, and its limitations will be a key element of developing mitigations.

A socket is defined as a network communications connection point that can be named and addressed [9]. Sockets are most often referred to in the context of exchanging information between remote processes; however, they can also support processes exchanging information on the same host [9]. For purposes of customization, the socket layer refers to a logical area between the application and transport layers of the standard transmission control protocol (TCP)/internet protocol (IP) model. A common use of the socket layer is to implement of SSL or transport layer security (TLS) during network communications [10]. Its relationship to the other layers of the TCP/IP stack is depicted in Figure 1.1. Payload modification occurs when application data enters the kernel space. During this process,

there is no interaction between the application and socket layers to ensure packet integrity. Similarly, when the data is passed to the transport layer, there is no designated process for checking the integrity of the application-layer payload. The data is simply accepted as is and continues to move through the stack. Auditing occurs at the kernel level during system calls to detect anomalies, but it is currently unknown whether or not standard rule sets can detect malicious manipulation of transmitted data. This presents a potential opportunity for malicious actors to embed data within an application payload without detection and thereby improve upon traditional methods of obfuscated exfiltration.

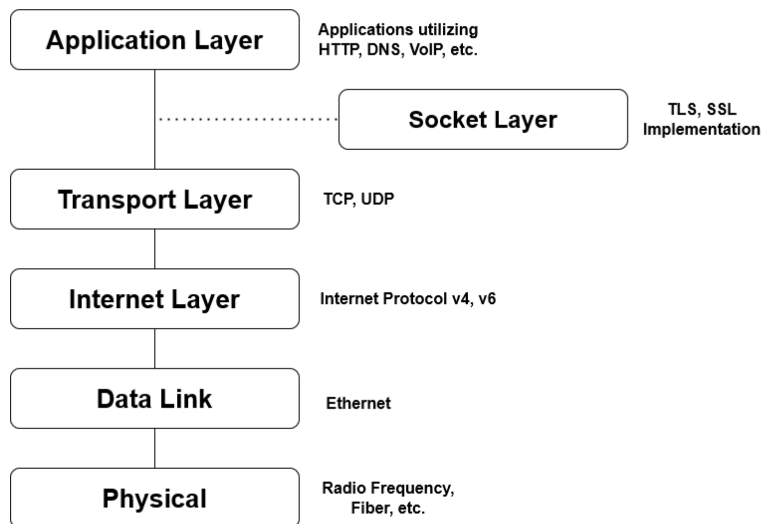


Figure 1.1. Socket Layer Reference to TCP/IP Model

Socket layer customization can further enhance obfuscated data exfiltration by dynamically embedding data to be exfiltrated into the legitimate network traffic of common applications. Thus, in situations where malicious actors might not be able to transmit data over traditional covert channels without detection, they could potentially embed data within legitimate traffic generated by host-based applications. The embedding of data could be performed by a kernel module that is both application and protocol agnostic. This implies that the module can potentially leverage any application or protocol. Additionally, since the embedding of the data happens at the kernel level, the packet modification is transparent to the layers of the TCP/IP stack potentially making it undetectable by host-based security solutions that are designed to detect unauthorized access to files or data by certain applications (e.g.,

common web-browsers). Finally, the dynamic nature of the socket layer module allows it to embed data of variable length and type (e.g., encrypted or encoded), and to embed data at any position within an application payload. This presents the added ability for the data to be customized to potentially bypass additional network security controls such as IPSs, firewalls, content-filtering proxy servers, and other data loss prevention (DLP) solutions.

This thesis simulates the process of data exfiltration between a host residing on a proprietary network and an external server residing somewhere outside the proprietary network. This simulation allows the evaluation of the performance of socket layer customization as a means of data exfiltration using common application protocols. Generally speaking, there are specific attributes of application protocols that can make them more preferable for data exfiltration, and these attributes are evaluated in concert with the use of socket layer customization. Additionally, security controls are implemented within the proprietary network to determine the effectiveness of this data exfiltration method in bypassing these controls. By evaluating this approach to data exfiltration in the context of current defenses, this work attempts to illuminate potential mitigation strategies.

1.2 Problem Statement

Based on recent work, socket layer protocol customization presents the potential to automatically implement various protocols such as SSL/TLS that is transparent to the application and other layers of the TCP/IP model utilizing a socket layer architecture referred to as Layer 4.5 [11]–[14]. The utilization of Layer 4.5 has the potential benefit of not only reducing overhead, but also of standardizing security configurations for applications communicating over the Internet. However, this capability also introduces the ability to transparently modify data packets as they move from the application layer down the TCP/IP stack for transmission. This capability could potentially be utilized for malicious purposes to include data exfiltration. The hypothesis of this thesis is that utilizing socket layer customization to dynamically embed data within standard application payloads will enhance traditional methods of obfuscated data exfiltration via commonly used application protocols.

1.3 Research Questions

In order to effectively evaluate this method of data exfiltration, this thesis made several assumptions. The first was that the adversary had persistent access to the host on the internal network with root privileges (i.e., to load the kernel modules). Exploitation methods necessary to gain access and execution privileges were not addressed. Rather, the scope was limited to specifically evaluating the method of data exfiltration as data was embedded and transmitted through and ultimately beyond the perimeter of the internal network. Additionally, this thesis assumed adversary read access to the files to be exfiltrated. Finally, it was also assumed that the external server was a registered server under full control of the adversary and that it was not blacklisted. As such, it only needed to be configured to host the appropriate application services. With this in mind, the performance of the external server was not evaluated beyond the execution of the kernel module and successful receipt of the exfiltrated data.

Socket layer protocol customization is flexible enough that many different application protocols were available to evaluate this exfiltration method's effectiveness. This thesis performed experiments with six different application protocols: hyper text transfer protocol (HTTP), hyper text transfer protocol secure (HTTPS), simple mail transfer protocol (SMTP), domain name system (DNS), network time protocol (NTP), and voice over internet protocol (VoIP). These application protocols were chosen for their common usage among organizations, as well as their association with traditional methods of data exfiltration. This facilitated the use of related works to provide comparisons during evaluation. This thesis developed custom kernel module configurations for socket layer customization to embed data into legitimate payloads of each of these application protocols and tested against various security controls to answer the following questions:

1. What characteristics of common application layer protocols (HTTP, DNS, SMTP, etc.) are better suited for obfuscating data for exfiltration, particularly when used in concert with socket layer protocol customization?
2. Is this method of data exfiltration detectable by typical host-based security controls designed to monitor and limit an application's access to data and files?
3. Are common network-based security controls able to detect or prevent this method of data exfiltration?

4. If proven to be a viable option, what specific recommendations can be made to mitigate or prevent this method of data exfiltration?

1.4 Thesis Organization

This thesis is divided into four additional chapters. Chapter 2 provides necessary technical background and concepts relative to the experimentation and the evaluations present in this thesis. It begins by discussing the details of the socket layer interface and socket layer architecture, or Layer 4.5 [12]–[14], that were utilized for this thesis. It then proceeds to discuss the application protocols and associated applications, transport protocols, and security controls that were utilized throughout experimentation. Lastly, the chapter highlights related work on obfuscated data exfiltration their mitigation strategies and techniques.

Chapter 3 introduces the experimental design of this thesis. It begins by discussing the development and design of the kernel level customization modules and how they were modified for each protocol. It then discusses the testbed design and describes how the testbed evolved with the addition of security controls over the course of the four phases of testing. The performance metrics themselves are then discussed, to include both quantitative and qualitative measurements. Lastly limitations associated with elements of the experimental design are discussed in order to clarify how results should be interpreted.

Chapter 4 presents the experimental configurations and the process of experimentation. For each phase of testing, the customization module configurations for each protocol are presented as are the application configurations for both the staging and exfiltration servers. A detailed summary of the test runs, test conditions, variable module configurations, and security controls is provided for each phase. Chapter 4 concludes with presentation of the experimental results, which are discussed in detail.

Finally, Chapter 5 presents the main conclusions drawn from the experimental results in the context of the research questions presented in Chapter 1. Following the main conclusions, the key limitations of the exfiltration method and customization modules are discussed. Chapter 5 concludes with a discussion of specific mitigation recommendations and provides recommendations for future research related to data exfiltration via socket layer protocol customization.

CHAPTER 2:
Technical Background

This chapter provides technical background regarding the software, protocols, and security controls utilized for the experimentation in this thesis. The first topics discussed are the socket layer architecture, or Layer 4.5, and how socket layer customization can be used to embed data into application payloads to conduct obfuscated data exfiltration. These topics are the primary enablers for the exfiltration method being evaluated in this research. A brief discussion of the software components utilized for the design and implementation of the virtual network environment is then provided. The various applications and application protocols that are utilized throughout experimentation are then presented. Detailed explanations of the security controls tested against the exfiltration method are also provided in this section. The chapter concludes a discussion of related work on obfuscated exfiltration.

2.1 The Socket Interface and Layer 4.5

Socket layer protocol customization is the key element in the potential enhancement of the obfuscated data exfiltration techniques evaluated in this thesis. The term "protocol customization" refers to the process of customizing or modifying the functionality and features of existing protocols [15]. This process can involve various techniques such as the addition of parameters or the modification or manipulation of messaging formats (referred to as *protocol dialecting*) [16]. Conversely, it can involve the reduction of functionality, or *debloating*, in order to tune or remove parameters or disable features in specific use cases [17], [18]. This allows for the enhancement of a protocol, typically for security or performance, without fundamentally altering its original design. This lightweight customization, or extension of functionality, allows existing protocols to be enhanced with new features, thus saving the overhead associated with implementing completely new protocols into the TCP/IP framework. Current protocol customization and extension methodologies, however, are typically limited to a single protocol and have proven problematic on implementation [19]–[22]. This has led to recent work on the development of a more flexible and efficient approach to protocol customization that relies on a novel socket layer architecture and application programming interface (API) that has been dubbed Layer 4.5 [12]–[14].

2.1.1 The Socket Networking Interface

A socket interface is used to define a connection between two endpoints associated with processes that are communicating and exchanging data [23]. On each end of a connection (client or server), the kernel creates a socket to facilitate passing messages between the application process and the transport layer. This basic socket is made up of a transport protocol identifier (e.g., TCP or user datagram protocol (UDP)) and what is commonly referred to as the socket pair: the source and destination IP addresses and port numbers [23]. Applications and programs that monitor or utilize socket connections often employ their own messaging formats to manipulate socket connections, however the basic semantics are fairly standardized. Two examples of how socket connections are often displayed using these basic properties are as follows:

- HTTP
  - *TCP: source = 192.168.1.1:1024 -> destination = 192.168.1.2:80*
- DNS
  - *UDP: source = 192.168.1.1:1025 -> destination = 192.168.1.3:53*

These examples show how an application on a single host may identify a socket connection for outbound traffic to two separate hosts providing basic services, in this case HTTP (web) and DNS (name resolution). The source and destination socket pair could also be reversed to indicate inbound traffic. The API creates the socket according to these properties and provides a descriptor or socket handle to identify the interface and associate it with a specific process or application [23]. Once a connection is established, the sockets on both endpoints are bound to each other and form the socket pair. This allows a host's operating system to identify data bound for a specific process or application and to handle it accordingly. Socket layer customization utilizes these defining properties of a socket interface to identify and intercept application data for customization.

2.1.2 Layer 4.5

Unlike traditional approaches, The Layer 4.5 methodology utilizes kernel-level modules to apply and process customizations on a per application message basis by tapping the socket API [12]–[14]. The basic Layer 4.5 device architecture, depicted in Figure 2.1, involves the application or application protocol, socket API, customization loader, and customization

module. The application data is pulled from the TCP/IP stack in its entirety using the socket API and is manipulated by the customization module before being inserted back into the stack [12]–[14]. Thus, customization is essentially independent from the application protocols and their original design and allows them to gain or reduce functionality without affecting compatibility. Since customization modules are not protocol specific, they can be applied to multiple protocols without significant modification [12]–[14]. While this methodology is meant to enhance functionality and security, it can also potentially be used by malicious actors to add or embed any type or amount of additional data to the application payload. This concept is the basis for the hypothesis of this thesis.

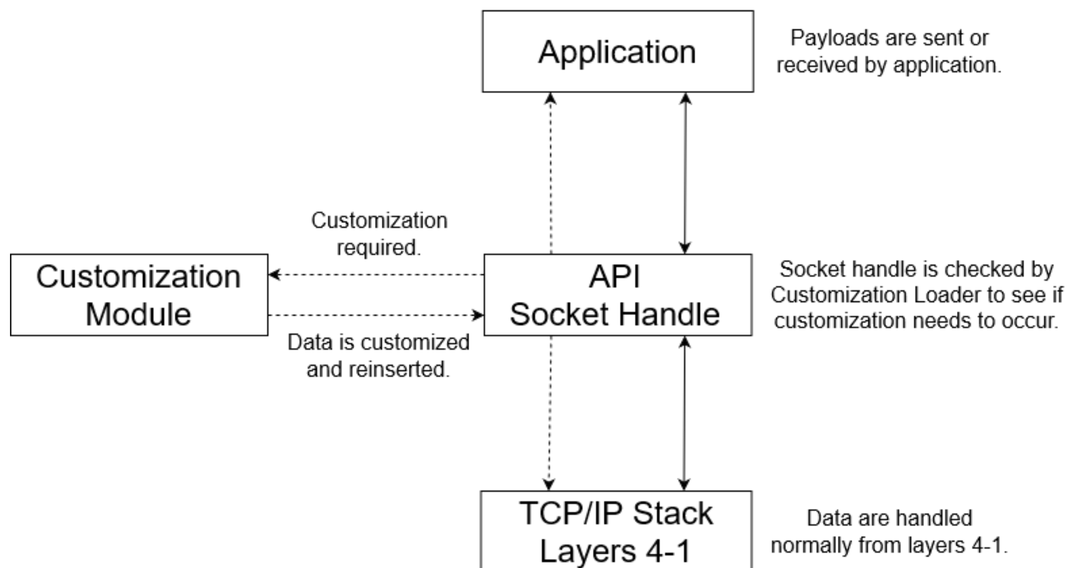


Figure 2.1. Basic Layer 4.5 Methodology. Adapted from [11]

2.1.3 Customization Modules

Customization modules are kernel-level modules that provide the basic functionality of the Layer 4.5 architecture by implementing specific protocol customization requirements [12]–[14]. These modules are developed with standard functions that fulfill specific customization

requirements and are configured for a specific socket connection between a sender and a receiver. At run-time, modules pull and process packets during transmission and apply customization based on the socket connection. At a minimum, customization modules must include the following metadata to aid the customization loader in making decisions [12]–[14]:

- Application/Process name,
- Socket pair,
- Layer 4 protocol (i.e., TCP or UDP),
- Send and/or receive functions, and
- Customization Identification.

The sole requirement for the experimentation described in this thesis was to embed specific data into specific application payloads. This required only one module per host (sender or receiver) that was configured for each tested application protocol. These modules are currently only available for use in the Linux operating system environment of the host machines used for this thesis.

2.1.4 Customization Loader

The customization loader is a separate kernel-level process used to invoke or load a customization module on a host-machine on a given socket connection [12]–[14]. Figure 2.2 shows the standard process of invoking a specific module via the customization loader. The loader is configured with specific socket parameters and is itself invoked when a new socket connection is identified. As specified by its configuration it then determines which modules that are loaded on a host have configured parameters that match the socket connection. When a match is found that module is bound to the socket, and all future data transmitted over the socket is customized according to the module's configuration. If no match is found the loader marks the socket for no customization to prevent future customization queries and to reduce overhead. For this thesis, each host was only loaded with one module so that the customization loader only had to invoke that module to embed data during experimentation.

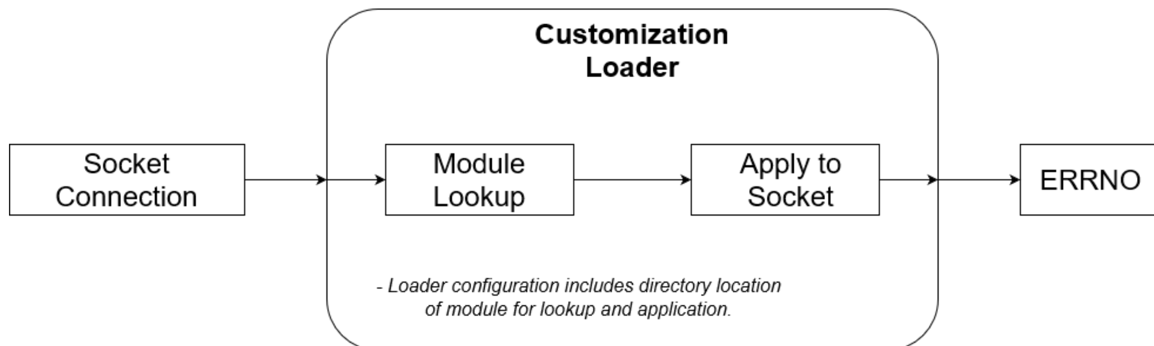


Figure 2.2. Basic Customization Loader Logic. Adapted from [11]

2.2 Application Protocols

Network protocols, by definition, are a set of rules or procedures for formatting and processing electronic data for transmission between devices [24]. The Internet Engineering Task Force, is predominantly responsible for developing, maintaining, and promulgating protocols as standards for use in inter-networking communications [25]. From this standpoint, protocols are usually referenced within the TCP/IP framework and exist at every layer of the TCP/IP stack to ensure the reliable transmission of data between applications communicating on different hosts. Together these protocols form the TCP/IP protocol suite and provide the basic functionality of communicating over the Internet.

Application layer protocols govern how applications interface with the lower levels of the TCP/IP stack to transmit and receive data [26]. These protocols function at the top layer of the TCP/IP stack and prepare the application data payload for encapsulation as it moves through the stack and is ultimately transmitted. Conversely, these protocols ensure that data is interpreted properly by the receiving application during decapsulation.

The following subsections describe the various application protocols and their associated applications used in this thesis for experimentation. These standard application protocols are in common usage and represent some of the most basic network-based services that organizations use for daily operations such as web-browsing and email. As such, they are

perfect candidates for obfuscated data exfiltration because they are commonly identified as "normal" and permitted within an organization's network.

These subsections are divided into two distinct parts. The first part describes the application protocol in detail and highlights its key attributes that make it ideal for experimentation. The second part describes the tools, services, and other technologies specific to the application protocol that were used for experimentation.

2.2.1 Hyper Text Transfer Protocol (HTTP) and Hyper Text Transfer Protocol Secure (HTTPS)

HTTP is a stateless application-level protocol for distributed, collaborative, hypermedia information systems [27]. It has been primarily used for supporting client web browsing and accessing web hosted services since as early as 1990 [27]. It has been updated frequently since its inception to meet the evolving and growing requirements of web-based communication. The most current widely adopted version is HTTP/2 [28]. As a popular generic protocol, it has proven very flexible in its ability to provide basic communication between user agents and other information systems, even those that supported by other application protocols. As such, HTTP, is still a common widely used protocol standard [27].

HTTPS is not considered a separate application protocol, but rather an extension of the original HTTP protocol standard. HTTPS specifically references the use of HTTP over TLS (formerly SSL) links that provide channel-oriented security for sensitive information [29]. TLS in its current version (1.3) is designed to provide confidentiality and authenticity, while also being application protocol independent [30]. This allows higher level application protocols such as HTTP to implement security without changing the overall protocol standard. HTTPS via TLS 1.3 is currently the recommended standard for web-based communications [31].

HTTP/HTTPS has several attributes that make it ideal for obfuscated data exfiltration. For one it is the standard protocol for web browsing and accessing web-based services, which means organizations often require its use for daily operations. It is a high usage protocol and as such a high volume of associated traffic will not be suspicious. Additionally, it incorporates a variety of request methods that are initiated by the client giving the adversary more potential options for embedding data.

Python 3

Python 3 is the most recent version of the popular high level object-oriented programming language designed to be an easy-to-use programmable interface to many system calls and libraries [32]. One of its most significant attributes is its versatility, which enables it to be used in most operating system environments without modification [32]. Within the scope of this thesis, Python 3 was used primarily to develop configuration programs for the customization modules. It was also used to program an Exfiltration Server to host basic HTTP web services. This allowed for testing of the HTTP protocol using a simple application instance [33], [34].

Apache2

Apache2 represents the latest version of the open source web server developed and maintained by the Apache HTTP Server Project [35]. Apache2 provides simple and efficient HTTP services using the current HTTP standards [35]. With regards to this thesis it is used in conjunction with the OpenSSL library to host a simple web server with SSL/TLS capability on the Exfiltration Server for testing with HTTPS [36], [37].

Client for URLs (cURL)

cURL is a small software suite for conducting data transfers using one or more supported Internet protocols [38]. It also provides a basic command-line tool for sending and receiving data using Uniform Resource Locator syntax [38]. cURL supports many protocols for data transfer, but for this thesis it was used for initiating client based web requests for testing with HTTP and HTTPS.

2.2.2 Simple Mail Transfer Protocol (SMTP)

SMTP is a connection-oriented, text-based application protocol developed to transfer electronic mail reliably and efficiently [39]. SMTP utilizes a reliable ordered data stream channel via TCP to transfer mail messages from a client to one or more SMTP servers [39]. Servers then act as relays and send messages to other SMTP servers in order to ultimately be received by the intended recipient [39]. In this sense, SMTP is considered to be a delivery protocol only. SMTP has been modified several times since its initial implementation to add additional features predominantly relating to security.

Data exfiltration via email services in the form of attachments is quite common and well documented, but embedding data within email protocols themselves is less so [6], [40]. Given that e-mail is essential to the daily operations of most organizations, SMTP is an attractive protocol for obfuscated data exfiltration.

