UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Design and Implementation of Glavlit: A Transparent Data Confinement System

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science

in

Computer Science

by

Nabil Adam Schear

Committee in charge:

Professor Amin Vahdat, Chair
Professor Geoffrey M. Voelker
Professor George Varghese

2007

The thesis of Nabil Adam Schear is approved:

Chair

University of California, San Diego

2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge the assistance of the faculty and students in the sysnet group at UCSD for providing feedback, encouragement, and resources that proved invaluable to the success of this work. I also acknowledge the aid provided by the chair of my committee, Amin Vahdat, for his continued support of my education and research opportunities.

Chapter 2, in part, has appeared previously in the 5th ACM Workshop on Hot Topics in Networks, 2006, ACM SIGCOMM. Nabil Schear Carmelo Kintana, Qing Zhang, and Amin Vahdat. The thesis author was the primary investigator and author of this work.

Chapters 1 and 2, in part, and chapter 4, in full, have been submitted for publication in the Proceedings of Ninth International Conference on Information Hiding, 2007, Springer-Verlag. Nabil Schear, Carmelo Kintana, Qing Zhang, and Amin Vahdat. The thesis author was the primary investigator and author of this work.

ABSTRACT OF THE THESIS

The Design and Implementation of Glavlit: A Transparent Data Confinement System

by

Nabil Adam Schear

Master of Science in Computer Science

University of California San Diego, 2007

Professor Amin Vahdat, Chair

Given the landscape of threats currently facing the Internet today, it is challenging to secure a network – not only to prevent attack, but also to ensure that sensitive data is not stolen. The Glavlit project considers the problem of efficiently confining protected data transferred using HTTP. Our goals for Glavlit are transparency, high speed, generic content analysis, and minimal covert channel bandwidth. While there have been a variety of solutions involving custom hardware, software, and practices, we focus on delivering high-performance data confinement that operates transparently using standard network protocols. The key techniques used by Glavlit to mitigate unauthorized communication are: i) verifying data transfer in both covert and overt channels; ii) employing a restricted, but compliant HTTP protocol subset; iii) verifying the semantics of protocol fields and behavior; and iv) separating the process of vetting authorized objects from line-speed data verification.

This thesis considers the implementation of the Glavlit system. We provide a summary of our leak mitigation techniques with a focus on their practical implementations. This thesis also presents the implementation of the three components of the Glavlit system: Warden, Client, and Guard. We detail how each is designed and the confinement and performance trade-offs we made. We also present a thorough evaluation of the performance and effectiveness of the Glavlit system. The evaluation shows that the mitigation techniques used by Glavlit may be deployed at the network perimeter without serious penalty. It also shows that our implementation is efficient compared to other network services.

Professor Amin Vahdat
Thesis Committee Chair

1

Introduction

1.1 Problem

Protecting sensitive data from leaving the network is an increasingly difficult problem. Businesses, governments, and individuals must process sensitive data. Improper release of such data can incur deleterious consequences. Consider the recent release of search data on hundreds of thousands of AOL users [44]. Organizations must be able to process information not fit for release like intellectual property, personal data, medical information, and financial records while at the same time processing data that can be released all on one computer network. The ubiquity of Internet-connected computers makes inadvertent and malicious unauthorized release difficult to control. Since information must be processed heterogeneously on one network, the risk of release due to attack or carelessness is also increased. Attackers may be interested in ex-filtrating sensitive information from the network to compromise its secrecy for profit or fame.

Due to censorship and intrusion detection methods employed in the network today, hackers have resorted to more dexterous techniques of exfiltrating sensitive data. For instance, as the values of the data increases, attackers wanting to ex-filtrate data from large organizations or governments may attempt to defeat confinement mechanisms by establishing covert channels embedded within

existing communications or by embedding data within otherwise innocent binary objects. The use of covert channels to transport unauthorized data is both hard to discover and to circumvent. Ensuring that sensitive data does not leak over a network to unauthorized machines remains an open problem.

We focus on the confinement of unauthorized data by allowing only approved data from crossing from an organization's internal network to the Internet at large. We wish to prevent the transmission of sensitive data in the payload channel of layer 7 protocols such as HTTP or FTP or in the protocol channel itself. To prevent information leaks through these channels, everything beyond the TCP header of every packet must be inspected and verified before allowing a packet to exit the network. Performing such verification at line speed has thus far been limited to pattern searching for sensitive information such as social security or credit card numbers. On the other hand we wish to enable verification and inspection at the granularity of entire objects (e.g., files).

To address the shortcomings of previous solutions, we created Glavlit[1] to confine sensitive data within a protected network while allowing authorized information to pass unhindered and with existing interactive protocols. Our goals for Glavlit are:

1. Transparency
2. High speed network operation
3. Support for generic powerful content review
4. Minimal covert channel bandwidth at layer 7

Glavlit provides stringent security guarantees by enforcing complex exit policy while trusting only the machines responsible for approving individual files for release and for inspecting the packets leaving a network. We assume that other

[1] We have named the system *Glavlit* for the organization of the former Soviet Union. Glavlit is the Russian acronym for the Chief Agency for Protection of Military and State Secrets that handled official state censorship matters [9].

machines in the system, such as Web and storage systems may be arbitrarily compromised.

Digital review of complex static objects such as Microsoft Office documents, PDF, or multimedia files can be compute intensive, lengthy, and potentially require the entire file to be examined at once. Additionally, some organizations are not willing to depend entirely on digital review and require human intervention. The need for complex, compute intensive, and human assisted object vetting is contrary to our goals of transparency and performance. One could imagine a new protocol where users external to some network boundary queue requests for particular data. Upon verifying that the requested data is authorized for release, the object could then be transmitted to the external user. Unfortunately, this process imposes significant per-request overhead and makes object access highly asynchronous. Further, it would remain difficult to ensure that the data actually crossing the network corresponds to the named data requested by the external user. Hence we provide a means to *decouple* the complex analysis process for objects from their verification at a gateway.

Although completely preventing all potential overt and covert channels for unauthorized release is an intractable problem, we address several mechanisms to mitigate and limit the bandwidth of covert channels. We target HTTP for verified data transfer across a network boundary since it is the dominant protocol used to disseminate information and since it is general to a variety of deployment settings. We believe that our techniques generalize to other layer 7 protocols, but we leave it to future work to test that hypothesis. We utilize a restricted but compliant protocol subset to significantly reduce the capacity of any covert protocol channels. We can also prevent unstructured channels and timing channel attacks by correlating request-response pairs and normalizing server response times.

The purpose of this thesis is to translate the mitigation techniques of

Glavlit into a practical implementation. Herein, we describe the architecture, design, and operation of the components of the Glavlit system. To aid the reader's understanding of the implementation, we provide a brief overview of the leak mitigation techniques used by Glavlit. Further background on these techniques can be found in prior work by the thesis author [50, 51].

1.2 Terms

To properly describe channels used for leaking information and how our system mitigates them, we must define some terms. Since HTTP is designed to serve information, we present methods for employing arbitrarily time-consuming analysis tools on the payload of HTTP. Key to our approach is splitting data confinement of payload objects into two distinct phases: *vetting* and *verification*. Vetting is the process a designated authority follows to determine whether an object is appropriate for external release. Verification ensures an object was previously vetted before releasing it across a designated network boundary. Glavlit maintains the invariant that only vetted content may leave the network while operating in real-time on per-packet information. Glavlit is analogous to a capability-based system design; for any object to cross a protected network boundary in the payload channel, the sender must obtain a capability. This capability is granted when the object is vetted.

The primary goal of Glavlit is to allow exit policy to be enforced for each *information release*. Information release occurs when information is moved over a protection boundary from one domain D_1 into a different domain D_2. This boundary could be from an organization to the public, an organization to a business partner, or from one level of protection (classification) to another. Once the information is transmitted, administrative control by the originating domain D_1 is relinquished to D_2. The mechanism used to perform information release is a channel.

Channels have many attributes including medium, visibility, carrier, and algorithm. The *medium* of a channel is the substrate over which information passes and is stored. The *visibility* is the degree to which the channel is known to exist. The *carrier* is the data inside of which the channel is embedded [27]. The *algorithm* is the synchronization and coordination element of the channel that allows the establishment of a connection to transmit information. We also introduce a subset of visibility which we have named intent. *Intent* shall be classified based upon whether the information release is authorized by the originating authority in D_1. The intent of a channel is related in some cases to its visibility. However, similar intent can occur across multiple levels of visibility.

1.3 Related Work

1.3.1 Covert Channels

There is substantial prior work on covert channels that are relevant to the content off this thesis. We decompose channels for information release using the terms described in 1.2 while including examples of known channel algorithms. Others have presented similar taxonomies of covert channels [25, 39, 8].

As stated, the carrier of a channel is the data in which the channel is embedded, imposed or encoded. A covert carrier is the cover data which masks the channel's existence [27]. We expand on this definition to include overt carriers. An overt channel has no need to be hidden; therefore, it is simply the data stream where the information is stored.

Carriers can be *structured* or *unstructured* as classified by Fisk et al [18]. All data storage is inherently organized in some way such that the data may be interpreted accurately. A structured carrier is one whose data is organized with explicit semantics. Structured data is most common in computing environments and is generally not meant for human consumption. Unstructured carriers

store data subjectively and often rely upon human interpretation. Unstructured carriers can also store data in subjectively random data like order, timing, and counting. We separate unstructured from structured data by the degree to which the organization of the data is apparent.

Structured Carriers

Any data with explicit organization can potentially store information as a structured carrier. In structured data, each individual field can be identified and located. An explicit generator exists for each field or item. Some channels simply use specific structured fields to store the information directly [48]. The most difficult to detect covert channels are typically targeted at storage in fields that have high entropy or are completely random. Murdoch et al and others have observed that many fields that seem random are actually pseudo-random (e.g., TCP ISN) and can be characterized to facilitate detection [42, 58]. Structured carriers have also been referred to as *storage channels* because they involve the direct or indirect writing of a specific storage location [11].

Numerous covert channels have been proposed for use over network mediums. There are numerous examples of covert channels that modify header fields in the TCP/IP protocol [49, 48, 21]. ICMP messages can also be used to create channels [30, 10]. Channels have also been created over higher layer protocols. Estrada et al modify HTTP protocol headers to create a unidirectional channel in [13]. Other application layer protocols (e.g., SSH and HTTP) have been layered on the DNS protocol to covertly communicate [26, 28].

Storage mediums also provide a number of structured carriers that can be used to create channels. Information can be embedded inside fields of the file system like time stamps, access controls, and block maps. The alternate data stream (ADS) feature of Microsoft's NTFS file system has been known to be used to covertly store information [7]. Information in an ADS is preserved when copied across a network to another NTFS file system. The slack space of file

system blocks has been used to store information [37].

Complex file formats such as Microsoft Office and Adobe PDF abound with structured carriers. The files normally contain numerous hidden fields that can be modified clandestinely. Covert information can be stored in document property information, macros, styles, views, past revisions, and comments [17]. Given the proliferation of files of these types, the danger of covert channels embedded in complex file format fields is high.

Unstructured Carriers

Unstructured carriers present a considerable challenge for detection and defeat. Because of the subjectivity of the data stream, the method used to distribute information and the amount of data stored affect detectability. The channel developer must apply organization to an unstructured carrier to make it feasible for data storage. In order to evade detection, the channel developer must minimize the organization required to reliably store data. Some detection strategies try to locate *artifacts* of organization in subjective data [47]. The artifacts are detected because of a flaw in the channel algorithm or by statistical analysis. Generic methods to search for arbitrary artifacts have not yet been explored.

Protocols and algorithms must be developed that organize some portion of the unstructured data. Protocols are often obfuscated and dependent on random information. For example: JPHide randomly chooses discreet cosine transform (DCT) coefficients to modify and limits the number that may be modified before the image is considered *full* of hidden information [35]. In some cases, an unstructured carrier is based upon a structured data stream. For example: a JPEG file is explicitly structured tables of data, however careful modifications to the subjective data in these tables can steganographically store information in an unstructured manner [46].

