Glavlit: Preventing Exfiltration at Wire Speed

Nabil Schear, Carmelo Kintana, Qing Zhang, Amin Vahdat
Department of Computer Science and Engineering, University of California at San Diego
{nschear, ckintana, qzhang, vahdat}@cs.ucsd.edu

ABSTRACT

Protecting sensitive data is no longer a problem restricted to governments whose national security is at stake. With ubiquitous Internet connectivity, it is challenging to secure a network – not only to prevent attack, but also to ensure that sensitive data are not released. In this paper, we consider the problem of ensuring that only pre-authorized data leave a network boundary using either overt or covert channels, i.e., preventing exfiltration. We identify the goals of *transparency*, *performance*, and *simplicity*. A system designed to prevent exfiltration should not adversely affect the transfer of authorized data and should work with existing protocols. Key to our approach is: i) separating the process of vetting authorized objects from line-speed data verification; and ii) employing a restricted, but compliant, HTTP subset to limit covert channels. In our evaluation, we show that Glavlit adds little overhead to the operation of a software network bridge.

1 INTRODUCTION

The protection of sensitive electronic data is an increasingly difficult problem. All businesses, governments, and individuals must process sensitive data, and improperly releasing such data can have significant consequences. Consider the inadvertent release of the search queries of hundreds of thousands of AOL users [11]. While organizations must be able to process information not fit for release like intellectual property, financial records, and medical information on their internal network, they must simultaneously process and distribute public data to the outside world.

The goal of our work is to ensure that only approved data exit an organization's protected internal network. We wish to prevent the transmission of data either overtly in the *payload channel* of layer 7 protocols such as HTTP or FTP or in hidden covert channels in the *protocol channel* of these protocols. To prevent information leaks through these channels, everything beyond the TCP header must be verified before allowing each 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. We, on the other hand, wish to enable verification at the granularity of entire files.

There are known techniques for detecting covert channels in layer 3 and 4 protocols [3, 8, 12]. There is relatively little work in detection at layer 7 despite the existence of many layer 7 channels [4, 6, 9]. We target HTTP for verified data transfer across a network boundary because it is the dominant layer 7 protocol used for interactive data transfer and it is general to a variety of deployment settings. We believe that our techniques extend to other layer 7 protocols, but we leave it to future work to test this hypothesis.

We have developed a system, *Glavlit*[1], which can prevent unauthorized release from a protected network while allowing authorized information to pass unhindered. Our goals for Glavlit are *transparency, performance,* and *simplicity.* Glavlit provides stringent security guarantees by enforcing complex *exit policies* while trusting only the systems responsible for authorizing individual files for release and for inspecting the packets leaving the network. At the same time, the common case performance of Glavlit is comparable to a software network bridge.

We first partition information into *objects*. An object is contiguous related information that can be analyzed independently, such as a file. We split the process of content control on 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 is the process of ensuring an object was previously vetted before releasing it across a designated network boundary. Glavlit verification assumes that *any* machine within the protected network is subject to compromise or negligence.

Many powerful vetting techniques cannot be implemented directly in the network because of performance overheads and the need to operate on entire files rather than individual packets. For example, digital review of complex files such as Microsoft Office formats, PDF, or multimedia can be compute-intensive and potentially require whole-file analysis. Additionally, some organizations are not willing to depend entirely on digital review and require human intervention. One could imagine a new protocol where external users queue requests for particular data. Glavlit allows objects to exit the network

[1]We have named Glavlit for the organization of the former Soviet Union that handled official state censorship matters.

upon verification that they are authorized for release. Unfortunately, this process imposes significant per-request overhead and makes object access highly asynchronous. Hence, we provide a means to *decouple* the vetting process for objects from their verification at a gateway.

To address information leaks in the protocol, we enforce HTTP protocol compliance to detect or limit most covert channels. We perform on-the-fly parsing of the protocol, verify the contents of structured fields, and restrict the HTTP RFC where necessary. We also prevent unstructured channels and timing attacks by correlating request-response pairs and normalizing server response time. In general, eliminating all covert channels for transferring unauthorized data is difficult to impossible. Therefore, we limit the bandwidth of such channels and essentially "raise the bar" for attackers.

The remainder of the paper is organized as follows: We introduce and motivate the Glavlit system architecture in Section 2. Our techniques for preventing leaks are given in Section 3. We briefly evaluate our system's performance in Section 4.