Postfix

Postfix is a popular open-source Mail Transfer Agent that can both send and receive electronic mail at servers for delivery to a client [41]. Postfix employs numerous client/server programs to determine routes and send emails, and it implements a number of standard security techniques [41]. Over the course of this thesis, it was used to setup basic SMTP servers on the Staging Host and Exfiltration Server to simulate the sending of email messages from the internal network to a remote server for testing of data exfiltration via SMTP [42], [43].

2.2.3 Domain Name System (DNS)

DNS is an application layer protocol whose core functionality is the mapping of assigned host or domain names to specific IP addresses for locating hosts and services on the Internet [44]. The DNS protocol is designed to be extensible and as such has been subject to many updates and extensions over the years. Since its adoption as a standard, it has evolved into an indispensable protocol that implements a hierarchical and decentralized naming system that supports the basic functionality of the Internet as it is widely used today.

DNS is one of the more common protocols that adversaries use to establish covert tunnels for data exfiltration [45]. This is mainly due to the recursive and hierarchical properties of the DNS architecture that make it more feasible for a malicious actor to register a domain and receive exfiltrated data on a controlled authoritative server [46]. The fact that DNS also provides an essential service makes it a prime candidate for testing with this data exfiltration method.

Dnsmasq

Dnsmasq is open-source software designed for small networks that provides lightweight network infrastructure services, the most notable of which are DNS and dynamic host

configuration protocol (DHCP) [47]. Its lightweight design makes it perfect for resource-constrained networks such as the virtual networking environment used for this thesis. The DNS subsystem provides a local DNS server for the network with the ability to forward queries to recursive servers. For this thesis, Dnsmasq was solely used to host a DNS server on the Exfiltration Server and handled only queries from the Staging Host for the purpose of testing data exfiltration with the DNS protocol [48], [49].

2.2.4 Network Time Protocol (NTP)

NTP, now on its fourth version, is a widely used application protocol used to synchronize computer clocks on the Internet [50]. NTP is implemented among a set of distributed time servers and clients [50]. The protocol is implemented within a network such that the time servers are in communication with the clients at regular intervals to ensure all systems maintain synchronization [50]. Thus, NTP accounts for a significant portion of frequent traffic between client-based hosts and time servers. Similar to DNS, NTP provides an essential service and is therefore also potentially exploitable for obfuscated data exfiltration.

NTPDaemon

NTPDaemon, commonly referred to as ntpd, is an operating system daemon or program instance that runs in the background to synchronize system clocks via communication with an NTP server [51]. Ntpd can actually be setup as either a client or a server and directly implements the NTP protocol. Ntpd was used in support of this thesis to run a client instance on the Staging Host to communicate with a time server instance on the Exfiltration Server to test data exfiltration using the NTP protocol [52].

2.2.5 Voice Over Internet Protocol (VoIP) and Session Initiation Protocol (SIP)

VoIP is not actually a single protocol or technology, instead it is a layered hierarchy of protocols and technologies that provide voice or multimedia sessions over IP links [53]. One of the more widely used protocols implementing VoIP is SIP, an application layer protocol that can establish, modify, maintain, and terminate real-time multimedia sessions [54]. It is a notably flexible protocol that can work in conjunction with other application protocols such as real-time transport protocol (RTP) and is designed to be independent of transport layer

protocols for a variety of implementations [54]. The fact that VoIP encompasses multiple protocols and technologies and involves frequent variations in traffic size and type makes it potentially vulnerable to being exploited for data exfiltration.

Real-time Transport Protocol (RTP)

RTP provides end-to-end delivery for data with real-time characteristics, such as interactive audio and video [55]. While its name implies that it is a transport protocol, it actually functions above the transport layer to carry media data streams, usually in conjunction with UDP [55]. In this sense, RTP is malleable enough to enable its integration into the processing of different data applications. Thus, it can be tailored for any application that requires it through simple modifications and customization. For this thesis, a RTP/UDP stream within a VoIP session is targeted to test this method of data exfiltration.

Linphone

Linphone is free open-source software providing VoIP services via SIP and RTP that is available for the Linux environment [56]. It was utilized in this work to test data exfiltration via an established VoIP session between the Staging Host and the Exfiltration Server [57].

2.3 Transport Protocols

Transport layer protocols are responsible for the delivery of the application data payload to the remote application via connections that are defined by port numbers [58]. Application protocols can be characterized as both connection-oriented or connectionless based on the transport layer protocol that is used to meet the application's requirements [59]. Connection-oriented implies that the transport layer provides both reliability and in-order delivery to the remote application. Connectionless, in contrast, implies that the delivery is best effort and subject to data loss during transmission. Both types of transport layer protocols can provide multiplexing and have advantages that can be leveraged based on the application's requirements [59]. The primary connection-oriented and connectionless transport layer protocols are TCP and UDP respectively.

2.3.1 Transmission Control Protocol (TCP)

TCP is a transport layer protocol that is utilized to provide reliable, connection-oriented data transmission between specific numbered ports on two hosts [60]. It accomplishes this by first establishing a connection between the two hosts, usually a client and a server, through the use of an electronic three-way handshake [60]. Once the connection is established, data can be reliably exchanged between the client and the server. Reliability is maintained through the use of packet numbering and acknowledgements, whereby data receipt is confirmed [60]. If the receipt of data is not acknowledged within a specified time frame, it is automatically resent. TCP data transmissions are often broken up into manageable chunks, the size of which are dictated by other network parameters such as maximum transmission unit (MTU) and window size [60]. Once receipt of all data is confirmed, the connection is closed.

Application protocols that require the reliable transfer of data can be identified by their use of TCP. In fact, many common applications where data loss is considered unacceptable utilize TCP to ensure quality of service. These application protocols are particularly ideal for obfuscated data exfiltration because they will ensure the transfer of the desired data to the remote host.

2.3.2 User Datagram Protocol (UDP)

UDP is transport layer protocol that provides basic transport services between numbered ports on two hosts [61]. Unlike TCP, UDP provides no reliability of transmission or receipt [61]. It simply transmits data as prescribed by the socket connection while providing no guarantee of delivery. As a result, UDP is not subject to the overhead associated with reliable data transmission. This makes it more suitable for time-sensitive or lightweight protocols where the loss or corruption of some packets can either be accepted or solved with a simple retransmission [61]. Many lightweight application protocols that provide essential system services, such as DNS and NTP utilize UDP because of their small packet size and frequent use [44], [50]. Many streaming application protocols utilize UDP since they can afford some loss of packets without significantly degrading quality of service [53], [55]. The ports associated with these specific application protocols are typically open to accommodate daily network and business operations. This, coupled with the fact that UDP-based protocols make up a high volume of traffic in any network, makes UDP protocols ideal candidates for obfuscated data exfiltration.

Table 2.1 presents a summary of the application protocols that were utilized for experimentation in this thesis and their relevant attributes.

Table 2.1. Summary of Application Protocols

Application Protocol	**Transport Protocol**	**Service**	**Encryption**
HTTP	TCP	Web	No
HTTPS	TCP	Web	Yes
SMTP	TCP	Mail	Yes
DNS	UDP	Name Resolution	No
NTP	UDP	Time Synchronization	No
VoIP	UDP	Voice/Video Chat	No

2.4 Security Controls

The term security controls describes a broad range of parameters, safeguards, and countermeasures that can be implemented to protect various forms of data and information system infrastructure that are deemed important to an organization [62]. Security controls are often categorized based on the type of security they provide and the types of assets they are intended to protect (e.g., physical and network security controls) [63]. Most often, organizations will perform risk or vulnerability assessments to determine exactly what security controls they need to implement based on their unique business or operational requirements. Several frameworks exist to help organizations perform assessments and implement appropriate security controls, such as those provided by the National Institute of Standards and Technology and the Center for Internet Security [63]. The security controls utilized to support this thesis' experimentation primarily fall into the category of network cybersecurity controls since they enforce rules preventing unauthorized connections or transmission of valuable data outside the internal network. The specific controls that were utilized are detailed in the following subsections.

2.4.1 Iptables

Iptables is the standard firewall utility program for Linux systems. Iptables uses basic Netfilter modules to provide traffic-filtering services for standard network or socket connections [64]. Iptables allows for the implementation of filtering policies on a host-by-host

basis. If configured correctly it can be very effective in enforcing network traffic restrictions and filtering unauthorized traffic. It can also emulate many of the more advanced functions of an inline IPS to filter network traffic without introducing significant overhead. This makes it a capable security control to supplement an overall network security infrastructure [64]. For this thesis, iptables was used to enforce a basic restrictive network traffic policy on packets transmitted between the Staging Host and the Exfiltration Server and to act as a perimeter router to provide simple traffic forwarding at the virtual host.

2.4.2 Wireshark

Wireshark is a free and open-source network protocol analyzer [65]. It allows for the deep inspection of network traffic on a byte-by-byte basis at each layer of the TCP/IP stack. Additionally, it is designed to allow identification of the protocols being used and to conveniently label all parts of the transmitted data to aid in analysis. In this thesis, Wireshark was used to capture data packets during experimentation to help analyze the effectiveness of data exfiltration. Figure 2.3 provides a screenshot of Wireshark being utilized to capture baseline HTTP traffic.

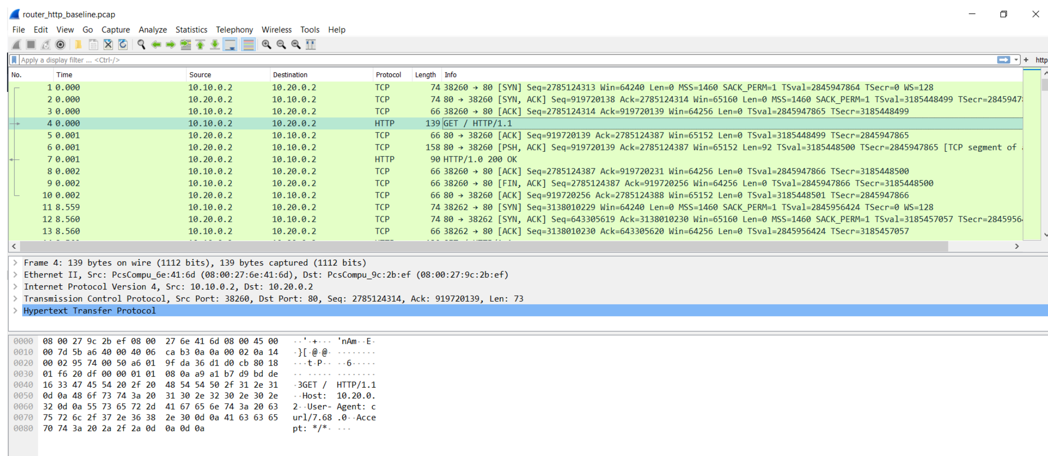


Figure 2.3. Wireshark Packet Capture of HTTP Traffic

2.4.3 AppArmor

AppArmor is a Linux-based security application designed to provide basic host-based security [66]. AppArmor protects the host operating system and applications from both internal

and external threats by enforcing basic mandatory access control policies and preventing application flaws from being exploited [66]. This allows administrators to effectively restrict applications' capabilities according to individual access control profiles. In this thesis, AppArmor was utilized as a basic host-based security control to supplement discretionary access control policies on restricted files and to apply restrictive profiles to applications used for data exfiltration experimentation [67].

2.4.4 Snort

Snort is free, open-source, lightweight IDS/IPS software for Linux and Windows [68]. It is rule or signature-based, and once configured it establishes a network policy that characterizes malicious network activity. The policy is used to detect packets that are identified as malicious traffic and to generate user alerts [68]. Snort can also be deployed inline, much like iptables, to act as a network IPS to further analyze and filter traffic [68], [69]. In this thesis, Snort was deployed and configured as an inline IPS during experimentation to provide additional traffic analysis and testing against common rule sets designed to prevent data exfiltration [70], [71]. Figure 2.4 depicts a typical deployment of Snort as an inline IPS.

2.4.5 Data Loss Prevention (DLP) Controls

DLP controls are solutions specifically designed to detect and prevent data breaches or data exfiltration [72]. They often take the form of software developed to identify, monitor, and protect sensitive or valuable organizational data from being accidental or intentionally shared outside of an organization's proprietary network [72]. While typical network security controls are often deployed with the intent of protecting data from the outside, DLP controls are focused on protecting data from the inside. DLP solutions make distinctions between data in three stages: at rest, in use, and in motion [72]. For this thesis, specific DLP controls were implemented to inspect and protect data in motion. These controls performed deep packet inspection, or content-filtering, in order to test against exfiltration of data embedded in the tested protocols. The specific controls were based on recommendations from a showcase report by the Software Engineering Institute (SEI) at Carnegie Mellon University on detecting data exfiltration through deep packet inspection [6]. These controls are detailed in the following subsections.

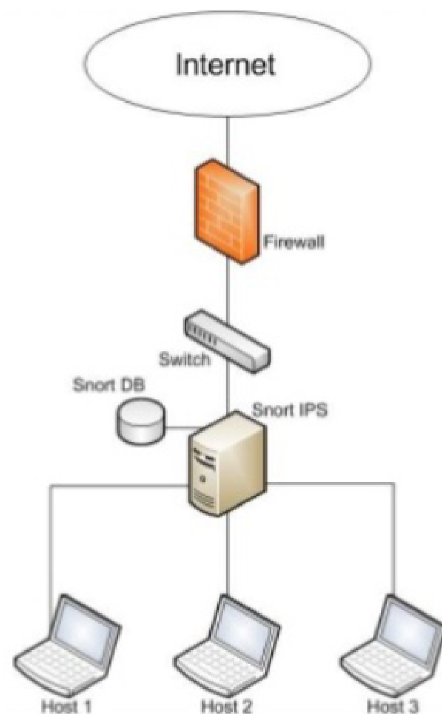


Figure 2.4. Snort Deployed in Inline Mode as an IPS. Source: [69]

Squid Proxy Server

Squid is a caching proxy server software suite supporting a variety of web-based application protocols [73]. It is primarily used to maximize bandwidth and improve network performance by caching and reusing frequently requested web pages. However, as detailed in the SEI report [6], Squid can also be used as a content-filtering or traffic inspection proxy server for web-based traffic. In this configuration, Squid effectively acts as a man-in-the middle (MitM) proxy that is able to inspect both encrypted and unencrypted web traffic in order to detect and prevent attempts at data exfiltration. Squid was used in this thesis for this purpose [6], [74].

In order to inspect traffic, the communications channel from the internal host was terminated at the Squid proxy server rather than the external destination as depicted in Figure 2.5. The Squid proxy inspected the packets and reencrypted them (if necessary) before forwarding them to the external destination host. This is one of the more effective ways an organization

can inspect encrypted web-based traffic for data exfiltration before it leaves the internal network.

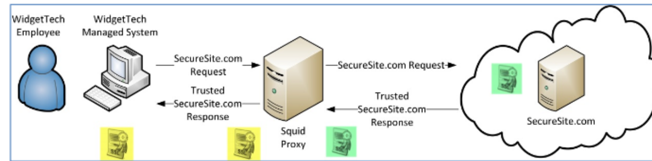


Figure 2.5. Traffic Inspection at Squid Proxy Server. Source: [6]

ClamAV

ClamAV is a free, open-source antivirus software engine for detecting various forms of malware and other malicious threats [75]. It is most often deployed on Linux-based systems and includes a number of utilities including a command-line scanner, an automatic database updater, and a multi-threaded daemon [75]. ClamAV was installed on the Squid proxy server in support of this thesis to assist in the scanning and blocking traffic containing customized file signatures. These signatures were tailored for application to the files that were to be exfiltrated during experimentation. Essentially, these signatures were manually added to the ClamAV signature database to falsely flag exfiltration attempts as viruses in order to block exfiltration attempts. [6], [76].

2.5 Related Work

A fair amount of research has been conducted on various ways to perform and enhance obfuscated data exfiltration using common protocols, including the application protocols that are evaluated in this thesis. This work has included a number of controls that are frequently implemented on proprietary networks. HTTP, for example, is very often heavily monitored and controlled to prevent its misuse. Considering that browser-based attacks are still very common, security conscious organizations are prone to implement controls to monitor and secure transmissions over HTTP. Nevertheless, exfiltration techniques continue to evolve in response to these defenses.

Ede presented a novel method for data exfiltration via HTTP that utilizes a dynamic and adaptive approach [77]. In this approach the adversary uses malware to analyze the regular

browser traffic present on a target host to create a template for exfiltration that adapts to and mimics regular benign browser traffic [77]. Although, an effective heuristic method for detecting the exfiltration method was also presented, the paper acknowledged that the exfiltration method would be harder to detect if it increased the volatility of the rate and amount of exfiltrated data [77]. Additionally, it was noted that this sort of dynamic and adaptive approach could be applied generally to other protocols [77]. This observation aligns with the intent of this thesis to evaluate data exfiltration via socket layer customization to determine if it can potentially allow for this sort of adaptive approach (i.e., by implementing protocol-agnostic modules with dynamic module parameters that can change how data is embedded into application payloads).

Born presented research on embedding data within application payloads and reassembling it at a remote server in "Browser-Based Covert Data Exfiltration" [78]. In his paper, Born discusses traditional methods of data exfiltration via the HTTP and DNS application protocols using tools such as *httptunnel* and *Iodine* respectively [78]. Born then presents an exfiltration method that utilizes JavaScript executed by a target host's browser to encode and embed data into the payload of DNS queries. This approach effectively performs obfuscated data exfiltration without privilege escalation [78]. The presented results indicate that the method is effective, but Born highlights the difficulty of separating the DNS queries and HTTP requests as the JavaScript executes within the browser application [78]. This can partly be attributed to the general difficulty of embedding data within payloads at the application layer. Related research conducted at the Oak Ridge National Laboratory in 2016, references Born's work and notes that most common techniques for data exfiltration involve natively installed utilities that are able to remain undetected [40]. Data exfiltration via socket layer protocol customization presents the ability to leverage use of native applications and avoid application layer difficulties by embedding data into application payloads as they transition from the application layer to the transport layer.

Malicious actors will often try to enhance obfuscated data exfiltration by establishing exfiltration servers that also host legitimate services. DNS, for instance, is an appealing protocol for malicious actors to target since it is critical to maintaining the functionality of the Internet as it exists today. As a result, many organizations implement controls to monitor and limit DNS traffic to specific local and authoritative servers to prevent its use for exfiltration. However, this form of exfiltration is often difficult to detect and prevent given the recursive

hierarchical nature of the DNS architecture [46]. Research has shown that malicious actors can successfully exfiltrate data via DNS on a target host by registering an authoritative domain and then embedding data into legitimate queries for that domain [46]. The global DNS architecture ensures that resolvers will forward the queries to the authoritative server. Since it is under the control of the malicious actor, the server can then retrieve the exfiltrated data [46]. This thesis attempted to simulate a similar scenario in testing by assuming that the Exfiltration Server was a registered server that was also hosting legitimate services.

A team from Northumbria University researched how typical data exfiltration via DNS could be enhanced via HTTPS [79]. Originally developed with security in mind, DNS over HTTPS (DoH) can ensure privacy and makes eavesdropping and MitM attacks more difficult. As their work indicates, however, the ability to encrypt DNS traffic could also facilitate typical methods of data exfiltration. The team performed tests against a variety of security controls and concluded that if an adversary could establish a tunnel between a target host and a rogue DNS server, it would be very difficult to detect exfiltration via DoH [79]. The team concluded that exfiltration could be detected by sophisticated traffic filtering proxies but that this would essentially nullify the privacy benefits of DoH [79]. Organizations are often hesitant to implement DLP solutions that might be viewed as invading privacy or inducing overhead. Even if DLP solutions are implemented, socket layer protocol customization might still make this type of exfiltration method possible if it can bypass typical DLP controls.

VoIP has seen rapid evolution and multiple advances over the last few years, and its growth has only been accelerated by the recent COVID-19 pandemic. Many organizations now see VoIP services as critical to their daily operations. As such, VoIP and the various protocols it incorporates make interesting candidates for data exfiltration. A study performed by a team from the School of Electronics and Information and Automation, Civil Aviation University of China researched VoIP's potential to hide and embed data for both security and malicious purposes [80]. Specifically related to data exfiltration, the study highlighted the fact that VoIP provides multiple attack vectors against the various protocols and payloads that are utilized to provide voice and video streaming [80]. Adversaries, for instance, might make use of the redundancy in both voice and video streams to embed data without detection [80]. Additionally, a study conducted at the Warsaw Technical University in 2008, pointed out that the high volume of data exchanged during a VoIP connection coupled with the potentially

high bandwidth associated with streaming protocols makes for an ideal environment to potentially exfiltrate a significant amount of data very quickly [81]. Further, protocols such as RTP often stream at dynamic and variable rates, which can make it difficult to define and inspect for detectable signatures without degrading quality of service [81]. A possible limiting factor of exfiltration over VoIP is the lack of reliability associated with UDP-based streaming protocols. This may affect the ability to transfer exfiltrated data in its entirety if using a dynamic approach through socket layer customization. This is one of the questions this thesis intended to answer through experimentation.

Similar to DNS, NTP is a critical application protocol that has been researched for its potential to establish covert channels for obfuscated data exfiltration. A study by Ameri and Johnson presented at the International Conference on Cryptography, Security, and Privacy in 2017 evaluated how a covert channel could be established between a receiving host and an NTP server without being detected [82]. Their model involved segmenting a one megabyte (MB) file into 32-bit segments and embedding them within all NTP responses to clients so that they could be retrieved by the listening receiver. They noted that the main limiting factor was the throughput of the channel, which was mostly a byproduct of the number of clients making NTP requests [82]. Another study by a group from Otto von Guericke University, noted that while data exfiltration over NTP was limited by available bandwidth, NTP provides malicious actors with a reliable and effective way to deeply hide persistent data exfiltration when bandwidth is not a priority [83]. Additionally, normal NTP packets exhibit high entropy, and it was discovered that embedding a small amount of encrypted data resulted in very comparable entropy values. This makes it potentially very difficult to detect based on anomalies or signatures [83]. The basic concept of embedding smaller amounts of data to avoid detection explored in these studies could perhaps also be applied to other protocols with more capable bandwidth via socket layer protocol customization.

