Data Exfiltration via Flow Hijacking at the Socket Layer

Eric Bergen
erberge@radium.ncsc.mil

Daniel Lukaszewski
dflukasz@nps.edu

Geoffrey Xie
xie@nps.edu

Department of Computer Science
Naval Postgraduate School

Abstract

The severity of data exfiltration attacks is well known, and operators have begun deploying elaborate host and network security controls to counter this threat. Consequently, malicious actors spare no efforts finding methods to obfuscate their attacks within common network traffic. In this paper, we expose a new type of application transparent, kernel level data exfiltration attacks. By embedding data into application messages while they are held in socket buffers outside of applications, the attacks have the flexibility to hijack flows of multiple distinct applications at a time. Furthermore, we assess the practical implications of the attacks using a testbed emulating a typical data exfiltration scenario. We first prototype required attack functionalities with existing Layer 4.5 application message customization software, and then perform flow hijacking experiments with respect to six common application protocols. The results confirm the flexibility of socket layer attacks and their ability to evade typical security controls.

Keywords: network security, data exfiltration, protocol customization

1. Introduction

Many cybersecurity incidents involve some form of data exfiltration and can be perpetrated by both insider and outsider threats (Ullah et al., 2018). Obfuscated data exfiltration is routinely used by 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. This is evidenced by the recent SolarWinds and Anchor DNS attacks. In the SolarWinds incident, once the attackers 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 their own command and control (C2) server (Chesney, 2020). Subsequently, they were able to conduct a host of malicious activities to include data exfiltration.

In response to data exfiltration attacks, elaborated rules aiming to detect and prevent unauthorized traffic from leaving a proprietary network have been added to host-based and network-based security controls (Bertino et al., 2011). As a result, malicious actors are increasingly using obfuscation via common application protocols such as HTTP(S), DNS and VoIP that are allowed within the bounds of the organization's policy (Collins et al., 2016; MITRE, 2022).

Prior works (Ede, 2017; Schlicher et al., 2016) reveal that emulating message flows of an application (e.g., a web browser) for obfuscated data exfiltration requires significant effort on the part of the attackers. This is because the pattern of legitimate message flows may vary from application to application and from host to host. Accordingly, the data exfiltration must be carefully customized per application and/or per host to avoid detection by aforementioned security controls.

In this paper, we expose a new *application transparent*, *kernel level* method of hijacking application flows for the purpose of obfuscated data exfiltration. Termed "socket layer flow hijack attack", the new attack method is conceptualized from two observations as follows. First, socket buffers in the kernel, where application messages are stored before transport layer processing, present a point of entry for embedding data in application flows. Second, modern operating systems support dynamic kernel extensions that can add capabilities to process kernel data structures such as the socket buffers, without rebooting the host (Jones, 2001; The Linux Foundation, 2022). We believe that the new attacks can *amplify data exfiltration from a host* because they can simultaneously hijack flows of multiple applications and more importantly, it is relatively straightforward for them to blend in with normal traffic by embedding a small amount of data within each application message.

Furthermore, we assess the practical impact of socket layer flow hijack attacks using a VM based

URI: https://hdl.handle.net/10125/103435
978-0-9981331-6-4

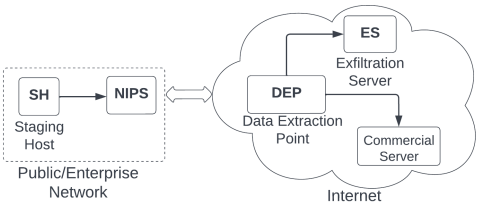

Figure 1. Data exfiltration attack scenario

testbed emulating a typical data exfiltration scenario as shown in Figure 1. In this scenario, the attacker gains control of a Staging Host (SH) in a specific public or enterprise network and exfiltrates data to a Data Extraction Point (DEP), while attempting to evade a network intrusion prevention system (NIPS). The attack hijacks legitimate traffic (e.g., to a commercial web server) and/or generates new camouflage flows to a pre-established Exfiltration Server (ES).

Leveraging an existing Linux based Layer 4.5 message customization system (Lukaszewski et al., 2022), we have prototyped two kernel level attack modules that run on the staging host (SH) and data exfiltration point (DEP), respectively, and performed a series of hijack experiments involving six common networked application protocols (HTTP, HTTPS, SMTP, DNS, NTP, and VoIP) with typical host and network based security controls.

The remainder of the paper is organized as follows. We review related work in Section 2. Section 3 details the technical considerations and building blocks for launching a socket layer flow hijacking attack, including an overview of the existing Layer 4.5 software framework. Section 4 presents the experimental design for evaluating the attack's feasibility and detectability with respect to a range of common application protocols. We present the results of the evaluation in Section 5. Section 6 details an extension to the Layer 4.5 software to properly evaluate attacks on application flows that utilize TLS. We outline other potential extensions and discuss mitigation of the attack in Section 7. Finally, Section 8 concludes the paper.

2. Related Work

A large body of related work exists for data exfiltration attacks. In this section, our review focuses primarily on (i) ways to perform and enhance data exfiltration obfuscated within common network flows and (ii) potential security controls against such attacks.