Because of their highly subjective data and sheer size, multimedia files are

well-suited to data hiding. In addition to the JPEG steganography techniques mentioned above, algorithms exist for other image, audio, and video formats. The least significant bit (LSB) of any type of subjective data can be used to store information covertly. LSB is often used with lossless formats like bitmaps, TIFF, and WAV audio files [19, 34]. Compression increases the subjectivity and randomness of a multimedia file. Channels have been proposed that store information in the compression data itself (e.g., DCT coefficient tables), in areas that won't be lost after compression (e.g., video key frames [23]), or in areas that are modified by compression (e.g., high frequency noise in audio). Watermarking differs from the steganography because of its inherent robustness to destruction after transformation (e.g., re-compression or editing).

Another type of unstructured carrier is one that uses signals to transmit information. Signals can be created using timing and order. These carriers are difficult to detect because they can be based upon random data like network traffic patterns. Timing is often random with respect to system load, scheduling, and network latency. Order can be affected by network conditions or the arbitrary nature of the data. The channel protocol must account for these problems of skew and reordering with organization to reliably pass data.

Order as a carrier of information has been examined in a variety of unstructured data streams. Network packet order was used by Galatenko et al to encode information between clients over a virtual private network [20]. Sequences of protocol commands can also be used to create unidirectional channels [24, 14, 36]. The use of the order of pipelined HTTP/1.1 requests could be used to create a bidirectional channel. The Gifshuffle program reorders the color table of a Gif file to store information [32]. Natural language has long been a subject of steganographic research. Wayner describes a mimic algorithm that creates streams of words that follow a pattern [60]. One application of this algorithm is spammimic which encodes messages using language common to spam emails.

[54]. The use of timing on packet networks as a carrier for covert communications has been often discussed but rarely researched until recently [6, 3].

Preventing Covert Channels

There are known techniques for detecting covert channels in layer 3 and 4 protocols [1, 42, 58]. These techniques focus on verifying the validity and allowable entropy in specific TCP and IP protocol fields. Fisk et al introduced minimum requisite fidelity (MRF) to allow an active warden to modify suspect fields within the MRF without disrupting the function of the protocol [18].

The first link between information theoretic channel capacity and computer covert channels was provided by Millen [40]. Giles and Hajek discuss jamming timing channels, but focus on continuous time channels rather than discrete [22]. Cabuk et al and Berk et al discuss statistical methods for detecting covert channels which use packet inter-arrival times as their carrier [3, 6]. Both techniques become less effective when the attacker maliciously inserts noise into the channel. We find that HTTP response time is easier to model because when monitored from the same network the server responses are almost entirely a function of the offered load. This allows maliciously inserted noise to also imply a covert channel or other problem exists.

Singh developed Eraser to prevent covert channels in media distribution systems, but it is not transparent [53]. It requires a new protocol between clients and the network gateway. Eraser detects when a compromised workstation in a secure network is communicating to a malicious Web server on the outside. The client must request permission for content to leave; therefore, the system relies upon client host integrity. Lucena et al discuss methods for creating application layer covert channels which preserve protocol syntax and semantics, and they performed a case study with SSH [38]. We focus on the same class of protocol channel in or analysis of HTTP, however rather than proposing a single channel mechanism we exhaustively examine all possible fields to minimize exfiltration

risk.

The privacy firewalls created by Yin et al provide a means to prevent information leaks from failed servers in a Byzantine fault tolerant system [61]. Their approach requires $h + 1$ filter nodes to handle h failures. The agreement protocol controls ordering information to prevent covert channels. The authors touch upon timing analysis but admit that some elements of nondeterminism are not covered by their system. Glavlit requires the trusted operation of its components. We do not consider the problem of component failure.

1.3.2 Content Protection

There are commercial network security systems which monitor the HTTP payload for sensitive content, though none address the problem of covert channels. There are products which audit HTTP traffic off line and are unable to actively confine protected data [56, 59]. Other products proxy outgoing Web traffic and lack transparency [45]. A solution from Fidelis performs simple analysis directly on the gateway and like Glavlit uses TCP reset packets to terminate unauthorized connections [15].

There is a substantial body of prior work on tools which can be used during Glavlit vetting to detect and defeat covert channels and steganography in file payloads. Because image steganography has become more prevalent and easy to exploit, detection tools could be used in the Warden to prevent it [55, 47]. Additionally, Los Alamos National Laboratory has developed File Scrub which detects and cleanses hidden metadata from complex office formats such as Word, Excel, PowerPoint, and PDF [17].

1.4 Acknowledgment

Chapter 1, in part, has been submitted for publication in the Proceedings of Ninth International Conference on Information Hiding, 2007, Springer-Verlag.

Nabil Schear, Carmelo Kintana, Qing Zhang, and Amin Vahdat. The thesis author was the primary investigator and author of this work.

2

System Design

2.1 Architecture

We have designed Glavlit to ensure that all information leaving a protected network has been approved for release and minimize unauthorized communication channels. The Warden and Guard are the primary components of the Glavlit system (Figure 2.1). A central server called the Warden vets objects. Client software provides an interface to content providers to manage exit policy at the Warden (Step 1). We can envision a Web based client or one using an email submission system. The Client provides the file to be vetted to the Warden as well as some metadata and information about how the file will be used. Authentication allows the Warden to enforce additional vetting policy. For example, the Warden can enforce that only project leaders may vet files of a certain type. The Warden provides a framework to employ various review and subliminal channel detection techniques for files leaving the network (however, the techniques themselves are orthogonal to this effort). The Warden shares a repository of white-listed content called *signatures* with the Guard. The signatures store hash values and protocol meta-data to later verify the content and mitigate protocol channels at the Guard (Step 2).

The Guard is a high-speed transparent network bridge at the perimeter

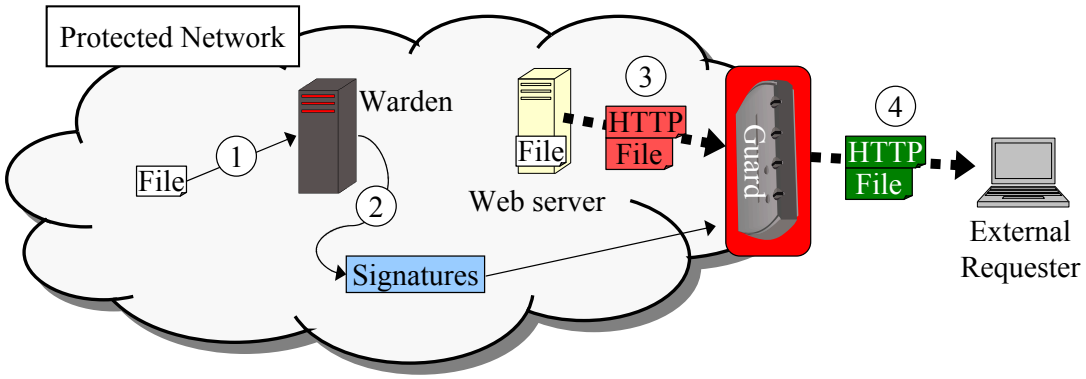

Figure 2.1: Glavlit System Architecture

of the protected network. When an external request arrives for a new object, the Guard parses the request and waits for the server response. The Guard checks verifiable fields in the server response, enforces ordering, and performs timing analysis on the response time to mitigate protocol channels (Step 3). The Guard determines the boundary between the protocol header and object payload. For every packet containing payload data, the Guard verifies that the (partial) contents match some previously vetted object. If verification fails, the Guard can actively stop data from leaving the network by terminating the connection. Otherwise, the authorized file and protocol are allowed to leave the network (Step 4).

2.1.1 Threat Scenarios

We further explore the motivation behind developing this system by examining the threats to information leaks. We categorize these threats into the following:

- *Accidental Release* – In this scenario, a valid system user inadvertently releases information. For example, this could happen because the user did not know a file's content was sensitive and places it on a public-facing Web server. Glavlit counters it by matching outgoing content against a white-list of allowed content.

- *Malicious Use of Standard HTTP* – An attacker (e.g., disgruntled employee, external hacker, or host compromised by a virus or malware) compromises a host in the protected network and maliciously places sensitive content on an existing Web server. Since Web servers are often replicated, allowed through firewalls, and accessed routinely by many parties, it follows that an attacker can use an existing server to deface existing content or use new/existing content to leak sensitive data. For example, Web servers often serve content from shared file systems (e.g., from user home directories) that can be accessed by underprivileged users. Glavlit detects any modification to vetted files and rejects files that are not vetted.

- *Malicious Use of Another Layer 7 Protocol* – An attacker can use a non-HTTP protocol to steal information. We believe that our techniques for preventing leaks extend to other protocols, but must now rely upon other systems and policy to prevent leaks in these protocols. For example, many protected network firewalls only allow certain ports to be accessed externally (thus, only allowing the associated protocols to be used). Glavlit provides the additional assurance that another protocol is not being used on an HTTP port or that unauthorized HTTP communications on other ports go unmonitored.

- *Compromised Web Server* – An attacker has full access to a protected Web server and may modify its configuration or replace it with a rogue server. The attacker can now embed covert channels in HTTP protocol or timing to encode information. Glavlit detects this activity by verifying the validity of all protocol responses and normalizing the server's response time. Even if an attacker creates a rogue server that does not use covert channels, Glavlit only allows it to serve *vetted* content.

- *Compromised Warden or Guard* – Unfortunately, we must prevent this type

of attack by assumption. For example, a malicious insider with appropriate permissions can use the Warden to vet unauthorized content, allowing it be released by the Guard. We assume that access to the Warden and Guard is more closely controlled than other hosts in the system like user workstations or Web servers. We also assume that the Warden uses its own *independent* authentication system to grant vetting access.

2.1.2 Comparison to Other Approaches

Other solutions have attempted to achieve data confinement. For various reasons we did not find these solutions to be appropriate for achieving our goals of high speed and transparency. We examine some of these previous approaches below to provide some background on the design and implementation decisions that we made.

- *Policy* – Organizations can write complex policies to achieve data confinement. Policy may dictate user practices, computer configuration requirements, and accountability. Unfortunately, most policy to prevent information release is difficult to enforce. Users may misbehave and computers may malfunction or be attacked [2, 43].

- *Private Unconnected Networks* – Organizations can establish unconnected networks to *air-gap* protected content from the Internet. There are four problems with this approach: it requires considerably more resources to operate multiple networks (often restricting this approach to governments and large organizations), the sensitivity levels for each network are too coarse-grained, information is duplicated across sensitivity levels leading to inconsistency, and lack of Internet connectivity decreases productivity for users of the private networks. Most organizations operate only one network and must rely upon policy to mitigate information leaks.

- *Firewalls* – Sophisticated firewalls can scrub network traffic transparently at line speed and detect malicious use [57]. Firewalls can search network data streams for sensitive keywords and data, e.g., customer information or credit card numbers. However, firewalls lack the ability to do more than rudimentary analysis for sensitive data. They are unable to read complex file formats or perform analysis where the whole object is required to determine its sensitivity (such as images or Microsoft Office files).

- *Application Proxies* – Proxy servers can look for protected content leaving the network and actively block it when it is detected. These systems are often used for email and web traffic. They can employ complex content analysis because the proxy can delay the traffic until completing its analysis. These systems incur large overhead to operate and require special client configuration.

- *Passive Content Control* – Content monitoring systems can examine outgoing network traffic for sensitive content. These systems can only audit rather than verify real-time traffic like HTTP. These systems can employ powerful (and potentially time consuming) techniques for sensitive content identification because the analysis is done off the critical network path.

Many of these systems also contain one additional important flaw: the system can no longer guarantee data confinement upon compromise of the hosting machine. Generally, even if the security system is remains functional, the compromised host could still transmit data through covert channels. In fact, the only system that can guarantee against this is a disconnected system, which does so trivially.