In this thesis, the customization modules that were utilized for embedding data into application payloads are described as loadable, kernel-level modules. This is an accurate description; however, their ability to execute potentially malicious code without detection with the privilege of the Linux kernel makes their classification as a type of kernel-level rootkit reasonable as well. Traditional Linux kernel-level rootkits typically take advantage of loadable kernel modules and rely on kernel privilege to access restricted areas of the operating system to include those allowing system calls and unrestricted access to user-

space applications [84]. While the original intent of socket protocol customization was to enhance the functionality of standard protocols, it is worth noting that the customization modules utilized in this thesis make use of all these kernel-level techniques. While it falls out of the scope of experimentation for this thesis, from a mitigation perspective some of the host-based detection mechanisms related to detecting traditional rootkits might also be useful in detecting malicious actions performed by this sort of customization module [84].

2.6 Summary

This chapter presented the significant technical aspects of this thesis. The Layer 4.5 architecture was introduced and the process of socket protocol customization was discussed in detail. The various, software, applications, protocols, and security controls that were utilized throughout experimentation were also presented to provide the reader with the necessary background information to interpret the results of this thesis. The chapter concluded with a discussion of works related to obfuscated data exfiltration and kernel-level modules.

The next chapter discusses the experimentation phases and experiment design. The virtual environment in which the experiments were conducted is also described.

CHAPTER 3: Experiment Design

This chapter details the development and design of the kernel-level customization modules and testbed environment that were utilized in this thesis. It begins by describing the general Layer 4.5 customization loader and its role in performing protocol customization. The protocol specific customization modules developed for this research are introduced, and details are provided on how they were designed to dynamically embed data in the payloads of each targeted application protocol. The logic behind the chosen configuration parameters of the customization modules and the design of the configuration programs themselves are also discussed. The chapter goes on to describe the network setup for each phase of testing. In particular, details on the testbed evolution across test phases are provided. Finally, a description of the performance metrics that were used at each phase of testing to evaluate the overall effectiveness and viability of the exfiltration method is presented. The chapter concludes with a discussion of the significant assumptions and limitations associated with the experimental design.

3.1 Socket Layer Customization Implementation

The socket layer protocol customization that was utilized in this research to embed data for exfiltration relied on two distinct kernel-level modules written in the C programming language and compiled using *gcc* version 9.4.0: the Layer 4.5 customization loader and the external customization module. The customization loader manages the registration and loading of external customization modules. It encodes the configurable parameters for each customization module (Table 3.1) as specific data structures that enable it to identify and track specific socket connections for customization [12]–[14]. When a socket connection to which a customization module is to be applied is identified, taps into standard socket calls are used to create a customized socket in place of the original application socket [12], [14]. Functions defined in the module are applied to future transmissions as long as the socket connection is open. In this thesis, the Layer 4.5 customization loader was installed on both the Staging Host and the Exfiltration Server to enable the embedding and extraction of data on specific socket connections.

Table 3.1. Customization Module Configurable Parameters

Parameter	**Applicable Host**	**Implementation**
Application Name	Both	Hard-coded
Destination/Server IP Address	Both	Hard-coded
Source/Client IP Address	Server	Hard-coded
Destination Port Number	Both	Hard-coded
*Source Port Number	Both	Hard-coded
Protocol Number	Both	Hard-coded
File to Exfiltrate	Client	Dynamic
File Size	Both	Dynamic
Byte Size of Data Customization	Both	Dynamic
Byte Position of Data Customization	Both	Dynamic

* *For experimentation, source port is a hard-coded wildcard value since it changes for every client-initiated connection, however, it can be set to a constant value if required.*

Two customization modules were developed: one for the Staging Host and one for the Exfiltration Server. Both of these versions of the customization module had the same default functions and designs that allowed them to register and interface with the Layer 4.5 customization loader. However, their internal send and receive functions were modified to accommodate the embedding or extraction of data respectively. Each version of the module was designed to be protocol agnostic with configurable parameters that were used to designate the socket connection for customization. These parameters also dictated what and how much data was embedded within the application payload during transmission. The parameters could also be modified at run-time, which enabled a dynamic approach to data exfiltration. This allowed the modules to be deployed with a wide-range of capabilities to accommodate the different application and transport layer protocols. The send version, or client-side module deployed on the Staging Host was designed to embed data within specific application payloads based on a registered socket connection. Conversely, the receive version, or server-side module deployed on the Exfiltration Server was designed to extract data from the application payloads based on the same registered socket connection. Table 3.1 presents the specific configurable parameters that were utilized for the customization modules of this thesis.

Of the Table 3.1 parameters, six were hard-coded in the customization module configuration for each application protocol prior to loading the module. These parameters could only be changed while the module was unloaded. The reason for this was that these parameters were also utilized by the Layer 4.5 customization loader to identify socket handles for customization. The hard-coded parameters were application name, destination address, source address, port number, and protocol number. The remaining four parameters; file, file size, byte size, and byte position; were modifiable while the customization module was loaded. This allowed the parameters dictating how and what data was embedded into the application payload to be configured dynamically without reloading the module. The only limiting factor was that both the send- and receive-side hosts had to have the same parameter configurations before data was embedded.

Algorithms 1 and 2 provide a high-level overview of the logic used by the customization loader and modules. The overall process consisted of three primary events: identify specific socket connections for customization, embed and send data from the client-side host, and identify and receive data on the server-side host. Socket connections can only be identified for customization on the first socket call, or initial socket creation. If a customization module is loaded after a socket has been created, then that socket will not be identified for customization regardless of whether or not it matches the module's configuration parameters.

Two simple bash shell scripts were utilized to configure the dynamic parameters of the client and server customization modules. During execution the shell scripts ask the user for specific inputs associated with configurable experimentation parameters. These shell scripts enable the module parameters to be changed while the customization modules were loaded to allow for quick transitions between experiments. Additionally, both bash shell scripts start *tcpdump* instances to capture client and server network traffic for post-test analysis and evaluation. The source code for the Layer 4.5 deployment module, customization modules, and configuration bash shell scripts is hosted on the Naval Postgraduate School (NPS) GitLab page as referenced in the appendix (A.1).

3.2 Testbed Design

At its highest level, the testbed design, depicted in Figure 3.1 is a simple point-to-point connection between the Staging Host and the Exfiltration Server. For experimentation

Algorithm 1 Client-side Customization Module

```
1:  Init Module Event:
2:     Module registers Layer 4.5 customization
3:     Loader will check all sockets against parameters for a match
4:  while Customization Module is Registered do
5:     First Socket Call Event:
6:        Application initiates socket call
7:        Loader creates customization socket and checks against module parameters
8:     if Match = True then
9:        Customization = True
10:    else
11:       Customization = False    //Socket is not customized and TX proceeds as normal
12:    end if
13:    while Customization = True do
14:       Send Func Event:
15:       if First Customization then
16:          Create custom memory buffer for file data based on file size parameter
17:          Read data from file to be exfiltrated into custom buffer
18:          Establish variable to keep track of how much file data is sent
19:       end if
20:       if More Data to Exfiltrate then
21:          Read byte position and byte size parameters
22:          Copy and embed file data from custom file buffer into original data payload
23:          Return customized payload with embedded data to send buffer
24:       else
25:          Free allocated memory
26:          Customization = False    //All file data has been sent
27:       end if
28:    end while
29: end while
```

purposes, these two hosts resided on a virtual network hosted on a MacBook Pro with a Intel Core i7 processor and macOS Big Sur version 11.4 installed. These two hosts both had Ubuntu operating system version 20.04/5.13 installed and were the only two host machines on the experimental local area network. This simple network topology allowed for traffic to be easily identified and analyzed as it was transmitted between the client and server using the various testing protocols. Additionally, it allowed for easier implementation and analysis of the security controls being tested against this method of data exfiltration. High

Algorithm 2 Server-side Customization Module

```
1: Init Module Event:
2:    Module registers Layer 4.5 customization
3:    Loader will check all sockets against parameters for a match
4: while Customization Module is Registered do
5:    First Socket Call Event:
6:       Server receives message and initiates socket call
7:       Loader creates customization socket and checks against module parameters
8:    if Match = True then
9:       Customization = True
10:   else
11:      Customization = False    //Socket is not customized and RX proceeds as normal
12:   end if
13:   while Customization = True do
14:      Recv Func Event:
15:      if First Customization then
16:         Create custom memory buffer for exfiltrated data based on file size parameter
17:         Establish variable to keep track of how much exfiltrated data is received
18:      end if
19:      if More Data Expected then
20:         Read byte position and byte size parameters
21:         Extract and copy exfiltrated data from receive buffer to custom buffer
22:         Return original data prior to client-side customization to receive buffer
23:      else
24:         Write exfiltrated data to kernel log file and free allocated memory
25:         Customization = False    //Data exfiltration is complete
26:      end if
27:   end while
28: end while
```

traffic volume outside a virtual environment is more realistic and often has its own effects on security control effectiveness; however, this simple setup allowed for analysis of the exfiltration method's ability to bypass typical security controls in isolation (i.e., without variable factors influencing the results).

3.2.1 Phase One Testing

Phase One testing predominantly focused on verifying the ability of socket layer customization to be adapted to any application or protocol. More specifically, it identified what

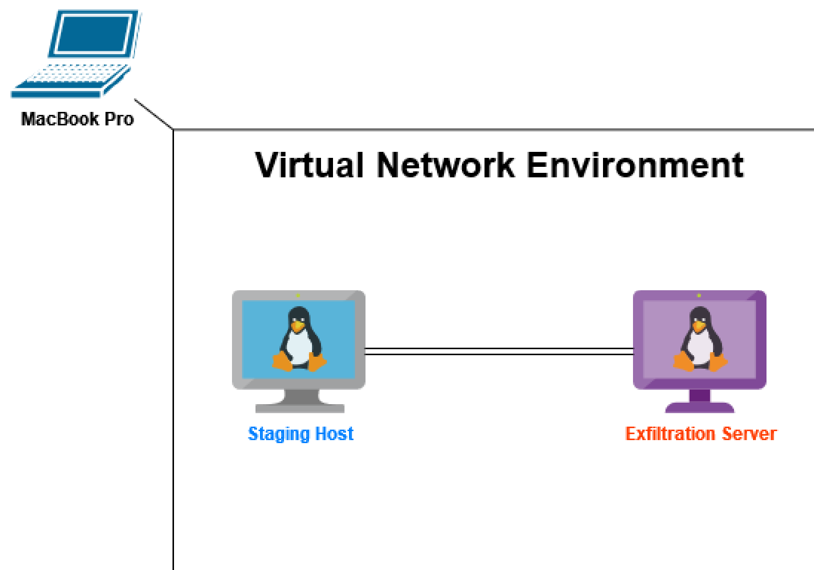


Figure 3.1. Basic Testbed Design

characteristics of application protocols were more conducive to obfuscated data exfiltration via socket layer protocol customization. As such, the testbed network design for Phase One was relatively simple and consisted of only the internal Staging Host, a perimeter router, and the external Exfiltration Server.

The Staging Host represented the target host machine located on a proprietary network and contained the files to be exfiltrated. This host utilized Linux's uncomplicated firewall (UFW) tool in its default configuration to block all incoming connections not initiated by the Staging Host.

The perimeter router was considered part of the proprietary network and served two purposes. One was to act as the default gateway for routing internal network traffic to the Internet (to include the network on which the Exfiltration Server resided). The second was to provide a basic iptables firewall to filter unauthorized traffic into and out of the internal network. All of the application protocols being tested were among those that are commonly used by organizations, so none of their associated ports were blocked since they would be considered required for daily operations. Iptables was configured to drop all traffic associated with other ports and to block all non-established inbound connections.

The Exfiltration Server was configured with three separate interfaces to host all of the relevant services. It also utilized Linux's UFW tool, which was configured to allow only traffic destined for ports related to hosted services. To this end, the Exfiltration Server was considered a registered server that was hosting legitimate services and as such, was not blacklisted. Figure 3.2 depicts the network setup for Phase One testing, and Table 3.2 lists the application protocols, port numbers, and client/server applications utilized for testing. The applications presented in this table were chosen based on their ease of implementation, widespread use, and ability to simulate realistic traffic for the application protocols being tested. Specific configurations for the host machines, perimeter router, and iptables can be found hosted on the NPS GitLab page as referenced in the appendix (A.1).

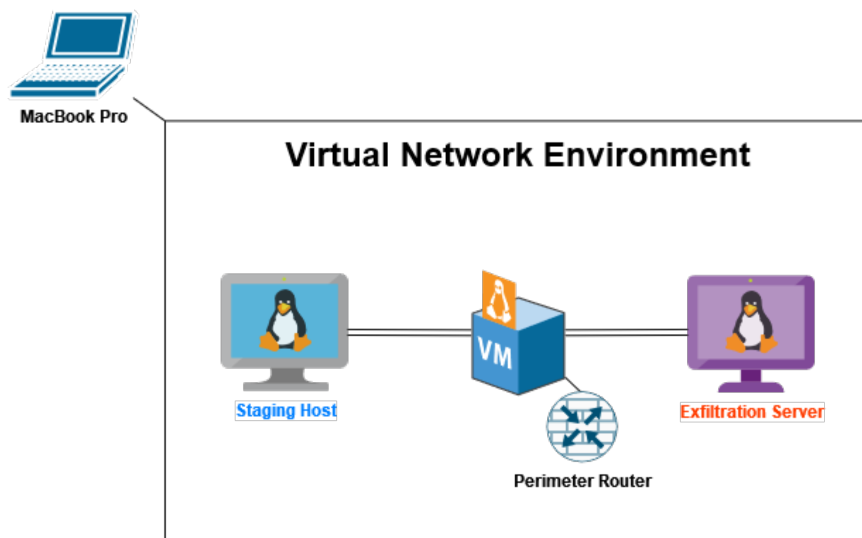


Figure 3.2. Phase One Testbed Design

Table 3.2. Virtual Machine Testing Configurations

Application Protocol	Port Number	Client Application	Server Application
HTTP	80	cURL	Python3
HTTPS	443	cURL	Apache2
SMTP	25	Postfix	Postfix
DNS	53	nslookup, dig	Dnsmasq
NTP	123	ntpdate	ntpd(aemon)
VoIP	9078, 9079, 7078, 7079	Linphone	Linphone

3.2.2 Phase Two Testing

The testbed was modified slightly for Phase Two to enable testing of the customization module's ability to perform customization against host-based security controls. During this phase, AppArmor was enabled and configured on the Staging Host to restrict access permissions to the files that were to be exfiltrated. AppArmor comes preinstalled on most versions of the Linux operating system and implements access controls through flexible profiles that can be applied to individual applications [67]. AppArmor profiles were implemented and configured to specifically limit the ability of applications to access files for the purpose of data transmission. Figure 3.3 depicts the network setup for Phase Two testing.

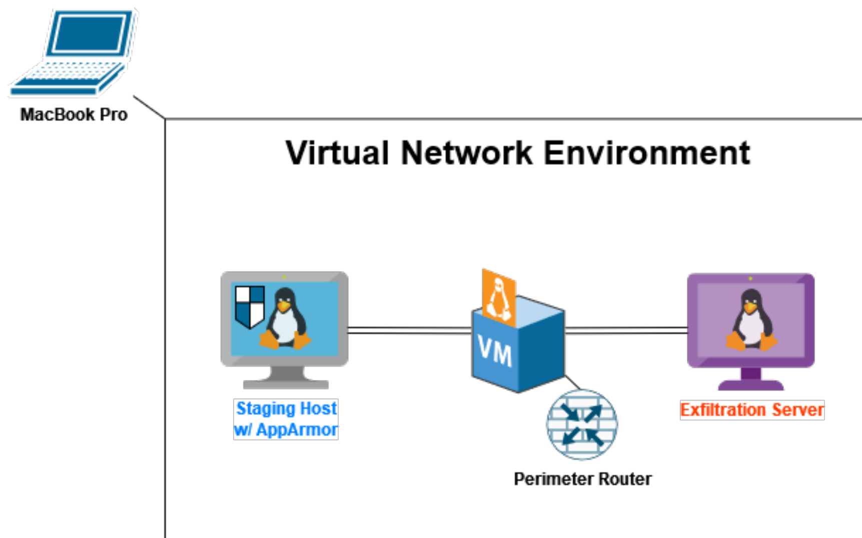


Figure 3.3. Phase Two Testbed Design

3.2.3 Phase Three Testing

The testbed was further modified for Phase Three to test the customization's ability to perform data exfiltration against more sophisticated network-based security controls. In this phase an additional virtual host was added inline between the perimeter router and the Staging Host. Snort was installed and configured on this additional host as a network IPS security control to inspect traffic as it left the target host network en route to the Exfiltration Server. Snort was utilized on this host with its default IPS configuration, which included alert generation, logging, deep packet inspection, and the ability to filter and drop traffic. Snort was installed and configured with the latest registered rules provided

by Cisco's threat intelligence research team, Talos. These rules are commonly utilized to identify malicious traffic and malware conforming to known signatures [68]. After testing using Snort's default configuration and registered rules, additional preprocessor rules and custom rules were enabled to target specific application protocols and content signatures of the target files to see if Snort could be configured to consistently detect and prevent data exfiltration attempts. Figure 3.4 depicts the network setup for Phase Three testing.

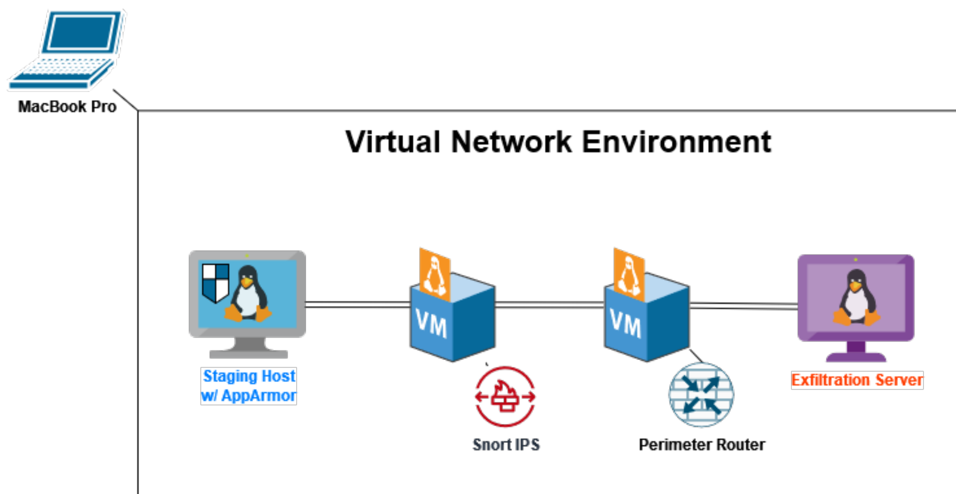


Figure 3.4. Phase Three Testbed Design

3.2.4 Phase Four Testing

Phase Four required adding an additional host to the testbed between the Snort IPS and the Staging Host. This host used the Squid and ClamAV software packages to provide a simple content-filtering proxy. Squid was configured as a transparent inline proxy to perform deep packet inspection on web-based traffic. In addition to the default signatures provided by ClamAV, additional customized hexadecimal signatures related to the files to be exfiltrated were added to the ClamAV configuration. These signatures took the form of actual data and included an added tag (FOUO, 0x464f554f) at the beginning, middle, and end of the proprietary file. These sorts of tags are often utilized by organizations to identify proprietary and other sensitive documents that may be targeted for exfiltration [6]. This final network configuration tested the customization modules' abilities to bypass more robust and comprehensive network security solutions specifically implemented to prevent data exfiltration of specific files. Figure 3.5 depicts the network setup for Phase Four testing.

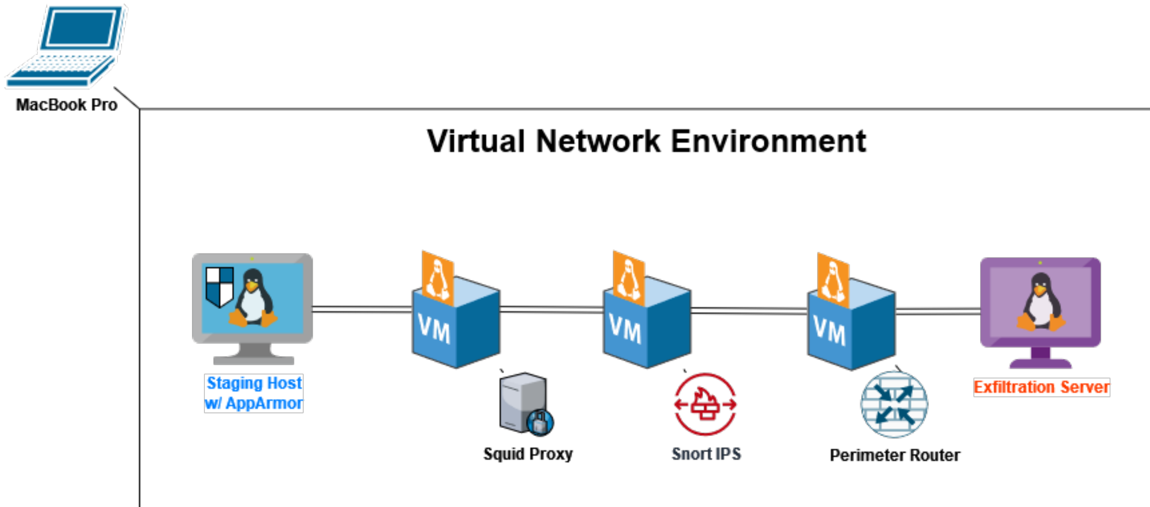


Figure 3.5. Phase Four Testbed Design

3.3 Performance Metrics

The performance metrics were varied for each phase of testing based on the goals of each phase. In Phase One there were two goals. The first was to successfully exfiltrate two separate files from the Staging Host to the Exfiltration Server. An associated objective of these tests was to gather measurable data on the performance of each application protocol when the customization module was utilized. Each protocol was placed into one of three categories: UDP stream, TCP session, or VoIP conversation. Exfiltration for both files was attempted for each protocol with three different embedded-byte sizes (i.e., the number of bytes simultaneously embedded). A maximum segment size of 1500 bytes was assumed to account for the largest number of simultaneously embedded bytes and to provide for accurate comparison. The specific metrics used to measure the performance of the data exfiltration with the customization module were as follows:

- Total number of packets transmitted,
- Total number of bytes transmitted, and
- Total time to completion of data exfiltration.

