Transport and Application Protocol Scrubbing

G. Robert Malan, David Watson and Farnam Jahanian
University of Michigan
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
{*rmalan,dwatson,farnam*}*@umich.edu*

Paul Howell
Merit Network
4251 Plymouth Road
Ann Arbor, Michigan 48105-2785
grue@merit.edu

Abstract—

This paper describes the design and implementation of a *protocol scrubber*, a transparent interposition mechanism for explicitly removing network attacks at both the transport and application protocol layers. The *transport scrubber* supports downstream passive network-based intrusion detection systems; whereas the *application scrubbing* mechanism supports transparent fail-closed active network-based intrusion detection systems. The transport scrubber's role is to convert *ambiguous* network flows into *well-behaved* flows that are unequivocally interpreted by all downstream endpoints. As an example, this paper presents the implementation of a TCP/IP scrubber that eliminates *insertion* and *evasion* attacks – attacks that use ambiguities to subvert detection – on passive network-based intrusion detection systems, while preserving high performance. The application protocol scrubbing mechanism is used as a substrate for building fail-closed active network-based intrusion detections systems that can respond to attacks by eliding or modifying application data flows in real-time. This paper presents the high-performance implementation of a general purpose transparent application-level scrubbing toolkit in the FreeBSD kernel.

I. Introduction

As society grows increasingly dependent on the Internet for commerce, banking, and mission critical applications, the ability to detect and neutralize network attacks is becoming vitally important. Security administrators use *network intrusion detection systems* (NID systems) as a tool for detecting attacks and misuse in real-time [7]. Network-based intrusion detection systems identify these attacks using passive monitoring techniques to recognize patterns of misuse as they occur. Organizations rely on NID systems to identify misuse within the protocol streams that pass through the firewall as well as those that originate within the network's perimeter. As such, they have become the second line of defense within an organization after firewalls. However, as attacks are increasing in sophistication it is becoming difficult to determine when an internal network has been compromised [11]. There are two serious problems with network-based intrusion detection systems. The first problem is that attackers can use ambiguities in network protocol implementations to deceive NID systems, bypassing their watchful eyes. The second problem is that NID systems are passive mechanisms by design. As passive entities they can only notify administrators or active mechanisms whenever intrusions are detected. However, the response to this notification may not be timely enough to withstand some types of attacks – such as attacks on infrastructure control protocols – where only immediate intervention can sustain the network's operation. This paper presents the design and implementation of a *protocol scrubber* that specifically addresses these two problems. The protocol scrubber is a transparent interposition mechanism for explicitly removing network attacks at both the transport and application protocol layers. The *transport scrubber* addresses the problem of transport attacks by removing protocol ambiguities, enabling downstream passive network-based intrusion detection systems to operate with high assurance. The *application scrubbing* mechanism allows the creation of active, interposed intrusion detection systems that can be used to elide or modify important network protocols in real-time; effectively enabling an immediate response upon detection of severe misuse.

The transport scrubber's role is to convert *ambiguous* network flows – flows that may not be interpreted in the same manner at different endpoints – into *well-behaved* flows that are interpreted identically by all downstream endpoints. As an example, this paper presents the implementation of a TCP/IP scrubber that eliminates *insertion* and *evasion* attacks – attacks that use ambiguities to subvert detection – against passive network-based intrusion detection systems. This paper argues that passive NID systems can only effectively identify malicious flows when used in conjunction with an active interposition mechanism. Through interposition, the transport scrubber can guarantee protocol invariants that enable downstream intrusion detection systems to work with confidence. Because the Internet protocols are well described, *correct* implementations exchange packets with deterministic results. However, sophisticated attackers can leverage subtle differences in protocol implementations to wedge attacks past the NID system's detection mechanism by purposefully creating ambiguous flows. In these attacks, the destination endpoint reconstructs a malicious interpretation; whereas the passive NID system's protocol stack interprets the protocol as a benign exchange. Examples of these ambiguities are IP fragment reconstruction and the reassembly of overlapping out-of-order TCP byte sequences. The role of the transport scrubber is to pick one interpretation of the protocols and to convert incoming flows into a single representation that all endpoints will universally interpret. The transport scrubber's conversion of ambiguous network flows into unequivocal interpretations is analogous to that of network traffic shaping. Shapers modify traffic around the edges of a network to generate predictable utilization patterns within the interior. Similarly, the transport scrubber intercepts protocols at the edges of an interior

network, and modifies them in such a way that their security attributes are predictable.

In addition to transport scrubbing, we introduce an application scrubbing mechanism that is used as a substrate for building fail-closed active network-based intrusion detections systems that can respond to attacks by eliding or modifying application data flows in real-time. An application scrubber is used to protect highly sensitive flows – such as infrastructure control protocols – that cannot wait for the attention of an administrator or the intervention of a remote countermeasure. In contrast, when an NID system identifies an attack, it notifies an administrator or some other agent so that appropriate action can be taken to neutralize the threat. This process introduces latency between the detection of an attack and its response. While this is acceptable for some types of attacks, there are other network services that cannot be compromised while maintaining the integrity of the network – examples include attacks on the network's routing infrastructure. As an interposed active mechanism, the application scrubber is *fail-closed*. Specifically, if the scrubber is incapacitated it will not let the attack through to its destination. This contrasts with the *fail-open* behavior of passive NID systems. Once an attacker has neutralized a NID system, the network remains open to unobserved attack. This paper presents the high-performance implementation of a general purpose transparent application-level scrubbing toolkit based on the FreeBSD kernel. The modifications to the kernel include additions to the socket API that allow a user-level application scrubber to bind a local socket to a set of remote network addresses. This simple primitive allows the easy creation of transparently interposed application scrubbers.