2 Glavlit Design

We have designed Glavlit to ensure that *all* information leaving a protected network has been approved for release. In this section we first describe the motivation for creating such a system by examining usage scenarios. We then describe our system architecture and threat model.

Consider the following scenarios: A company X outsources its customer service to another company Y, and they have established a shared network connecting their corporate LANs. X must provide proprietary documents and manuals to Y, but it does not wish to share its customers' personal information or financial records. X can use Glavlit to ensure that private information does not leak from its network to the network shared with Y.

Glavlit could also be used to enforce classified data handling policies among government agencies. Glavlit can dictate that all data that cross an organizational boundary are appropriate for release (e.g., to foreign nationals or to other networks with different classification levels). Glavlit's centralized vetting can also strictly enforce and track the set of granted *exit capabilities*. For example, Glavlit can require that only project leaders may vet files. It can further ensure that each vetting action is fully documented to provide an audit trail.

2.1 System Components

Figure 1 shows the system architecture of Glavlit. 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). Mechanisms for submitting objects for review and the actual approval process are orthogonal to this effort. The Warden can implement

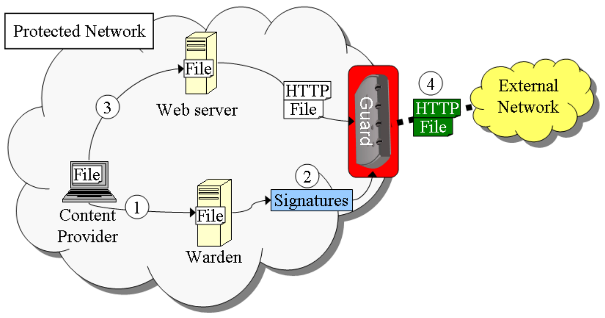

Figure 1: Glavlit System Architecture

any type of digital and/or human reviews to determine if the object is fit for release. The Warden shares a repository of white-listed content allowed to leave the network with the Guard. A *signature* stores hash values and metadata to later verify the content at the Guard (Step 2). After vetting, the user can place the file on a Web server accessible from outside the protected network (Step 3).

The Guard is a high-speed transparent network bridge at the perimeter 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. 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 (Step 4). If verification fails, the Guard can actively stop data from leaving the network by terminating the connection.

To support verification of HTTPS, the Guard can use techniques from intrusion detection systems [2]. Each Web server's private keys can be shared with the Guard, allowing it to perform in-line decryption of all traffic. This sharing is possible because the owner of the Glavlit protected network also controls the internal Web servers.

2.2 Threat Model

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 the it on a public-facing Web server. Since this threat is most common, Glavlit counters it through high-speed content control.

Malicious Use of Standard HTTP – An attacker (e.g., disgruntled employee, external hacker, user inadvertently compromised by 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., for 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 (by intention implying that only the protocols associated with those ports are allowed). Glavlit provides the additional assurance that another protocol is not being used on an HTTP port.

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.

3 Preventing Information Leaks

In this section we describe the techniques Glavlit uses to prevent unauthorized content release. Since we must control data release through authorized channels with the same vigor as covert ones, we use two complementary techniques: content control and protocol mitigation.

3.1 Content Control

Since powerful vetting can be time-intensive, we perform content control by splitting vetting from verification. The Guard can then verify at high speed that all content crossing the network boundary is pre-vetted.

There are commercial content control solutions that perform vetting and verification simultaneously on the gateway [7]. The primary drawback to this approach is that it is unable to perform whole-object analysis, critical for supporting modern transfer protocols. Thus, these approaches restrict vetting and verification to pattern matching, e.g., ensuring that individual packets do not contain substrings that resemble social security or credit card numbers. The rate at which this approach can send packets is limited by the speed of its vetting process. Another previous approach is to proxy access to protected content and individually analyze the authorization of each request [5]. Since this requires special client configuration which is not feasible for external clients, it has been restricted to intra-organizational protected networks or for *outgoing* client requests.

3.1.1 Vetting and Static Verification

Users send objects they wish to vet to the Warden, which grants or denies each object the ability to leave the network as specified by policy. This process can be as simple as a keyword search, or as rigorous as requiring approval from a committee of human analysts. Once approved, the Warden partitions the object into *chunks* of 1024 bytes, each hashed with SHA-1. The resulting collection of hashes, named a *signature*, supports high-speed verification at the Guard.