Additionally, these metrics were utilized as the basis for a relative comparison of the

performance of each application protocol for exfiltration. For these relative comparisons two distinct measurements were utilized: relative throughput and relative overhead. Relative throughput captures the amount of file data exfiltrated in kilobyte (KB)'s per second, and relative overhead captures the number of packets transmitted per KB of exfiltrated file data. These measurements were useful for a relative comparison because they detailed each application protocol's performance as a function of the total size of the exfiltrated data. In short, they showed exactly how much data each application protocol was able to exfiltrate per second and how much overhead was incurred for each KB of exfiltrated data.

Successful data exfiltration was verified in two ways. The first involved gathering and reassembling the data on the Exfiltration Server to reconstruct the original file. A message digest algorithm (MD5) hash was calculated for the reconstructed file and compared with a calculated MD5 hash of the original file on the Staging Host. A match indicated successful exfiltration of the data as depicted in Figure 3.6. The second method of verifying successful exfiltration involved instances where customization module or buffer errors occurred. In these instances Wireshark was utilized to analyze a packet capture of traffic on the Exfiltration Server interface to verify the receipt of all exfiltrated data.

Figure 3.6. MD5 Hash Comparison to Verify Successful Exfiltration

A simple text file and an image file in the tag image file format (TIFF) format were used for experimentation. The text file was left in its original ASCII encoding for exfiltration. The TIFF image file, which is in a binary format, was converted to ASCII-based hexadecimal character bytes using the `xxd` utility prior to exfiltration. This encoding made the process of exfiltrating, gathering, and reassembling the data into its original form on the Exfiltration Server simpler and more manageable. The files utilized for exfiltration and their associated sizes were as follows:

- *constitution.txt* (44.84 KB) and
- *Penguin.tif* (229.8 KB original and 459.5 KB encoded).

The second goal of Phase One was to evaluate the effectiveness of the customization

modules for performing data exfiltration with each application protocol. For each test run, data was embedded into application payloads at a specific byte position using one of the three test embedded-byte sizes and exfiltrated. Embedded-byte position was varied for each application protocol and was chosen based on the total size and format of the application payload. *Tcpdump* was utilized to capture network traffic on the Staging Host, perimeter router, and Exfiltration Server. The traffic was then analyzed using Wireshark and evaluated, taking any errors reported into account, to identify specific characteristics of each application protocol that made it more or less conducive for obfuscated data exfiltration. The results of these tests drove customization module configurations for the subsequent phases of testing against security controls.

The ideal module configurations for subsequent phases were identified based on the results of Phase One experimentation. The data exfiltration experiments were then repeated for Phases Two through Four to test whether the customization modules could successfully bypass the security controls introduced in each phase. Boolean values of *True* and *False* were utilized to report the successful or unsuccessful detection or prevention of the data exfiltration as a result of the added security controls. Successful detection or prevention events were subsequently analyzed and evaluated to determine what signature or rule-set was able to detect or prevent the data exfiltration attempt.

3.4 Research Assumptions and Limitations

With regards to the experimental design, some significant assumptions were made to accommodate the testing of the customization modules in this thesis, and a number of limitations must be noted. One assumption, which has been mentioned previously, was that the adversary had already achieved persistent, root-level access on the Staging Host. In the context of Lockheed Martin's Cyber Kill Chain [85], the actual execution of data exfiltration being evaluated in this thesis was representative of the final phase, Actions on Objective, whereby the malicious actor achieves intended objectives by executing specific actions on the target host. The six phases that would be required prior to the Actions on Objective phase represent critical steps to covertly obtain the persistent system access necessary for data exfiltration [85]. This thesis, however, was only concerned with the obfuscated data exfiltration performance against specific host and network-based security controls, so it was reasonable to assume the adversary's success up to that point. That said, the experiments

in this thesis were not an attempt to diminish the importance of defending against these preliminary steps, but rather an exploration of a specific type of attack at a specific point in the kill chain.

It is also worth noting that the functionality of the exfiltration customization modules could be incorporated into a version of the operating system kernel (e.g., by compromising the supply chain) rather than through a Layer 4.5 approach. Basically, the customization module could be developed to be independent of the Layer 4.5 deployment module platform. Assessment of that approach, however, is beyond the scope of this thesis and it was therefore not accounted for in this work.

The ability of a malicious actor to compromise the kernel of a target host presents a worst case scenario for the defender. Consequently, there are kernel-level host-based controls available that are specifically designed to mitigate this threat. Many of these kernel-level controls are event-based and take advantage of robust logging at the kernel level to observe, detect, and in some implementations even prevent potential malicious modules from loading on a host [84]. Additionally, research has been conducted on kernel-level controls that can monitor or intercept system calls and make policy-based decisions to either allow, flag, or deny them [84], [86]. These types of controls often blend attributes of both network-based and host-based security controls. Their implementation introduces additional administrative and resource overhead, but they are important for any organization to consider for any enterprise-level environment to mitigate or prevent exploitation of kernel-level modules such as the method being explored in this thesis. Nevertheless, experimentation in this thesis was limited to the evaluation of the customization module's ability to bypass a readily available host-based control, in this case AppArmor, implemented to detect and prevent data exfiltration in real-time. Thus, the use of such security controls and analysis of kernel-level logs to detect the unauthorized use of kernel-level customization modules, both during and following exfiltration, falls outside the scope of this thesis.

A significant limitation of the customization modules in their current forms is the difficulty to integrate them with TLS. TLS version 1.3 is comprised of two primary components that complicate integration with the customization modules: the TLS handshake and the record protocol or record layer. The TLS handshake is used to authenticate the communicating parties, negotiate cryptographic modes and parameters, and establish shared keys [30]. It

works primarily to secure the connection between hosts and does not impede the ability of the customization modules to embed and extract data within associated payloads. The record layer, on the other hand, works to protect traffic between communicating peers by dividing traffic into a series of records, each of which is independently protected using the traffic keys [30]. While the record layer does not directly inhibit the embedding or extracting of data by the customization modules, it does induce significant errors when the data is extracted at Layer 4.5 and passed to the application layer. This is due to one record in particular, length, that specifies the size of the TLS application payload outside of the record layer. Since TLS is implemented above Layer 4.5, the length value is calculated before data is embedded. Embedding data, therefore, results in a transmitted payload length that is larger than the length record which often leads to a *decode_error* on the server-side (this error indicates that a message could not be decoded because of a field error or incorrect length) [30]. To complicate matters further, TLS-based applications seem to designate specific length records for certain packets, particularly during the TLS handshake. In these situations, specific sizes for receive buffers are allocated to match the record layers and payload lengths. Some applications will request the five-byte record layer before processing the payload to determine the exact length of payload to allocate and push to the TLS process. When the receive-side customization module extracts the embedded data it will often cause corruption in these buffers due to the TLS process receiving data that it was not expecting. Figure 3.7 provides a generic illustration of this limitation as data is embedded at the Staging Host and received at the Exfiltration Server.

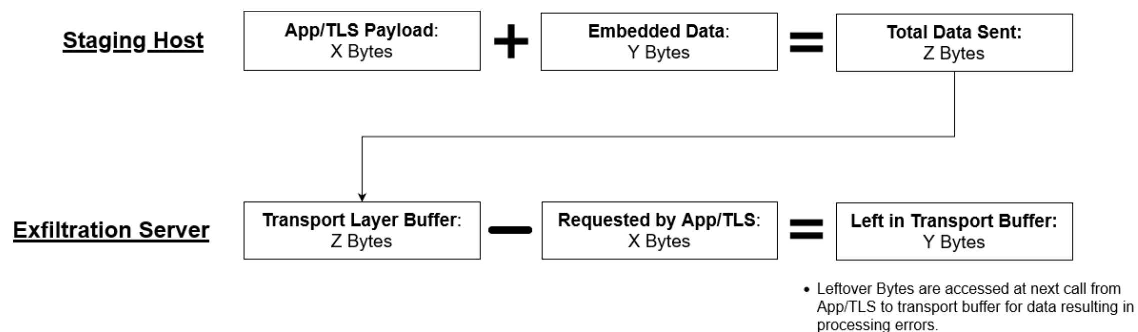


Figure 3.7. TLS Buffer Limitation

Figures 3.8 and 3.9 present examples of this limitation in the experimental test environment for a test in which 12 bytes were embedded into a TLS *Client Hello* packet (517 bytes) on the Target Host and received by a simple Python-based web server on the Exfiltration Server. Figure 3.8 shows a trace log of five bytes being pulled from the transport buffer on the Exfiltration Server. This is the record data that tells the TLS process the version being used, what type of data is in the packet (*Client Hello*), and the content length. The next trace log entry of the figure shows 512 bytes being read from the transport buffer, as dictated by the content length of the record data. Embedded within this 512 bytes are 12 bytes of illegitimate (i.e., embedded) data. This data is extracted properly and the original data is put back into its original position. However, as designed, the customization is independent of the TLS process, so TLS still reads 512 bytes which now include 12 bytes of padding from the original *Client Hello* packet. This data does represent any legitimate TLS record or application data and causes processing errors as a result. This scenario is played out to completion in Figure 3.9. Here the data is extracted successfully, but the TLS process interprets the additional bytes as an unintended message and terminates the connection. Future research efforts will focus on addressing this limitation by adding additional buffering capabilities to the Layer 4.5 architecture to act as an intermediary between the transport buffer and the TLS process.

```
python3-4061    [000] ....   826.736557: modify_buffer_recv: Customization has started
python3-4061    [000] ....   826.736559: modify_buffer_recv: Message size is 5
python3-4061    [000] ....   826.736560: trace_print_cust_hex_dump: Message Hex: 000000007b0aa44e: 16 03 01 02 00    Record Data
.....
python3-4061    [000] ....   826.736561: modify_buffer_recv: No customization
python3-4061    [000] ....   826.736607: modify_buffer_recv: Customization has started
python3-4061    [000] ....   826.736607: modify_buffer_recv: Message size is 512
python3-4061    [000] ....   826.736608: trace_print_cust_hex_dump: Message Hex: 00000000c570818c: 01 00 01 fc 03 03 c5 15 22
........"M.q..B.
```

Figure 3.8. TLS Client Hello Message Received by the Exfiltration Server

```
python3-6116    [000] ....  4726.706726: modify_buffer_recv:  is a test.\n    Embedded Data
python3-6116    [000] ....  4726.706740: modify_buffer_recv: Customization has started          Leftover Data
python3-6116    [000] ....  4726.706741: modify_buffer_recv: Message size is 12
python3-6116    [000] ....  4726.706742: trace_print_cust_hex_dump: Message Hex: 000000001c0fbb8b: 00 00 00 00 00 00 00 00 00 00 00 00
...........
```

Figure 3.9. Corruption of TLS Buffer by Leftover Data

There are also limitations associated with how the customization modules prepare data for

exfiltration. In their current states, the customization modules open and read the files to be exfiltrated in kernel space and write the data to a buffer that is subsequently accessed when data is embedded into application payloads. This technique allows targeted files to be accessed directly by the customization module without further interaction with the user. While this technique can potentially offer advantages, it limits the amount of data that can be written from the target file to a buffer to approximately one MB due to limits of the *kernel_read* function. This necessitates data exfiltration in separate segments for files greater than one MB, which limits the ability of the customization modules to perform automated data exfiltration.

Finally, encoding of targeted files for exfiltration during experimentation was conducted by the bash script configuration program rather than within the customization module. This was done for the sake of convenience and simplicity; however, it would be feasible to implement code within the initialization function of the customization module to perform basic encoding and encryption as the file is read into a buffer. Furthermore, the customization modules currently cannot confirm or acknowledge the receipt of exfiltrated data. This does not pose notable problems with regards to the objectives of this thesis since all testing is done on an isolated virtual network. This could be a significant limitation when testing this method in more realistic network environments where packet loss is more likely.

3.5 Summary

In summary, this chapter provided an overview of the experimental design for this thesis. The design and development of the Layer 4.5 customization modules were discussed. The parameters and logic that allow the modules to perform customization to existing sockets to exfiltrate and receive data were presented in detail. Additionally, the basic testbed design and its evolution through each phase of testing were also presented. The performance metrics were then discussed in detail to show how the exfiltration method was tested and evaluated for each test phase. The chapter concluded with a discussion on the significant assumptions and limitations associated with the experimental design for the reader's consideration.

Chapter 4 covers experimentation and provides a detailed discussion of the results from each test phase.

CHAPTER 4: Experimentation and Results

This chapter presents the specific configuration settings for the experiments conducted in each phase of testing and explains the execution flow for individual experiments. It concludes with the reporting of the results and analysis of the customization modules' performance against the specific security controls implemented in each phase of testing.

4.1 Experimental Configurations

As previously stated, there were four distinct experimental testing phases that were utilized to evaluate the ability of socket layer protocol customization to enhance obfuscated data exfiltration. Each phase adds to the experimental network testbed by implementing specific host-based and network-based security controls to prevent data exfiltration. A detailed guide on the setup and configuration of the virtual machines used in experimentation can be found hosted on the NPS GitLab page referenced in the appendix (A.1). All source code for the various customization modules, the Layer 4.5 deployment module, the Python configuration files, and the associated bash scripts can also be found on the GitLab page.

Table 4.1 presents the virtual machine configurations utilized throughout experimentation. It also includes the significant applications and services that were configured and utilized on each machine. Finally, it indicates the test phases in which each virtual machine was used.

Table 4.1. Virtual Machine Configurations

Phases	Virtual Machine	Operating System	Network Adapters	Applications/Services
1, 2, 3, 4	Staging Host	Ubuntu 20.04	1	cURL, Postfix, Linphone
1, 2, 3, 4	Exfiltration Server	Ubuntu 20.04	3	Apache2, Dnsmasq, Postfix, ntpd, Linphone
3, 4	Snort IPS	Ubuntu 20.04	2	Snort IPS 2.9
4	Proxy Server	Ubuntu 20.04	2	Squid, ClamAV
1, 2, 3, 4	Perimeter Router	Ubuntu 20.04	3	Iptables

The Staging Host was configured with the cURL software utility installed to accommodate simple command-line HTTP and HTTPS GET and POST requests. The Postfix utility was installed to enable the sending of simple test email messages. The Linphone application was installed to enable VoIP calls to be initiated. The `dig` and `ntpdate` command-line utilities were utilized to perform manual queries to test the DNS and NTP protocols respectively.

The Exfiltration server had all the applications and services necessary to receive and respond to requests initiated by the Staging Host installed. These included Apache2/Python 3 (web servers), Dnsmasq, Posttfix, ntpd, and Linphone.

The Snort IPS and proxy server virtual machines were configured with Snort, Squid, and ClamAV as described in Sections 3.2.3 and 3.2.4 respectively.

Table 4.2 presents the network configurations utilized to connect the different nodes for each phase of experimentation. Phases One and Two had the same network configuration with the only difference being the implementation of host-based security controls (i.e., AppArmor) on the Staging Host for Phase Two. Phase Three added the virtual machine hosting the Snort IPS in inline mode. This network node acted as a transparent bridge for inspecting network traffic without altering the baseline network configuration. Phase Four added a virtual host that acted as a transparent content-filtering proxy for DLP. The addition of this machine changed the baseline network configuration to intercept and inspect all traffic from the Staging Host before using network address translation to forward approved traffic to the perimeter router.

Table 4.3 presents the hard-coded customization module configurations for each application protocol that was tested in each phase of experimentation. These configurations remained unchanged throughout each respective phase of testing.

4.1.1 Phase One: Exfiltration Method Evaluation and Analysis

Figure 4.1 depicts the network environment for Phase One testing. For each application protocol, three rounds of 13 data exfiltration test runs for both files were completed for a total of 78 runs. Different embedded-byte sizes were utilized for each file in each round. The embedded-byte sizes for *constitution.txt* were 10, 118, and 236. The embedded-byte sizes for *Penguin.tif* were 10, 125, and 250. The embedded-byte sizes were slightly different

Table 4.2. Network Configurations by Testing Phase

Phase	Staging Host	DLP Proxy	Snort IPS	Perimeter Router	Exfiltration Server
1	10.10.0.2	N/A	N/A	10.10.0.1 10.20.0.1 10.0.2.15 (WAN)	10.20.0.2 (Web, VoIP) 10.20.0.3 (DNS, NTP) 10.20.0.4 (Mail)
2	10.10.0.2	N/A	N/A	10.10.0.1 10.20.0.1 10.0.2.15 (WAN)	10.20.0.2 (Web, VoIP) 10.20.0.3 (DNS, NTP) 10.20.0.4 (Mail)
3	10.10.0.2	N/A	Bridge to Perimeter Router	10.10.0.1 10.20.0.1 10.0.2.15 (WAN)	10.20.0.2 (Web, VoIP) 10.20.0.3 (DNS, NTP) 10.20.0.4 (Mail)
4	10.30.0.2	10.30.0.1 10.10.0.2	Bridge to Perimeter Router	10.10.0.1 10.20.0.1 10.0.2.15 (WAN)	10.20.0.2 (Web, VoIP) 10.20.0.3 (DNS, NTP) 10.20.0.4 (Mail)

Table 4.3. Customization Module Hard-coded Configurations

HTTP						
Host	**App. Name**	**Dest. Addr.**	**Src. Addr**	**Dest. Port**	**Src. Port**	**Protocol Nbr.**
Staging Host	curl	10.20.0.2	N/A	80	*	6
Exfiltration Server	python3	10.20.0.2	10.10.0.2	80	*	6
HTTPS						
Host	**App. Name**	**Dest. Addr.**	**Src. Addr**	**Dest. Port**	**Src. Port**	**Protocol Nbr.**
Staging Host	curl	10.20.0.2	N/A	443	*	6
Exfiltration Server	apache2	10.20.0.2	0.0.0.0	443	*	6
SMTP						
Host	**App. Name**	**Dest. Addr.**	**Src. Addr**	**Dest. Port**	**Src. Port**	**Protocol Nbr.**
Staging Host	smtp	10.20.0.4	N/A	25	*	6
Exfiltration Server	smtpd	10.20.0.4	10.10.0.2	25	*	6
DNS						
Host	**App. Name**	**Dest. Addr.**	**Src. Addr**	**Dest. Port**	**Src. Port**	**Protocol Nbr.**
Staging Host	*	10.20.0.3	N/A	53	*	17
Exfiltration Server	dnsmasq	0.0.0.0	10.10.0.2	53	*	17
NTP						
Host	**App. Name**	**Dest. Addr.**	**Src. Addr**	**Dest. Port**	**Src. Port**	**Protocol Nbr.**
Staging Host	ntpdate	10.20.0.3	N/A	123	*	17
Exfiltration Server	ntpd	10.20.0.3	10.10.0.2	123	*	17
VoIP						
Host	**App. Name**	**Dest. Addr.**	**Src. Addr**	**Dest. Port**	**Src. Port**	**Protocol Nbr.**
Staging Host	linphone	10.20.0.2	N/A	9078	*	17
Exfiltration Server	linphone	0.0.0.0	0.0.0.0	9078	*	17

for the two files to make them whole number factors of the total file size. Additionally, the byte positions were varied with each run and were chosen based on the application protocol being tested. The purpose of these test runs was to establish basic performance metrics of the exfiltration method for each application protocol to allow future comparison and identification of any characteristics of specific application protocols that make them more or less conducive to this exfiltration method.

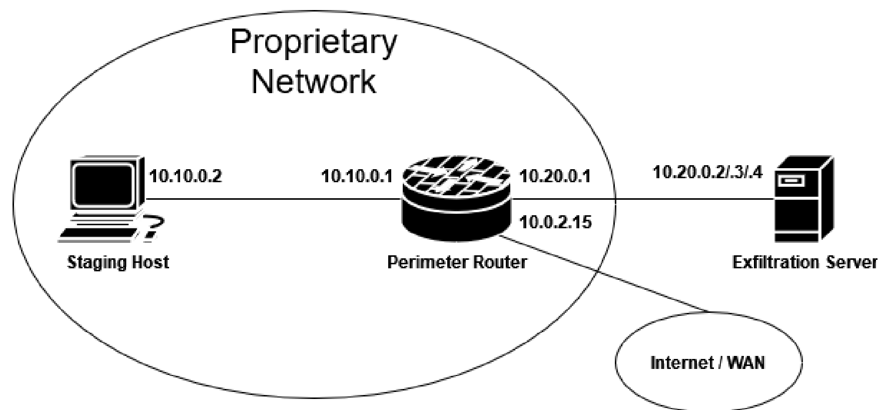


Figure 4.1. Phase One Network Topology

4.1.2 Phase Two: Exfiltration Method Testing against Host-Based Controls

Figure 4.2 depicts the network environment for Phase Two testing. For each application protocol, 13 data exfiltration test runs were completed for both files for a total of 26 runs. For these tests, the byte position was fixed to the end of the payload and the embedded-byte sizes for *constitution.txt* and *Penguin.tif* were fixed to 236 and 250 respectively. These specific configurations were chosen to streamline testing across all the application protocols since embedded-byte size and byte position do not affect the exfiltration method's ability to bypass controls hosted on the Staging Host (i.e., they do not affect the ability or inability of host-based access control to detect or inhibit exfiltration). These test runs were performed specifically to evaluate the exfiltration method's ability to bypass a host-based control designed to prevent application-based access to these specific files (i.e., AppArmor configured on the Staging Host as depicted to limit access to the exfiltration targets).