Exfiltration by Hijacking Network Flows: MITRE Corp. actively tracks such incidents and has identified FTP, SMTP, HTTP/S, DNS, and SMB as the most common targets (MITRE, 2022). Additionally, several studies concluded that directly manipulating existing applications (e.g., browsers) or invoking shell utilities (e.g., `curl` or `dig`) requires careful examination of application usage patterns in order to evade host level security controls or a NIPS (Born, 2010; Ede, 2017; Schlicher et al., 2016). In contrast, the new socket layer attack exposed in this paper – by hijacking legitimate flows outside of applications – can be less prone to deviation from normal application usage patterns and thus, become less detectable.

Meanwhile, Mazurczyk (2013) and Wu et al. (2021) focused on VoIP applications after observing the applications incorporate multiple protocols during a session and stream a sizable amount of sound and video data for at least a few minutes at a time. Socket layer data exfiltration can amplify these attacks by targeting any of the sockets established during a VoIP session.

Security Controls: First, there are host specific security controls that can detect or prevent some data exfiltration attacks. The standard host level firewall utility program for Linux systems (i.e., `iptables`), provides traffic-filtering services that can restrict socket connections to only permitted sites, ports, and protocols (Rash, 2007). AppArmor, an access control application that is bundled within many standard Linux distributions, performs access control for both the local file system and network connections to external hosts, on a per-application basis (AppArmor, 2022). It should be noted that both `iptables` and AppArmor require a significant amount of configuration effort. More importantly, as shown in later sections, the type of attacks evaluated in this paper – with its ability to hide data within traffic of host-native applications – can evade them effectively.

Intrusion prevention systems, such as Snort, provide another line of defense against data exfiltration attacks. As presented in later sections, we experimented with Snort as the representative NIPS and evaluated the effectiveness of the published rule sets in detecting exfiltration attacks at the socket layer.

3. Data Exfiltration at the Socket Layer

As discussed in the introduction, the socket layer sits between the application and transport layers, and thus, provides an *application-transparent point of insertion* for data exfiltration. Furthermore, transport protocols currently cannot detect and prevent such attacks because there are no built-in mechanisms for them to seek validation of individual messages from applications.

This section is divided into three parts. First, we step into the attacker's shoes and discuss the main technical challenges, including the system privileges

required, for implementing and concealing socket layer data exfiltration attacks. Then, we overview an existing Layer 4.5 application flow customization framework from our prior work (Lukaszewski et al., 2022), which we used to prototype the attacks disclosed in this work. Finally, we present design details of the Layer 4.5 modules created for our experiments, each of which can tap into one or more target application sockets and embed data within selected application message flows.

3.1. Assumptions and Challenges

As socket buffers sit in the kernel space, intercepting and modifying application messages there requires kernel-level permissions. Therefore, we assume the attacker has taken control of the staging host (SH) and gained persistent kernel-level permissions, while avoiding detection through a "zero day" root-kit or by tampering with the supply chain and planting attack modules within select OS image distributions.

This type of attack requires new heuristics to correctly isolate a specific socket buffer for a target application message flow. The conventional 5-tuple identification (src_IP, dst_IP, src_port, dst_port, protocol) is inadequate for several reasons. First, we should not limit the attack to only application flows that are already active since the attacker will likely attempt to hijack future flow instances before the sockets are created. Second, multiple different applications (e.g., Chrome, Firefox, `curl`) invoke the same application protocol (HTTP/S). Last, an application may utilize outside processes to send traffic. For example, consider the Linux `dig` application, which utilizes `bind` services to conduct DNS requests and as a result can have a process ID that is different from the application name.

Finally, the attack must consider application transparency, i.e., how to minimize interference with the normal progress of the hijacked application flows. As shown in Figure 1, the data extraction point (DEP) is effectively a middlebox and as such, must be carefully designed on the part of the attacker to not disrupt the application flows. After extracting exfiltrated data, the DEP must ensure (i) the new messages will be deemed properly formed by the server end, and (ii) the corresponding control messages (e.g., TCP ACK) sent back to the SH will properly count the bytes of exfiltrated data.

3.2. Layer 4.5 Software Framework

Layer 4.5 is a Linux based open-source software framework[1] that allows customization of application flows per TCP or UDP socket (Lukaszewski et al., 2022). As illustrated in Figure 2, the software supports tapping the `sendmsg()` and `recvmsg()` system calls specific to TCP/UDP, loading kernel extension modules designed to match specific application sockets, and performing custom logic on outgoing (incoming) messages before the messages are sent down to the transport layer (passed up to the socket buffers). We have chosen this framework because it supports rapid prototyping of our needed functionality.

The rest of the discussion will focus on how Layer 4.5 leverages the assumed kernel access privilege and addresses the previously identified challenges.