2.1.3 Architecture Design Discussion

Simmons and Lampson's seminal discussion of confinement provides the basis for our Work on Glavlit and the classes of information release that we have

developed above [52, 33]. Unfortunately, program confinement is inadequate to cover all forms of information release. Computer system users also introduce factors that affect or create channels for release. For example, inadvertent release is often the result of policy not being followed or user confusion over what is sensitive and what is not.

Channels may be inadvertent or malicious, covert or overt, and authorized or unauthorized. Given the complexity of these factors, it is infeasible to develop a solution for each specific scenario that may combine any number of these factors. Furthermore, the danger and consequences to the owner of a protected network remain the same regardless of the nature of how the leak occurred. For example, the consequence of a valid user inadvertently leaking intellectual property via email and an attacker exfiltrating the same information using a covert channel are equal.

In designing Glavlit, we choose to treat all the threats to information release equally. This distinction is most important when considering different combinations of intent and visibility. We make assumptions neither on the intent of users nor the validity of software within the protected network. This decision allows Glavlit to cover a wider range of threats with a unified design.

Since our system relies upon transparent network monitoring and blocking, end-to-end encryption like SSL or IPSEC presents problems. Though our implementation of Glavlit does not support SSL, techniques used by network intrusion detection systems could be added to support it [5]. Since an organization owns the systems within its protected network, it can share the private SSL keys from a web server with Glavlit. Glavlit can now decrypt SSL traffic in-line in the network allowing the Guard to use clear-text analysis techniques. The Guard can re-encrypt the data if it is performing in-line modification or reuse the original packets if not. A similar technique can be used for IPSEC tunnels.

2.2 Techniques for Preventing Leaks in HTTP

In this section, we discuss the techniques we use in Glavlit to prevent information leaks using HTTP. This section provides an overview of the potential leak vectors and techniques. A further exploration of how our techniques and their motivation can be found in [51]. We first focus on protocol channels in HTTP and what Glavlit does to mitigate them. We then discuss how to prevent leaks in the payload of the HTTP response.

2.2.1 Structured Protocol Carriers

Parsing

HTTP is a command-based protocol that uses headers to specialize commands [4, 16]. An attacker can hide data in this structure to create covert channels (many protocols follow a similar pattern in their structure; therefore, this discussion can be extended to other protocols like FTP or SMTP). Since the HTTP protocol was designed for maximum flexibility, the syntax does not impose sufficient restrictions to limit covert channels. For example, variable amounts of white space can encode data because the RFC allows any amount of white space between tokens. A channel using a similar technique within HTML has been demonstrated [13].

Since we assume the Web server could be compromised, Glavlit parses outgoing responses using a more restrictive grammar than the one given in the RFC to prevent channels of this type. The outgoing parser restricts white space between tokens to a single space. Glavlit defines a fixed set of acceptable response headers and a predefined order per request type to mitigate channels which use the presence of headers to encode information. The parser also checks that date, numeric, and text fields are only composed of characters that befit those formats (e.g., alphabetic characters are not allowed in a numeric header like Content-

Length). Though we impose restrictions on the output syntax of the server, we have found that existing servers such as Apache already follow most of our guidelines. Glavlit also parses incoming requests to correlate values of the server response with the corresponding request. Since we do not control systems *outside* the protected network, the parser for incoming requests uses the standard RFC grammar to allow any client to connect.

Semantic Value Checking

An attacker can use header fields like Last-Modified or Content-Type to encode data [12]. Most HTTP fields are *verifiable* given sufficient information about the nature of the request and the content served. To accomplish this verification, we restrict the RFC such that header types are known and each field can be verified. In some cases, the Web server must be configured to comply with these restrictions. In most others, the file metadata can be used to verify response headers.

The Warden stores appropriate metadata about the file in the signature database. Because many headers like Content-Length and Last-Modified can be verified using simple file system metadata, the Warden can automatically determine these values during vetting. The Warden stores this metadata as part of the file signature. Some metadata is not available directly from the file system and must be determined by or provided to the Warden. For example, the Content-Language (if appropriate) should be specified to the Warden while the Content-Encoding can be determined using file type identification. Some headers require server configuration and cooperation. Often restricted configuration leads to less flexibility because a header may only take on a single value or few values based upon certain conditions from the corresponding request. In practice, operating a Web server only requires the use of a small number of these headers; therefore, we feel this limitation is acceptable.

2.2.2 Unstructured Protocol Carriers

Order Enforcement

Request order can encode information unidirectionally *into* a server [14]. Similarly, bidirectional communications can be made by reordering pipelined responses. The HTTP RFC defines that the order of responses should match with the order that they were received. Glavlit maintains this invariant by ensuring that the responses return in the same order as requested.

Timing Channel Mitigation

An attacker can encode information based on the timing of network activity. Specifically, we focus on timing channels which use delayed HTTP responses to encode information at the application layer. We jam potential timing channels by introducing *jitter* into the data stream. Kang et al used a similar idea in the Pump by normalizing and smoothing the timing of acknowledgments between a high sensitivity system and a low one [29]. Glavlit can delay HTTP responses by a uniformly random variable or follow a fixed probability distribution to minimize channel capacity. Normalized responses are often not apparent to a user but can be devastating to a covert timing channel.

We model the HTTP response timing channel with jitter as a timed Z-channel [41]. The server responds in time t_0 to encode a zero and in time t_1 for a one. We write the Z-channel as $Z(\epsilon)$ where $\epsilon = t_1 - t_0$ and $t_0 < t_1$. Since we are establishing an upper bound on the channel capacity, we only consider the artificially inserted jitter and do not consider either the additional propagation delay added by the network or the server processing time (hence, $t_0 = 0$ and $\epsilon = t_1$). Like a standard Z-channel, a zero may be transformed into a one with probability q and a zero is transmitted correctly with probability $p = 1 - q$. We will assume that zeros and ones are equally probable in the input (i.e., as though the input were compressed and/or encrypted). We also assume the attacker does

not use coding. Let J be a uniformly random variable representing the jitter delay. Let $P(J = j_i)$ be a uniform distribution from 0 to j_t where j_t is the maximum jitter factor. We wish to determine how j_t affects q and the usable bandwidth of the channel. If $j_i > \epsilon$, then the jitter has changed a zero into a one. The probability q that this occurs is given in (2.1).

$$P(j_t > \epsilon) = q = \begin{cases} 1 - (\epsilon/j_t) & j_t > \epsilon > 0 \\ 0 & j_t \leq \epsilon \end{cases} \tag{2.1}$$

Thus, if ϵ is 50ms, the attacker can theoretically transmit 40 bits/s with j_t=0. When j_t=250ms, the channel is completely noisy with probability 0.8 and the capacity is reduced greatly. To thwart the jitter, the attacker can increase ϵ until it dominates the jitter factor and $q = 0$. If we can disallow a response greater than j_t, then the attacker is forced to utilize coding to extract usable capacity from what remains of the disrupted channel.

Recall this analysis is a best-case scenario for the attacker, and realistically, usable capacity will be even lower due to network effects. The degree to which we can disrupt the channel usability depends on the delay used by the attacker to defeat the noise introduced by the network or the jitter system. Consider a worst case scenario for jitter insertion: the attacker uses a large $\epsilon = 100ms$ and is physically close with propagation $delay = 20ms$. The attacker has limited her bandwidth to 29 bit/sec; however, to induce the channel to be noisy with probability 0.9, the Guard must use j_t=1 second.

2.2.3 Payload Channels

Thus far we have only examined the protocol or its operation as the channel carrier. Seeing that the purpose of HTTP is to serve file content to clients, it follows that an attacker could easily utilize the payload to steal information. One technique is to place information to be stolen in the publicly accessible Web tree

[13, 14]. An attacker may also embed malicious data in an otherwise harmless looking file (e.g., using steganography to embed data in a banner image on the site without affecting its appearance to the naked eye).

When the channel carrier is a file, the attacker can utilize complex and exceedingly difficult to detect steganographic techniques to hide her data. Though some detection and defeat mechanisms exist for combating the use of steganography most are specific to a single file type. They are also potentially time consuming and often require the entire file to analyze. It is impractical to employ these defense mechanisms at the network boundary. We address this problem by splitting content analysis (*vetting*) from enforcement at the network perimeter (*verification*).

To allow generic and powerful vetting techniques to be employed we pre-vet the set of files that may leave the network. This allows us to use powerful review techniques during the vetting process. For the purposes of this work, we assume that existing techniques are sufficient to ensure that once a file is verified (either through automated techniques or human review) its content is safe for release. Once vetted, files can cross the network boundary unhindered while unvetted files are blocked by the Guard. We minimize the number of distinct server responses by verifying content. If the content requested for banner.jpg can be only one file that has been vetted, channel capacity is greatly reduced.

Once content is cleared for release it cannot be subsequently modified. File integrity detection tools can be used to ensure this invariant [31]. Unfortunately, these systems rely on the integrity of the system hosting the files. An attacker prepared to ex-filtrate information has likely already compromised the host rendering local file integrity checks useless. With Glavlit, any host in the network can be compromised, but only approved data will be released as long as the Warden and the Guard are secure.

Vetting

Users send objects they wish to vet to the Warden, which grants or denies the object the ability to leave the network as specified by policy. Once approved, the object is partitioned into *chunks*, each hashed. The resulting collection of hashes, named a *file signature*, enables high speed verification at the Guard. To preserve name transparency we choose to identify files by their content. For example, once a file has been vetted it can be placed on any server within the protected network.

The vetting process can be as simple as a keyword search, or as rigorous as requiring approval from a committee of human analysts. Organizations can customize the vetting process to their needs. For example, a health insurance company may want to screen for medical data while the military may want to perform more rigorous review for image steganography. Even if an organization is not concerned with covert channels, Glavlit provides an in-network mechanism to detect and prevent website defacement.

Verification

Verification ensures that data was previously vetted. This process consists of locating the data within the network stream and comparing the hash of individual chunks to a pre-existing object signature. We use two methods of identifying files by their content: lookup hashes and chunk signatures. If using the lookup hash method, Guard performs a hash on the beginning part of the file content and uses the result to locate the appropriate file signature. Collisions between lookup hashes are not allowed. If using the chunk signature method, the Guard hashes the first chunk of the file and locates all possible files which begin with that chunk. The Guard resolves collisions between files whose starting chunks match iteratively as more distinct chunks arrive.

Once the Guard identifies an object's signatures, it hashes each chunk

of object data and compares the result with the known hash. If any chunk does not match, the connection is bidirectionally terminated by injecting a TCP RESET to both client and Web server. Since chunks may cross packet boundaries and since packets may arrive out of order at the gateway, Glavlit may have to perform some packet buffering for further analysis before forwarding the packets. However, if the amount of required state becomes too large (e.g., as a result of some internally-mounted denial of service attack), it is always safe for Glavlit to drop buffered packets to maintain its data confinement guarantees.

2.2.4 Vulnerabilities and Attacks

While we acknowledge that covert channel defeat is generally intractable, we can eliminate many covert channels completely and reduce the capacity of the remaining channels. We provide *limitations* on the degree to which data confinement is violated. Our goal is to make it more difficult to maliciously or accidentally leak sensitive data while still remaining backward compatible with existing protocols. There is still the possibility that an attacker can leak information using covert channels that are not covered by our system. There are also methods by which an attacker can attack the system, which we describe below.

Theoretically, an attacker can use files which contain identical chunks to encode data. For example, an attacker can create a file that is very similar to a vetted file with a modification to the last chunk. Requests for the vetted file will succeed while requests for the modified file will stop after transmitting the modified chunk. The attacker can manipulate which file is returned for each request thereby encoding data to the outside. To mitigate this risk, the Guard could flag a file as suspicious and buffer more data to check future chunks before sending previously verified chunks. This method has limitations as the size of the server's transmission window will dictate the maximum buffering the Guard

can perform before stalling the connection.