The main contributions of this work are:

- *Identification of transport scrubbing:* The paper introduces the use of an active, interposed transport scrubber for the conversion of ambiguous network flows into *well-behaved*, unequivocally interpreted flows. We argue that the use of a transport scrubber is essential for correct operation of passive network-based intrusion detection systems. The paper describes the use of transport scrubbers to eliminate insertion and evasion attacks on NID systems [11]. The concept of transport scrubbing can easily be merged with existing firewall technologies to provide the significant security benefits outlined in this paper.
- *Design and implementation of TCP/IP scrubber:* The novel design and efficient implementation of the *half-duplex* TCP/IP scrubber is presented. The current implementation of the TCP/IP scrubber exists as a modified FreeBSD kernel [4]. This implementation is shown to scale with commercial stateful inspection firewalls and raw Unix-based IP forwarding routers.
- *Creation of a transparent application-level protocol scrubbing mechanism:* The protocol scrubber supports flexible transparent application protocol scrubbers that can elide or modify application level flows in real-time in response to attacks. By creating a lightweight transparent application scrubbing mechanism, we allow for the active scrubbing of critical infrastructure level protocols. The support comes through the custom socket-based API extensions to the FreeBSD kernel.

The remainder of this paper is organized as follows. Section II places our work within the broader context of related work. Section III describes the design, implementation and performance characteristics of our TCP/IP transport scrubber. Section IV presents our mechanism for providing transparent application-specific protocol scrubbing. Finally, Section V presents our conclusions and plans for future work.

II. Related Work

Firewall technologies [2] are closely related to protocol scrubbers. They are both active interposition mechanisms – packets must physically travel through them in order to continue towards their destinations – and both operate at the ingress points of a network. Modern firewalls primarily act as gate-keepers to a protected network, utilizing filtering techniques that range from simple header-based examination to sophisticated authentication schemes. However, due to performance reasons, once a firewall has identified an authorized flow, packets are routed through a fast-path and are not scrutinized further for attacks. In contrast to firewalls, the protocol scrubber's primary function is to homogenize network flows, identifying and removing their attacks in real-time. The scrubber is utilized to remove attacks present within the protocols once a firewall has authorized a flow's access. As such, scrubbing technology can easily be added to existing firewall technologies to to significantly enhance network security.

Older firewalls, such as the TIS Firewall Toolkit [18], that utilize application-level proxies are similar to protocol scrubbers. These types of firewalls provide the most security; however their performance characteristics are not acceptable for deployment in high-speed environments. Their utility has decreased as the Internet has evolved. In contrast, the protocol scrubber has been designed to achieve maximum throughput as well as a high level of security.

Firewall technologies changed with the advent of so-called *stateful inspection* of networking flows, exemplified by Checkpoint's Firewall-1 [19]. These types of firewalls examine portions of the packet header and data payloads to determine whether or not entry should be granted. After the initial check, a flow is stored in a table so that fast routing of the subsequent network packets can occur. These later packets are not checked for malicious content. The protocol scrubber differs in that it continues to remove malicious content for the lifetime of the flow.

Network Associates has recently introduced a new version of their Gauntlet firewall [5]. The approach taken in this firewall is a combination of application-level proxy and fast-path flow caching. At the beginning of a flow's lifetime, the flow is intercepted by an application-level proxy. Once this proxy authenticates the flow, it is cached in a lookup table for fast-

path routing. Again, the protocol scrubber differs by allowing detection of malicious content, not only at the beginning, but throughout the flow's lifetime.

Intrusion Detection Systems (ID systems) [7], [12] are also closely related to protocol scrubbers. There are two broad categories of intrusion detection systems: network-based and host-based. Network-based Intrusion Detection Systems (NID systems) are implemented as passive network monitors that reconstruct networking flows and monitor protocol events through eavesdropping techniques [20], [13], [10], [15]. As passive observers, NID systems have a vantage point problem [9] when reconstructing the semantics of passing network flows. This is the vulnerability that can be exploited by sophisticated network attacks that understand this inherent schism between the protocol's destination and an intermediary [11]. As active participants in a flow's behavior, the protocol scrubber removes these attacks, and can function as a fail-closed real-time NID system that can sever or modify malicious flows.

Host-based ID systems are located on the end-hosts in a network and monitor the resources and security procedures followed by co-resident users and applications. While host-based ID system techniques are very useful, they suffer from two limitations: they only reside on the systems that the administrator knows about; and they cannot observe events that do not manifest themselves high enough in the system. Protocol scrubbers and network-based intrusion detection are not mutually exclusive with host-based ID systems, but rather act as their complement.

Protocol scrubbers deal with virtual private networks (VPN) and header and payload encryption [1] in the same manner as network intrusion detection systems. There are two approaches to filtering encrypted flows: the first assumes that if the flow is end to end encrypted it is sanctioned; an alternative approach is to filter out any flows with unsanctioned security associations. As an active mechanism, the protocol scrubber could remove unsanctioned flows in real-time. When placed on the inside of a VPN, the protocol scrubber could be used to further clean protocols. This would apply to scrubbing e-commerce transactions and sensitive database accesses.

The TCP splicing work of Maltz [6] and Spatscheck [16] allows application-level proxies to push the association of the trusted and untrusted sockets down into the kernel for higher network performance. This association within the kernel allows the pair of sockets to route packets between each other without user-level intervention. In practice, this splicing of sockets takes place after the connection has been authenticated. After the splicing, the flow is not checked for malicious content.

Network address translation (NAT) is a general-purpose mechanism that can be used to support transparent proxying [3]. This is accomplished using a static translation rule that maps connections bound for external hosts on well-known ports to local ports on the loopback interface. The application scrubber's approach to transparency addresses two problems with NAT-based transparent proxies. First, application scrubbers are provided total bidirectional transparency; whereas a typical NAT-based proxy can only provide transparency to a single side of a connection. Moreover, the application scrubbing mechanism handles transparent UDP proxying by explicitly tagging incoming packets with the appropriate destination mappings. NAT-based UDP proxies have trouble looking up the reverse mappings, where multiple destinations can be bound to a single mapping, resulting in ambiguity.