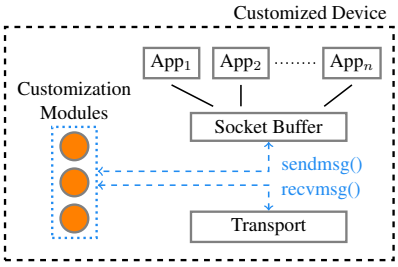

Figure 2. Layer 4.5 customization of application flows (Lukaszewski et al., 2022)

Taps of Socket Calls: Layer 4.5 provides a kernel tap between the socket and transport layers to allow customization modules to be invoked on matching application flows, which it accomplishes in two steps. First, it creates backups of the global function pointers to TCP/UDP send and receive calls, such as `tcp_prot.sendmsg`. Then, it replaces the global pointer with a pointer to a new function with necessary logic to hand application flows off to matching customization modules for intermediate processing before resuming TCP/UDP calls through the backup pointers.

Application Flow Isolation: The Layer 4.5 customization modules must specify the matching application flow parameters, which include the standard 5-tuple plus the target application name, such as `curl`. Recall that we assume an attacker will target future application flows. This means the entire 5-tuple is likely unknown when the attack modules are loaded due to dynamic port assignments. For this reason, the customization may use wildcard values in place of unknown parameters. Additionally, the target application name can be used to identify flows that have not established a full 5-tuple. After a Layer 4.5 customization module is loaded, the socket-transport tap will automatically identify new sockets and match them against customization modules using the specified parameters.

[1]Code available at GitHub: https://github.com/danluke2/software_defined_customization

Transparency to Applications: Layer 4.5 is designed to be application transparent (Lukaszewski et al., 2022). Each flow customization requires deploying *a pair of extension modules*, at the send and receive end, respectively. The receive-end module is responsible for returning messages to their pre-customization forms before passing them up to the socket buffer.

When the SH's target application sends a message to the socket, Layer 4.5 will redirect the flow to the matching customization module and report to the application that the amount of data transported matched what the application expected. When the DEP receives the custom traffic, Layer 4.5 will again redirect the flow to the customization module for processing. Note, this redirection occurs after the transport layer received the data, thus any TCP acknowledgements including the customized data have already been sent. Similar to the send path, Layer 4.5 will perform the required changes to the application buffer to prevent mismatches between the amount of data reported by the transport layer and what the application received.

3.3. Exfiltration and Extraction Modules

Two distinct Layer 4.5 customization modules were developed for this research: an "exfiltration module" deployed to the SH and an "extraction module" for the DEP. Both modules followed standard Layer 4.5 module design requirements and differed only in the send and receive customization logic to accommodate the embedding or extraction of data, respectively.

To streamline evaluating a range of application protocols as potential hijacking targets, the two modules were designed to be general purposed (especially, protocol agnostic) by using the configurable parameters presented in Table 1. The 5-tuple and `app_name` parameters were used to designate the socket connection to hijack at the SH and to extract data at the DEP, with the exfiltration source port and extraction destination port set as wildcard values to account for dynamic port assignments. The `bytes_total`, `bytes_per_message`, and `insert_pos` were used to infer the end of exfiltration and specify how the data exfiltration and extraction should be processed per application message. The `insert_pos` could be set to either the front, end, or a specific offset into the application message.

Furthermore, the modules were designed to be extensible to allow protocol specific logic. For example, after initial testing we discovered that hijacking VoIP flows necessitates a check of the message size in the exfiltration module to avoid creating messages that the IP layer will fragment due to exceeding the MTU size restriction. Exceeding this MTU threshold is possible, but we chose to emulate an attacker that would avoid causing IP fragmentation in an effort to avoid detection.

Table 1. Configurable parameters for modules

	Exfiltration	Extraction
5-tuple	`-src_port`	`-dst_port`
`app_name`	✓	✓
`bytes_total`	✓	✓
`bytes_per_msg`	✓	✓
`insert_pos`	✓	✓

4. Design of Experiments

In this section we describe the tests performed, configurations for each test, and the criteria used to determine if each test passed. We begin with a description of the testbed and machine settings used for the experiments to aid in reproducing each experiment.

4.1. Testbed Design

To simplify the scenario of Figure 1, we designed the experimental testbed shown in Figure 3 to combine the DEP and ES into a single machine and did not include the commercial server. This simplified setup allows for analysis of the exfiltration method's ability to bypass typical security controls, without the variability of more realistic inter-network transmissions influencing results.

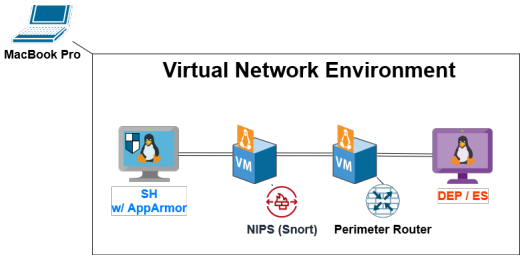

Figure 3. High level testbed design

The experiments in this section were performed under a testbed consisting of up to four Ubuntu 20.04/5.13 VMs running on a Quad-Core Intel Core i7 MacBook Pro with 16GB of RAM. Each VM was allocated 1 CPU, 4GB RAM, and configured to use an internal network with a 1000Mbps capacity.