Verification ensures that all information crossing the network boundary 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. To identify an object, the Guard hashes the first 256 bytes of the file content (lookup size). This hash keys the signature table shared with the Warden. We use the Content-Length header to identify files smaller than 256 bytes. Hash collisions are not yet implemented; however, a tree structure within the signature table could implement support for collisions.

Once the Guard identifies the object's signatures, it hashes each chunk of object data and compares the result with the known hash. If any hash value does not match, the connection is bidirectionally terminated by injecting a TCP RESET packet.

To perform verification transparently in the network, the Guard must be able to reconstruct the network communications on-the-fly. Since the Guard performs hashes at a chunk granularity, the packets associated with a chunk can egress as soon as all chunks within a packet are fully verified. Since chunks may cross packet boundaries and since packets may arrive out of order at the gateway, we may have to perform some packet buffering. Content verification is complicated by the presence of the layer 7 protocol in the network byte-stream. We discuss protocol verification in Section 3.2.

The Guard creates a data structure called a *flow* for each TCP connection. As packets arrive, it classifies them by protocol, type, and TCP connection. To sim-

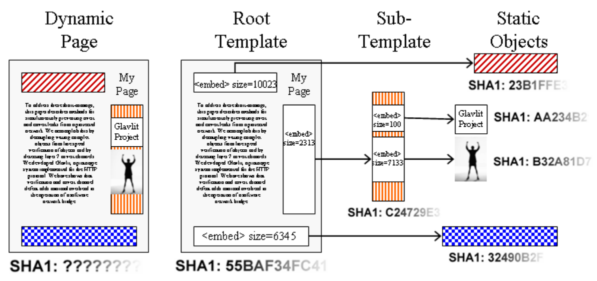

Figure 2: Dynamic Template Model

plify the packet vs. chunk boundary problem, the Guard reconstructs the TCP in-order byte stream in a structure called a *stream*. The Guard maintains an offset to the last verified byte in the stream. As the verifiers operate on the stream, this offset advances until it crosses a packet boundary. The corresponding fully verified packet can be sent out of the protected network at this point.

Since verification may cross packet boundaries, retransmitted packets cannot be re-verified. We handle retransmissions by keeping a window called a *packet cache* of already verified data within the stream. The Guard compares retransmitted packet data with the packet cache and sends it immediately without having to repeat verification.

3.1.2 Dynamic Content Verification

Web traffic is increasingly generated dynamically. The same mechanism used for static content cannot be directly deployed. To solve this problem, we must enhance the vetting and verification process to be aware of the structure of dynamic content. While we believe it is intractable to vet/verify arbitrarily generated dynamic content, we present an approach to handle a subset of dynamically-generated content.

We assume that the dynamic object fits a template model. The template defines static content as well as gaps that contain other templates and static content (Figure 2). The dynamic object can be modeled as a tree where the base template is the root and the leaves are static data. Each layer of the tree defines sub-templates and static content allowed in each gap. Gaps may be filled by another template or a list of valid static objects. For example, a dynamic Web page might randomly choose from a set of images for the header of the page. The web developer must express the template structure and valid objects for each gap as part of the content development [1]. Glavlit vetting and verification processes can now understand this structure because it is self-describing.

We cannot use the Content-Length header to determine the size of embedded objects *smaller* than the lookup size. To solve this problem without an exhaustive search, we propose a solution that requires the Web server to cooperate but does not assume trust using hints. Since the server constructs the content, we require the server to include the size of small dynamic objects within each gap. This does not place trust on the Web server since any false description will misalign the hash.

3.2 Protocol Channel Mitigation

The previous discussion shows how to prevent the transfer of unauthorized data in the payload of HTTP. However, one can transfer unauthorized data within the protocol using covert channels. In this section, we examine the channels for information release using HTTP and what Glavlit does to mitigate them. We discuss the carrier data where information could be encoded covertly. We classify carriers as structured or unstructured [8] and separate them by the degree to which the organization of the data is apparent.

3.2.1 Structured Carriers

Structured data have explicit organization and semantics. We further partition structured carrier channels by their syntactic compliance to the relevant RFC and semantic validity.

We first examine non-compliant channels that deviate from the protocol syntax. For example, a server response could include arbitrary information. Conventional parsing can easily detect this technique. A common case of a non-compliant protocol channel is one that uses the HTTP port to tunnel another protocol.