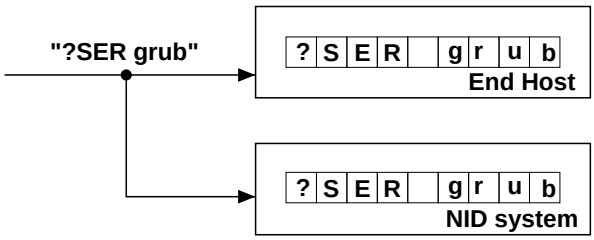
III. TCP/IP Scrubber

Network-based intrusion detection systems are based on the idea that packets observed on a network can be used to predict the behavior of the intended end host. While this idea holds for well-behaved network flows, it fails to account for easily created ambiguities that can render the NID system useless. Attackers can use the disparity between the reconstruction at the end-host with that of the passive NID system to attack the end host without detection. The TCP/IP scrubber is an active mechanism that explicitly removes ambiguities from external network flows, enabling downstream NID systems to correctly predict the end-host response to these flows. By enforcing protocol invariants on the downstream flows, the TCP/IP scrubber eliminates TCP/IP insertion and evasion attacks against NID systems that can render them useless. By utilizing a novel protocol-based approach in conjunction with an in-kernel implementation, the TCP/IP scrubber provides high performance as well as enforcement of flow invariants. The TCP scrubber only reconstructs the incoming half of the connection. By keeping a significantly smaller amount of state, the scrubber is able to scale to tens of thousands of concurrent connections with throughput performance that is comparable to commercial stateful inspection firewalls and raw Unix-based IP forwarding routers. This section describes the overall design and implementation of the TCP/IP scrubber and provides a comprehensive performance profile using both macro and microbenchmarks.

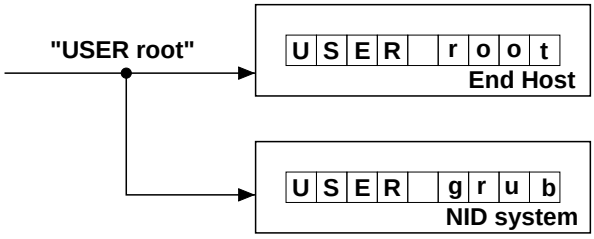
A. TCP/IP Ambiguities and ID Evasion

Sophisticated attacks can utilize protocol ambiguities between a network intrusion detection system and an end-host to slip past the watching NID system completely undetected. Network ID systems rely on their ability to correctly predict the effect of observed packets on an end-host system in order to be useful. In [11], Ptacek and Newsham describe a class of attacks that leave NID systems wide open to subversion. We borrow their description of the two main categories of these attacks: *insertion attacks*, where the NID system accepts a packet that the end host rejects; and *evasion attacks*, where the NID system rejects a packet that the end host accepts.

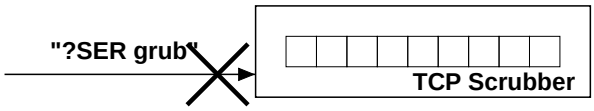
Figure 1 provides a simple example of how differences in the reconstruction of a TCP stream can result in two different interpretations, one benign and the other malicious. In this



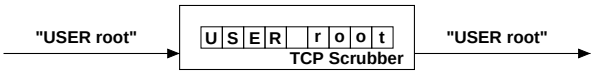
(a) Host and NID system after attacker sends a hole.



(b) Attacker filling in the hole and confusing the NID system.



(c) Scrubber enforces single interpretation.



(d) Attacker filling in the hole and sending new data.

Fig. 1. Example of ambiguity of transport layer protocol implementation differences between an interposed agent (NID system) and an end host.

simple example an attacker is trying to log into an end host as `root`, while fooling the NID system into thinking that it is connecting as a regular user. The attacker takes advantage of the fact that the end host and the NID system reconstruct overlapping TCP sequences differently. In Figure 1a the attacker sends a data sequence to the end host with a hole at the beginning (represented by the question mark). Since TCP is a reliable byte-stream service that delivers its data to the application layer in order, both the end-host and NID system must wait until that hole is filled before proceeding [17]. However, unbeknownst to the NID system – but not the wily attacker – the end host deals with overlapping sequences of bytes differently than the NID system. In Figure 1b when the attacker resends the data with the hole filled, but with a different username of the same length, the difference in implementation choice between the two systems allows the attack to dupe the NID system. Since a correct TCP implementation would always send the same data upon retransmission, it is not mandated in the specification as to which set of bytes the endpoint should keep. In this example, the end host chose to keep the new sequence of bytes that came in the second packet; whereas the NID system kept the first sequence of bytes. Neither is more correct than the other; just the fact that there is ambiguity in the implementation of the networking stacks allows sophisticated attacks to succeed.

In addition to the handling of overlapping TCP segments, there are many other ambiguities in the actual implementation of the TCP/IP stack [11]. To begin with, the handling of IP fragments and their reconstruction varies by implementation. Similar variations are seen with the reconstruction of TCP streams. End hosts deal differently with respect to IP options and malformed headers. They vary in their response to relatively new TCP header options such as PAWS [17]. Moreover, there are vantage point problems that passive NID systems encounter such as TTL-based routing attacks and TCP creation and tear-down issues. The large number of ambiguities with their exponential permutations of possible end-host reconstructions make it impractical for NID systems to model all possible interpretations at the end-host. They must pick some subset, generally a single interpretation, to evaluate in real-time. For this reason it is impractical to adequately address the problem within the context of a passive NID system.

To address this problem, we have created the TCP/IP scrubber. Specifically, the scrubber provides the invariants that NID systems need for confident flow reconstruction and end-host behavior prediction. Figures 1c and 1d demonstrate how an active protocol scrubber interposed between the attacker and the downstream systems eliminates the ambiguity. The scrubber enforces a single interpretation of the attacker's TCP/IP stream to eliminate downstream ambiguity. By picking a single way to resolve the TCP reconstruction – in this case the scrubber simply throws away the data after a hole – both the downstream NID system and end host both see the attacker logging in as `root`.