4.2. Test Objectives

Experimentation will be conducted in three phases to test the following objectives: Exfiltration Feasibility, Host-Based Detection, and Network Detection. First, we focus on verifying the ability of socket layer

customization to be adapted to any application or protocol. More specifically, we identify the characteristics of application protocols that are more conducive to obfuscated data exfiltration via socket layer protocol customization. This phase is also used to determine which protocols can be customized using Layer 4.5, without causing application disruptions.

Next, we evaluate the exfiltration module's ability to access restricted files and perform customization against a host-based IPS (HIPS). The HIPS will primarily be configured on the SH and restrict file access permissions. This phase is designed to detect if we can reliably restrict application-based access permissions to the targeted files to prevent data exfiltration.

Last, we expand detection to include more sophisticated network-based security controls. In this phase, we include a NIPS inline between the SH and DEP to provide alert generation, logging, deep packet inspection, and the ability to filter and drop traffic. This phase assumes the exfiltration attack is able to bypass the HIPS and requires detection within the network.

4.3. Test Configurations

For each test, the SH represents the client machine and is the target host located on a proprietary network that contains the files to be exfiltrated. The DEP is configured to host all of the relevant services corresponding to the application protocols in Table 2. We include the client/server applications used as well since applications may exhibit different behavior when sending/receiving data, such as multithreading/multiprocessing. Note that since we target the SH with the exfiltation module, we may need multiple connections to occur to exfiltrate the entire file.

Table 2. Application protocols tested

Protocol	Client	Server	Transport	Port
HTTP	curl	python3	TCP	80
HTTPS	curl	python3	TCP	443
SMTP	postfix	postfix	TCP	25
DNS	dig	dnsmasq	UDP	53
NTP	ntpdate	ntpd	UDP	123
VOIP	linphone	linphone	UDP	9078

Table 3 lists the exfiltration configurations used for each test phase. The targeted files for exfiltration consist of a small text file (E1: *constitution.txt*) and a larger image file (E2: *penguin.tif*). The embedded-byte sizes are slightly different for each file and were chosen to both determine any variability of metric data between larger and smaller byte sizes and to avoid IP fragmentation, while also being whole number factors of the total file size to simplify the logic required to embed/extract data and determine when exfiltration is complete. When varying the insert positions, the position is chosen based on the size of the payload and the header fields available for the specific application protocol being tested.

Table 3. Exfiltration configuration

	File size	Bytes/msg	Insert pos.
E1	44.84 KB	10, 118, 236	Variable
E2	459.5 KB	10, 125, 250	Variable

During the HIPS and NIPS detection phases, we utilize the security controls of Table 4. For HIPS, we utilize AppArmor and configure it on the SH to restrict access permissions to the targeted files using flexible profiles that can be applied to individual applications (AppArmor, 2022). We implement specific AppArmor profiles and configure them to limit the ability of the targeted applications to access files for the purpose of data transmission.

Table 4. Security control configuration

Control	Software	Version	Rule-set
HIPS	AppArmor	2.13	Cust. Profiles
NIPS	Snort IPS	2.9	Talos v2.9.19

When testing against the NIPS, we utilized Snort with both default and custom configurations. Data exfiltration test runs are completed in two distinct rounds using various dynamic module configurations that are chosen based on the results of the exfiltration feasibility tests. The first round tests various module configurations against the default configuration and registered rule set of Snort. The second round consists of test runs that include additional custom Snort rules with content modifiers designed to detect attempts to use any application protocol to exfiltrate the specific target files. With regards to the custom Snort rules, the content modifiers within each rule are designed to identify specific byte sequences within the two files present in a transmitted packet and drop detected packets.

4.4. Performance Metrics

The specific metrics used for each test phase are included in Table 5. The throughput captures the amount of file data exfiltrated in KBps, and the overhead captures the number of packets transmitted per KB of exfiltrated file data. These measurements are useful for a relative comparison between protocols because they detail each application protocol's performance as a function of the total size of the exfiltrated data. In short, they show exactly how much data each application protocol is able to exfiltrate per second and how much overhead is incurred for each KB of exfiltrated data.

Table 5. Application performance metrics

Metric	Unit	Test
Throughput	KBps	Feasibility
Overhead	Packets per KB	Feasibility
Application Transparency	Number of Errors	Feasibility
Intrusion Detection	Number of Detections	HIPS/NIPS

Successful data exfiltration occurs when the target application does not encounter any errors and the HIPS/NIPS do not detect the exfiltration. If HIPS or NIPS detection occurs, we will analyze and evaluate the detection to determine what signature or rule-set was able to detect or prevent the data exfiltration attempt. Additionally, we verify the file was successfully exfiltrated by comparing the MD5 hash of the reconstructed file at the DEP to the file on the SH.

5. Experimental Results

In this section we present the results of the exfiltration feasibility, host-based detection, and network detection experimentation. We then discuss complications experienced when attempting to exfiltrate data over TLS encrypted application flows.