By resetting the connection when it encounters bad data, the Guard is vulnerable to a denial of service attack against the servers in the protected network. If an attacker can man-in-the-middle a valid connection, then she can inject a properly formed packet with invalid data. Since the forged packet's TCP header is correct, but the contents are invalid, the Guard will detect that an unauthorized leak is in progress and will reset the existing external connection to the Web server. Conceptually, this problem is the same if the Guard drops packets rather than terminating connections. The attacker must be able to watch a valid connection (to obtain the correct TCP header fields) and potentially block the real sender from sending a valid packet. To minimize this risk in our implementation, we reset connections only because of invalid *outgoing* packets. Incoming invalid packets are dropped; however, the connection is not blocked to allow the *actual* sender's packets through.

2.3 Acknowledgment

Chapter 2, in part, has appeared previously in the Proceedings of the 5th ACM Workshop on Hot Topics in Networks, 2006, ACM SIGCOMM. Nabil Schear Carmelo Kintana, Qing Zhang, and Amin Vahdat. The thesis author was the primary investigator and author of this work.

Chapter 2, in part, has also been submitted for publication in the Proceedings of Ninth International Conference on Information Hiding, 2007, Springer-Verlag. Nabil Schear, Carmelo Kintana, Qing Zhang, and Amin Vahdat. The thesis author was the primary investigator and author of this work.

3

Implementation

3.1 Warden

The Warden generates and stores the table of file signatures and metadata used to enable the verification described in Chapter 2. The Warden also provides a network interface to allow Clients to send files to be vetted remotely. The Warden's operation is closely related to the data structures it uses to store file signatures.

The Warden implements a *null* vetting process whereby all files pass vetting. We have implemented neither secure communication nor authentication between the Warden and the Client since they are solved problems and tangential to the main thrust of our work.

3.1.1 Data Structures

The Warden uses three interrelated structures to store signatures:

- struct signatures – Stores the signature table and all the parameters related to signature generation. The primary member of this struct is the hash table of signatures.
  - chunk_size – the number of bytes in a chunk

  - hash_func – the string name of the hash function used to create signatures
  - method – the method used to store signatures, supports lookup or chunk methods
  - lookup_size – the number of bytes from the beginning of the file to hash to uniquely identify it
  - ht – a hash table which stores files keyed by their lookup hash (if using method lookup) or chunks keyed by their hash (if using method chunk). Figure 3.1 shows the layout of the file_sig and chunk_sig structures for each method.
  - all_files – a queue of all the files that are stored in the signature table. In chunk mode, the Warden uses this queue to easily iterate over all the possible files since the hash table will not contain files.

- struct file_sig – stores the signatures and meta-data for a single file. Every file_sig points to all the chunk_sigs it contains.
  - filename – the name of the file stored a string
  - max_size – the size of the file in bytes
  - key – the unique key of the file and is used to detect collisions. This is either a digest of lookup_size bytes (lookup mode) or a digest of the entire file (chunk mode)
  - num_chunks – the number of chunks in the file
  - chunks – an array of pointers to chunk_sigs for each chunk in the file

- struct chunk_sig – stores the signature for a single chunk. It also points back to all the files that contain this chunk. This allows file_sigs and chunk_sigs to be doubly linked.

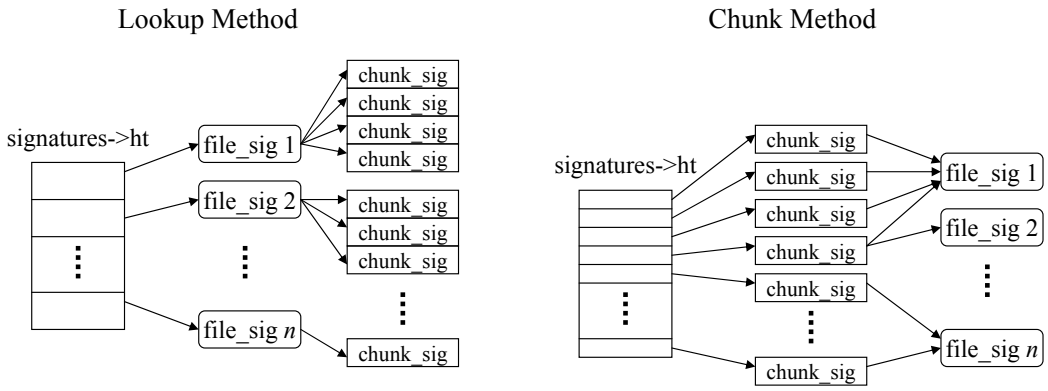

Figure 3.1: Signature Table Layouts

- sum – the digest of the chunk
- length – the length of the chunk in bytes
- num_files – the number of files that contain this chunk
- file_list – a queue of the files that point back to this chunk

3.1.2 Implementation Details

On start up, the Warden reads the signature file from disk or creates one if it does not exist. The Warden ignores command line parameters that conflict with the parameters stored in an existing signature file. It keeps the entire signature table in memory to resolve collisions during Client requests. When the signature table is updated, the Warden syncs the file to its text representation on disk. The signature file is text-based to enable quick human validation. Figure 3.2 shows a sample signature file containing a single file.

The Warden listens for Client requests on a socket. When a Client connects, the Warden parses the Client's request to determine the filename, size, and type of operation requested (add or delete). The Warden reads the file content over the network and creates a temporary copy of the file in the directory .warden_tmp. The Warden then vets the file. Since the file is saved locally, the

```
method: Lookup
chunk_size: 1024
lookup_size: 256
hash_func: SHA1

name: index.html
size: 6087
num_chunks: 6
21 B0 9A 46 A5 D1 9F 17 AF CE B 57 1B 81 BD 3B B8 74 42 62
0 1024 E5 1 25 B1 D 2E 74 34 90 6C 6E 2D 6D 94 1A 26 61 BE 33 6B
0 1024 7E B5 21 85 B2 68 D1 5C D8 73 4E E5 6B B3 5F 9D BC C3 74 AF
0 1024 BB 3C EE EE 35 61 50 52 85 BF 23 C7 64 67 A8 D0 7C B7 3E AD
0 1024 B6 D5 C5 57 A6 83 AB 1B DE 5E F8 EF CF CB 27 86 BB C9 A2 63
0 1024 4 DD DA F0 C8 84 E7 C3 E0 E6 2F E7 57 A4 87 73 18 7 9B E4
0 967 F9 5 8B AC E9 C2 26 AD 34 A1 40 B6 7B 31 C EF 5 39 2E 63
```

Figure 3.2: Sample Warden Signature File

Warden can use any review or steganalysis tool without modification.

Once the file is approved, the Warden generates signatures from the temp file and places them in a file_sig structure. At this point during vetting, the temporary file_sig is independent of the signature table. To generate signatures, the Warden splits the file into chunk_size segments and hashes each with the specified hash algorithm.

The temporary file_sig is then inserted into the signature table. If using the lookup method, the key of the new file_sig is compared against those in the hash table. If a collision occurs, the Warden returns an error to the user. If using the chunk method, the Warden inserts each of the file's chunks into the hash table keyed by its secure hash. It places the file_sig in the all_files queue in struct signatures.

For a delete operation using the lookup method, the Warden locates the file_sig in the hash table and removes it. It also reclaims all the memory used by the file's chunk_sigs. If using the chunk method, deletion is more complex. The Warden removes the file_sig to be deleted form the all_files queue. It then iterates over each chunk in the file. If the chunk is unique to the file to be

deleted, the Warden removes it from the hash table. If the chunk is not unique, the Warden removes the pointer to the file to be deleted from the chunk's file_list and decrements num_files.

3.1.3 Usage and Options

The Warden sets the parameters for signature generation through the following options:

warden -[chmolpqsf] [file1 file2...]

-c size set the default chunk size in bytes to size. If unspecified, it defaults to 1024 bytes.

-l size set the lookup size in bytes to size. If unspecified, it defaults to 256 bytes.

-h hash_func set the hash function used to create the signatures. Possible values for hash_func are SHA1, MD5, MD4, RIPEMD. SHA1 is the default if unspecified.

-m method define how the signatures are generated and stored in the signature table. Possible values are lookup and chunk.

-o filename read and write the signatures from this file. If it does not exist, the Warden will create it.

-p port sets the listening port number where the Warden will listen for Client connections. Not compatible with offline mode -f.

-q enable quiet mode where warning and informational messages are suppressed.

-f offline mode: accept file(s) from the command line to add to the signature table. Only adding to the signature table is supported offline.

3.2 Client

The Client is a UNIX command-line tool for sending files to be vetted to the Warden. It reads a variable number of files from the command line and transmits them over a socket connection to the Warden for vetting. The Client uses a simple protocol to specify to the Warden the name, size, and other meta-data used for vetting the file. The Warden returns a status message to the Client through the socket indicating the result of the vetting request. The Client can also instruct the Warden to remove files from the signature table remotely. The files to be removed must be available to the Client and must be unmodified from the time they were originally vetted.

3.2.1 Usage and Options

The Client takes a variable number of files from the command line. It also supports the following options:

client -[phd] file1 [file2 file3 ...]

-p *port* sets the remote port number of the Warden. If unspecified, it defaults to 8001.

-h host-name sets the remote host name or IP address of the Warden. If unspecified, it defaults to localhost.

-d Issue a delete request to the Warden. The file(s) passed on the command line will be processed and deleted from the signature table.

3.3 Guard Implementation

The Guard operates as a network bridge by capturing packets exiting the network in promiscuous mode or as a router where it is an explicit hop in

the network. At a high level, we created a pipelined, multi-threaded, event-driven system for the Guard. The Guard is broken down into components each powered by a thread that operates on the incoming or outgoing stream. In future work, we hope to increase the number of threads per component to increase overall throughput across many connections. Each thread has a work queue and processes events until it exhausts its work and goes to sleep. We have implemented rudimentary load shedding inside the work queues. In future work, we anticipate that this design will allow us to more easily perform dynamic thread allocation and batching control to exhibit well conditioned behavior.

3.3.1 Data Structures

A *flow* is conceptually one half of a full duplex TCP connection and is identified by the distinct tuple: source IP, source port, destination IP, and destination port. Our system treats incoming flows differently from outgoing flows. Both kinds of flows store a pointer to the memory for the reconstructed stream, TCP sequence numbers, object parsing offsets, and a mutex. Each outgoing flow called a *flow* also has a list for layer 4 and below packet headers stored in a structure called a *flow_pkt*. The flow_pkt keeps an offset to its corresponding data in the stream. Since incoming data is sent immediately, the *incoming_flow* does not need a packet header queue. It stores a queue of parsed HTTP requests to enable verification of outgoing responses.

When the Guard sees a new flow, it adds it to a hash table keyed by its distinct tuple called *all_flows*. This distinct tuple is called an *all_flows_key* and is used to pass work from one thread to another's work queue. This allows a thread to look-up the flow before processing to ensure it has not been asynchronously deallocated. There is a similar hash table for incoming flows called *all_incoming_flows*.

Locking Behavior

Since the Guard is multi-threaded with many shared data structures, we developed intricate locking rules to prevent dead lock. We describe the locking rules in each of the following structures:

- Work Queues (work_queue, work_heap) – Each work_queue contains a mutex and condition variable to allow threads to share work (flows and packets) between different stages of the packet pipelines. All work_queue locks are independent and may be acquired at any time. The thread that is the owner of the work_queue may wait for work using pthread_cond_wait. Any thread that adds work to the queue must signal that the work_queue state has changed to wake up the owner thread. Each work_heap also contains a mutex and condition variable and is used in the same manner as a work_queue. The only difference is that work on the heap is kept ordered. The jitter insertion thread uses a work_heap rather than a work_queue.

- Flow Hash Tables (all_flows, all_incoming_flows) – We use the all_flows lock to mutually exclude threads from accessing flows and the hash tables of flows simultaneously. When passing flows from thread to thread, the Guard uses an all_flows_key rather than a pointer to the flow. A pointer to a flow is only valid when it is locked. To access a flow that is not yet locked a thread must: 1) lock all_flows and search the hash table for the flow it wishes to access, 2) lock the flow if it is found, and 3) unlock all_flows (Figure 3.3). No flow may be locked without first locking all_flows. The all_incoming_flows structure mimics the locking and operation of all_flows. To simultaneously lock both all_flows and all_incoming_flows, the all_flows lock *must* be acquired first.