B. TCP/IP Scrubber Design and Implementation

The TCP/IP scrubber converts external network flows – sequences of network packets that may be ambiguously interpreted by different end-host networking stacks – into homogenized flows that have unequivocal interpretations, thereby removing TCP/IP insertion and evasion attacks. While TCP/IP implementations vary significantly in many respects, correct implementations interpret *well-behaved* flows in the same manner. The protocol scrubber's job is to codify what consists of well-behaved protocol behavior and to convert external network flows to this standard. To describe all aspects of a well-behaved TCP/IP protocol stack is impractical in a paper of this length; however we will illustrate this approach by detailing its application to the TCP byte stream reassembly process. TCP reassembly is the most difficult aspect of the TCP/IP stack and is crucial to the correct operation of NID systems.

B.1 TCP Scrubber Design

The TCP scrubber's approach to converting ambiguous TCP streams into unequivocal, well-behaved flows lies in the middle of a wide spectrum of solutions. This spectrum contains stateless filters at one end and full transport-level proxies – with a considerable amount of state – at the other. Stateless filters can handle simple ambiguities such as non-standard usage of TCP/IP header fields with little overhead; however, they are incapable of converting a stateful protocol, such as TCP, into a non-ambiguous stream. Full transport-layer proxies lie at the other end of the spectrum, and can convert all ambiguities into a single well-behaved flow. However, the cost of constructing and maintaining two full TCP state machines – scheduling timer events, round-trip time estimation, window size calculations, etc. – for each network flow is prohibitive from a performance and scalability standpoint. The TCP scrubber's approach to converting ambiguous TCP streams into well-behaved flows attempts to balance the performance of stateless solutions with the security of a full transport-layer proxy. Specifically, the TCP scrubber maintains a small amount of state for each connection; but leaves the bulk of the TCP processing and state maintenance to the end hosts. Moreover, the TCP scrubber only maintains data state for the half of the TCP connection originating at the external source. Even for flows originating within a protected network there is generally a clear notion of which endpoints are more sensitive and need protection; if a situation arises that needs bidirectional scrubbing, it can be configured in the scrubber. With this compromise between a stateless and stateful design, the TCP scrubber removes ambiguities in TCP stream reassembly with performance comparable to stateless approaches.

To illustrate the design of the TCP scrubber we compare it to a full transport layer proxy. TIS Firewall Toolkit's `plug-gw` proxy is one example of a transport proxy [18]. It is a user-level application that listens to a service port waiting for connections. When a new connection from a client is established, a second connection is created from the proxy to the server. The transport proxy's only role is to blindly read and copy data from one connection to the other. In this manner, the transport proxy has fully obscured any ambiguities an attacker may have inserted into their data stream by forcing a single interpretation of the byte stream. This unequivocal interpretation of the byte stream is sent downstream to the server and accompanying network ID systems for reconstruction. However, this approach has serious costs associated with providing TCP processing for both sets of connections in terms of throughput and scalability.

Unlike a transport layer proxy, the TCP scrubber leaves the bulk of the TCP processing to the end points. For example, it does not generate retransmissions, perform round trip time estimation, or any timer-based processing; everything is driven by events generated by the end hosts. The TCP scrubber performs two main tasks: it maintains the current state of the connection; and keeps a copy of the byte stream that has been sent by the external host, but not acknowledged by the internal receiver. In this way it can make sure that the byte stream seen downstream is always consistent; it throws away any packets that could lead to inconsistencies. While this strategy may seem somewhat Draconian in terms of performance, data flows from well-behaved clients will never have their flows tampered with. The perturbation to flows that are scrubbed is easily absorbed by the end host as packet loss.

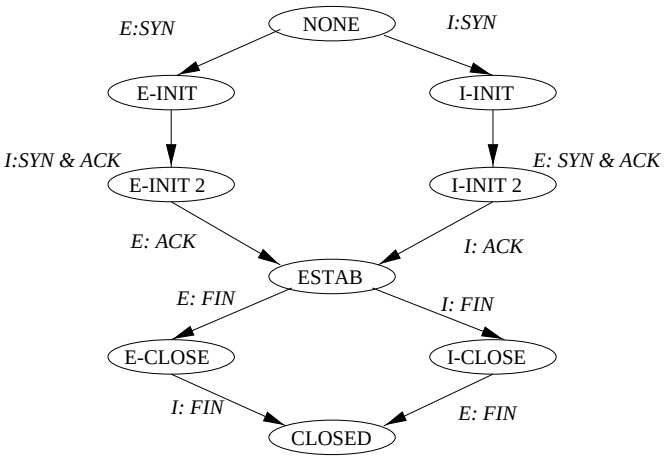
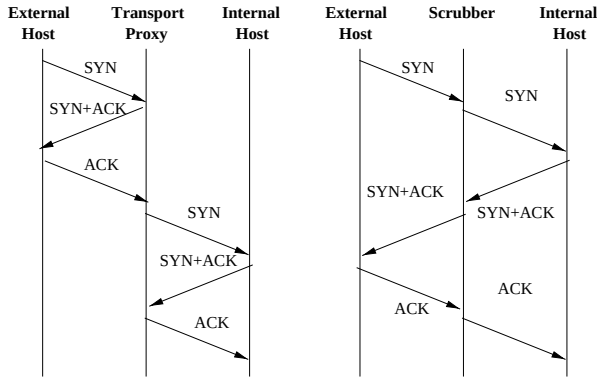


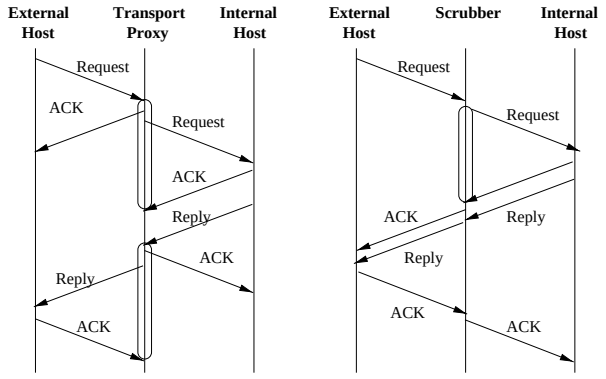
Fig. 2. TCP scrubber's state transition diagram for a single connection.