Since the RFC loosely defines header syntax, adding *Credit-Card: 1234-5678-9012-3456* as a header is actually compliant. The RFC does not explicitly require response headers be in a certain order or to be used at all. An attacker can use list order steganography to encode data by reordering the response headers [13]. In addition to custom headers, the presence or lack of headers could also encode data. Glavlit defines a fixed set of acceptable response headers per request type. The Guard parses outgoing responses using a more restrictive grammar than the one given in the RFC. It also parses incoming requests, but uses the standard RFC grammar to allow any client to connect to a protected Web server.

Responses that do not deviate syntactically are compliant; however, compliance does not imply that the server's response is *correct*. For example, a server may return Content-Length 1024 and 1025 on consecutive requests for the same file of size 1024. Such responses could encode a series of 1's and 0's for covert data delivery. The peak bandwidth in bits per second B of a compliant channel is:

$$B = c * log_2(n) * (t + \frac{RTT}{2})$$

where c = # *connections*, n = # distinct server responses, t = server CPU time, RTT = round trip time

For example, with a 10ms RTT, server processing time of 3ms with 2 connections and 8 distinct server responses, the bandwidth of the covert channel would be 48 bits per second.

In addition to Content-Length, a covert channel can encode data in fields like Last-Modified or Content-Type [4]. Most HTTP header fields are *verifiable* given sufficient information about the nature of the request and the content served. We have examined the HTTP RFC to determine the verifiability of each header. Once we verify a field, only a single response may leave the network, significantly reducing channel capacity. Some header fields are not verifiable because they are based upon server state and load. For example, the Content-Range header specifies the particular chunk of data being transmitted that could vary with respect to the OS buffer cache.

Headers often must be restricted in the server configuration to enable verification. This leads to less flexibility because a header may only take on a few values. In practice, operating a Web server only requires a small number of headers; therefore, we feel this limitation is acceptable. For example, the Allow header specifies the HTTP commands that the server can understand and could be set statically (e.g., only GET/HEAD/PUT methods). Restricted configuration can also be based upon the corresponding request. The Connection header defines the TCP connection state the server will use for the connection. To aid verification, this header can be standardized to always *keep-alive* when an HTTP/1.1 connection is requested and always *close* when HTTP/1.0 or close state is requested.

Our implementation limits the number of distinct responses by verifying protocol fields by combining object meta-data and the corresponding client request to completely verify the server response. The incoming flow data structure stores a queue of parsed requests. When an HTTP response returns, the Guard parses it and compares it with the incoming request and the signature meta-data to ensure that all fields are verified.

3.2.2 *Unstructured Carriers*

Unstructured carriers store data subjectively or randomly. For example, the order and timing of network events can encode data in an unstructured manner. We examine the *structured artifacts* that these channels produce and ways Glavlit can defeat them.

Request order can encode information unidirectionally *into* a server [6]. Similarly, reordering pipelined responses enables bidirectional communications. The Guard maintains the RFC specification that responses should be returned in the order they were requested by a single client by parsing and correlating incoming requests.

An attacker can encode information based on the timing of network activity [3]. Because of the inherent randomness in network timing, *increases* in the timing carrier (e.g., packet inter-arrival time or HTTP response time) frame covert data. These increases leave a structured artifact behind; therefore, a good channel implementation should affect the timing carrier as little as possible. One can model the Web server performance (e.g., as a function of offered load) and block deviations from the model. This technique has two drawbacks: the model may not be sufficient to cover the server's behavior and the distance metric may not detect carefully crafted deviations.

We can also disrupt rather than detect a covert timing channel by introducing *jitter* into the data stream. This *jitter* is impossible to defeat without increasing the amount of data sent across the network, e.g., a reliable channel, that further reduces usable bandwidth. Kang et al. employed a similar concept to prevent timing channels in the Pump with fully stochastic acknowledgements [10]. We can normalize Web server performance to defeat potential covert timing channels. Glavlit can delay responses by a uniformly random variable or follow a fixed probability distribution. Normalized responses are often not apparent to a user but devastating to a covert timing channel.

The above covert channel mitigation and content control techniques can *limit* the channel capacity. However, there still exist ways to encode information. It is computationally infeasible to completely defeat all covert channels. The only alternative is to limit the number of requests the server is allowed to process per unit time.