5.1. Feasibility

The first test loaded exfiltration and extraction customization modules to determine the throughput, overhead, and successful exfiltration of specific file data to completion. We begin by analyzing the throughput achieved for each protocol in Figure 4. Note that the E1 and E2 exfiltration experiments with intermediate byte sizes showed similar trends.

We see that all protocols, except NTP and HTTPS, were able to achieve relatively high throughput values. The DNS protocol resulted in the best throughput performance, while the VoIP protocol displayed the most flexibility in insert position and embedded-byte size for data exfiltration. We attribute the low throughput performance of NTP to using a deprecated client application with outdated algorithms (Network Time Foundation, 2022).

HTTPS throughput was not able to be measured due to application transparency failure during the experiment. This complication arose from the inability of the extraction module to properly handle the data buffers of application protocols that utilize TLS. In the case of HTTPS, TLS is implemented immediately upon a client-initiated request making it infeasible to perform extraction at the server without errors. SMTP

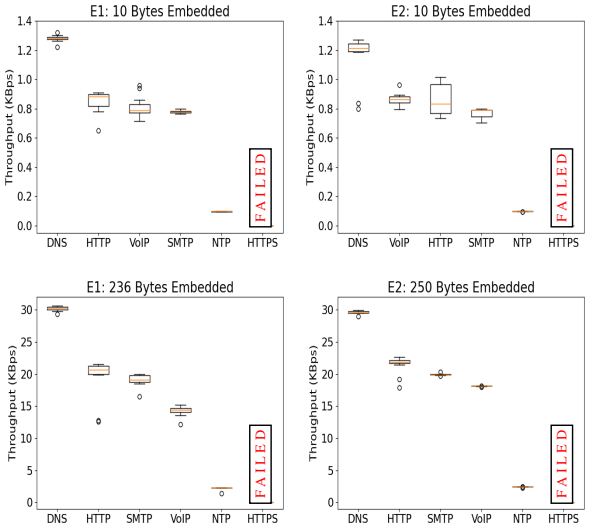

Figure 4. Data exfiltration throughput in KBps. Protocols ordered best to worst throughput.

transactions, however, incorporate several messages between hosts before initiating a TLS session. Thus, we successfully targeted these initial packets only and avoided errors associated with the TLS data buffers. We further expand on this TLS limitation in Subsection 5.4.

Application protocols that achieve a high throughput for data exfiltration may have unacceptable overhead. For this reason, we analyzed the overhead of exfiltrating data using each protocol. Figure 5 provides representative overhead results using E1 and E2 configurations.

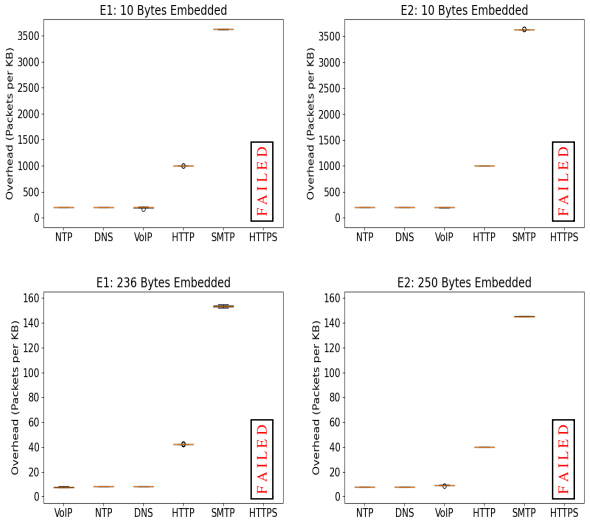

Figure 5. Data exfiltration overhead in packets/KB. Protocols ordered lowest to highest overhead.

From Figure 5 we see that even though HTTP and SMTP performed comparably to the UDP protocols in terms of throughput, the overhead experienced by these TCP protocols was significantly higher. The additional overhead of establishing the TCP connection and acknowledging the receipt of data resulted in an increase to the total number of packets sent before completely exfiltrating the target file.

Overall, the feasibility 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.

5.2. Host Based Detection

The results for Phase Two tests are summarized in Table 6. Across 26 test runs, there were only two instances where exfiltration was detected and prevented by AppArmor host-based access controls. Even though HTTPS failed feasibility testing, this may not always be the case. Therefore, we included it when testing host and network detection capabilities.

Table 6. AppArmor detections

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

Out of all the protocols tested, AppArmor only detected HTTP and HTTPS. The first detection occurred when utilizing HTTP and resulted from the customization module executing a system call to open and read a restricted file in preparation for exfiltration. The first detection was nullified by modifying the exfiltration module to access the targeted file during initialization when the module has root permissions instead of during customization where the module executes with application privileges. The second detection occurred when the exfiltration module targeting HTTPS embedded the file data within encrypted application data. By placing file data within only the first packet of the TLS handshake, this detection is no longer triggered. Once these changes were implemented, no further HIPS detections occurred.

The results show that the advantage associated with the kernel-level privileges of the exfiltration 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 basic host-based access controls by executing functions associated with file access outside customization-specific functions.