Figure 2 graphically represents the reduced TCP state processing that occurs at the TCP scrubber. This simple combined bidirectional state machine allows for high scalability by leaving the complex protocol processing to the end-points. The scrubber has only three general states: connection establishment ($INIT$ and $INIT2$ states), established operation ($ESTAB$), and connection termination ($CLOSE$ and $CLOSED$); whereas the endpoint TCP's keep track of much more complex state: fast retransmit, slow start, etc. While this paper deals mainly with the TCP byte stream reconstruction aspect of the TCP scrubber, it is also worthwhile to note that enforcement of the TCP state machine at the scrubber eliminates a set of TCP stack fingerprinting attacks by disallowing random protocol events through the scrubber. For example, an actual connection must be established between a client and a valid service before further TCP packets will pass the scrubber. Even after this, the type of packets that are exchanged are carefully filtered by the scrubber.

Another difference between the TCP scrubber and a transport level proxy is the handling of connection establishment. Figure 3a compares the connection establishment TCP messages that are exchanged with an interposed transport proxy and TCP scrubber. Notice that the approach taken by the scrubber obviates some of the denial of service (and general performance) problems that accompany full transport proxies that must buffer incoming data. Upon connection from an external host, the transport proxy establishes a connection to the appropriate internal host. However, the internal host may not be able to service this connection – for example, the serving



(a) Connection establishment diagram.



(b) Data transfer protocol diagram.

Fig. 3. Examples of TCP messages exchanged between endpoints and interposed mechanism.

host may be down or the service is not running. In the interim, a full incoming socket buffer can be filled by the external host at the transport proxy. This can lead to serious resource allocation problems. In contrast, the TCP scrubber does not keep any data state until the internal service host has acknowledged and reciprocated the TCP connection.

The TCP scrubber scales significantly better than a full transport proxy because the amount of state that must be kept by the scrubber is much less than that kept at a transport proxy. TCP is a reliable byte stream service; therefore a sender must keep a copy of any data it has sent buffered until it receives a message from the receiver acknowledging its receipt. Figure 3b illustrates a data transfer operation from an external client to an internal service using TCP. The circled portions at the center time-line represent the amount of time that data from either the client or server is buffered at the transport proxy or scrubber. Notice that both the scrubber and the transport proxy must buffer the incoming external request until it's receipt is acknowledged by the internal server. However, the server's reply is not modified or buffered by the TCP scrubber; whereas the transport proxy must buffer the outbound reply until it is acknowledged. This is a somewhat subtle point: the outbound reply will generally be held for much longer than the incoming request by an interposed mechanism. This is due to the fact that the *distance* – measured as round trip time and packet losses – from the scrubber to the server will be short relative to the long distance to an external client. It is fair to assume that the scrubber and services it protects are collocated on a fast enterprise network; the scrubber and external client are separated by a wide area network with widely varying loss and latency characteristics. The TCP scrubber's approach to homogenization of TCP flows improves scalability in the number of simultaneous connections it can service.

B.2 TCP Scrubber Implementation

In addition to a novel protocol processing design, the TCP scrubber's in-kernel implementation provides for even greater performance advantages over a user-space transport proxy. Figure 4 shows the software routines that comprise the current implementation of the TCP scrubber within the FreeBSD 2.2.7 kernel [4]. The FreeBSD kernel's networking stack is derived from the BSD 4.4 code[21]. The arrows in Figure 4 show the path packets take through the system. Below the bottom line are the link-level interfaces to the networking code – the Ethernet subsystem is used as the link level transport mechanism in the figure. Incoming packets are handed to the IP processing code through a soft interrupt that invokes `ipintr`. The code between the two dotted lines show the code path of an IP forwarded packet. The routines above the dotted lines comprise the TCP scrubber (`ts_*`). For those packets that belong to scrubbed flows, the packet is given to `ts_input` and then `ts_tcpin`. If the source of the packet is external, the packet is given to `ts_basUpkt`, this scrubs the packet and modifies the data payload if necessary – retransmitted sequences are copied from the scrubber's buffer. If the packet is internal, `ts_forward` is invoked. This routine checks the TCP header and discards any external data that is acknowledged by the internal end-host. The packet is then given to `ts_output` which modifies the next-hop link level address and directly gives the packet to the correct output device driver's link level interface (e.g. `ether_output`).

C. TCP/IP Scrubber Performance

This section presents the results from a series of experiments that profile the TCP/IP scrubber's performance characteristics. They show that, in general, the current implementation of the TCP/IP scrubber can match the performance of both commercial stateful inspection firewalls and raw Unix-based IP forwarding routers when used in networks of up to 500 Mbit per second. For all of the experiments, the interposed machine that ran the TCP/IP scrubbing kernel, the IP forwarding kernel, and the TIS FWTK `plug-gw` proxy was the same: a 300 MHz Pentium II CPU; 128 megabytes main memory; and two Intel EtherExpress Pro 10/100B Ethernet (`fxp` de-

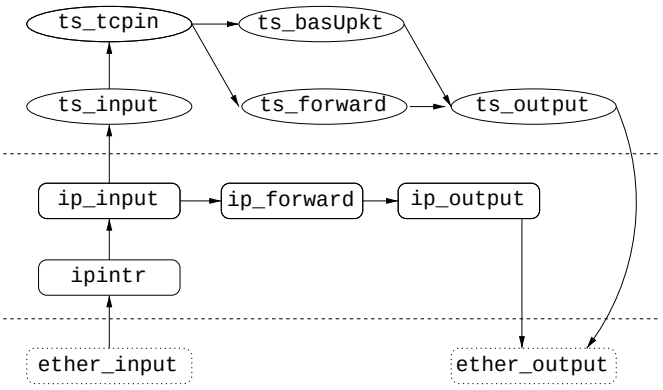


Fig. 4. FreeBSD implementation's kernel software architecture.

vice driver) cards. The TCP/IP scrubbing kernel was used to generate the scrubber's statistics. An unmodified FreeBSD 2.2.7 kernel was used for the IP forwarding numbers. Finally, a modified 2.2.7 kernel was used as a substrate for the `plug-gw` experiments.