4 PERFORMANCE EVALUATION

We evaluated the performance of our prototype Glavlit system compared to a direct wire connection and a standard Linux kernel software bridge (Figure 3). We also tested the Guard with verification *off* to evaluate our network implementation. Our test setup consists of three machines running Linux 2.6.9-34 with dual 2.8 GHz Pentium 4 Xeon processors, 2 GB of memory, and gigabit Ethernet interfaces. The Guard is connected to a host running a custom HTTP client on one interface and a host running Apache version 2.2.2 on the other. The custom client spawns 20 threads, and each thread requests files using HTTP 1.1 for a specified time.

The Guard with verification does not impose significant overhead for most file sizes compared with the kernel bridge or direct connection. The Guard *without* verification performs slightly better than with verification showing that the primary overhead is packet reconstruction and forwarding. The overhead of setting up new connections (protocol parsing and TCP stream allocation) reduces the performance of the Guard with verifi-

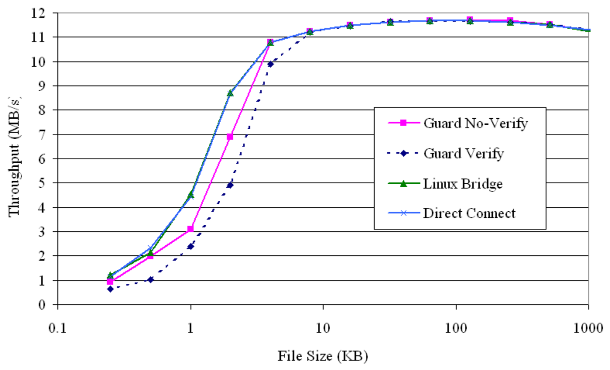

Figure 3: Throughput vs. File Size

cation up to 50% for small files (<8KB). Despite this, the Guard sustained processing approximately 3000 requests per second. For the cases where file size is greater than 10KB, the Guard can hash and forward packets as fast as the server can produce them. In these cases, the per-connection overhead is amortized over larger data transfers. Given this performance with a largely un-tuned implementation, we believe it possible to perform verification at higher speeds with a highly tuned kernel-level software implementation or with hardware acceleration.

5 Conclusion

This paper presents a network content protection system that avoids many of the drawbacks with previous attempts to ensure that only authorized data cross a network boundary. Specifically, the system maintains client-server transparency, while only marginally decreasing throughput. Additionally, our system can secure all files on a network regardless of the security at a particular host.

The key insight behind our approach is the decoupling of the object-vetting process (which can be variably slow) and the object-verification process (which can be performed at high speed and on a per-packet basis). Our prototype implementation of Glavlit performs nearly as well as a standard software bridge.

Malicious users often employ covert channels to transport data outside the network boundary. As an overture to this remaining exit channel, we have implemented the first system to transparently mitigate application-layer covert channels in HTTP.

References

[1] Brabrand, C., Møller, A., and Schwartzbach, M. I. Static Validation of Dynamically Generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01* (June 2001), pp. 221–231.

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

[3] Cabuk, S., Brodley, C. E., and Shields, C. IP Covert Timing Channels: Design and Detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), ACM Press, pp. 178–187.

[4] Dyatlov, A., and Castro, S. Tunneling and Covert Channels over the HTTP Protocol, June 2003. http://www.gray-world.net/projects/papers/covert_paper.txt.

[5] Entelligence^{TM}Content Control Server. Entrust. www.entrust.com/content-control.

[6] Feamster, N., Balazinska, M., Harfst, G., Balakrishnan, H., and Karger, D. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 247–262.

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

[8] Fisk, G., Fisk, M., Papadopoulos, C., and Neil, J. Eliminating Steganography in Internet Traffic with Active Wardens. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding* (London, UK, 2003), Springer-Verlag, pp. 18–35.

[9] Handel, T. G., and Maxwell T. Sandford, I. Hiding Data in the OSI Network Model. In *Proceedings of the First International Workshop on Information Hiding* (London, UK, 1996), Springer-Verlag, pp. 23–38.

[10] Kang, M. H., Moskowitz, I. S., and Chincheck, S. The Pump: A Decade of Covert Fun. In *ACSAC* (2005), pp. 352–360.

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

[12] Tumoian, E., and Anikeev, M. Network Based Detection of Passive Covert Channels in TCP/IP. In *LCN* (2005), pp. 802–809.

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