- Flows (flow, incoming flow) – Each flow contains a lock which prevents other threads from accessing the data stored in the struct or any data

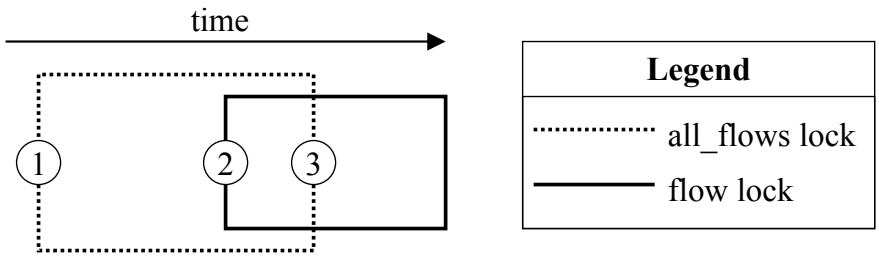

Figure 3.3: Flow Locking Procedure

to which it points (e.g., the reconstructed stream). A thread can hold a flow lock without holding the all_flows lock. A thread may search all_incoming_flows while holding a flow lock. A thread *may not* search all_flows while holding an incoming_flow lock. Only during stream reallocation, flow deallocation, and object verification are both incoming and outgoing flows locked simultaneously.

3.3.2 Incoming Packet Pipeline

The incoming packet pipeline processes the packets that are received on the external interface and forwarded into the protected network. We monitor these packets to parse the requests that correspond with the HTTP responses that exit the network. Figure 3.4 shows the data flow for the incoming packet pipeline.

External Receiver

The incoming receiver listens on the external Ethernet interface and uses libpcap to capture packets. As incoming TCP packets arrive, the external receiver identifies, checks, and forwards them immediately to the packet sender. We do not delay the incoming packets until after they are parsed like we do for outgoing packets. Therefore, the external receiver operates passively in the network.

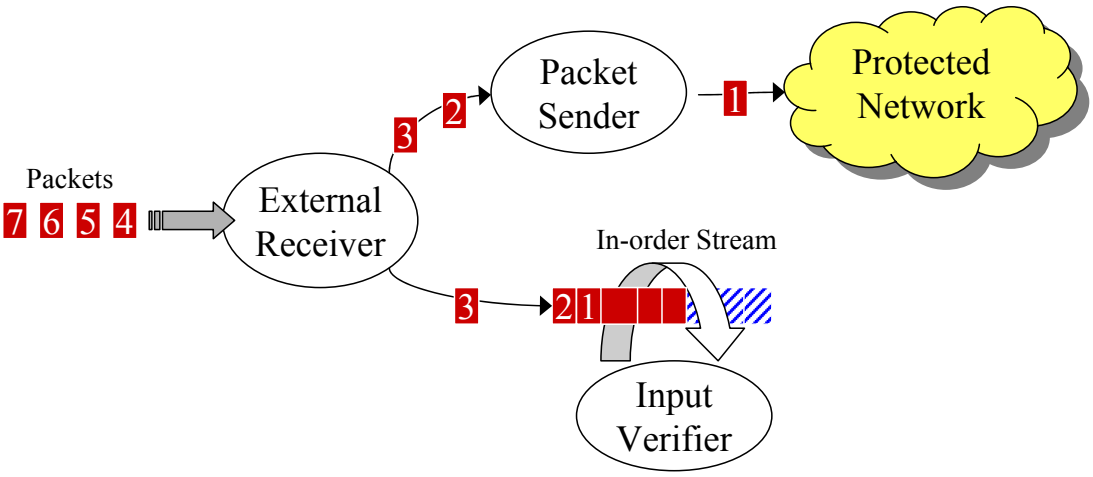

Figure 3.4: Incoming Packet Data Flow

When it receives a SYN packet, the external receiver creates an incoming_flow structure and stores it in the all_incoming_flows hash table. It keeps the initial sequence number for the flow and notes that the flow will have one phantom byte created during the TCP handshake. As later TCP packets arrive, the external receiver checks their sequence number against the expected sequence number based upon the packets the Guard has already received. If the packet is the next expected, the external receiver copies their data into an in-order byte stream for the input verifier. It drops future packets because the stream is append only and we do not fully support SACK.

Input Verifier

The purpose of the input verifier is to parse requests to the protected Web server to correlate the server's response with the external Web client's request. Like the outgoing verifiers, the input verifier operates on an in-order TCP stream. It waits for incoming_flows to be placed on its work queue and wakes up when new requests arrive. As an optimization, the external receiver only sends packets to the input verifier which have non-zero payload length to prevent spurious wake-ups.

The input verifier parses incoming HTTP requests and stores the results

in a queue in the incoming_flow. The parser uses the grammar and syntax described in the RFCs directly without the restrictions we describe for mitigating covert channels. We implemented the parser using flex and bison that is fully reentrant. The parser creates a *req_parser_result* which contains all the relevant information from the request. Because the outgoing verifiers will need this result an unspecified amount of time later, the parser copies all data placed in the req_parser_result. This allows the stream memory to be reclaimed immediately if necessary. The parser stores information such as the file requested, connection status, HTTP version, and accept header.

Packet Sender

The packet sender thread takes flow_pkts from a work queue and sends them on the protected network interface. The Guard creates two packet sender threads: one for the protected interface and one for the external. The packet sender is also responsible for modifying the Ethernet headers for forwarded packets. If the Guard is running in router mode, the packet sender replaces the destination MAC address to be the gateway for the network (specified on the command line). It also sets the source MAC address to the MAC address of the sending interface. If running in bridge mode, it does not modify the Ethernet header.

3.3.3 Outgoing Packet Pipeline

Protected Receiver

The protected receiver captures packets leaving the protected network. As with incoming packets, it uses a classification mechanism to determine the protocol. Non-TCP packets are sent directly to the packet sender and transmitted immediately out of the protected network. When it receives a SYN-ACK packet, the Guard locates the corresponding incoming_flow. If an incoming_flow ex-

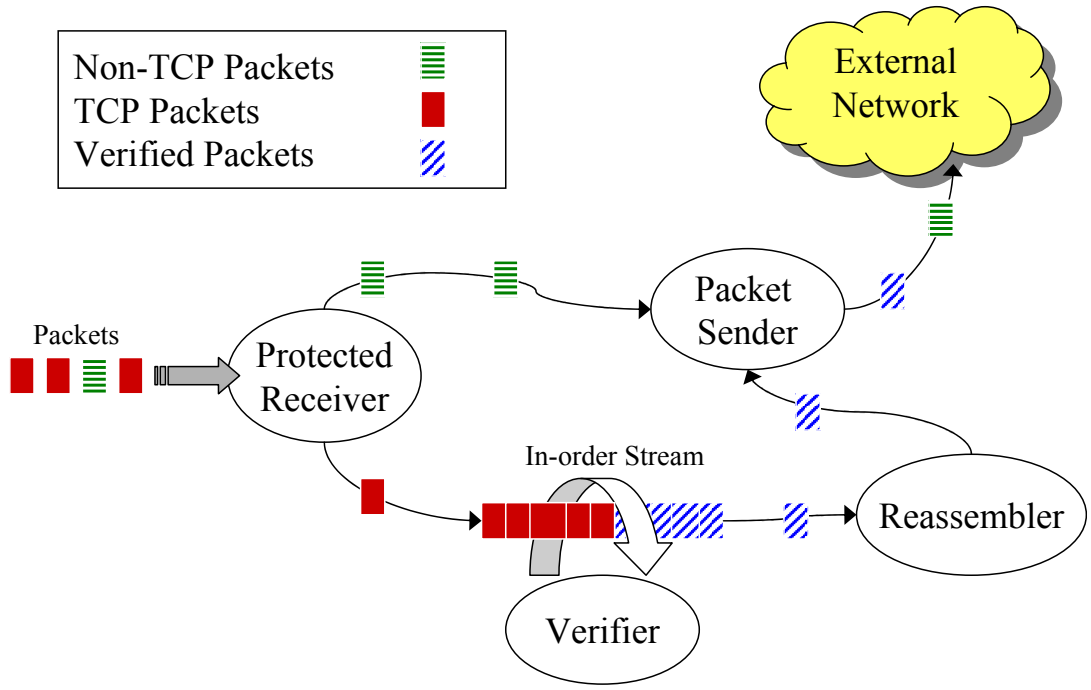

Figure 3.5: Outgoing Packet Data Flow

ists, it creates a new flow and inserts it into the all_flows hash table. After it checks the TCP sequence number is of a new packet, the packet data is copied into the stream for the flow. Corrupt packets, future packets and non SYN-ACK packets for which a flow does not exist are dropped. The packet headers are placed separately onto a queue stored in the flow. Since all TCP packets will be eventually reconstructed, the protected receiver preallocates enough space for the entire packet when creating space for the headers to be placed on the queue.

Because verification occurs in a linear one pass fashion, we keep a packet cache in the stream to handle retransmitted packets. The size of the packet cache grows with the window size of the TCP flow. The stream is a *rolling buffer* of in-order packet data as shown in Figure 3.6. As more data is received, Segment 2 moves to the left and the packet cache grows. When available space in Segment 3 empties, the Guard shifts all pending data and the data of all unacknowledged packets from the packet cache back to the start of the buffer. Using the ACK numbers from the incoming_flow, we can precisely determine the

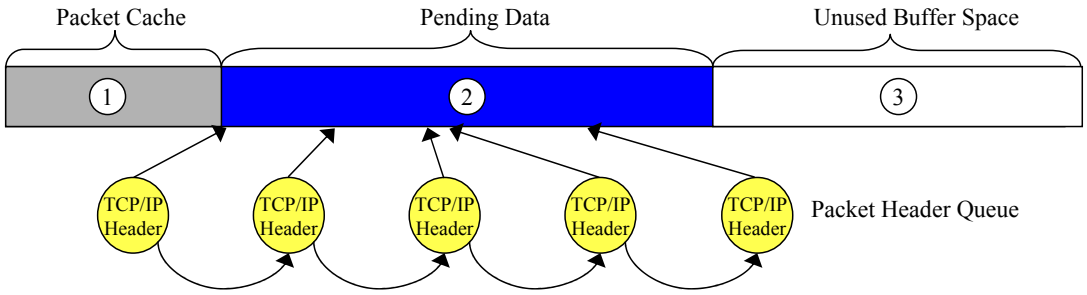

Figure 3.6: Flow Data Structure

minimum size of the packet cache for any outstanding packets in the network. If the entire stream is pending and new data has been received, the stream buffer is reallocated to double its original size. In future work, we plan to tune how the Guard utilizes memory by balancing reallocation versus memory copy operations.

When it receives a retransmitted packet, the protected receiver locates the data in the packet cache using the packet's sequence number. It then compares the payload of the packet to the verified bytes in the packet cache rather than repeating the verification. If the data is the same, the packet is sent immediately by the packet sender. If the packet is not in the packet cache, it is dropped. If a packet is not in the cache, then by definition it must have already been acknowledged would eventually be dropped by the client. If the packet is still pending verification, it is dropped to allow the original pending packet to finish. If the old packet data does not match, the flow is marked to be reset.

The packet header queue stores offsets from the beginning of the flow rather than pointers into the stream. Using the pointer to the buffer storing the stream and per-flow correction factor the Guard can compute a pointer to the packet data from the offset stored in the flow_pkt. This allows us to shift the stream without having to update all the active pointers in the header queue.

Reassembler

After verification (described below), the reassembler thread recombines the header and payload of each packet. Each flow contains an offset to the last verified byte in the stream called *obj_pos*. It checks obj_pos against the offset and length of the first packet in the header queue. If obj_pos has crossed a packet boundary, the packet can be sent.