TABLE I

THROUGHPUT FOR A SINGLE EXTERNAL CONNECTION TO AN INTERNAL HOST (MBPS, $\pm 2.5\%$ AT 99% CI)

IP Forwarding	Scrubbing	Plug Proxy
83.84	82.87	82.71

Several experiments were undertaken to determine the maximum sustainable bandwidth for the TCP/IP scrubber. The results in Table I provide a baseline measurement of the maximum TCP throughput for a single connection. This throughput was measured using the Netperf benchmark [8]. Three machines were used for the test; all were connected through a 100 Mbps Ethernet switch. The performance of all three forwarding mechanisms were comparable; the networking bandwidth was clearly the first-order bottleneck. In the absence of larger capacity networking resources, we undertook a series of microbenchmarks to pinpoint the TCP/IP scrubber's maximum throughput. These microbenchmarks measured the amount of time it took for a packet to complete the kernel's `ip_input` routine (see Figure 4). For an IP forwarding kernel, the time spent in `ip_input` corresponds to the amount of time needed to do IP processing and forwarding, including queuing at the outbound link-level device (Ethernet). For the TCP/IP scrubber it represents the time to scrub the packet and queue it on the outbound link-level device. Numbers were not gathered for the plug-proxy due to difficulty in matching incoming packets bound for one socket buffer to the outgoing packets from another. Table II shows the results from this experiment. From these numbers it is possible to calculate the optimal sustained throughput (excluding interrupt handling overhead) of both the IP forwarding and TCP scrubber. For scrubbing a stream of TCP packets with full-sized data payloads, the current implementation's ceiling on our test hardware is 366Mbps. We believe that with optimizations and fewer data copies we could increase this ceiling to 891Mbps (13.19 usec latency for scrubbing 1460 byte data payloads).

TABLE II

LATENCY OF TCP/IP FORWARDING AND TCP SCRUBBING (IN MICROSECONDS)

Forwarding Type	Mean	Std Dev
IP Forwarding	8.00	2.91
TCP Scrub (1 byte)	13.19	3.38
TCP Scrub ($>$ 1000)	31.85	5.72

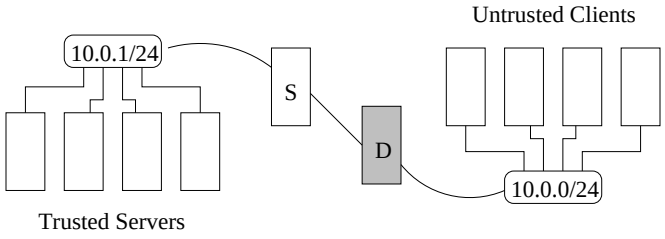
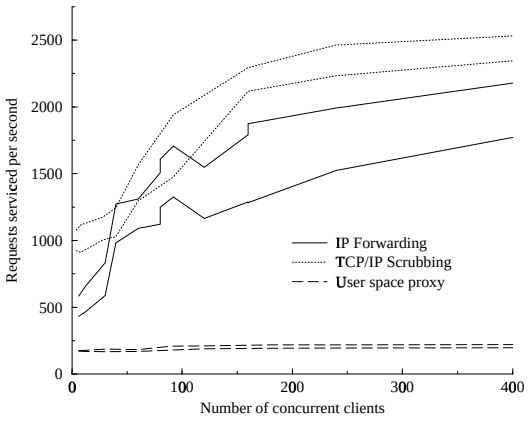


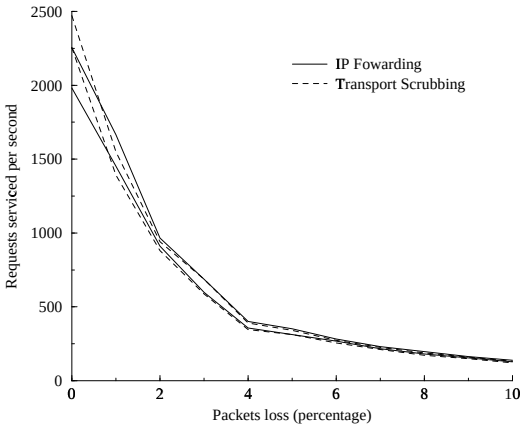
Fig. 5. Experimental apparatus for measuring the protocol scrubber's implementation.

The next set of experiments show that the TCP scrubber does not have a negative impact on the performance characteristics of well-behaved TCP streams. They show this by measuring the sustainable client-server connections per second (similar to transactions) from a set of external client machines to a set of internal server machines. Specifically, an external set of custom web clients made identical fetches to an internal set of Apache web servers. The clients repeatedly fetched 1k byte pages from the servers, stressing the connection setup and teardown process. Figure 5 shows the experimental configuration used in these experiments. The server's `10.0.1/24` network is comprised of an Intel Express 10/100 Ethernet switch; whereas the client's `10.0.0/24` network is an Intel Express 10/100 Ethernet hub. The experiments were measured using a promiscuous mechanism on the client-side hub. The TCP scrubber, IP forwarding router; and `plug-gw` proxy all ran on the *S* machine. For a second set of experiments, a `dummynet` router was used as machine *D* [14]. All ten machines were equipped identically to the TCP/IP scrubber described above. The clients and servers all ran a modified FreeBSD 2.2.7 kernel that was compiled with a large `maxusers` constant.

Figure 6a shows the number of sustained connections per second measured for the TCP/IP scrubber, the IP forwarding router, and the user-space `plug-gw` proxy. The pairs of lines in the graph represent the 99% confidence intervals for the mean sustainable connections per second. The results are twofold: the TCP scrubber's performance is comparable, even better than the raw IP forwarding kernel; and the user-level proxy's performance is extremely low compared to the



(a) Connections per second with no artificial loss.



(b) Connections per second with 480 clients and varied artificial loss.

