Self-Securing Network Interfaces: What, Why and How

Gregory R. Ganger, Gregg Economou, Stanley M. Bielski
May 2002
CMU-CS-02-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Self-securing network interfaces (NIs) examine the packets that they move between network links and host software, looking for and potentially blocking malicious network activity. This paper describes self-securing network interfaces, their features, and examples of how these features allow administrators to more effectively spot and contain malicious network activity. We present a software architecture for self-securing NIs that separates scanning software into applications (called scanners) running on an NI kernel. The resulting scanner API simplifies the construction of scanning software and allows its powers to be contained even if it is subverted. We illustrate the potential via a prototype self-securing NI and two example scanners: one that identifies and blocks known e-mail viruses and one that identifies and inhibits rapidly-propagating worms like Code-Red.

We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. This material is based on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

Keywords: Network security, intrusion detection, firewall, proxy, virus, worm, NIC

1 Introduction

Multi-purpose computer systems are, and likely will remain, vulnerable to network intrusions for the foreseeable future. Implementers and administrators are unable to make them bulletproof, because the software is too big and complex, supports too many features and requirements, and involves too many configuration options. As a result, most network environments rely on firewalls and service proxies, with moderate success, to keep malicious parties from exploiting OS and service weaknesses. These special-purpose components observe and filter network traffic before it reaches the vulnerable systems. Usually placed at the boundary between a LAN and the "rest of the world," these components generally limit themselves to trivial filter rules.

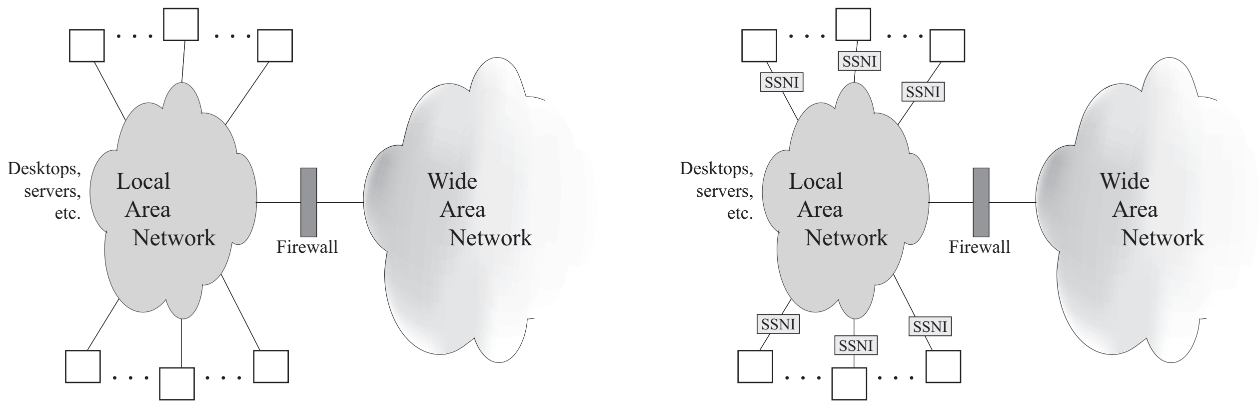
More complete protection could be provided by extending individual network interfaces (NIs) to observe and even contain malicious network activity. Embedding such functionality in each machine's NI has a number of advantages over firewalls placed at LAN boundaries. First, distributing firewall functionality among end-points [20] avoids a central bottleneck and protects systems from local machines as well as the WAN. Second, a misbehaving host can be throttled at its source; as with firewalls, embedding checks and filtering into NIs [17] isolates them from vulnerable host software, preventing a successful intruder or malicious insider from disabling the checks. Third, and most exciting, each NI can focus on a single host's traffic, digging deeper into the lower-bandwidth, less noisy signal comprised of fewer aggregated communication channels. For example, reconstruction of application-level streams and inter-connection relationships becomes feasible without excessive cost. We refer to NIs extended with intrusion detection and containment functionality as *self-securing network interfaces*.

Counters to two common network attacks highlight the potential of self-securing NIs. First, e-mail viruses can be contained much more effectively than the current approach of sending e-mail warnings. Specifically, once a new virus is discovered, the administrator can update all self-securing NIs to identify propagation attempts to and from their hosts, prevent them, and identify machines already infected. Second, the recent Code Red worm [8] (and follow-ons [9, 41]) can be readily identified by the traffic pattern at a self-securing NI. Specifically, these worms spread exponentially by the abnormal behavior of targeting large numbers of randomly-chosen IP addresses with no corresponding DNS translations.