The Reassembler uses the preallocated space in the flow_pkt to store the TCP payload from the stream. Since the Guard does not modify the data, the Reassembler does not need to recompute the header checksums. If a flow is marked to be reset, the Reassembler generates two TCP RESET packets using the sequence numbers from the offending packet. We choose to reset the connection because it is very difficult to locate where in the stream good verifiable data starts. Recovery is possible, but the overhead finding good data and nullifying the bad data is too large.

3.3.4 Content Verification

The hash and protocol verifiers are collocated in a single thread and work together on the stream to verify outgoing data before allowing it to be sent. Figure 3.7 shows the data flow of the outgoing verifiers.

Protocol Verifier

The protocol verifier (PV) first parses the HTTP response. The response parser uses the specifications in the HTTP RFCs [4, 16] with the restrictions from Section 2.2. Because the parser modifies its input, the PV copies the stream data to a temporary buffer before parsing. The parser returns a *parser_result* structure that contains pointers to all the HTTP response metadata in the temporary buffer. Since the Web server most often returns the HTTP response and enough content for a lookup simultaneously, the temporary buffer can be immediately

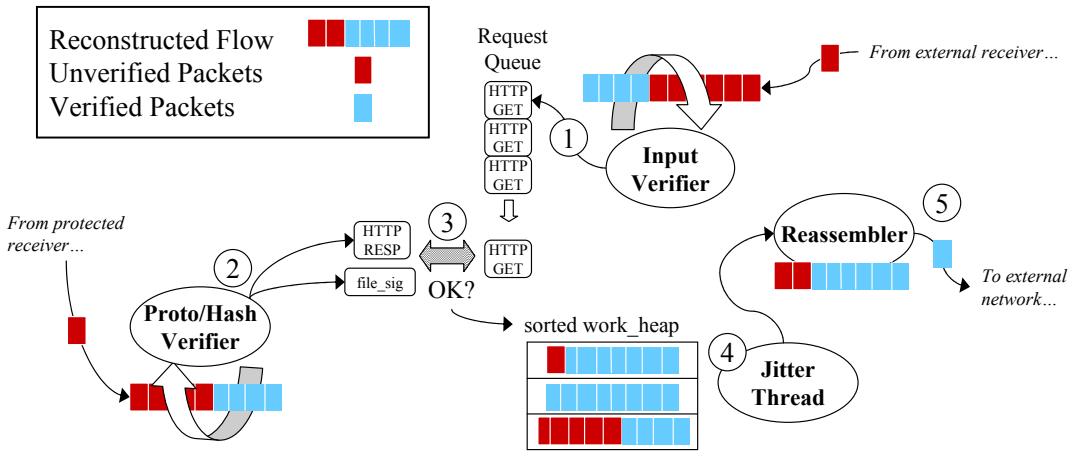

Figure 3.7: Verifier Data Flow

used to verify the protocol fields. If sufficient lookup data is not available, the PV will create a new temporary buffer allowing the original buffer to persist until its protocol fields are verified.

As described above, the input verifier places parsed requests on a queue, which ensures that responses are returned in the order they were requested. The PV locks the *incoming_flow* to retrieve the corresponding pre-parsed HTTP request (Step 1 in Figure 3.7). The hash verifier (HV) next identifies the file content (Step 2). To identify the file content using the lookup method, the HV hashes a lookup_size segment of the file and uses the result to query the signature table. When using chunk identification, the PV hashes the first chunk of the file and queries the signature table using resulting hash. If the chunk is not unique to a single file, the PV attempts to uniquely identify the appropriate file_sig by its name in the parsed request. If the PV still cannot uniquely identify the file, it iteratively calls the semantic value checking system for each possible file_sig and allows the response if any one matches.

The semantic value checking code takes the parsed response, parsed request, and file signature to verify that all the protocol information is correct to minimize protocol channels (Step 3). Currently it checks the file name, content

length, encoding, HTTP version, server type, and status code.

To be most resilient and efficient with respect to packet boundaries we could have split protocol objects into chunks, however this requires considerable parser state between chunks. Not splitting protocol objects results in two potential drawbacks: i) there is a small possibility that the HTTP server will not send the entire protocol object before exhausting its transmission window causing the Guard to stall the connection indefinitely; ii) we must incur the overhead of parsing from the beginning each time more data is received on an unfinished protocol object. Practically, protocol objects do not tend to be large and rarely cross more than one packet boundary.

We also set a maximum size of 1024 bytes for protocol objects to detect failure quickly. If there was no maximum size, the parser could potentially operate over a large amount of file content that followed bad protocol before failing. In practice this size is sufficient to cover HTTP responses from standard Web servers.

Hash Verifier

Next, the hash verifier (HV) takes over to verify the file content. The HV splits the available file content into chunks and performs a secure hash on each. In lookup mode, it checks the result of each hash against the one stored in the *file_sig*. When the hash verifier encounters a chunk that does not match, the flow is marked to be reset. As the hash verifier moves over the stream, it advances the offset to the last verified byte (*obj_pos*). On updates to obj_pos, the HV sends the flow to the reassembler to check if any packets on the flow are ready to be delivered.

When using chunk identification, the operation of the HV is complicated by chunk collisions between files. If the first chunk is not unique to a single file_sig, the HV creates a list of the possibilities stored as *possible_file* structs. The possible_file struct stores a pointer to each potential file_sig and an index

to the last verified chunk of the file. As further chunks on the file are hashed, the HV iterates over the possibilities to determine if it can eliminate any of the possible_files from the list. This process continues until there is only a single possible_file and verification can proceed as it does for the lookup method. Chunk collision resolution adds some overhead to the system, but we expect that extensive collisions are rare, and that most will be resolved after the first few chunks are received.

Jitter Insertion

The Guard can also insert uniformly random delays into each server's response time to disrupt potential timing channels. After parsing the response header, the PV marks the flow to be delayed. It then adds a random delay time within a pre-specified range (JITTER_FACTOR) to determine when the flow should be released. We delay the entire flow rather than just first packet of the response to prevent the attacker from simply using the subsequent packets as the signal that the request has arrived. The PV places the flow on a priority queue (*work_heap*) implemented as a Fibonacci heap. The jitter thread queries the work heap for the flow with the earliest send-time and sleeps until that time (Step 4 in Figure 3.7). When the sleep terminates, the jitter thread removes the head flow from the priority queue and enqueues it to the reassembler thread for delivery (Step 5). If the PV has a new flow to delay whose send time is earlier than the head of the priority queue, it signals the jitter thread to wake-up and reset the duration of the sleep to the shorter time. As the clock_gettime function we use has nanosecond accuracy whereas the Linux kernel can only provide accuracy on the order of microseconds, all time comparisons are fuzzy within $\pm 1\,\mu sec$ to prevent unnecessary nanosleep calls and heap reorganizations.

3.3.5 Flow Deallocation

After flows are closed or terminated, the Killer thread asynchronously recycles flow data structure memory. The Killer maintains a list sorted by age (called the *kill_list*) of flows that have been marked completed or reset. The thread wakes up on a specified interval and determines if the head of the kill_list has timed out. It continues processing the list until the head of the list has not yet timed out. This timeout is defined by TCP as exceeding two segment lifetimes (120 seconds). The Killer deallocates outgoing and incoming flows simultaneously; therefore, both must be timed out before deallocation can proceed. Since the Killer holds the flow hash table locks while it operates, we chose to make its operation infrequent by setting the wake-up interval to 2 seconds.

The external and protected receivers can also force deallocation of a flow if its tuple is being reused by a host. They do so by moving the flow to be reused to the head of the kill_list and calling the killer directly.

3.3.6 Usage and Options

The Guard takes the following command line options:

guard -e ifname/gateway -p ifname/gateway [-f filename]

-*e* ifname/MAC_ADDR sets the external interface and gateway mac address. For example: eth0/00:01:A5:20:13:DA

-p ifname/MAC_ADDR sets the protected interface and gateway mac address. For example: eth1/00:01:A5:20:45:C4

-f filename optionally set the path to the signature file to be used for verification. Defaults to ./output_sigs.

The Guard also supports the following compile time performance options:

ENABLE_VERIFY When true, the Guard will parse and hash all HTTP traffic leaving the protected network. If false, the Guard performs no verification and forward packets only.

CHECKSUM_HEADERS If true, the Guard checks of the TCP checksums of all received packets. Defaults to false for better performance.

NET_MODE Tells the Guard whether it is operating as a transparent bridge (BRIDGE_MODE) or as an explicit hop in the network (ROUTER_MODE) and adjusts the handling of forwarded packet Ethernet headers appropriately.

ENABLE_JITTER If true, the Guard uniformly delays HTTP responses between 1 and JITTER_FACTOR *µsecs*. This option requires ENABLE_VERIFY to be true.

ALLOW_UNSOLICITED_OUTGOING If true, the Guard will allow connections to originate from within the protected network. This option may not be used with ENABLE_VERIFY set to true. It is only used to test forwarding performance using Iperf. Defaults to false because in normal operation, all connections are initiated by external requests.

ENABLE_LATENCY_TEST When enabled, the Guard will time stamp the packet arrival and departure to determine latency of forwarding each packet from one network to the other. The Guard reports the average and standard deviation in *µsecs* of all forwarded packets when issued a SIG_TERM signal.

4

Performance Evaluation

4.1 Experimental Setup

In a real deployment, the Glavlit Guard would run as a transparent bridge connecting a protected network router to the gateway. However, to evaluate the performance of our prototype, we emulated this configuration with three serially connected machines. All the test machines were Dell PowerEdge SC1450s with two 2.8 Pentium 4 Xeon processors and 2 GB of RAM. Each machine has gigabit network interfaces connected with cross-over cables running CentOS 4.3 with a 2.6.9-34 Linux kernel.

The first machine emulates the protected network, and it runs the Apache Web server version 2.0.52-28. The second is the bridge between the two LANs. The last machine emulates the external requester using a custom client HTTP load generator that spawns a variable number of threads, each requesting a set of files from the protected network. The client uses HTTP version 1.1 using requests formatted similarly to the Mozilla Firefox Web browser. It requests 5 files using a persistent connection before restarting. We performed all experiments with 50 threads requesting files for 20 seconds unless otherwise noted. We performed all tests of the Guard without jitter enabled except those in Section 4.4.

4.2 Micro-Benchmarks

4.2.1 Profiling

We instrumented the Guard using GNU gprof to analyze the time it spends performing its tasks. We tested the Guard using 50 clients each requesting files between 256 bytes and 512 bytes for approximately twenty minutes. We found that the Guard spent upwards of 48% of its time receiving and reassembling packets leaving the protected network. Copying packet data onto the stream accounted for 2.05% of that time. The Guard also spent 28.97% of its execution time manipulating the flow hash tables. The work queues used to pass packets and flows between threads accounted for an additional 20.46%. The remaining 2-3% was spent verifying chunks and parsing protocol. In future work we plan to optimize the flow and work queue data structures to further enhance the performance of our system.

4.2.2 Forwarding Latency

We measured the outgoing forwarding latency of single file transfers using the Guard without verification (Guard-NV) and with hash/protocol verification enabled. We define forwarding latency to be the amount of time from when the packet is first received by the Guard until it is sent out of the protected network. Table 4.1 shows average per-packet forwarding latencies for a range of file sizes. The value in parenthesis is the standard deviation of each run. When verification is running, the high standard deviations are due to the hash/protocol verifier's batching behavior. Because the server will send as many packets as its current congestion window will allow, the Guard will often have more than one packet pending verification at once. The verifier will attempt to verify as much data as is available before releasing the flow to be sent. This causes some ready-to-send packets to wait for later packets to be verified. The Guard-NV shows less

Table 4.1: Forwarding Latency (μsec) of Guard and Guard-NV

File Size	256B	4KB	16KB	64KB	512KB
Guard-NV	500 (437)	526 (478)	574 (523)	605 (525)	635 (532)
Guard	676 (4771)	721 (5694)	1028 (6639)	1632 (12033)	1799 (13221)

variation because the Verifier (and its batching behavior) is removed from the pipeline.