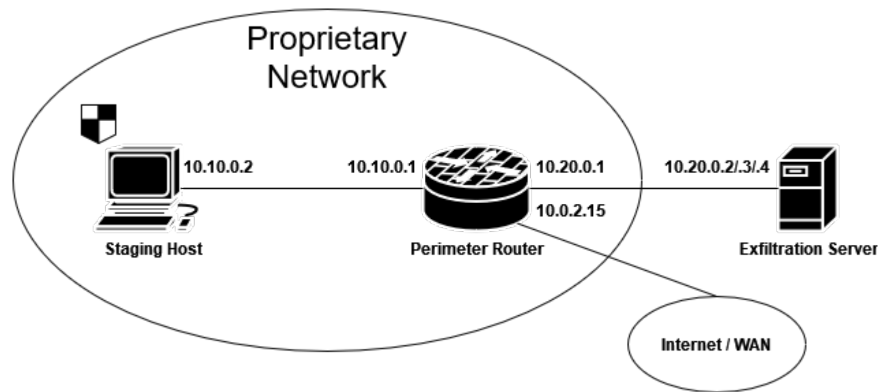


Figure 4.2. Phase Two Network Topology

4.1.3 Phase Three: Exfiltration Method Testing against Network-Based IPS

Figure 4.3 depicts the network environment for Phase Three testing. For each application protocol, two rounds of data exfiltration test runs were completed using various dynamic module configurations that were chosen based on the results of Phase One. The first round consisted of 13 tests runs per file testing various module configurations against an inline instance of the network-based IPS, Snort, with its default configuration and registered rule set (v2.9.19). The second round consisted of thirteen test runs for each file with variable embedded-byte sizes and positions for each application protocol. This round included 20 additional custom Snort rules with content modifiers designed to detect attempts to use any application protocol to exfiltrate the specific target files over UDP or TCP. With regards to the custom Snort rules, the content modifiers within each rule were designed to identify specific byte sequences within the two files. These sequences varied in size between 20 and 30 bytes. Snort was to log an alert and drop packets if these bytes were detected anywhere within the application payload of an inspected packet. These rules as written were in keeping with the best practice of creating rules with content modifiers that are specific enough to identify target data without inducing false positives. Previously disabled Snort preprocessor rules from the baseline configuration specific to the application protocols were enabled as well. These tests were used to determine whether or not the exfiltration method can be effective against an inline network-based IPS.

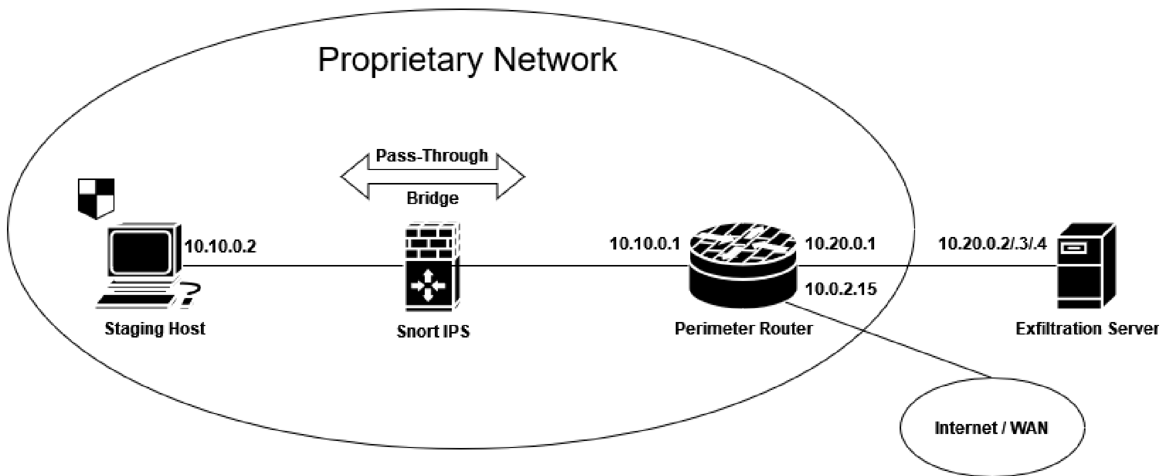


Figure 4.3. Phase Three Network Topology

4.1.4 Phase Four: Exfiltration Method Testing against a DLP Proxy Server

Finally, Figure 4.4 depicts the network topology for Phase Four testing. For each application, 13 data exfiltration test runs were intended with fixed dynamic module configurations chosen based on the results of Phases One and Three. These tests were to be performed to determine the exfiltration method's effectiveness against a robust DLP solution specifically configured to prevent the exfiltration of the targeted files.

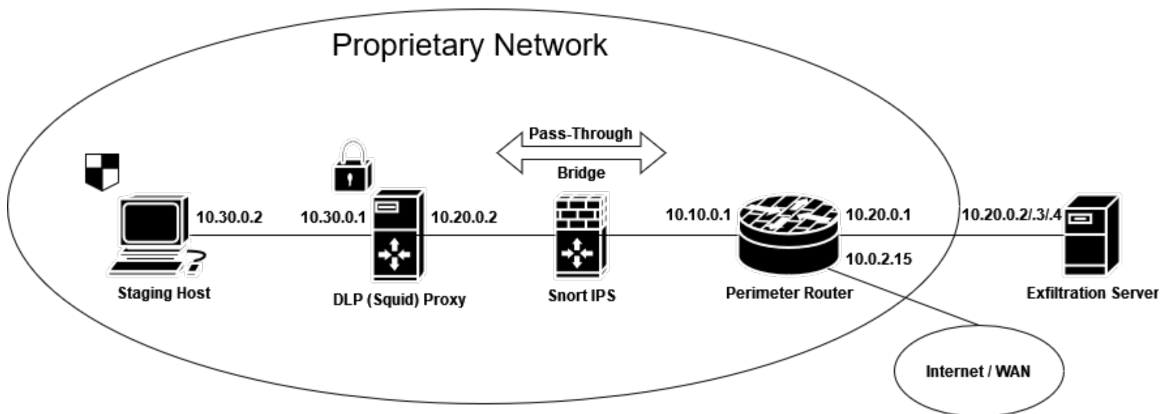


Figure 4.4. Phase Four Network Topology

4.2 Experimental Flowchart

Figure 4.5 provides a depiction of the generic experimental flow for each round of test runs for each application protocol. The process described in this section remained largely unchanged throughout all phases of testing. The steps depicted in the diagram can be described as follows:

- The *Layer4_5.ko* kernel module was the customization loader for all experiments and was loaded onto the Staging Host and Exfiltration Server. After confirming that the Layer4_5 module was loaded, the *in_rm_mod.sh* shell script was utilized to load each application-specific customization module for each round of test runs.

- The *host_config.sh* and *server_config.sh* shell scripts were executed on the Staging Host and Exfiltration Server respectively. These simple bash shell script configuration programs let the user choose to exfiltrate the data as raw bytes or in its original encoded format. They also allowed for the dynamic parameters of the modules to be reset and configured for each test run. Additionally, these shell scripts started *tcpdump* sessions on a user specified port and interface to capture traffic associated with the experiment's application protocol.

- After both the Staging Host and Exfiltration Server were configured, the application-specific services were started on the Exfiltration Server. In some cases, DNS and NTP for instance, these services were running indefinitely, but they were confirmed to be running for each test run. An application specific bash shell script program was then executed on the Staging Host to commence data exfiltration. This program performed automated application-specific transmissions to the server based on the embedded-byte size and total file size until data exfiltration was completed or a security control-based interruption occurred.

- Once the application-specific bash shell scripts concluded the *tcpdump* session was terminated on the Staging Host and analyzed to ensure that all data was exfiltrated as expected. The same process was repeated on the Exfiltration Server. At this point data exfiltration was considered successful as long as all data was seen in the capture file at the Exfiltration Server's receiving interface. Depending on the application, the customization modules could facilitate the gathering and reassembly of the data

into its original readable form. The *()_gather_data.py* and *()_reassemble_data.sh* are protocol-specific variations of script programs that were used to gather the data from the kernel trace logs and reassemble it. A MD5 hash digest was calculated for the reassembled files and compared with digests from the original files on the Staging Host. Matching digest values indicated successful exfiltration.

- If additional runs were required, the *host_config.sh* and *server_config.sh* shell scripts were simply executed again to reset and reconfigure the already loaded customization module. If no more runs were required, the *in_rm_mod.sh* shell script was executed to unload the current customization module, and the process was repeated for the next round of test runs.

4.3 Results

The experimentation conducted throughout the first three phases produced mostly expected results. In the first phase of testing, the loaded customization modules applied socket layer protocol customization in the successful exfiltration of specific file data to completion. This demonstrated a protocol-agnostic mechanism for exfiltration of data over traditionally established sockets. UDP-based application protocols outperformed their TCP counterparts with a few notable exceptions. The DNS application protocol resulted in the best performance as measured by the metrics, and the VoIP protocol displayed the most flexibility in byte position and embedded-byte size for data exfiltration.

Overall, socket layer protocol customization showed an ability to enhance obfuscated data exfiltration predominantly in the ability of customization modules to be adapted to any application protocol to bypass some basic host-based access controls. However, the customization modules in their current versions did show some limitations. The most notable limitation was the inability of the receive-side module to work correctly with applications that implement TLS (i.e., to properly handle exfiltrated data embedded into the TLS data buffers). This method, like other methods of obfuscated data exfiltration, was also susceptible to network-based security controls that employed robust content-filtering. The following sub-sections present the results of each phase of testing in detail.

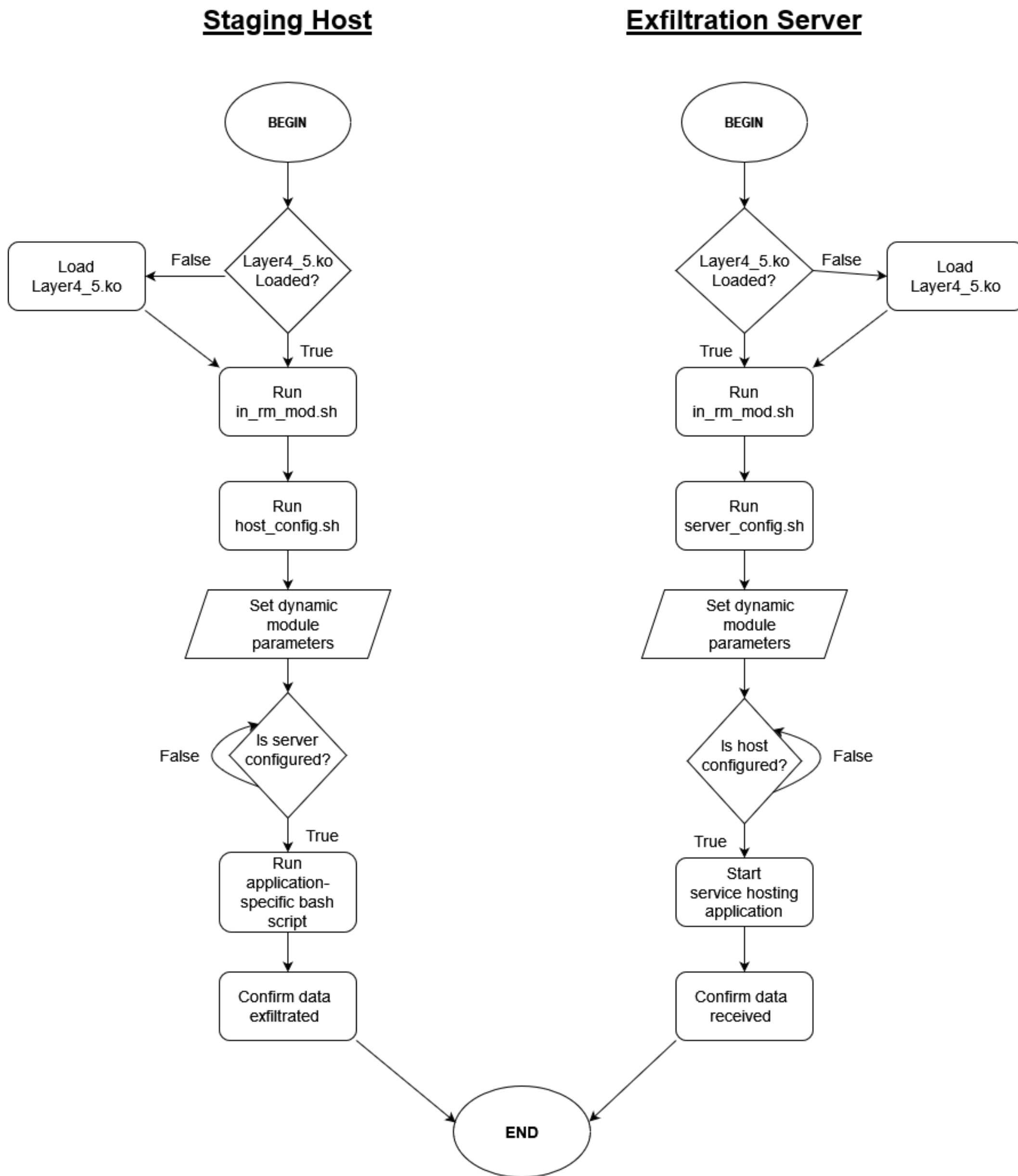


Figure 4.5. Experimental Flowchart

4.3.1 Phase One Results

The following subsections provide summaries of the Phase One results. Each subsection contains an analysis of the performance of each application protocol detailing the characteristics of each one that make it more or less conducive to this method of data exfiltration.

The analysis was conducted primarily by capturing traffic during data exfiltration using Wireshark and determining whether or not the embedded data significantly altered the application payload or displayed any notable signatures beyond the embedded data itself. The experimental results are also presented in these subsections as tables displaying the mean values for the total bytes, total packets, and total time performance metrics introduced in Section 3.3 for each application protocol. The average values were calculated over all 13 test runs for each file and embedded-data size. Finally, a series of box plots for each file are presented to provide an overall comparison of the performance of each application protocol in terms of throughput and overhead relative to the total amount of data exfiltrated (total file size).

HTTP Results Analysis

Tables 4.4 and 4.5 depict the results of experiments conducted for the HTTP protocol. Table 4.4 presents the average totals over all 13 test runs for bytes, packets, and time to the completion of data exfiltration of both files at each embedded data size. Table 4.5 presents the values of these same metrics after 500 KB were added to the server response to HTTP GET requests.

Table 4.4. HTTP Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	4.12	4.49×10^4	53.2
118 Bytes	3.90×10^{-1}	3.80×10^3	5.63
236 bytes	2.18×10^{-1}	1.90×10^3	2.37
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	42.2	4.60×10^5	544
125 Bytes	3.80	3.68×10^4	46.5
250 Bytes	2.13	1.84×10^4	21.5

The HTTP application protocol includes a variety of request methods for communication between the client and the server. For testing, two of the more common methods, GET and POST, were utilized. From a client perspective, each type of request has a particular header structure designed to communicate the details of the requesting client application and the

Table 4.5. HTTP 500 KB Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	1.07×10^3	2.32×10^5	160
118 Bytes	90.9	1.96×10^4	13.6
236 bytes	45.5	9.81×10^3	6.69
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	1.10×10^4	2.30×10^6	1.59×10^3
125 Bytes	879	1.90×10^5	145
250 Bytes	440	9.49×10^4	71.1

specific resources that are being requested. This structure presents a variety of places to embed data within the application payload. Generally, embedding data anywhere within the HTTP application payload did not lead to specific warnings from Wireshark, however, a few significant observations were made:

- Embedded data in the front of the payload essentially disguised the type of HTTP request. Therefore, any device that inspected the traffic saw unspecified data being sent over port 80 (Figure 4.6).
- Embedded data within specific header fields disguised the type of header field and that of subsequent header fields. Therefore, any device that inspected the traffic saw legitimate HTTP requests with unspecified or incorrectly formatted header fields.
- Embedded data at or near the end of a GET or POST request outside of any header field was seen as superfluous HTTP data and not part of the original request.
- Embedding data at or near the end of the payload produced the least amount of change to the structured formatting of HTTP request methods and proved to be preferable. This was especially true for POST requests where the end of the payload was variable depending on what the user was pushing to the server. This in turn would presumably make it harder to implement signature detection (Figure 4.7).

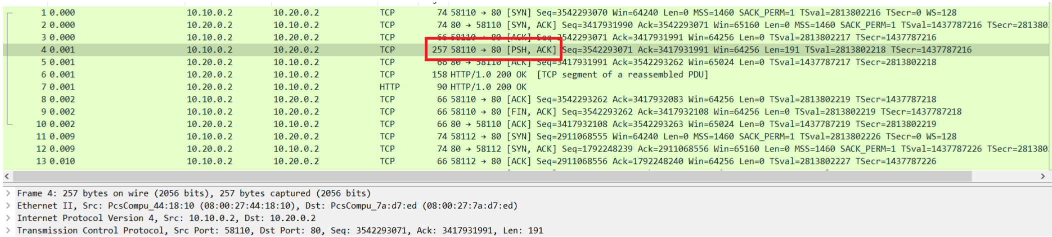


Figure 4.6. Wireshark View of Data Embedded at the Beginning of an HTTP Payload

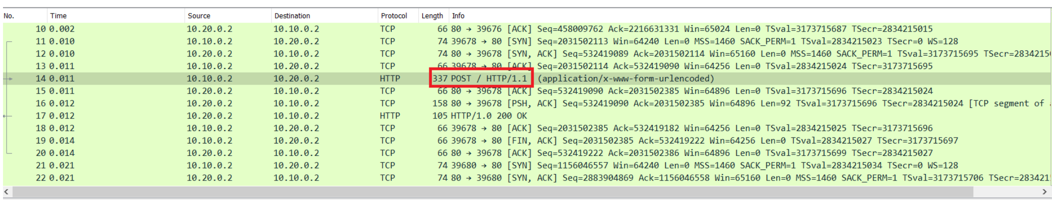


Figure 4.7. Wireshark View of Data Embedded at the End of an HTTP Payload

HTTPS Results Analysis

Table 4.6 depicts results from experiments conducted for the HTTPS protocol. This table presents the average totals over all 13 test runs for bytes, packets, and time to the completion of data exfiltration for both files at each embedded data size.

The HTTPS application protocol utilizes TLS and therefore provided more opportunities

Table 4.6. HTTPS Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	71.2	1.22×10^5	152
118 Bytes	6.08	1.04×10^4	13.6
236 bytes	3.06	5.12×10^3	5.75
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	729	1.24×10^6	1.38×10^3
125 Bytes	58.8	9.99×10^4	126
250 Bytes	29.6	4.99×10^4	61.9

to embed data within the several packets that make up a single HTTPS request. For each request a TLS handshake was required before application data could be sent over the encrypted channel. Since, TLS is implemented just below the Application Layer on the same socket connection, data could be embedded in these packets as well. However due to the previously discussed limitations associated with TLS, embedding data was limited to the end of the second-to-last packet sent by the client for a simple GET request. Since TLS specifies strict lengths within its record layer for the data it processes on the receive side, the extra embedded data at the end of the payload was essentially ignored and left untouched in memory on the Exfiltration Server while the application proceeded as normal. This caused memory issues on the server since the customization modules could not handle the buffer structure utilized by TLS, but it allowed for the performance metrics to be collected. As previously discussed, the data could not be properly accessed and reassembled from the TLS buffer for a hash verification with the original file. As a result, successful data exfiltration was measured by receipt of all data by the Exfiltration Server was verified through Wireshark analysis. Some additional tests were conducted in which data was embedded in various locations of each available packet leading to the following observations:

- Embedded data in TLS handshake packets anywhere other than the end of the payload produced formatting errors visible in Wireshark. This included the over 200 bytes of padding in the initial *Client Hello* packet.
- Embedded data within the payloads of application data packets outside the TLS record

layer and within the encrypted application data was preferable. Potentially, any device that inspects the traffic would see the embedded data as part of the original encrypted data. The exception to this would be devices configured to specifically check the *Length* value in the TLS record layer, which would reveal that the TLS payload was larger than originally calculated by the sending host (Figure 4.8).

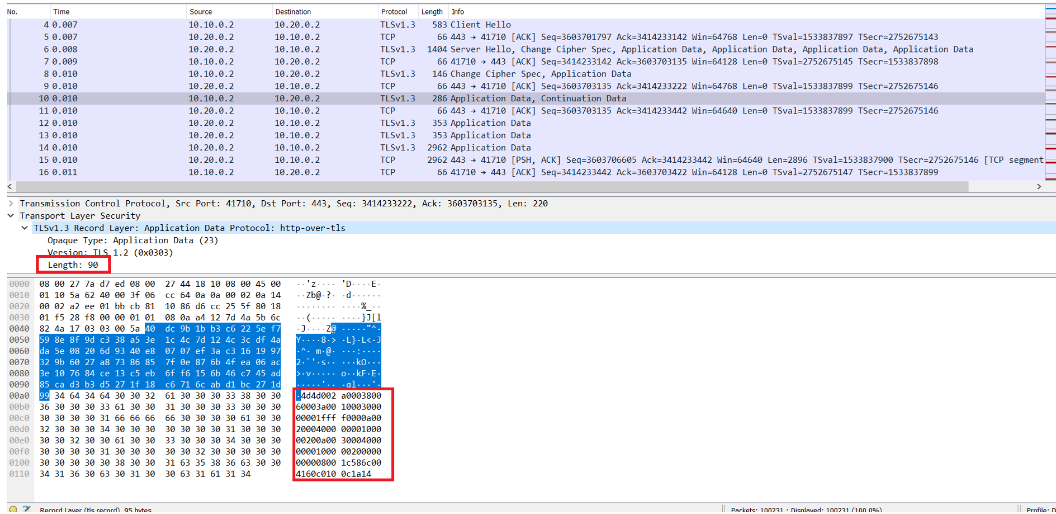


Figure 4.8. Wireshark View of Data Embedded in TLS Application Data

SMTP Results Analysis

Table 4.7 depicts results from experiments conducted for the SMTP protocol. This table presents the average totals over all 13 test runs for bytes, packets, and time to the completion of data exfiltration for both files at each embedded data size.