5.3. Network Detection

Table 7 presents the results for 26 exfiltration test runs against Snort's baseline configuration, which only produced alerts for exfiltration attempts over HTTP. These alerts were triggered by a preprocessor rule that detected irregular traffic with POST requests when the E1 configuration insert position was not at the end of the message. These alerts did not occur during E2 configuration testing over HTTP, regardless of insert position. No other alerts were generated by Snort's baseline configuration and registered rule set for any of the remaining application protocols.

Table 7. Snort IPS baseline configuration detections

Protocol	Number of Detections
HTTP	10
HTTPS	0
SMTP	0
DNS	0
NTP	0
VoIP	0

Table 8 presents the results for exfiltration testing against a customized rule set. Snort was able to detect and block E1 data exfiltration attempts for every tested application protocol when the larger embedded-byte sizes of 118 and 236 were used. This was because with these byte sizes the content modifiers were always included within the embedded data. Snort was unable to detect E1/E2 data exfiltration for test runs utilizing the 10 byte size, which was less than the content byte size specified in the custom 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.

Unexpectedly, Snort was unable to detect any E2 data exfiltration attempts. Investigation revealed that the *xxd* program utilized to encode the original file 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

Table 8. Summary of Snort detection results

		HTTP		HTTPS		SMTP		NTP		VoIP		DNS	
File	Data Size	Byte Position	Detected	Byte Position	Detected	Byte Position	Detected	Byte Position	Detected	Byte Position	Detected	Byte Position	Detected
E1	10	END	FALSE	END	FALSE	END	FALSE	END	FALSE	END	FALSE	END	FALSE
	118	45	TRUE	320	TRUE	5	TRUE	12	TRUE	5	TRUE	47	TRUE
	236	32	TRUE	375	TRUE	END	TRUE	35	TRUE	11	TRUE	50	TRUE
	10	END	FALSE	310	FALSE	18	FALSE	35	FALSE	3	FALSE	49	FALSE
	118	END	TRUE	END	TRUE	8	TRUE	END	TRUE	6	TRUE	END	TRUE
	236	END	TRUE	END	TRUE	13	TRUE	END	TRUE	END	TRUE	END	TRUE
E2	10	END	FALSE	END	FALSE	END	FALSE	END	FALSE	END	FALSE	END	FALSE
	125	58	FALSE	376	FALSE	5	FALSE	3	FALSE	3	FALSE	END	FALSE
	250	32	FALSE	450	FALSE	11	FALSE	8	FALSE	13	FALSE	47	FALSE
	10	58	FALSE	376	FALSE	6	FALSE	35	FALSE	5	FALSE	47	FALSE
	125	END	FALSE	415	FALSE	END	FALSE	END	FALSE	7	FALSE	49	FALSE
	250	END	FALSE	320	FALSE	END	FALSE	END	FALSE	15	FALSE	END	FALSE
	10	32	FALSE	450	FALSE	15	FALSE	22	FALSE	7	FALSE	49	FALSE

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. For completeness, we performed the experiments again after changing the associated custom rules to account for the correct encoding to verify Snort was able to detect the exfiltration. As with E1, Snort was able to detect and block the E2 data exfiltration attempts for every tested application protocol, but only when the larger embedded-byte sizes of 118 and 236 were used.

5.4. TLS Receiving Failures

The most notable limitation experienced throughout testing was the inability of the extraction module to work correctly with applications that implement TLS (i.e., to properly handle exfiltrated data embedded into the TLS data buffers). To determine why TLS caused errors, we need to understand the receive-side processing of TLS and Layer 4.5.

TLS messages, in general, follow a length-data pattern where the length specifies the amount of data expected for the current message. Since TLS is implemented above Layer 4.5, this length value is calculated before data is embedded by the exfiltration module. Embedding data, therefore, results in a transmitted payload length that is larger than indicated in the length record, which leads to a *decode_error* on the server-side that indicates a message could not be decoded because of a field error or incorrect length (Dierks et al., 2008). The root cause of this error is the result of Layer 4.5 leveraging the application's buffer to receive potentially customized messages in an effort to minimize the customization memory footprint. Thus, customization modules may be restricted by the size of the allotted application buffer and the requested bytes from the transport layer.

Figure 6 provides an example illustration of a TLS processing failure caused by customization. The SH begins by sending a 517 byte payload to the DEP (1). Layer 4.5 intercepts the transmission and the exfiltration

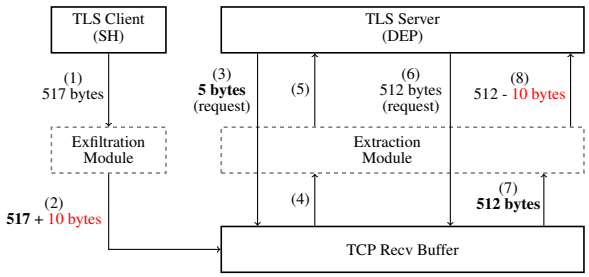

Figure 6. Example of TLS processing failure with numbers in () indicating flow order

module embeds 10 bytes of data into the TLS payload (2) prior to sending to the DEP. When the DEP receives the data, the application first requests the five-byte record layer (3), which is intercepted by Layer 4.5 and sent through the extraction module (4) prior to returning to the application (5). These first five bytes are not customized, so the extraction module does not modify their contents. The application then requests the 512 byte payload, based on the record layer length field, from the TCP buffer (6). Before these bytes reach the application, the extraction module removes the 10 embedded bytes (7), which results in the application only receiving 502 bytes (8). Since the application did not receive the correct number of bytes, payload decryption fails and the TLS connection is terminated.