Fig. 6. TCP scrubber scalability results.

two in-kernel implementations. The first result is a somewhat surprising; however, when looking closely at the data it can be explained by buffering at the TCP scrubber. By buffering the incoming TCP connections, the TCP scrubber shapes the traffic that the servers see, effectively smoothing the request streams so that they are more easily handled at the receivers. These results only apply with very short-lived bursty traffic; the TCP scrubber's performance would decrease relative to IP forwarding when scrubbing long-lived flows. However, this decrease would be relatively small on low-bandwidth networks (100Mbps) as shown in Table I. The second result is not a surprise. The original `plug-gw` code was modified so that it did no logging and no DNS resolutions, which resulted in a large performance increase. The proxy's kernel was also modified so that a large number of processes could be accommodated. A custom user-space proxy optimized for speed would certainly do better (the `plug-gw` proxy forks a child for each incoming connection). However, the multiple data copies and context switching will always resign any user-space implementation to significantly worse performance than the two in-kernel approaches [6], [16].

Finally, we conducted a set of experiments to determine the effects of a lossy link between the external clients and the interposed machine. In these experiments, the number of web clients was fixed at 480, while artificial packet loss was forced on each network flow by a `dummynet` router, labeled *D* in Figure 5. The results of this experiment are shown in Figure 6b. The vertical axis represents the number of requests serviced per second; the horizontal axis represents the proportion of bidirectional packet loss randomly imposed by the dummynet router. The pairs of lines represent the 99% confidence intervals for the mean sustained connections per second. The main result from this experiment is that the TCP scrubbed flows behave comparably to the raw IP forwarded flows.

To put these results in perspective it is useful to compare them with the performance of a fast commercial firewall. CheckPoint reports in a performance white paper that the peak throughput for their FireWall-1 product on a dual 167 MHz UltraSparc with four 100 Mbps Ethernet adapters (200 Mbps on each side) is 89.75 Mbps [19]. While it is difficult to accurately compare the results from separate performance experiments, the TCP scrubber's performance is clearly as good as current firewall technology.

IV. Application Protocol Scrubbing

While the TCP/IP protocol scrubber enables NID systems to accurately detect attacks, there is still a need to actively *prevent* attacks rather than merely identifying them. While this is easily accomplished by application level proxies, such solutions require client modification. These modifications are practically impossible when attempting to protect important infrastructure level protocols such as Internet routing protocols. For this reason we have developed an application protocol scrubber that supports flexible transparent application protocol scrubbing through both a custom socket-based API. By creating a lightweight transparent application-level protocol scrubbing substrate, we allow for the low-cost monitoring and scrubbing of important infrastructure level protocols in a fail-closed manner. The protocol scrubber supports transparent application-level scrubbing by enabling user-level processes to bind a local socket to a set of remote network addresses.

The application protocol scrubber's approach to lightweight transparent interposition contrasts sharply with the explicit approach taken by traditional application-level proxies. As a transparent mechanism, neither endpoint is aware of the application scrubber, thus removing the need to modify the server or the client. Moreover, the application scrubber's approach

addresses two problems with NAT-based transparent proxies: it allows for bidirectional transparency; and it provides explicit mappings for the unambiguous reverse lookup of UDP datagram destinations. The application scrubbing mechanism allows a security administrator to implement an application-level scrubber using any standard Unix programming language. This greatly simplifies the creation of application-level proxies for new protocols.

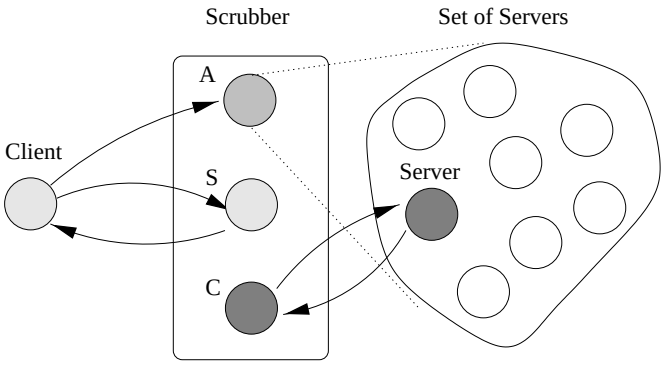


Fig. 7. Overview of transparent application-level socket mechanism.

```
1   int server_socket;
    struct sockaddr_in addrClient;
    struct sockaddr_in addrServer;

5   server_socket = socket(AF_INET,
                           SOCK_STREAM, 0);

    setsockopt(server_socket, SOL_SOCKET,
               SO_REMOTEBIND, &on,
10             sizeof(on));

    bzero((char *) &addrClient,
          sizeof(addrClient));
    addrClient.sin_family = AF_INET;
15  addrClient.sin_addr.s_addr =
      addrActiveRemote.sin_addr.s_addr;
    addrClient.sin_port =
      addrActiveRemote.sin_port;

20  bind(server_socket,
         (struct sockaddr *)&addrClient,
         sizeof(addrClient));

    connect(server_socket,
25          (struct sockaddr *)&addrServer,
           sizeof(addrClient));
```

Fig. 8. Example application level code.

The application-level protocol scrubber requires no modifications of the client or server code. It supports transparent application-level scrubbing through a single conceptual change to the socket programming interface: the protocol scrubber enables a user-level process to bind a local socket to a set of remote addresses. In this way an application scrubber can masquerade as an entire network's set of services. Conceptually, this is shown in Figure 7. The user-level scrubber creates a socket A and binds it to the well-known port as well as the addresses that represent a set of servers. After binding the socket, the scrubber uses it as any other: it performs a `listen` and `accept` call using its descriptor. When a client initiates a connection to a server covered by the scrubber, a new socket is created (as usual) and bound to the specific remote server's address and port. This socket is returned by the kernel as S. This is the same socket a traditional proxy uses to communicate with the client. As an interposed mechanism, the scrubber then creates a new socket, C, for communicating with the server. The scrubber binds C to the client's address and connects it to the true server using a normal `connect` call. The source code in Figure 8 provides an actual example of this process. The `setsockopt` call is used with a new value (`SO_REMOTEBIND`) to flag a socket as remote. After this, the `bind` system call is used to associate the remote address with the socket. The remote address can be widened by supplying a network mask, the default is 32 bits, using the new `SO_REMOTEBIND_MASK` option.