The SMTP application protocol also implements TLS and offered the same opportunities to embed data in various packets during data transmission. However, given the previously discussed limitations associated with TLS, the options for SMTP were limited to packets containing SMTP-protocol-specific data. Since only simple test emails were sent between servers the only two options for embedding data were the first two packets sent by the Staging Host for each transaction. These packets effectively identified the client as a Postfix server and initiated a TLS connection. These packets contained the commands *EHLO* and *STARTTLS* respectively along with an associated parameter. In the case of the *STARTTLS* packet, the parameter is limited to TLS. For this reason testing was limited to the first packet,

Table 4.7. SMTP Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	20.5	1.63×10^5	57.5
118 Bytes	1.78	1.38×10^4	4.95
236 bytes	9.12×10^{-1}	6.89×10^3	2.36
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	210	1.67×10^6	598
125 Bytes	17.2	1.33×10^5	47.2
250 Bytes	8.85	6.67×10^4	23.1

which contained the domain name of the server and could therefore be of more variable length and content. Testing led to the following key observations:

- Embedded data at the front, end, or at byte positions 1-4 and 21 was characterized by Wireshark as a data fragment. This was most likely due to the fact that when data was embedded in these byte positions the data overlapped the command or parameter fields and was therefore not recognized as legitimate.
- Embedded data in the middle of the payload was preferable because it avoided possibly corrupting the payload's command and parameter fields (Figure 4.9).
- SMTP protocol payloads are small, simple, and somewhat predictable making them less conducive to embedding larger amounts of data or specific bytes that do not conform to common command or parameter content.

DNS Results Analysis

Table 4.8 depicts results from experiments conducted for the DNS protocol. This table presents the average totals over all 13 test runs for bytes, packets, and time to the completion of data exfiltration for both files at each embedded data size.

The DNS application protocol outperformed all the other tested application protocols in terms of metrics, but was determined to be limited in the number of byte positions that could be used to embed data within the payload. From testing the following key observations were made:

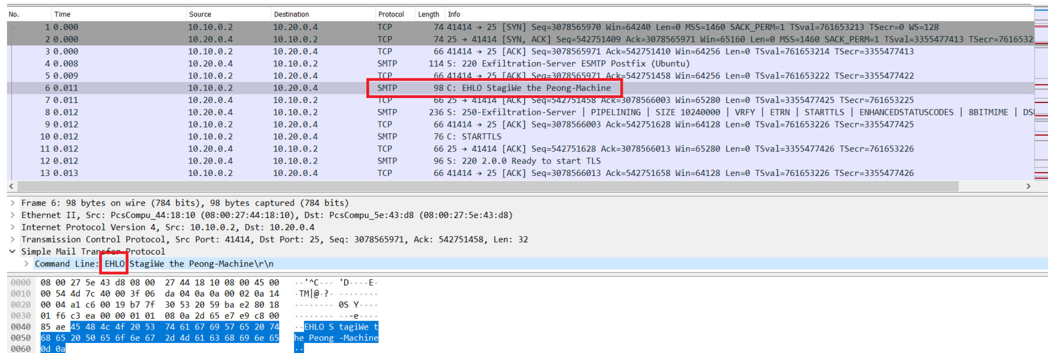


Figure 4.9. Wireshark View of Data Embedded after the SMTP Command Parameter

Table 4.8. DNS Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	9.01×10^{-1}	8.97×10^{3}	35.0
118 Bytes	1.17×10^{-1}	760	2.97
236 bytes	8.11×10^{-2}	380	1.47
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	9.24	9.19×10^{4}	403
125 Bytes	1.16	7.35×10^{3}	30.8
250 Bytes	8.11×10^{-1}	3.68×10^{3}	15.5

- Any data embedded outside the *Additional Records* field corrupted the formatting of the payload and was flagged by Wireshark as a malformed DNS packet (Figure 4.10).
- Within the *Additional Records* field itself, embedded bytes would still cause the packet to be flagged as malformed if the byte position was not within 11 bytes of the end of the payload.
- The DNS protocol appears flexible in terms of the amount of bytes that can be embedded at the end of the payload, eliciting no warnings from Wireshark up to the maximum of 250 bytes that was tested (Figure 4.11).

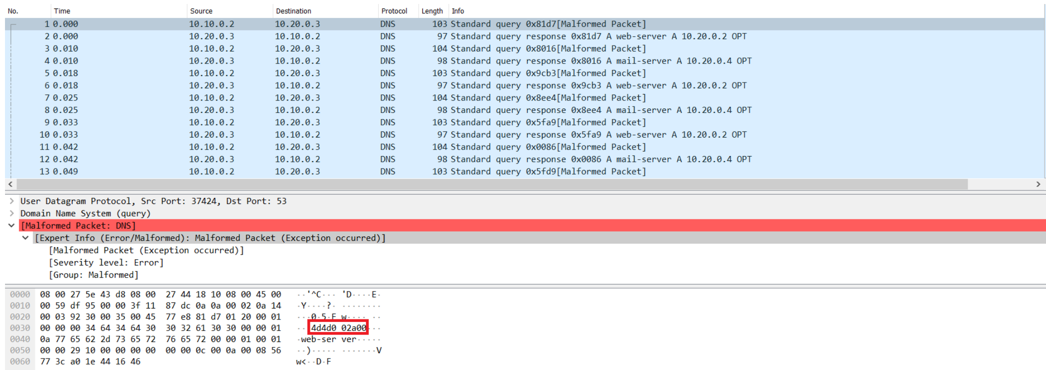


Figure 4.10. Wireshark View of Embedded Data Leading to a DNS Malformed Packet

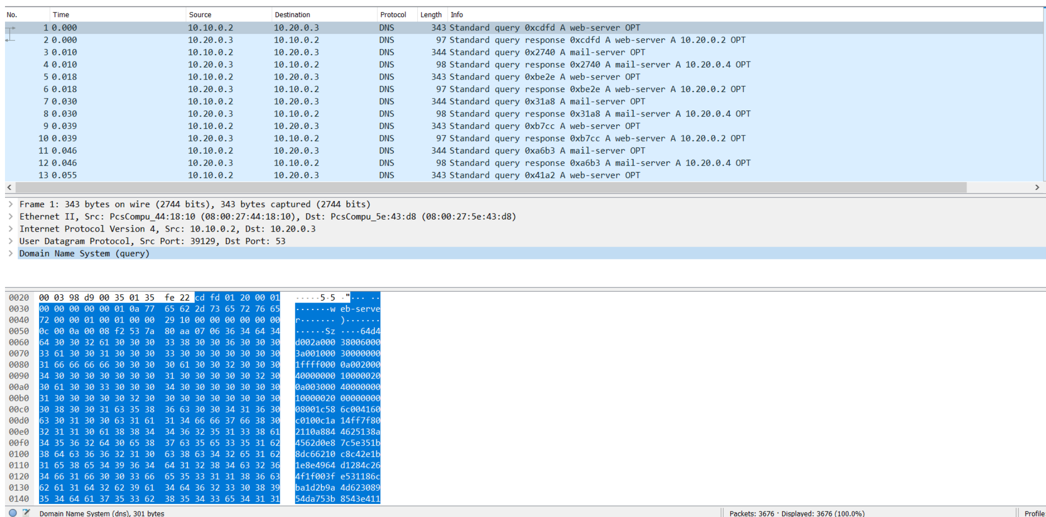


Figure 4.11. Wireshark View of Data Embedded at the End of a DNS Payload

NTP Results Analysis

Table 4.9 depicts results from experiments conducted for the NTP protocol. This table presents the average totals over all 13 test runs for bytes, packets, and time to the completion of data exfiltration for both files at each embedded data size.

The NTP application protocol, overall, induced limited overhead, but also performed the worst among all application protocols in terms of throughput. However, it proved to be fairly flexible with regards to the byte positions and embedded-byte sizes that could be utilized for embedding data. From testing the following key observations were made:

Table 4.9. NTP Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	8.52×10^{-1}	8.97×10^{3}	467
118 Bytes	1.13×10^{-1}	760	39.3
236 bytes	7.90×10^{-2}	380	20.6
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	8.73	9.19×10^{4}	4.80×10^{3}
125 Bytes	1.12	7.35×10^{3}	382
250 Bytes	7.90×10^{-1}	3.68×10^{3}	191

- Wireshark would produce irregular output when data was embedded in various fields of the NTP payload, but overall it would accept bytes where they were embedded and interpret them for what they were. For example data embedded at the very front of the packet would cause the version of NTP being utilized to be interpreted as *reserved* and would sometimes identify the mode as *server* when it was actually a standard version-4 client query. None of this output, however, induced warnings from Wireshark.
- Embedded data at or near the end of the payload was preferable as those bytes were interpreted as extra field options available in the NTP payload. During experimentation these fields were the optional fields utilized for the message authentication code. These fields can be variable in their content, which made them ideal for embedding data (Figure 4.12).
- NTP may be less conducive to data exfiltration overall because most if not all systems implement NTP with fixed polling. This polling value is usually between 64 (default) and 1024 seconds [50]. Therefore, system configuration changes for NTP would be required on the host to increase the rate of polling to exfiltrate data efficiently.

VoIP Results Analysis

Table 4.10 presents the Phase One results for the VoIP application protocol which can be categorized as a conversation incorporating multiple ports and transport protocols. This table presents the average totals over all 13 test runs for bytes, packets, and time to the

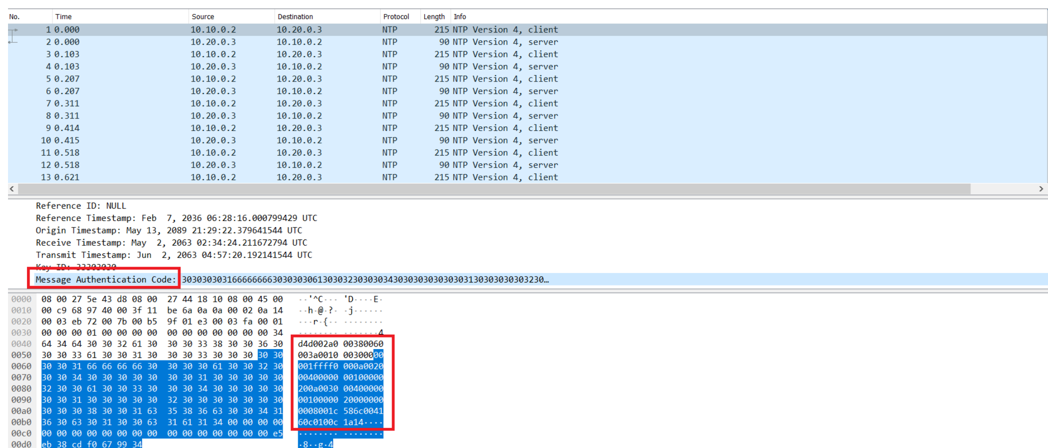


Figure 4.12. Wireshark View of Data Embedded Near the End of a NTP Payload

completion of data exfiltration for both files at each embedded data size.

Table 4.10. VoIP Mean Test Results

constitution.txt			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	7.68	9.01×10^{3}	55.7
118 Bytes	6.14×10^{-1}	732	5.43
236 bytes	2.68×10^{-1}	340	3.15
Penguin.tif			
Data Size	**Total Bytes (MB)**	**Total Packets**	**Total Time (s)**
10 Bytes	80.6	9.23×10^{4}	532
125 Bytes	6.73	7.53×10^{3}	45.0
250 Bytes	4.00	4.21×10^{3}	25.4

Of all of the application protocols tested, VoIP displayed the most flexibility in terms of byte position and embedded-byte sizes that could be utilized effectively for data exfiltration. This was primarily due to the fact that it utilizes multiple transport protocols and ports to continuously stream data between hosts. However, awareness of how the sockets are set up by the application is required to properly configure the customization modules. In the case of Linphone, four ports are utilized for streaming data, and corresponding ports are set up to receive and send on the initiating and receiving hosts respectively (and vice versa). This

limited the number of ports that could be utilized for exfiltration to two since the ports are set up in pairs. From testing the following key observations were made;

- For the Linphone application, port 9078 was preferred since it was utilized for straight UDP data transfer.
- Embedded data anywhere in the UDP payload did not induce any warnings from Wireshark since it was interpreted as innocuous UDP data (Figure 4.13).
- It was preferable to embed data at positions that accounted for the smallest packet size to avoid appending null bytes to the original payload and causing errors at the receiving application. This limited the number of effective embedded-byte positions to the size value of the smallest packet of the UDP stream, which in this case was 17.
- Embedding data only in payloads that would not exceed the 1500 MTU prevented fragmented IP protocol errors and enhanced the effectiveness of the exfiltration by avoiding fragmentation of the embedded data itself.

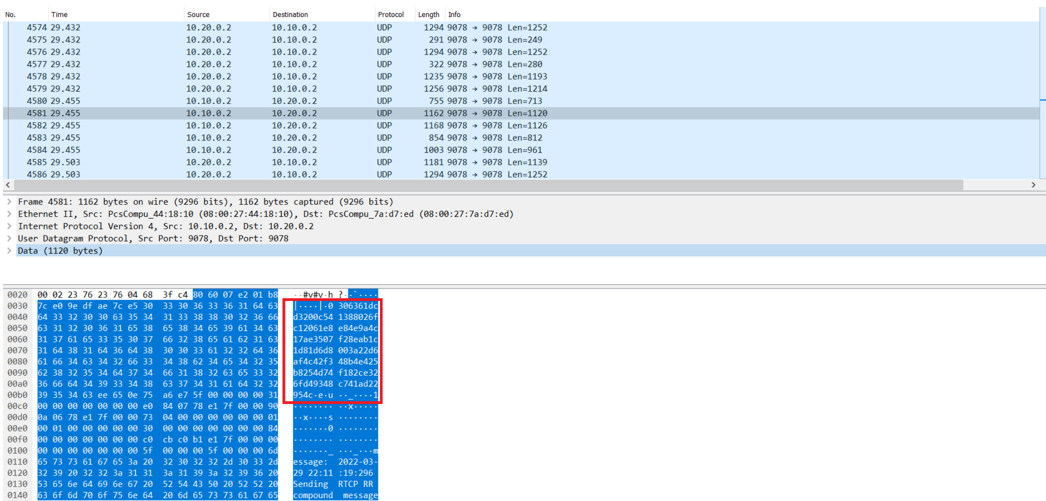


Figure 4.13. Wireshark View of Data Embedded in a VoIP Payload

Phase One Results Summary

The analysis of the exfiltration method utilizing these application protocols determined that the more structured format of TCP protocols made it more difficult to embed data within the payload than to embed data within UDP protocols. Regardless, for most of the protocols it proved preferable to utilize a byte position at or near the end of the payload.

This avoided altering the format or corrupting data fields with the embedded data, thereby reducing detectable signatures. The exception to this was VoIP, which streams audio and video data over UDP and thus has no structured format. This allowed data to be embedded at any feasible position within the packet based on the total packet size.

As the data presented in Tables 4.4 through 4.10 indicate, the UDP protocols produced less overhead than their TCP counterparts across all embedded-byte size variations. Among all the protocols tested, NTP produced the least amount of overhead and HTTPS produced the most. These results were largely expected given that TCP protocols, by design, require more overhead to ensure the reliable transmission of data. The exception was HTTP, which performed comparably to the UDP protocols in total bytes transmitted. This can be explained by the fact that the simple Python web server utilized for experimentation sent responses of no more than 21 bytes to the client. However, the additional overhead of establishing the TCP connection and acknowledging the receipt of data, did manifest in the total number of packets sent. This overhead was especially prominent with SMTP and HTTPS, both of which utilize TLS in addition to TCP. The additional set of test runs performed for HTTP using a 500 KB server response showed an increase in overhead relative to the size of the server response. Additional overhead increases the total time of exfiltration due to the fact that the client cannot initiate follow-on requests until all data from the in-progress request has been sent by the server.

With the exception of NTP, the UDP protocols also predominantly outperformed their TCP counterparts in total time to completion of data exfiltration. However, there were notable exceptions where SMTP performed comparably to VoIP in terms of total time to completion at larger embedded-byte sizes despite greater overhead. Among all protocols tested, DNS produced the shortest time to completion and NTP produced the longest time to completion. NTP's poor performance despite having the least amount of overhead can possibly be explained by the NTP protocol's design, which implements an algorithm to calculate the delay and offset of the message stream in order to transmit an accurate timestamp from the server to the client [50].

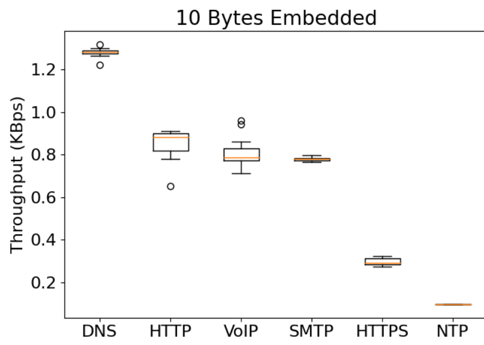
Figures 4.14 and 4.15 provide consolidated relative comparisons of the performance of each application protocol for exfiltrating both files at each embedded data size. The measurements utilized for these comparisons are relative throughput and relative overhead as introduced in

Section 3.3. As the plots indicate, the test results were largely unaffected by embedded-byte size and total file size. This result implies that the process of customization has a minimal effect on overhead or the total time of exfiltration regardless of how many customizations are required to exfiltrate an entire file. Additionally, it shows that data exfiltration via socket layer customization can be performed with any of the tested application protocols with predictable results despite varying embedded-byte sizes and increasing file sizes. VoIP was the lone exception in that it appeared to have increased throughput relative to the other protocols at smaller embedded data sizes and a larger total file size. Conversely, it displayed decreased relative throughput at higher embedded-byte sizes causing it to fall behind both HTTP and SMTP. Based on these observations, VoIP displayed the potential to have scalable and increasing relative throughput as file size increases and embedded-data size decreases. However, more tests would need to be conducted at larger file sizes to confirm this observation. A notable outlier in the results was one test involving NTP in which the total time of exfiltration was approximately 13 seconds longer than the average of the other 12 test runs. In this case the client had to resynchronize with the NTP server before performing less time-consuming NTP queries. This reinforces the notion that while performance may be predictable to a certain extent, this exfiltration method is still subject to the design and performance of the application protocol itself, even under known or controlled network conditions.

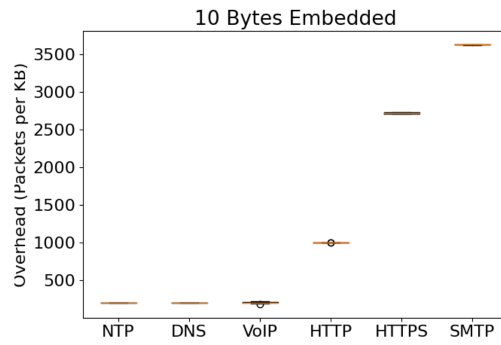
4.3.2 Phase Two Results

The results for Phase Two tests are summarized in Table 4.11. Across all test runs (26 for each application protocol), there were only two instances where exfiltration was detected and prevented by host-based access controls implemented in AppArmor. The first detection occurred during the first run utilizing the HTTP protocol and resulted from the customization module executing a system call to open and read one of the restricted files into a memory buffer in preparation for exfiltration. The second detection occurred on the first run utilizing the HTTPS protocol and resulted from the embedded data being placed within encrypted application data.

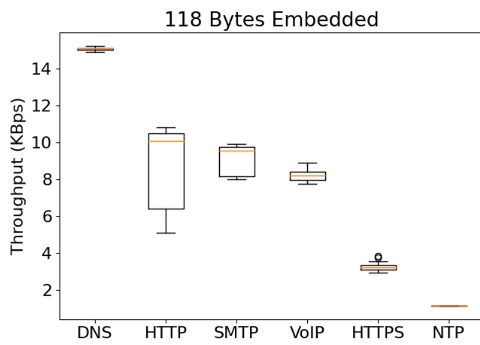
In the first instance where exfiltration was detected, the version of the customization module being utilized accesses the file to be exfiltrated after the module had been initialized and registered for Layer 4.5 customization, and after it had identified a socket for customization.