4.2.3 Packet Forwarding Performance

To evaluate the performance of our network reassembly and forwarding implementation, we also tested raw forwarding performance of a single TCP connection using Iperf. The kernel bridge achieved an average bandwidth of 778 Mbits/s. The Guard-NV achieved 640 Mbits/s into the protected network and 341 Mbits/s out. The difference in performance depending on the direction of the data stream is due to the way the Guard forwards packets. It forwards *incoming* packets directly while outgoing packets are copied onto the stream, reconstructed, and then sent.

4.3 HTTP Performance

We tested total request latency and throughput while varying the file size using the HTTP client described above (Figure 4.1 and 4.2). We define total request latency as the time taken to request and retrieve a single file completely. The throughput is the rate of bytes in full responses received during the test time. Tested files ranged between 256 bytes and 512 KB on an exponential scale. We tested the Guard with full hash/protocol verification, the Guard-NV, the

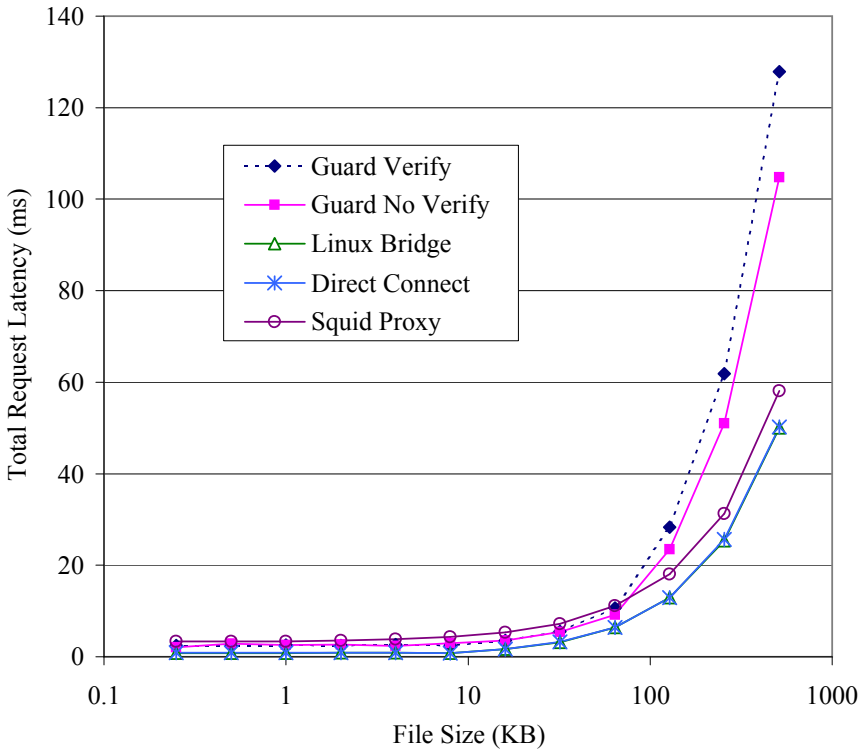

Figure 4.1: HTTP Request Latency versus File Size

Linux kernel software bridge, the Squid 2.5 Web proxy, and directly connecting the server and client. Each data point on these graphs represents the averaged result of five runs.

We see that all the systems perform similarly up to 32KB. After that point, the performance diverges as raw file data transmission time dominates connection establishment. The kernel bridge benefits from only having to copy each packet twice and being agnostic to connection overhead. Since Squid is a user space application, it incurs two additional copy operations to move the data to and from the kernel. The Guard trails the most because it also copies data in and out of the in-order stream. Verification adds only small overhead to the Guard. The latency of very small files for the Guard with verification increases because of the higher overhead of protocol request and response parsers.

For throughput, the transfer rates ramp up quickly as file sizes increase and then flatten asymptotically, as shown in Figure 4.2. The throughput of the

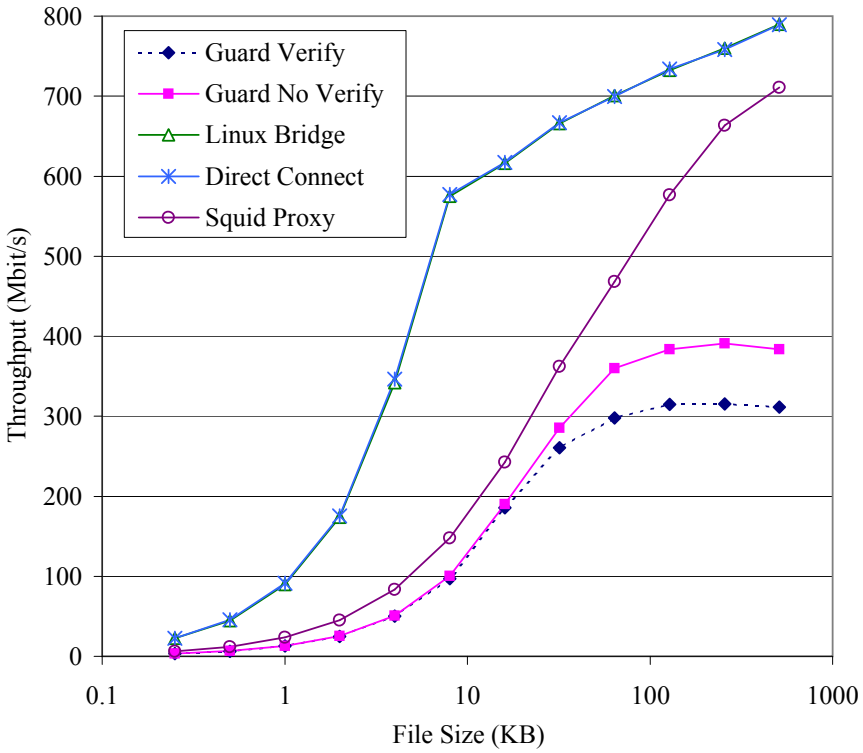

Figure 4.2: Throughput versus File Size

Guard and Squid grow more slowly because they incur a constant per-connection overhead. Verification adds little overhead to the Guard for files smaller than 32 KB. Since files on the Internet tend to average around 10 KB, we feel the Guard can be deployed without serious performance penalties in many settings.

4.4 Jitter Request Latency

As we found in Section 2.2.2, the jitter factor should be as long as possible to disrupt all potential timing channels. To determine the effects of jitter on the perceived request latency to the client, we measured the total request latency while varying the max jitter factor. The jitter factor defines the maximum time that a response may be artificially delayed. Since the average forwarding latency was on the order of 1-2ms, we used 2ms as the lower bound. We tested jitter factors between 5ms and 1 second.

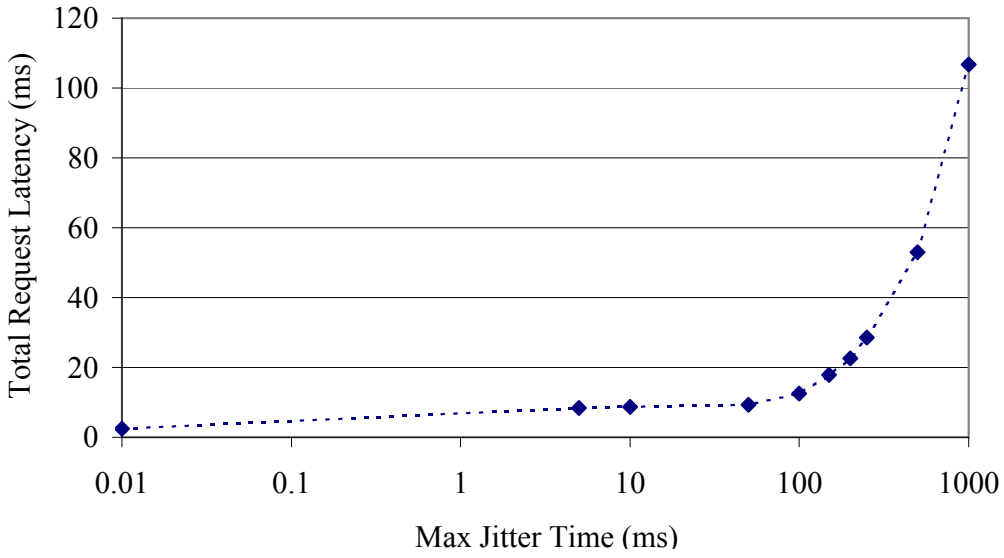

Figure 4.3: Guard Total Request Latency of 16KB file with Jitter

We show the effects of jitter on total request latency for a 16KB file in Figure 4.3. There is little different between 5, 10, and even 50ms jitter times because of the accuracy of the timing mechanisms we use. In practice, jitter factors of this length are not very effective, so this diminished resolution is not a concern. The average latency increases exponentially after 50 ms. However, we found that the average latency is still acceptable to an external client compared to larger file transfer even with a long jitter factor (recall that a 1 second jitter factor can theoretically disrupt a 100ms delay timing channel up to 90% and practically it can be even more disruptive because of synchronization). Though jitter factors longer than 1 second have the potential to be most disruptive to timing channels, they will adversely affect interactivity for external requesters and therefore we believe their usability is limited.

4.5 Acknowledgment

Chapter 4, in full, has been submitted for publication in the Proceedings of Ninth International Conference on Information Hiding, 2007, Springer-Verlag. Nabil Schear, Carmelo Kintana, Qing Zhang, and Amin Vahdat. The thesis author was the primary investigator and author of this work.

5

Conclusion

5.1 Future Work

Through our performance evaluation we found a number of bottle neck areas which affect the Guard's performance. We describe some methods for improving the performance of our system by addressing the overhead in these areas. We also describe improvements to the Glavlit design to support more protocols and scenarios.

Copying Overhead

The Guard spends considerable time copying packet data into and out of the in-order stream (as evidenced by the profiling results in Section 4.2.1). We believe that the asymptotic curve of the throughput of the Guard for large file sizes is also due to extra copying (Figure 4.2). We chose to arrange the packet data in-order in memory to simplify the discrepancies between packet and chunk boundaries. However, to improve performance we could reverse this decision.

With minor modifications, the verifiers could operate on noncontiguous memory thereby reducing the Guard's memory copying burden. This would allow the Guard to keep all packet data in a queue rather just the layer 4 and below packet headers. There are three times that the stream is accessed by the Guard:

when parsing protocol, hashing chunks, and handling retransmitted packets. The response parser already copies the data from the stream into a temporary buffer. This copy could be replaced by a loop which iterates over available packets copying their payload until the temporary buffer is full. Similarly, using the openssl *digest*_Init, *digest*_Update, and *digest*_Final functions (where *digest* is one of the supported hash functions), a wrapper routine can iteratively update the hash buffer until a full chunk has been hashed.

Handling retransmitted packets is less straight-forward. The Guard still needs to keep a packet cache, so old packets would need to be placed on a cache list. When it receives a retransmitted packet, the Guard would search through the cache list until it found the appropriate cached packet. It could then compare the two packet payloads. Since retransmissions are comparatively rare, searching the cache list would not add considerable overhead to the Guard. The Guard would also need to periodically flush unneeded packets from the cache list. The Killer thread could mark a flow to be flushed during its wake-up interval or the receiver could automatically flush the cache list after receiving n packets.

Work Queues and Flow Hash Tables

The second most time consuming operation the Guard performs is queue and hash table data structure manipulation. To allow a flow to be asynchronously deallocated at any time, we choose to pass keys to flows through the work queues rather than the flow pointers themselves. This requires every thread receiving a flow on its work queue to lookup the flow in one of the two flow hash tables (all_flows and all_incoming_flows). This results in considerable time spent querying the hash tables.

To cut back on the number of hash table lookups that must be performed, the Guard could pass pointers to flows directly through the work queues. To prevent a memory fault on asynchronous deallocation, each flow would contain a reference counter that indicates the number of work queues on which the flow

is residing. This allows the Killer to avoid deallocating flows that are still in use by some other thread in the Guard.