6. Extension of Layer 4.5 for TLS Traffic

To overcome the TLS receive side processing problem observed in our experiments, we extended the Layer 4.5 software to allow a customization module (such as the extraction module) to request data beyond what was requested by the application. Now the module is capable of holding enough data to allow processing the customization, but this requires modules to potentially buffer data instead of delivering all data to the application as previously done.

For example, consider the example in Figure 6. Using the extended prototype, when the DEP

application requests the first 5 bytes (3), Layer 4.5 will instead request all 527 bytes from the TCP buffer and deliver them to the extraction module (4). The extraction module will remove the embedded 10 bytes, buffer the 512 byte payload, and return the requested 5 bytes to the application (5). Now when the application requests the 512 byte payload (6), the extraction module returns the bytes without causing a *decode_error*.

Using this Layer 4.5 extension on the DEP, we refined the HTTPS extraction module and repeated the previously failed feasibility experiments. The representative E1 and E2 throughput and overhead results are shown in Figure 7 and Figure 8, respectively. Since TLS adds additional server packets to the HTTPS connections, both the throughput and overhead of HTTPS are worse than the unencrypted HTTP protocol.

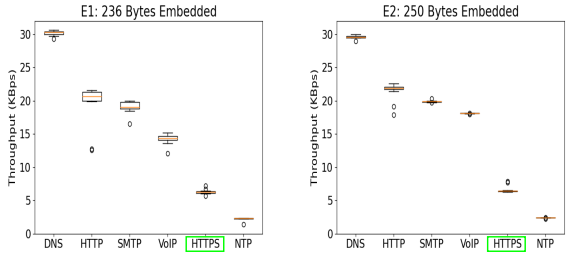

Figure 7. Data exfiltration throughput including HTTPS. Protocols ordered best to worst throughput.

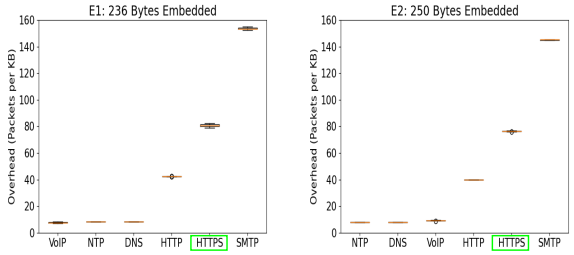

Figure 8. Data exfiltration overhead including HTTPS. Protocols ordered lowest to highest overhead.

The experiments conducted in Phase 2 and 3 were not repeated for HTTPS utilizing this extension since the security controls implemented are designed to detect exfiltration at the SH and during data transmission. Thus, the extension implemented at the DEP would not have any effect that would change the original results.

7. Discussion

In this section we first present potential strategies to counter the type of attacks exposed in this paper. Then, we discuss some future extensions to this research.

7.1. Mitigation

Fundamentally, the customization module is similar to a traditional rootkit. There are known methods to prevent and detect rootkits (Joy et al., 2011), but within the scope of this paper there are a few specific mitigations that might prevent or detect this particular method from exfiltrating data if a module does get installed on a host system.

One mitigation would be to provide priority event alerts if unregistered or restricted kernel-level modules are loaded onto a system. These alerts are already generated locally on a Linux host via the kernel log, but tamper-resistant features and other refinements are needed to ensure reliable delivery of these alerts to the operator in real time. A more comprehensive network-based management solution could involve the implementation of a security information and event management (SIEM) system, which could automate the process of detecting and responding to these types of events, from alerting the operator to isolating the host from the rest of the internal network (Bhatt et al., 2014).

Another mitigation would be to enforce protocol specific filtering rules at a NIPS. As shown in our experimentation, attackers can leverage the format of a protocol payload to perform data exfiltration. For example, the byte space allocated for domain names in a DNS query can be used to exfiltrate data since it is not common to use the entire space for most queries (Born, 2010). Snort and other NIPS solutions provide the capability to determine the size of such fields for DNS and other protocols and drop traffic that exceeds an acceptable threshold. The threshold can be configured to match common network traffic patterns of a network.

The Layer 4.5 architecture (Lukaszewski, 2022; Lukaszewski et al., 2022) could also be used to detect unauthorized customization modules being used within the network. For example, the architecture includes a central customization orchestrator responsible for the distribution and continuous management of all customization modules. If a rogue Layer 4.5 customization module is being used to exfiltrate data on the network, then the orchestrator would detect the presence of the module and either remove it automatically or generate an alert. However, if an attacker installs the customization module outside the Layer 4.5 architecture, then it would likely not be discovered by the orchestrator.