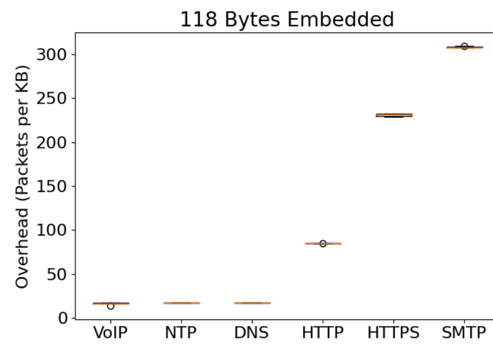
(a) Relative Throughput at 10 Bytes



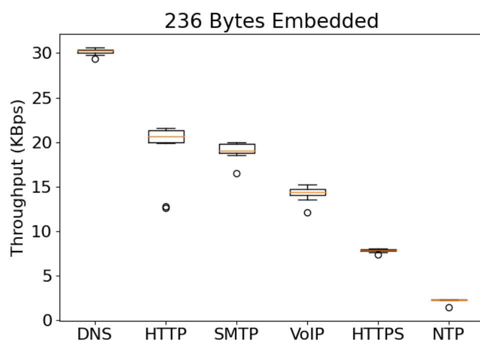
(b) Relative Overhead at 10 Bytes



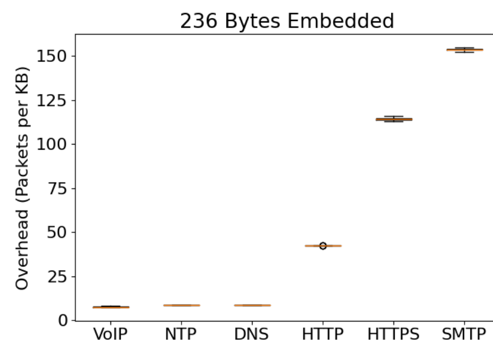
(c) Relative Throughput at 118 Bytes



(d) Relative Overhead at 118 Bytes



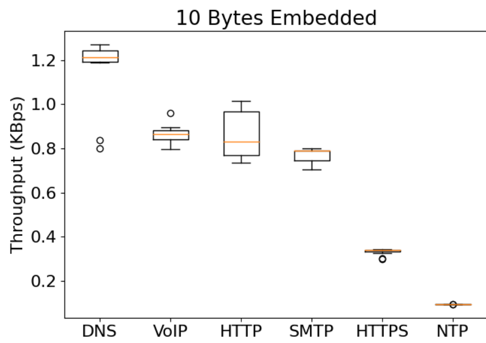
(e) Relative Throughput at 236 Bytes



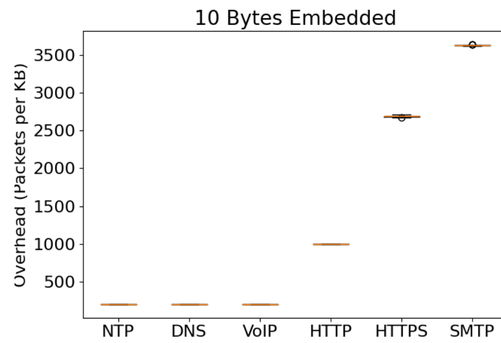
(f) Relative Overhead at 236 Bytes

Figure 4.14. Relative Results of Exfiltration for *constitution.txt* 48.5 KB

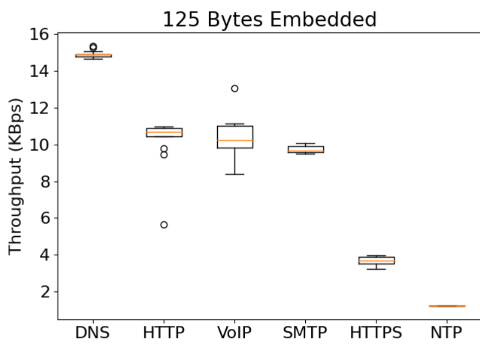
This was preferred since the file would only need to be accessed and loaded into memory when a socket connection was being customized. While customizing the socket, the module



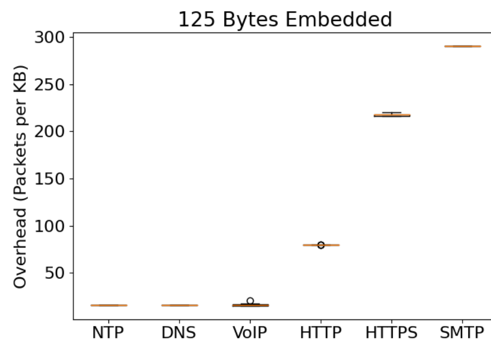
(a) Relative Throughput at 10 Bytes



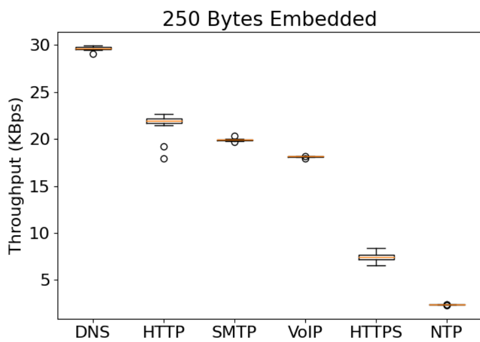
(b) Relative Overhead at 10 Bytes



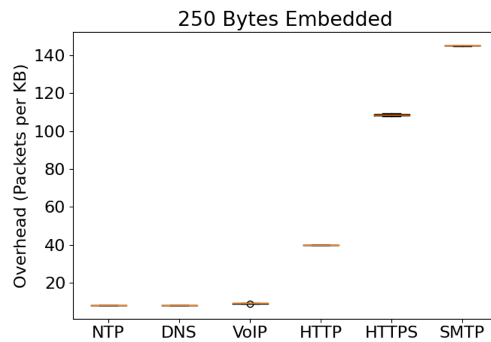
(c) Relative Throughput at 125 Bytes



(d) Relative Overhead at 125 Bytes



(e) Relative Throughput at 250 Bytes



(f) Relative Overhead at 250 Bytes

Figure 4.15. Relative Results of Exfiltration for *Penguin.tif* 459 KB

executed its functions under the process ID of the application associated with the socket. The access controls implemented by AppArmor would not allow this process to access

Table 4.11. Phase Two: Host-Based Access Control Detections

Application Protocol	Number of Detections
HTTP	1
HTTPS	1
SMTP	0
DNS	0
NTP	0
VoIP	0

the targeted file under normal circumstances. However, there was an expectation that the *filp_open* function that was utilized to access the file for exfiltration would execute at the kernel level under privilege zero and therefore bypass any implemented access controls. This turned out not to be the case, as AppArmor identified the cURL application attempting to access the file as the trace log depicted in Figure 4.16 depicts. This indicates that once the module identified a socket for customization, it executed its send and receive functions under the execution privilege of the application associated with the socket.

```
newuser@Staging-Machine: /var/log
Apr  3 14:03:24 Staging-Machine kernel: [ 6413.164423]  ? apparmor_socket_getsockopt+0x29/0x50
Apr  3 14:03:24 Staging-Machine kernel: [ 6413.164435]  ? apparmor_socket_getsockopt+0x29/0x50
Apr  3 14:03:24 Staging-Machine kernel: [ 6413.172260]  ? apparmor_file_permission+0x1c/0x20
Apr  3 14:03:24 Staging-Machine kernel: [ 6413.180122]  ? apparmor_file_permission+0x1c/0x20
```

Figure 4.16. Trace Log of AppArmor Prevention of Exfiltration over HTTP

Another version of the customization module was developed to access the file during initialization when the module was first loaded. This was a less preferred method since it placed the file data in a memory buffer before a socket was even identified for customization; however, it did allow module functions to execute with kernel-level privilege and thus bypass AppArmor's access protections. This version was also utilized with the SMTP protocol to bypass the *chroot* configuration of the Postfix server to access the files during Phase One testing. After identifying and resolving the cause of the detection, this approach was applied to the remaining protocols and prevented AppArmor from detecting restricted file access

for on any of the remaining test runs.

The cause of the second instance where exfiltration was detected was unclear. The cURL process was killed, which is the standard response when AppArmor detects a policy violation; however, the kernel log did not reference AppArmor as the reason for terminating the process. Instead, a message indicating that the process was tainted was produced (Figure 4.17), and the program was subsequently killed on the command line. This behavior only occurred when AppArmor controls were implemented, so it is likely that they are related. It could also be related to the module's inability to handle the TLS buffers properly during customization, which in combination with AppArmor controls might have produced an error when the module attempted to access the TLS buffer to embed data. Remaining HTTPS tests for this phase were conducted by embedding data only in the initial *Client Hello* packet of the TLS handshake, and no further detections occurred. This provides evidence in support of the hypothesis that the cause was linked to the module's inability to properly handle the TLS buffer structure.

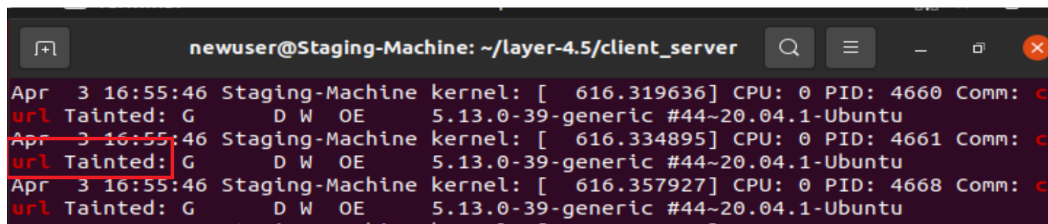


Figure 4.17. Trace Log of Anomaly Detection Prevention of Exfiltration over HTTPS

The results of Phase Two show that the advantage associated with the kernel-level privileges of the customization module can be negated by host-based access control measures if the module runs with privileges of the application once the socket has been identified for customization. However, it also demonstrates the ability of socket layer customization to bypass host-based access controls by executing functions associated with file access outside of customization-specific functions (i.e., before the access can be associated with a non-kernel-level application). In the end, these tests clearly demonstrated the inability of host-based access controls to inhibit exfiltration over any of the tested protocols.

4.3.3 Phase Three Results

Tables 4.12 and 4.13 display the results for Phase Three tests. The initial exfiltration tests against Snort's baseline configuration produced 10 alerts out of 26 runs for exfiltration attempts over HTTP. The HTTP alerts were triggered by a preprocessor rule that detected irregular traffic with POST requests where ASCII formatted data from the *constitution.txt* file was not embedded at the end of the payload. This is unsurprising given the results from Phase One that identified the end of the payload as the best placement for embedded data for exfiltration over HTTP. These alerts did not occur during testing over HTTP for any of the exfiltrated data from the *Penguin.tif* file that was encoded, regardless of byte position.

No other alerts were generated by Snort's baseline configuration and registered rule set for any of the remaining application protocols. This can potentially be explained by observing that Snort's default configuration and rule set are designed to protect primarily against external threats and therefore lack internal network rules. Detailed investigation of the rule set revealed that Snort comes with rules that are designed to protect against some basic internal threats but that they are primarily disabled by default.

Table 4.12. Phase Three: Snort IPS Baseline Configuration Detections

Application Protocol	Detections Over 26 Test Runs
HTTP	10
HTTPS	0
SMTP	0
DNS	0
NTP	0
VoIP	0

When utilizing a customized rule set, Snort was able to detect and block data exfiltration attempts of the *constitution.txt* file for every tested application protocol when the larger embedded-byte sizes of 118 and 236 were used (Figure 4.18). This was because at these byte sizes the content modifiers were always included within the embedded data. Snort was unable to detect data exfiltration for test runs utilizing the 10 byte size, which was less than the content bytes size specified in the custom rules. Snort does not have the ability to match partial content of a byte sequence or string, so separate rules would be required to account for smaller embedded-bytes sizes. The additional preprocessor rules that were enabled did

Table 4.13. Phase Three Results: Snort IPS Customized Rule Set Detections

HTTP				**HTTPS**			
File	**Data Size**	**Byte Position**	**Detected**	**File**	**Data Size**	**Byte Position**	**Detected**
constitution.txt	10	END	FALSE	*constitution.txt*	10	END	FALSE
	118	45	TRUE		118	320	TRUE
	236	32	TRUE		236	375	TRUE
	10	END	FALSE		10	310	FALSE
	118	END	TRUE		118	END	TRUE
	236	END	TRUE		236	END	TRUE
Penguin.tif	10	END	FALSE	*Penguin.tif*	10	END	FALSE
	125	58	FALSE		125	376	FALSE
	250	32	FALSE		250	450	FALSE
	10	58	FALSE		10	376	FALSE
	125	END	FALSE		125	415	FALSE
	250	END	FALSE		250	320	FALSE
	10	32	FALSE		10	450	FALSE
SMTP				**DNS**			
File	**Data Size**	**Byte Position**	**Detected**	**File**	**Data Size**	**Byte Position**	**Detected**
constitution.txt	10	END	FALSE	*constitution.txt*	10	END	FALSE
	118	5	TRUE		118	47	TRUE
	236	END	TRUE		236	50	TRUE
	10	18	FALSE		10	49	FALSE
	118	8	TRUE		118	END	TRUE
	236	13	TRUE		236	END	TRUE
Penguin.tif	10	END	FALSE	*Penguin.tif*	10	END	FALSE
	125	5	FALSE		125	END	FALSE
	250	11	FALSE		250	47	FALSE
	10	6	FALSE		10	47	FALSE
	125	END	FALSE		125	49	FALSE
	250	END	FALSE		250	END	FALSE
	10	15	FALSE		10	49	FALSE
NTP				**VoIP**			
File	**Data Size**	**Byte Position**	**Detected**	**File**	**Data Size**	**Byte Position**	**Detected**
constitution.txt	10	END	FALSE	*constitution.txt*	10	END	FALSE
	118	12	TRUE		118	5	TRUE
	236	35	TRUE		236	11	TRUE
	10	35	FALSE		10	3	FALSE
	118	END	TRUE		118	6	TRUE
	236	END	TRUE		236	END	TRUE
Penguin.tif	10	END	FALSE	*Penguin.tif*	10	END	FALSE
	125	3	FALSE		125	3	FALSE
	250	8	FALSE		250	13	FALSE
	10	35	FALSE		10	5	FALSE
	125	END	FALSE		125	7	FALSE
	250	END	FALSE		250	15	FALSE
	10	22	FALSE		10	7	FALSE

not result in detections for any of the tested application protocols. Despite the ineffectiveness of default rules and preprocessor rules, consistent detection across applications as a result of

the customized rule set indicates that obfuscated data exfiltration via socket layer protocol customization can be defeated by robust content-filtering.

```
04/23-13:28:56.488584  [Drop] [**] [1:10000008:1] !!!!!!!! DATA EXFILTRATION RU
LE !!!!!!!!! [**] [Priority: 0] {UDP} 10.10.0.2:35314 -> 10.20.0.3:53
04/23-13:29:03.183407  [Drop] [**] [1:10000009:1] !!!!!!!! DATA EXFILTRATION RU
LE !!!!!!!!! [**] [Priority: 0] {UDP} 10.10.0.2:41099 -> 10.20.0.3:53
```

Figure 4.18. Detection and Prevention of Exfiltration over DNS by Customized Snort Rules

Unexpectedly, Snort was unable to detect any data exfiltration attempts of the *Penguin.tif* file. At first it was not initially clear why, since the content modifier rules accounted for the fact that the file was being exfiltrated as encoded binary. A closer comparison of the Snort content rule bytes with the embedded bytes, however, revealed that the byte positions were equal but reversed. Further investigation revealed that the `xxd` program utilized to encode the original file for exfiltration provided hexadecimal output in big-endian format while `hexdump`, which was utilized to develop the Snort content-modifier rules, output the file contents in little-endian format. Although unintentional, these results emphasize that while in-line content filtering and signature detection can be effective tools against data exfiltration, the advantage still lies with the malicious actor. The defender must account for all of the ways in which an adversary might try to exfiltrate data while the malicious actor simply has to find one way that is not addressed. This might be easily accomplished through a custom encoding scheme designed for obfuscation that mimicks payloads of a specific application protocol. Data exfiltration via socket layer customization can leverage this advantage by providing flexibility to use arbitrary applications for exfiltration and to modify where and how much data is embedded within an application payload. Beyond this additional flexibility, however, the socket layer customization modules in their current form present no inherent ability to bypass network-based security controls.

4.3.4 Phase Four Results

Thesis timeline constraints prevented meaningful Phase Four test results from being obtained. Implementation of Squid as a transparent, inline proxy server capable of receiving, inspecting, and forwarding traffic on all ports proved difficult. Typically, when Squid is deployed inline for content filtering, it is utilized as a transparent proxy for web-based

traffic only. The desired configuration for this thesis was to deploy Squid for this purpose to perform signature-based detection via ClamAV on all web traffic while forwarding all other traffic. Unfortunately, there were issues configurating Squid to effectively segregate some traffic while simply forwarding other traffic. Additionally, the value of performing these tests decreased as a result of the issues associated with customization modules for TLS since the goal was to evaluate the data exfiltration against a proxy server that decrypts and inspects HTTPS traffic. The alternative was to forgo the use of Squid to segregate traffic of interest and to simply inspect all traffic against the custom ClamAV signatures. Unfortunately, a compatible data acquisition library or generic solution utilizing open-source resources for packet capture and retransmission could not be identified and tested in a timely manner. Nevertheless, this phase of testing still has the potential to provide meaningful results in evaluating this method of data exfiltration in future work once the issues regarding TLS implementation with the customization modules are resolved.

4.4 Summary

In this chapter, the experimental configurations and settings for each phase of testing were discussed in detail. More specifically, the various virtual machine configurations, customization module, and application configurations were presented to clarify what settings were implemented to test each individual application protocol with the exfiltration method for each phase of testing. The network environments were presented and discussed as well in order to clarify how data was transmitted from the Staging Host to the Exfiltration Server during each testing phase. Additionally, the number of test runs conducted and significant settings for each phase were presented in order to frame the scope of the results. The chapter then discussed the logical flow of experimentation from loading of the customization module to successful exfiltration of a file.

Finally, the results were presented for each phase of testing. Phase One testing showed that exfiltration via socket layer protocol customization could be applied successfully to any application protocol. The UDP-based application protocols with less overhead and less structured format proved more conducive to this method of exfiltration in general when compared to the TCP-based protocols. Phase Two results showed that the exfiltration method could successfully bypass basic host-based access controls regardless of application protocol. Phase Three results revealed that the exfiltration method is susceptible to network

based controls implementing robust content-filtering; however, it was surmised that simple encoding or encryption techniques could be utilized to bypass these controls. Phase Four experiments were not conducted due to time-constraints and issues associated with integrating the customization modules with applications implementing TLS.

Chapter 5 will present the broad conclusions of this work, discuss potential mitigations targeting this form of data exfiltration, and make recommendations concerning future work in this area.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion

This thesis investigated the viability of applying recent research associated with socket layer protocol customization to improve upon traditional methods of obfuscated data exfiltration. A simple experimental network was used to simulate data exfiltration between a host residing on an internal proprietary network and an external server. Various security controls were implemented in separate phases of testing to evaluate the effectiveness of the exfiltration method. In this chapter we present the overall conclusions of the research, discuss limitations of the exfiltration modules that were identified during testing, recommend mitigations that might address the exfiltration vulnerability, and suggest future work to build upon the results of this thesis.

5.1 Research Conclusions

The primary conclusions drawn from this research are listed below. Each numbered conclusion addresses the correlating numbered research question presented in Chapter One.

1. The results from Phase One showed that application protocols utilizing UDP are more conducive to obfuscation of embedded data when used in concert with socket-layer protocol customization than TCP protocols. Predominantly, UDP protocols were shown to require less overhead and had less structured formats than their TCP counterparts and therefore allowed for more variation in byte position and byte size for the embedded data. The VoIP protocol in particular stood out for its flexible format and the ease with which various sizes of data could be embedded within application payloads streamed as UDP data over multiple ports. DNS also allowed for a highly variable data to be appended at the end of its payload without inducing warnings from tools such as Wireshark. NTP also did not induce warnings when its data was supplanted by embedded data, and its protocol incorporates options at the end of its payload for a message authentication code that often goes unused. In contrast, the TCP protocols such as HTTP have very specific formats for the different types of requests initiated by the client, which limited the placement of exfiltrated data bytes

to the end of the payload. SMTP and HTTPS both implement TLS, which increases the number of available packet types available for customization in a single TCP transaction, but limits the ability to embed and extract data because of nature of the TLS buffers. Based on results of all phases of experimentation, the VoIP protocol displayed the most potential to be utilized effectively with this exfiltration method and should therefore be a focus for future research.

2. Results from Phase Two demonstrated that the customization modules can execute functions with kernel-level privilege during initialization and before customization. Kernel-level privilege enabled them to bypass basic host-based discretionary access control and mandatory access control policies that might be implemented on restricted files targeted for exfiltration. Additionally, the results showed that functions within the module that are executed once a socket was identified for customization did not maintain their kernel-level privilege and instead were executed under the privilege of the associated application. This added a layer of obfuscation to the customization module's execution during exfiltration, but also prevented modules from executing privileged functions to access restricted files during customization. The conclusion drawn from this observation is that the placement of functions that must access restricted files requires careful consideration to bypass basic host-based controls.
3. Much like traditional methods of obfuscated data exfiltration, the method tested in this work was susceptible to detection by network-based security controls that implement robust content-filtering, particularly if an organization develops a content matching scheme that can identify restricted files based on a relatively small number of bytes. It was concluded, however, that the customization modules might be developed to embed a byte size that is too small to detect or in an application payload location that would be undetectable by a content-filtering solution. This would likely prove difficult since it would require in-depth knowledge of the controls that are in place for the proprietary network.
4. Finally, the results across all experimentation phases showed that data exfiltration via socket layer protocol customization is a viable method that potentially improves upon traditional methods of obfuscated exfiltration. This method is protocol-agnostic and offers enough flexibility to tailor configurations for a wide variety scenarios. A customization module can be configured for a very specific socket connection to target a particular application, for instance, or it can be configured with more

generic options to target ports or IP addresses of interest more broadly. This gives a malicious actor a variety of options to establish covert channels for exfiltration. The modules also do not establish the sockets themselves; they merely customize the data that is being transferred over application sockets. Therefore, data is exfiltrated over legitimately established socket connections that are more likely to be treated permissively by network security measures. The customization modules also allow for data of any size to be embedded anywhere within a packet making it easy to adapt it to the formats of certain types of application-specific traffic.

5.1.1 Limitations

The customization modules that enable the socket layer protocol customization for this exfiltration method are not without limitations in their current form. The primary limitations that were identified through research and experimentation conducted for this thesis are as follows:

1. As previously discussed, the customization modules are unable to properly account for the buffer structures and record layer implemented by TLS when embedding and extracting data. Since most network traffic these days is encrypted, this represents a significant limitation for utilizing this exfiltration method with TCP-based application protocols in real-world network environments.
2. The customization modules do not yet have the ability to verify the receipt of exfiltrated data. If UDP packets are lost or corrupted for instance, the sending module would not know to resend them or abstain from further customization. Similarly for TCP-based transmissions, different applications can access the TCP buffer in unexpected ways resulting in the modules performing customization improperly. This might lead to errors affecting the legitimate transmission of data between the sending and receiving hosts.
3. The customization modules do not have inherent functions implemented to encode or encrypt data. In their current form all encoding or encrypting has to be done prior to the module accessing the file for exfiltration.