Increased Parallelism

We designed the Guard such that it would be straight-forward to add parallelism to support more concurrent connections. The work queue infrastructure allows us to replicate the verifier thread and reassembler so that more connections can be serviced at once. Since the receivers pass work to the verifiers by enqueuing a flow each time new data is available, there may be multiple instances of a single flow on the work queue. If a flow is currently being processed by one thread, any other thread that attempts to work on that flow will block trying to acquire the mutex. To avoid this problem, the verifiers can use the try_lock function to test a flow and move on to another flow if it fails. Another approach is to use per work queue reference counters to determine where a flow is being processed.

Dynamic Content

The Glavlit system as described here does not support the verification of dynamically generated Web content. Given the rise in use of such content, we feel a major focus of future work on Glavlit is to verify dynamic content. We do not believe that verification can be employed on any arbitrary dynamically generated page (i.e., one that is completely random). However, most dynamically generated pages are largely static with some generated content scattered throughout. Though Glavlit would be unable to provide as strong guarantees as it does currently, it would provide an extra line of defense from malicious or accidental information release from dynamic pages.

There are two possible techniques that we envision for vetting and verifying dynamic content: self-describing templates and chunk search algorithms. Using self-describing templates, the content developer would express the struc-

ture of the page as it is being developed. This allows the Warden to parse the source of the dynamic page to find the object and sub-object boundaries. The Guard uses a tree of possible object locations generated by the Warden to verify the dynamic page.

Using chunk search algorithms, the Warden would collect many examples of the dynamic page during vetting and split them into chunks and store secure hashes as it does now. It would then perform a fast checksum on the chunk to allow it to be quickly identified. The Guard in turn would use the fast checksum (with a rolling property) to search network data for valid chunks. Regions of the file that do not match any chunk would be passed off to a verification engine that could dynamically analyze the region for keywords or similarity to regions learned during vetting.

5.2 Discussion

This thesis explores the design and implementation of an efficient data confinement system called Glavlit. Glavlit considers a form of the data confinement problem, ensuring that only pre-authorized data crosses a protected network boundary. As the value of the data within a protected network increases, attackers wanting to ex-filtrate data may attempt to defeat confinement mechanisms by establishing covert channels embedded within existing communications or by embedding data within otherwise innocent binary objects. To address these challenges, we have designed and implemented Glavlit to perform data confinement for both explicit data transfer in payloads and implicit data transfers established through channels in network protocol headers.

Glavlit maintains transparency with existing protocols, specifically HTTP, and high performance using the key insight that the file vetting process should be separated from the file verification process. Before any object can cross a network boundary, it must be submitted for vetting to some trusted authority.

Once an object is approved for external release, the authority uploads a signature to the Guard software bridge running at the network boundary. We identify a restricted version of HTTP amenable to detection of covert channels; the Guard ensures that all protocol communication complies with this subset and that individual header fields take on semantically correct values. The Guard compares individual chunks of data in the protocol payload against the set of pre-loaded signatures to ensure that only previously vetted objects cross the protected network boundary. Lastly, the Guard implements a response latency timing channel jamming system by inserting uniformly random delays into the network.

Though mitigating covert channels is a difficult problem, we have proved effective methods for mitigating many of the risks they pose. Our system successfully defeats several classes of covert channels. Through our experiments, we have demonstrated that our system can operate at the edge of a network with acceptable performance compared to a Web proxy or a software bridge. We expect that further software enhancements (better memory management and pulling features directly into the kernel) could further improve performance. We can also envision a hardware based Glavlit system high speed cryptographic coprocessors with even better performance.

Most networks would reap great security benefits from the installation of a robust version of our Glavlit on their network. We have implemented the first system to prevent application layer covert channels transparently. We expect that Glavlit or its successors will be an essential part of standard network security in the future.

Bibliography

[1] Matthias Bauer. New Covert Channels in HTTP: Adding Unwitting Web Browsers to Anonymity Sets. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2003)*, Washington, DC, USA, October 2003.

[2] Kevin Beaver. Don't Spring a Leak. *Information Security Magazine*, January 2006.

[3] Vincent Berk, Annarita Giani, and George Cybenko. Detection of Covert Channel Encoding in Network Packet Delays. Technical Report TR2005-536, Dartmouth College, Computer Science, Hanover, NH, August 2005.

[4] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational), May 1996.

[5] Breach*View* SSL. Breach Security, Inc. www.breach.com/products_breachviewssl.asp.

[6] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP Covert Timing Channels: Design and Detection. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications security*, pages 178–187, New York, NY, USA, 2004. ACM Press.

[7] H. Carvey. The Dark Side of NTFS, 2002. www.madchat.org/vxdevl/vxmags/bcvge3.

[8] Simon Castro. Covert Channels Through The Looking Glass, 2003. www.gray-world.net/projects/papers/cc.txt.

[9] Censorship. http://en.wikipedia.org/wiki /Censorship.

[10] daemon9 AKA route. Project Loki. *Phrack Magazine*, 7(49), August 1996.

[11] Department of Defense. *DoD 5200.28-STD: Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)*, 1985.

[12] Alex Dyatlov and Simon Castro. Tunneling and Covert Channels over the HTTP Protocol, June 2003. www.gray-world.net/projects/papers/covert_paper.txt.

[13] Nick Estrada, Nick Feamster, and Michael Freedman. An Application Layer Covert Channel: Information Hiding with Chaffing, 1999. www.scs.cs.nyu.edu/~mfreed/docs/6.857/paper.pdf.

[14] Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium*, pages 247–262, Berkeley, CA, USA, 2002. USENIX Association.

[15] Fidelis XPS^{TM}Extrusion Prevention System. Fidelis Security Systems. www.fidelissecurity.com.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[17] Filescrub. Los Alamos National Laboratory, 2007. http://filescrub.lanl.gov.

[18] Gina Fisk, Mike Fisk, Christos Papadopoulos, and Joshua Neil. Eliminating Steganography in Internet Traffic with Active Wardens. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 18–35, London, UK, 2003. Springer-Verlag.

[19] Jessica Fridrich, Miroslav Goljan, and Rui Du. Detecting LSB Steganography in Color and Gray-Scale Images. *IEEE MultiMedia*, 8(4):22–28, 2001.

[20] Alexei Galatenko, Alexander Grusho, Alexander Kniazev, and Elena Timonina. Statistical Covert Channels Through PROXY Server. In *MMM-ACNS*, pages 424–429, 2005.

[21] John Giffin, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts. Covert Messaging through TCP Timestamps. In *Privacy Enhancing Technologies*, pages 194–208, 2002.

[22] James Giles and Bruce Hajek. An Information-Theoretic and Game-Theoretic Study of Timing Channels. *IEEE Transactions on Information Theory*, 48(9):2455–2477, 2002.

[23] MSU Graphics & Media Lab Video Group. MSU StegoVideo. http://www.compression.ru/video/stego_video /index_en.html.

[24] Xin guang Zou, Qiong Li, Sheng-He Sun, and Xiamu Niu. The Research on Information Hiding Based on Command Sequence of FTP Protocol. In *KES (3)*, pages 1079–1085, 2005.

[25] Theodore G. Handel and II Maxwell T. Sandford. Hiding Data in the OSI Network Model. In *Proceedings of the First International Workshop on Information Hiding*, pages 23–38, London, UK, 1996. Springer-Verlag.

[26] Florian Heinz. Nstx. http://nstx.dereference.de/nstx.

[27] Neil F. Johnson and Sushil Jajodia. Steganalysis: The Investigation of Hidden Information. In *Proc. of the 1998 IEEE Information Technology Conference*, pages 113–116, September 1998.

[28] Dan Kaminsky. Ozymandns. http://www.doxpara.com.

[29] Myong H. Kang, Ira S. Moskowitz, and Stanley Chincheck. The Pump: A Decade of Covert Fun. In *ACSAC*, pages 352–360, 2005.

[30] Peter Kieltyka. ICMP Shell. http://icmpshell.sourceforge.net.

[31] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, New York, NY, USA, 1994. ACM Press.

[32] Matthew Kwan. Gifshuffle, 2003. http://www.darkside.com.au/gifshuffle.

[33] Butler W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16(10):613–615, 1973.

[34] Andreas Lang and Jana Dittmann. Transparency and Complexity Benchmarking of Audio Watermarking Algorithms Issues. In *MM&Sec '06: Proceeding of the 8th workshop on Multimedia and security*, pages 190–201, New York, NY, USA, 2006. ACM Press.

[35] Allan Latham. JPHide and JPSeek, 1999. http://linux01.gwdg.de/ alatham/stego.html.

[36] Stefan Lebelt. Webknocking. German only, 2005. http://webknocking.de/.

[37] Vincent Liu. Metasploit Anti-Forensics Project. www.metasploit.com/projects/antiforensics.

[38] Norka B. Lucena, James Pease, Payman Yadollahpour, and Steve J. Chapin. Syntax and Semantics-Preserving Application-Layer Protocol Steganography. In *Information Hiding*, pages 164–179, 2004.

[39] J. McHugh. *Handbook for the Computer Security Certification of Trusted Systems*, chapter 8. Naval Research Laboratory, Washington D.C., 1995.

[40] Jonathan K. Millen. Covert Channel Capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.

[41] Ira S. Moskowitz, Steven J. Greenwald, and Myong H. Kang. An Analysis of the Timed Z-channel. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 2, Washington, DC, USA, 1996. IEEE Computer Society.

[42] Steven J. Murdoch and Stephen Lewis. Embedding Covert Channels into TCP/IP. In *Information Hiding*, pages 247–261, 2005.

[43] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *CSFW '04: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, page 172, Washington, DC, USA, 2004. IEEE Computer Society.

[44] Ryan Naraine. AOL 'Screw-up' Causes Search Data Spill. EWeek, August 2006. www.eweek.com/article2/0,1895,2000225,00.asp.

[45] Portauthority. PortAuthority Technologies. www.portauthoritytech.com.

[46] Neils Provos. Outguess: Universal Steganography, 2004. http://www.outguess.org.

[47] Niels Provos and Peter Honeyman. Detecting Steganographic Content on the Internet. In *NDSS*, 2002.

[48] Craig H. Rowland. Covert Channels in the TCP/IP Protocol Suite. *First Monday*, 2(5), 1997.

[49] Joanna Rutkowska. The Implementation of Passive Covert Channels in the Linux Kernel, 2004. www.invisiblethings.org/papers/passive-covert-channels-linux.pdf.

[50] Nabil Schear, Carmelo Kintana, Qing Zhang, and Amin Vahdat. Glavlit: Preventing Exfiltration at Wire Speed. In *Proceedings of HotNets-V*. ACM SIGCOMM, November 2006.

[51] Nabil Schear, Carmelo Kintana, Qing Zhang, and Amin Vahdat. Mitigating Covert Channels in HTTP. In *Information Hiding*, 2007. Submitted Pending Acceptance.

[52] Gustavus J. Simmons. The Prisoners' Problem and the Subliminal Channel. In *CRYPTO*, pages 51–67, 1983.

[53] Abhishek Singh. Eraser: An Exploit-Specific Monitor to Prevent Malicious Communication Channels. Technical Report 04-28, Georgia Tech CERCS, 2004.

[54] Spammimic. http://spammimic.com.

[55] Stego–Suite^{TM}. WetStone Technologies. www.wetstonetech.com.

[56] Content Alarm NW. Tablus Inc. www.tablus.com.

[57] TippingPoint Intrusion Prevention System. 3Com. www.tippingpoint.com/products_ips.html.

[58] Eugene Tumoian and Maxim Anikeev. Network Based Detection of Passive Covert Channels in TCP/IP. In *LCN*, pages 802–809, 2005.

[59] Vontu network prevent^{TM}. Vontu. http://www.vontu.com/products/prevent.asp.

[60] Peter Wayner. *Disappearing Cryptography: Information Hiding: Steganography and Watermarking (2nd Edition)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[61] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 253–267, New York, NY, USA, 2003. ACM Press.