The simple interface to the application scrubber hides significant complexity in the actual implementation. The in-kernel modifications required significant changes to the FreeBSD socket code. The changes to the kernel's socket code were made difficult by the code's original assumptions that a socket only has a single address, and that the socket's IP address is bound to one of the host machine's interfaces. These changes were contained within the base functions for mapping incoming packets to socket buffers. In normal operation, a Unix machine only sends incoming packets with IP addresses that match one of the interface card's IP addresses to the host's TCP stack and eventual socket input processing routines. This is a relatively simple check that does not consume significant overhead; however when a protocol scrubber has a number of application level scrubbers with sockets bound to sets of remote IP addresses, the check consumes a larger amount of processing resources.

V. Conclusions and Future Work

This paper presented the design and implementation of a protocol scrubber, an active interposed mechanism for transparently removing attacks from both transport and application protocol layers in real-time. The key contributions of this work are: the identification of transport scrubbing as a mechanism that enables passive NID systems to operate correctly; the design and implementation of the high performance half-duplex TCP/IP scrubber; and the creation of an active transparently interposed application-level protocol scrubbing mechanism.

The transport scrubber is an active interposed mechanism for converting ambiguous network flows into well-behaved flows that are interpreted identically at all downstream endpoints. The transport scrubber eliminates a class of insidious attacks that subvert passive NID systems by explicitly removing these ambiguities. When used in conjunction with a NID system, a transport scrubber removes these insertion and evasion attacks insuring a high confidence in their detection.

While the security community has examined application proxies, the concept of removing transport level attacks through a transport scrubber has not been previously introduced.

The paper presented the novel design and implementation of the TCP/IP scrubber that removes attacks from the Internet's most common transport protocols. The removal of ambiguities from the TCP reassembly process – one of the most difficult aspects of NID system correctness – was presented as a specific example of the TCP scrubber's operation. The transport scrubber achieves high scalability and performance by leaving the bulk of the TCP processing to the end points. It uses packet arrivals as the only mechanism for driving protocol processing. When coupled with the half-duplex scrubbing design and an in-kernel implementation, the TCP/IP scrubber achieves performance for well-behaved flows comparable to Unix-based routers and stateless commercial firewalls. The TCP scrubber's effect on well-behaved flows is a negligible increase in transmission delay due to the interposed mechanism. However, this effect is increased for flows with ambiguities, effectively trading performance for security.

The paper also presented the creation of a transparent active application-level protocol scrubbing mechanism that can be used to elide or modify application level flows in real-time response to attacks. By creating a lightweight, transparent application scrubbing mechanism, we allow for active scrubbing – a method of active intrusion response – of critical infrastructure level protocols. We envision application level scrubbers removing attacks in real-time from Internet control flows that threaten the infrastructure's stability. The support comes from the custom socket-based API extensions to the FreeBSD kernel.

There are two main directions we are taking this work in the future. One direction is to improve the transport and application scrubbing mechanisms. Specifically, we plan to incorporate zero-copying techniques to the TCP/IP scrubber's data handling routines, bringing the performance even closer to high speed networking levels – 1Gbps and beyond. We also plan to improve the performance of the application scrubber's socket-based mechanisms and dispatching routines. A second area for future work is the construction of application scrubbers for Internet control protocols, such as BGP and OSPF. We believe that intrusion detection will become increasingly important as society's organizations grow more dependent on the Internet. We have shown how protocol scrubbers can be used to significantly benefit an organization's network security through both improved detection and active prevention of protocol attacks.

Acknowledgments

The Intel Corporation provided support for this work through a generous equipment donation and gift. This work was also supported in part by a research grant from the Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Research Laboratory under Grant F30602-99-1-0527. Rob Malan was supported by a Horace H. Rackham Predoctoral Fellowship.

References

[1] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.

[2] D. Brent Chapman and Elizabeth D. Zwicky. *Building Internet Firewalls*. O'Reily and Associates, Inc., 1995.

[3] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, May 1994.

[4] FreeBSD Homepage. http://freebsd.org.

[5] Network Associates Inc. Gauntlet Firewall. `http://www.iss.net/prod/rs.html`.

[6] David Maltz and Pravin Bhagwat. TCP Splicing for Application Layer Proxy Performance. Technical Report RC 21139, IBM Research Division, March 1998.

[7] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network Intrusion Detection. *IEEE Network*, 8(3):26–41, May and June 1994.

[8] Netperf web page. `http://www.netperf.org`.

[9] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proceedings of ACM SIGCOMM '97*, Cannes, France, September 1997.

[10] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[11] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Originally Secure Networks, Inc., now available as a white paper at the Network Associates Inc. homepage at http://www.nai.com/, January 1998.

[12] Marcus J. Ranum. Intrusion Detection: Challenges and Myths. Network Flight Recorder, Inc. whitepaper at http://www.nfr.com/, 1998.

[13] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing a Generalized Tool for Network Monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, San Diego, CA, October 1997.

[14] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[15] Internet Security Services. RealSecure^{TM}. `http://www.iss.net/prod/rs.html`.

[16] Oliver Spatscheck, Jorgen S. Hansen, John H. Hartman, and Larry L. Peterson. Optimizing TCP Forwarder Performance. Technical Report TR98-01, Dept. of Computer Science, University of Arizona, February 1998.

[17] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[18] Trusted Information Systems. TIS Firewall Toolkit. `ftp://ftp.tis.com/pub/firewalls/toolkit`.

[19] Check Point Software Technologies. FireWall-1. `http://www.checkpoint.com`.

[20] Gregory B. White, Eric A. Fisch, and Udo W. Pooch. Cooperating Security Managers: A Peer-Based Intrusion Detection System. *IEEE Network*, 10(1):20–23, January and February 1996.

[21] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.