5.1.2 Mitigation Recommendations

Fortunately, even though this work demonstrates that this method of exfiltration can be effective, it does require loading a kernel-level module, which normally requires root-level access. In this sense the customization module is similar to a traditional rootkit [84]. There are known methods to prevent and detect rootkits, but within the scope of this thesis there are a few specific mitigations that might prevent or detect this particular method from exfiltrating data if a module does somehow get installed on a host system.

1. One possible mitigation would be to provide priority alerts if unregistered or restricted kernel-level modules are loaded onto system. These alerts are already generated locally on the system, but it would be ideal to have it sent to a remote server in case an intruder was able to erase or modify the local logs. Host-based controls could be implemented to disconnect the machine from the network since this type of alert would indicate a root-level intrusion.
2. Another possible mitigation for this particular method of exfiltration would be to generate alerts for restricted files that are accessed with root- or higher-level access. This sort of access would be highly unusual in a network where only internal user accounts typically access these files and would therefore be highly indicative of malicious activity.
3. Based on results from Phase Three tests, robust content filtering rules for specific files could be employed to detect restricted file content embedded within application payloads. Phase Four testing was not completed, but appending byte signatures to restricted files that can be detected by antivirus or other host-based controls might potentially provide additional detection capability [6], [87].

Notwithstanding the potential effectiveness of these mitigations, the vulnerability that is taken advantage of by this technique is more fundamental. Specifically, there is no mechanism to ensure the integrity of data across network stack layers. Ideally, a way to ensure the integrity of the application payload data as it transits the TCP/IP stack, specifically when it transitions from layer 5 to layer 4 (where the customization module is currently able to embed data without detection) would provide a more thorough mitigation. TLS tries to accomplish this through its record layer and strict buffer structures; however, those can still be manipulated by the customization module and do not account for applications that do not utilize TLS. This type of capability would fundamentally change the design of TCP/IP and

might impose unacceptable overhead requirements. A more realistic solution might involve cross-layer monitoring that generates alerts when actions that manipulate application payload buffers are taken. In addition, future research could be conducted to identify possible mitigation techniques at layer 4.5.

5.2 Future Work

This thesis showed that socket layer protocol customization can enhance the effectiveness of traditional methods of data exfiltration. Associated future work should include further evaluation of the effectiveness of this exfiltration method as well as exploration of additional uses of socket layer protocol customization to provide further capabilities. A suggested list of specific potential future research topics follows.

- Develop and evaluate customization modules capable of integrating with application protocols that implement TLS. The TLS record layer and buffer structures represent unique obstacles to data exfiltration using socket layer protocol customization. Efforts should be made to understand how TLS allocates memory for buffer structures and how it initializes values for its record layer to determine how these values can be effectively manipulated.
- Develop customization modules that make use of the customization send and receive functions on both the Staging Host and the Exfiltration server. Modules that make use of both these functions simultaneously could potentially establish a command and control channel capable for near-real-time parameter modification during exfiltration of a target file. These functions might also be used to facilitate the confirmation of the receipt of exfiltrated data between the sending and receiving hosts to make them less vulnerable to dropped or corrupted data.
- Implement symmetric or hybrid encryption within the customization modules to further enhance the exfiltration method. This could potentially allow the modules to more effectively bypass network-based controls utilizing content-filtering and signature detection.
- Perform experimentation of the exfiltration method over the Internet using real-world infrastructure to assess performance and effectiveness. This will allow determination of whether or not the method is still viable across all application protocols in a more dynamic and realistic network environment.

- Test the exfiltration method from different host roles and perspectives. This could include exfiltration from a proprietary server to a client or the use of customization at an en route network node to extract the data before it reaches its final destination.

A final recommendation for future work would involve researching ways to detect or prevent this exfiltration method by utilizing socket protocol customization as the inhibitor instead of the enabler. This might involve intercepting application data as it leaves layer five, calculating a hash digest, and then verifying it before it transitions to layer four. The flexibility and capability inherent in socket layer protocol customization makes it a prime candidate for future research and development.

5.3 Summary

In summary, socket layer protocol customization shows the ability to enhance traditional methods of obfuscated data exfiltration through its ability to leverage unrestricted kernel-level access to manipulate data as it transitions from the application layer to the transport layer. The method is also flexible enough with regards to identifying sockets that it can potentially be utilized with any application protocol. The results of this research show that this method can be successfully utilized to bypass some basic host- and network-based security controls to consistently and successfully embed restricted file data into legitimate application payloads for transmission to an external host. Some basic mitigation recommendations are made based on the experimentation results to prevent or at least detect this method of exfiltration. In addition, the customization loader and modules do display some limitations in their current forms. These limitations were discussed in the context of suggestions for future research to further understand the capabilities of this exfiltration method.

APPENDIX:

A.1 Network Configurations and Source Code

The configuration files and source code for the various virtual hosts, applications, programs, and bash shell scripts utilized for experimentation in this thesis can be found consolidated on NPS GitLab at the following link: https://gitlab.nps.edu/eric.bergen/data-exfiltration-layer-4-5. This includes the source code and files for the Layer 4.5 architecture and the various versions of the customization modules for each application protocol.

A.2 Script Descriptions

Several bash and Python scripts were created to provide a level of automation for configuring the modules between test runs and executing the individual tests. Scripts were also created to gather and reassemble the exfiltrated data on the receive side for evaluation. These scripts helped to streamline the execution of tests for each application to help prevent any configuration errors between runs.

A.2.1 Bash Shell Scripts

- **in_rm_mod.sh:** Load or unload a customization module for quick transitions between test rounds and individual test runs. This script is very useful during test runs against security controls where the module must be loaded and unloaded between runs to load the file to be exfiltrated into memory. This script can be easily chained with the configuration bash scripts below for easy transitions during these test runs.
- **host_config.sh:** Configures the customization module for the client-side host and prepares file for exfiltration. The script allows for the specification of the file to be exfiltrated and the choice to encode it using `xxd` or leave it as is. Directs user input for the customization module's configurable parameters. The script also indicates whether the file will need to be segmented and performs user-directed segmentation to perform exfiltration if the file is over 1.0 MB. The script will also start a *tcpdump* instance to capture traffic on the module's configured destination port.

- **server_config.sh:** Configures the customization module for the server-side host. The script directs user input for the customization module's configurable parameters. The script will also start a *tcpdump* instance to capture traffic on the module's configured destination port.
- **clear_trace.sh:** Clears the kernel trace log after exfiltrated data has been copied from the log file.
- **text_reassemble.sh:** Reassembles exfiltrated data from text files into original format.
- **tif_reassemble.sh:** Reassembles exfiltrated data from TIFF files into original format.
- **jpg_reassemble.sh:** Reassemble exfiltrated data from JPEG files into original format.

A.2.2 Python Scripts

- **text_gather_data.py:** Parses the kernel trace log for exfiltrated data and writes it to a designated file. The script then calls *text_reassemble.sh* and *clear_trace.sh* as sub-processes.
- **tif_gather_data.py:** Parses the kernel trace log for exfiltrated data and writes it to a designated file. The script then calls *tif_reassemble.sh* and *clear_trace.sh* as sub-processes.
- **jpg_gather_data.py:** Parses the kernel trace log for exfiltrated data and writes it to a designated file. The script then calls *jpg_reassemble.sh* and *clear_trace.sh* as sub-processes.

List of References

[1] MITRE Corporation, "Exfiltration," Jul. 2019 [Online]. Available: https://attack.mitre.org/tactics/TA0010/

[2] J. Collins and S. Agaian, "Trends toward real-time network data steganography," *International journal of network security and its applications*, vol. 8, no. 2, pp. 1–21, 2016.

[3] E. Bertino and G. Ghinita, "Towards mechanisms for detection and prevention of data exfiltration by insiders: Keynote talk paper," *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 10–19, 2011.

[4] F. Ullah, M. Edwards, R. Ramdhany, R. Chitchyan, M. A. Babar, and A. Rashid, "Data exfiltration: A review of external attack vectors and countermeasures," *Journal of Network and Computer Applications*, vol. 101, pp. 18–54, 2018.

[5] R. Chesney, "Cybersecurity law, policy, and institutions (version 3.0)," *U of Texas Law, Public Law Research Paper*, no. 716, 2020.

[6] G. J. Silowash, T. Lewellen, D. L Costa, and T. B. Lewellen, "Detecting and preventing data exfiltration through encrypted web sessions via traffic inspection," *Software Engineering Institute*, 2013.

[7] S. Tari, "SSLproxy - transparent SSL/TLS proxy for decrypting and diverting network traffic to other programs for deep SSL inspection," Nov. 2021 [Online]. Available: https://github.com/sonertari/SSLproxy

[8] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of IP flow-based intrusion detection," *IEEE Communications Surveys Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[9] IBM Corporation, "Socket programming," 2010 [Online]. Available: https://www.ibm.com/docs/en/i/7.1?topic=communications-socket-programming

[10] D. M. Charjan, P. M. Bochare, and Y. R. Bhuyar, "An overview of secure sockets layer," *Int. J. Comput. Sci. Appl*, vol. 6, no. 2, pp. 388–393, 2013.

[11] D. Lukaszewski, "Systematic customization of network protocols with layer 4.5," 2021, Dissertation Proposal, unpublished.

[12] D. Lukaszewski and G. Xie, "Towards software defined layer 4.5 customization," in *IEEE 8th International NetSoft Conference*, 2022.

[13] D. Lukaszewski and G. Xie, “Demo: Towards software defined layer 4.5 customization,” in *IEEE 8th International NetSoft Conference*, 2022.

[14] D. Lukaszewski, *Software for Layer 4.5 Customization Framework*. Available: https://github.com/danluke2/software_defined_customization

[15] D. K. Hong, Q. A. Chen, and Z. M. Mao, “An initial investigation of protocol customization,” in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017, pp. 57–64.

[16] M. Sjoholmsierchio, B. Hale, D. Lukaszewski, and G. Xie, “Strengthening SDN security: Protocol dialecting and downgrade attacks,” in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. IEEE, 2021, pp. 321–329.

[17] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, “TRIMMER: Application specialization for code debloating,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 329–339.

[18] C. Qian, “REDUCING SOFTWARE’S ATTACK SURFACE WITH CODE DEBLOATING,” Ph.D. dissertation, Georgia Institute of Technology, 2021.

[19] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, “IP options are not an option,” Technical report, EECS Department, University of California, Berkeley, Tech. Rep., 2005.

[20] F. Gont, J. Linkova, T. Chown, and W. Liu, “Observations on the dropping of packets with IPv6 extension headers in the real world,” RFC 7872, RFC Editor, Tech. Rep., 2016.

[21] R. Zullo, T. Jones, and G. Fairhurst, “Overcoming the Sorrows of the Young UDP Options,” in *2020 Network Traffic Measurement and Analysis Conference (TMA), IEEE*, 2020.

[22] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, “The QUIC transport protocol: Design and internet-scale deployment,” in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[23] Oracle, “Programming Interfaces Guide, Chapter 7: Socket Interfaces,” 2010 [Online]. Available: https://docs.oracle.com/cd/E19683-01/816-5042/sockets-85885/index.html

[24] CompTIA, Inc., “What Is a Network Protocol, and How Does It Work?” 2022 [Online]. Available: https://www.comptia.org/content/guides/what-is-a-network-protocol

[25] IETF, “Mission and principles,” 2022 [Online]. Available: https://www.ietf.org/about/who/

[26] CsPsProtocol, “What is the TCP/IP Model?” 2022 [Online]. Available: https://www.cspsprotocol.com/tcp-ip-model/

[27] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1,” Internet Requests for Comments, RFC Editor, RFC 2616, June 1999, http://www.rfc-editor.org/rfc/rfc2616.txt. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[28] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2),” Internet Requests for Comments, RFC Editor, RFC 7540, May 2015, http://www.rfc-editor.org/rfc/rfc7540.txt. Available: http://www.rfc-editor.org/rfc/rfc7540.txt

[29] R. Khare and S. Lawrence, “Upgrading to TLS Within HTTP/1.1,” Internet Requests for Comments, RFC Editor, RFC 2817, May 2000.

[30] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” Internet Requests for Comments, RFC Editor, RFC 5246, August 2008, http://www.rfc-editor.org/rfc/rfc5246.txt. Available: http://www.rfc-editor.org/rfc/rfc5246.txt

[31] D. Benjamin, “Using TLS 1.3 with HTTP/2,” Internet Requests for Comments, RFC Editor, RFC 8740, February 2020.

[32] Python Software Foundation, “General Python FAQ,” 2022 [Online]. Available: https://docs.python.org/3/faq/general.html

[33] Python Software Foundation, “Active Python Releases,” 2022 [Online]. Available: https://www.python.org/downloads/

[34] A. Zarubin, “Simple Python HTTP(S) Server — Example,” Mar. 2016 [Online]. Available: https://blog.anvileight.com/posts/simple-python-http-server/

[35] The Apache Software Foundation, “Apache HTTP Server Project,” 2022 [Online]. Available: https://httpd.apache.org/

[36] The Apache Software Foundation, “Downloading the Apache HTTP Server,” Mar. 2022 [Online]. Available: https://httpd.apache.org/download.cgi

[37] J. Ellingwood, “How To Configure the Apache Web Server on an Ubuntu or Debian VPS,” Mar. 2022 [Online]. Available: https://httpd.apache.org/download.cgi

[38] D. Stenberg, "Curl: Command line tool and library for transferring data with URLs," Mar. 2022 [Online]. Available: https://curl.se/

[39] J. Klensin, "Simple Mail Transfer Protocol," Internet Requests for Comments, RFC Editor, RFC 5321, October 2008, http://www.rfc-editor.org/rfc/rfc5321.txt. Available: http://www.rfc-editor.org/rfc/rfc5321.txt

[40] B. G. Schlicher, L. P. MacIntyre, and R. K. Abercrombie, "Towards reducing the data exfiltration surface for the insider threat," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2016, pp. 2749–2758.

[41] W. Z. Venema, "The Postfix Home Page," 2022 [Online]. Available: https://www.postfix.org/start.html

[42] W. Z. Venema, "Postfix Source Code," 2022 [Online]. Available: http://cdn.postfix.johnriley.me/mirrors/postfix-release/index.html

[43] X. Guoan, "Build Your Own Email Server on Ubuntu: Basic Postfix Setup," Mar. 2022 [Online]. Available: https://www.linuxbabe.com/mail-server/setup-basic-postfix-mail-sever-ubuntu

[44] P. Mockapetris, "Domain names - implementation and specification," Internet Requests for Comments, RFC Editor, STD 13, November 1987, http://www.rfc-editor.org/rfc/rfc1035.txt. Available: http://www.rfc-editor.org/rfc/rfc1035.txt

[45] J. Steadman and S. Scott-Hayward, "DNSxD: Detecting data exfiltration over DNS," in *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2018, pp. 1–6.

[46] J. Ahmed, H. H. Gharakheili, Q. Raza, C. Russell, and V. Sivaraman, "Monitoring enterprise DNS queries for detecting data exfiltration from internal hosts," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 265–279, 2019.

[47] S. Kelley, "Dnsmasq," Sep. 2021 [Online]. Available: https://dnsmasq.org/

[48] S. Kelley, "Index of /dnsmasq," Sep. 2021 [Online]. Available: https://thekelleys.org.uk/dnsmasq/

[49] J. Mutai, "Install and Configure Dnsmasq on Ubuntu 22.04|20.04|18.04," Mar. 2022 [Online]. Available: https://computingforgeeks.com/install-and-configure-dnsmasq-on-ubuntu/

[50] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification," Internet Requests for Comments, RFC Editor, RFC 5905, June 2010, http://www.rfc-editor.org/rfc/rfc5905.txt. Available: http://www.rfc-editor.org/rfc/rfc5905.txt

[51] L. Frank Baum, "Network Time Protocol (NTP) Daemon," Mar. 2014 [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/ntpd.html

[52] Vitux, "How to Install NTP Server and Client(s) on Ubuntu 20.04 LTS," Mar. 2021 [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/ntpd.html

[53] A. Carroll, "VoIP protocols," Dec. 2020 [Online]. Available: https://www.erlang.com/support/voip-protocols/

[54] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," Internet Requests for Comments, RFC Editor, RFC 3261, June 2002, http://www.rfc-editor.org/rfc/rfc3261.txt. Available: http://www.rfc-editor.org/rfc/rfc3261.txt

[55] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," Internet Requests for Comments, RFC Editor, STD 64, July 2003, http://www.rfc-editor.org/rfc/rfc3550.txt. Available: http://www.rfc-editor.org/rfc/rfc3550.txt

[56] Linphone - Belledonne Communications SARL, "Linphone," 2020 [Online]. Available: https://linphone.org/

[57] Linphone - Belledonne Communications SARL, "Linphone: Download," Apr. 2022 [Online]. Available: https://linphone.org/technical-corner/linphone?qt-technical_corner=2qt-technical_corner

[58] T. Noergaard, "Chapter 4 - The Fundamentals in Understanding Networking Middleware," in *Demystifying Embedded Systems Middleware*, T. Noergaard, Ed. Burlington: Newnes, 2010, pp. 93–190. Available: https://www.sciencedirect.com/science/article/pii/B9780750684552000042

[59] D. Serpanos and T. Wolf, "Chapter 8 - Transport Layer Systems," in *Architecture of Network Systems* (The Morgan Kaufmann Series in Computer Architecture and Design), D. Serpanos and T. Wolf, Eds. Boston: Morgan Kaufmann, 2011, pp. 141–160. Available: https://www.sciencedirect.com/science/article/pii/B9780123744944000086

[60] 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

[61] J. Postel, "User Datagram Protocol," Internet Requests for Comments, RFC Editor, STD 6, August 1980, http://www.rfc-editor.org/rfc/rfc768.txt. Available: http://www.rfc-editor.org/rfc/rfc768.txt

[62] Joint Task Force, "Security and Privacy Controls for Information Systems and Organizations," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST Special Publication (SP) 800-53, Rev. 5, 2020.

[63] IBM Cloud Education, "What are Security Controls?" Dec. 2019 [Online]. Available: https://www.ibm.com/cloud/learn/security-controls

[64] M. Rash, *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort*. No Starch Press, 2007.

[65] Wireshark, "About Wireshark," 2022 [Online]. Available: https://www.wireshark.org/

[66] AppArmor, "AppArmor: Linux kernel security module," 2022 [Online]. Available: https://www.apparmor.net/

[67] Canonical Ltd., "How to create an AppArmor Profile," 2022 [Online]. Available: https://ubuntu.com/tutorials/beginning-apparmor-profile-development3-generating-a-basic-profile

[68] Cisco, "What is Snort?" 2022 [Online]. Available: https://www.snort.org/

[69] S. Tino, "Snort IPS," Dec. 2013 [Online]. Available: https://www.slideshare.net/SimoneTino/snort-ips

[70] Cisco, "Downloads," Apr. 2022 [Online]. Available: https://www.snort.org/downloads

[71] J. Ruostemaa, "How to install Snort on Ubuntu," Aug. 2021 [Online]. Available: https://upcloud.com/community/tutorials/install-snort-ubuntu/

[72] R. Tahboub and Y. Saleh, "Data leakage/loss prevention systems (DLP)," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*. IEEE, 2014, pp. 1–6.

[73] D. Wessels, "What is Squid?" May 2013 [Online]. Available: http://www.squid-cache.org/Intro/

[74] D. Wessels, "Squid Versions," Apr. 2022 [Online]. Available: http://www.squid-cache.org/Versions/

[75] Cisco, "ClamAV," 2022 [Online]. Available: https://docs.clamav.net/

[76] Cisco, "Download," 2022 [Online]. Available: https://www.clamav.net/downloads

[77] T. S. Ede, "Detecting adaptive data exfiltration in HTTP traffic," Master's thesis, University of Twente, 2017.

[78] K. Born, "Browser-based covert data exfiltration," *CoRR*, vol. abs/1004.4357, 2010.

[79] D. A. Haddon and H. Alkhateeb, "Investigating data exfiltration in DNS over HTTPS queries," in *2019 IEEE 12th International Conference on Global Security, Safety and Sustainability (ICGS3)*. IEEE, 2019, pp. 212–212.

[80] Z. Wu, J. Guo, C. Zhang, and C. Li, "Steganography and steganalysis in voice over IP: A review," *Sensors*, vol. 21, no. 4, p. 1032, 2021.

[81] J. Lubacz, W. Mazurczyk, and K. Szczypiorski, "Hiding data in VoIP," WARSAW TECHNICAL UNIV (POLAND), Tech. Rep., 2008.

[82] A. Ameri and D. Johnson, "Covert channel over Network Time Protocol," in *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, 2017, pp. 62–65.

[83] K. Lamshöft, J. Hielscher, C. Krätzer, and J. Dittmann, "The threat of covert channels in network time synchronisation protocols," *Journal of Cyber Security and Mobility*, pp. 165–204, 2022.

[84] J. Joy, A. John, and J. Joy, "Rootkit detection mechanism: A survey," in *International Conference on Parallel Distributed Computing Technologies and Applications*. Springer, 2011, pp. 366–374.

[85] Lockheed Martin, "The Cyber Kill Chain," 2022 [Online]. Available: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

[86] P. Biondi, "Kernel level security," *Context*, pp. 1–23, 2003.

[87] R. T. El-Maghraby, N. M. Abd Elazim, and A. M. Bahaa-Eldin, "A survey on deep packet inspection," in *2017 12th International Conference on Computer Engineering and Systems (ICCES)*, 2017, pp. 188–197.

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