Finally, some networks utilize TLS interception software (i.e., proxy) to allow inspecting encrypted traffic within the network (de Carnavalet et al., 2016). The presence of this proxy in the exfiltration network could inhibit an attacker's success based on the

limitations of TLS customization processing discussed in Subsection 5.4. If the attacker successfully infiltrated the SH, but did not also compromise the TLS proxy, then the exfiltrated data will cause the same *decode_error* we experienced during feasibility testing.

7.2. Future Extensions

Several extensions to this research to further evaluate the risk of socket layer exfiltration attacks are worth mentioning. First, our work is limited in scope due to using a simulated network environment. It would therefore be of value to measure the ability of this method to perform exfiltration across real network infrastructure where data must pass through multiple network devices and navigate typical routing configurations such as NAT to see if these conditions have any significant effects.

Extensions could also include testing against more capable and varied security controls, particularly at the SH. While AppArmor is a capable host-based control, it would be valuable to test against commercial and open-source alternatives that have more specialized suites of host-based security controls. Additional network-security controls, such as an inline proxy solution should also be considered.

Finally, more research should be performed with respect to emerging protocols that are becoming more prominent in everyday network traffic, such as DNSSEC and QUIC. Furthermore, testing various applications that utilize the same protocol, such as web browsers and their command line variants, could further assess the complexity and generality of the attack methodology. Our initial tests reveal that applications that use many concurrent communication processes, such as the Apache web server, may require additional capability to be programmed into the attack modules.

8. Concluding Remarks

This paper has exposed and evaluated practical implications of a new type of data exfiltration attacks that gain application transparency by hijacking background application flows at the socket layer. Our results indicate that the new attacks can be hard to detect and more research still is required to understand their full impact on network security. In particular, open questions arise as to if and how operating systems should be upgraded to support validation of application messages at the transport layer.

Acknowledgements

We thank anonymous reviewers and track chair for their helpful comments. This work was partially funded by the U.S. Office of Naval Research (ONR) under the TPCP program. The contents are those of the author(s) and do not necessarily represent the official views of, nor an endorsement, by the U.S. Government.

References

AppArmor. (2022). *AppArmor: Linux kernel security module*. https://www.apparmor.net/

Bertino, E., & Ghinita, G. (2011). Towards mechanisms for detection and prevention of data exfiltration by insiders: Keynote talk paper. *Proc. 6th ACM ASIACCS Symposium*.

Bhatt, S., Manadhata, P. K., & Zomlot, L. (2014). The operational role of security information and event management systems. *IEEE Security & Privacy*.

Born, K. (2010). Browser-based covert data exfiltration. *CoRR*. https://doi.org/http://arxiv.org/abs/1004.4357

Chesney, R. (2020). Cybersecurity law, policy, and institutions (v3.0). *U of Texas Law*.

Collins, J., & Agaian, S. (2016). Trends toward real-time network data steganography. *Int. Journal of Network Security and its Applications*.

de Carnavalet, X., & Mannan, M. (2016). Killed by proxy: Analyzing client-end tls interception software. *NDSS Symposium*.

Dierks, T., & Rescorla, E. (2008). *The Transport Layer Security (TLS) Protocol Version 1.2* (RFC No. 5246).

Ede, T. S. (2017). *Detecting adaptive data exfiltration in HTTP traffic* (Master's thesis). University of Twente.

Jones, K. (2001). Loadable kernel modules. *login: The Magazine of USENIX and SAGE, Special Focus Issue: Security*, *26*(7).

Joy, J., John, A., & Joy, J. (2011). Rootkit detection mechanism: A survey. *Int. PDCTA Conference*.

Lukaszewski, D. (2022). *Software defined customization of network protocols with layer 4.5* (Doctoral dissertation). Naval Postgraduate School.

Lukaszewski, D., & Xie, G. (2022). Towards software defined layer 4.5 customization. *IEEE 8th Int. NetSoft Conference*.

Mazurczyk, W. (2013). Voip steganography and its detection — a survey. *ACM Computing Surveys*.

MITRE. (2022). *Exfiltration over alternative protocol*. https://attack.mitre.org/techniques/T1048/

Network Time Foundation. (2022). *Deprecating ntpdate*. https://support.ntp.org/bin/view/Dev/DeprecatingNtpdate

Rash, M. (2007). *Linux firewalls: Attack detection and response with iptables, psad, and fwsnort*. No Starch Press.

Schlicher, B. G., MacIntyre, L. P., & Abercrombie, R. K. (2016). Towards reducing the data exfiltration surface for the insider threat. *49th HICSS*.

The Linux Foundation. (2022). *What is eBPF*. https://ebpf.io/what-is-ebpf/

Ullah, F., Edwards, M., Ramdhany, R., Chitchyan, R., Babar, M. A., & Rashid, A. (2018). Data exfiltration: A review of external attack vectors and countermeasures. *Journal of Network and Computer Applications*.

Wu, Z., Guo, J., Zhang, C., & Li, C. (2021). Steganography and steganalysis in voice over ip: A review. *Sensors*.