Digging deeply into network traffic, as promoted here, will greatly increase the codebase executing in an NI. Further, it will inevitably lead to less-expert and less-hardened implementations of scanning code, particularly code that examines the application-level exchanges. As a result, well-designed system software will be needed for self-securing NIs, both to simplify scanner implementations and to contain rogue scanners (whether buggy or compromised).

This paper describes a software architecture that addresses both issues. A trusted NI kernel controls the flow of packets between the host interface and the network wire. Scanners, running as application processes, look for and possibly block suspicious network activity. With the right permissions, a scanner can subscribe to see particular traffic via a socket-like interface, pass or clip parts of that traffic, and inject pre-registered packets. The simple API should simplify scanner implementation. More importantly, the separation of power offers a critical degree of safety: while it can disrupt traffic flow, a rogue scanner cannot act arbitrarily as a man-in-the-middle.

We have built a prototype of this self-securing NI software architecture and a number of interesting scanners, including scanners for the e-mail virus and Code Red examples. The internal protection boundary between scanners and the trusted base comes with a reasonable cost. More



(a) Conventional security configuration

(b) Addition of self-securing NIs

Figure 1: **Self-securing network interfaces.** (a) shows the common network security configuration, wherein a firewall protects LAN systems from some WAN attacks. (b) shows the addition of self-securing NIs, one for each LAN system.

importantly, our experiences indicate that the scanner API meets its programmer support goal. In particular, the network interaction part of writing scanner code is straightforward; the complexity is where it belongs: in the scanning functionality.

The remainder of this paper is organized as follows. Section 2 expands on the case for NI-embedded intrusion detection and containment functionality. Section 3 discusses the design of NI system software for supporting such functionality. Section 4 details our prototype implementation. Section 5 evaluates our prototype and some example scanners. Section 6 discusses related work.

2 Self-Securing NIs

The role of the network interface (NI) in a computer system is to move packets between the system's components and the network. A *self-securing NI* additionally examines the packets being moved and enforces network security policies. Like a firewall, the security administrator can configure each self-securing NI to examine packet headers and simply not forward unacceptable packets into or out of its computer system. A self-securing NI can also dig deeper into traffic patterns, slow strange behavior, alert administrators of potential problems, etc. By embedding this traffic management functionality inside the NI, one enjoys its benefits even when the host OS or other machines inside the LAN border are compromised. This section makes a case for self-securing NIs by describing the system architecture in more detail and discussing its features.

2.1 Basic architecture

A self-securing NI extends the NI's base functionality of transferring packets between the host software and the wire [18]. In most systems, this base functionality is implemented in a network interface card (NIC). For our purposes, the component labeled as "NI" will have three properties: (1) it will perform the base packet moving function, (2) it will do so from behind a simple interface with few additional services, and (3) it will be isolated (i.e., compromise independent) from the remainder of the host software. Examples of such NIs include NICs, DSL or cable modems, and NI emulators within a virtual machine monitor [39]. As well, leaf switches have these properties for the hosts directly connected to them. The concepts and challenges of embedding the new functionality in an NI applies equally to each of these.

We define such an NI to be a *self-securing NI* if it internally monitors and enforces policies on packets forwarded in each direction. No change to the host interface is necessary; these security functions can occur transparently within the NI (except, of course, when suspicious activity is actively filtered). For traditional NIs, which exchange link-level messages (e.g., Ethernet frames), examination of higher-level network protocol exchanges will require reconstruction within the self-securing NI software. Although this work is redundant with respect to the host's network stack, it allows self-securing NIs to be deployed with no client software modification. For NIs that offload higher-level protocols (e.g., IP security or TCP) from the host [11, 13], redundant work becomes unnecessary because the only instance of the work is already within the NI.

Self-securing NIs enforce policies set by the network administrator, just as a centralized firewall would. In fact, administrators will configure and manage self-securing NIs over the network, since they must obviously be connected directly to it — doing so allows an administrator to use the NI to protect the network from its host system as well as the other way around. This approach also decouples the NI-enforced policies from the host software; even the host OS and its most-privileged users should not be able to reconfigure or disable the NI's policies. Prior work provides solid mechanisms for remote policy configuration of this sort, and recent research [5, 7, 17, 20] and practice [2, 24] clarifies their application to distributed firewall configuration.

In addition to configuration over the network, alerts about suspicious activity will be sent to administrative systems via the network. The same secure channels used for configuration can be reused for this purpose. These administrative systems can log alerts, construct an aggregate view of individual NI observations, and notify administrators if so configured. As with any intrusion detection system, policy-setting administrators must balance the desire for containment with the damage caused by acting on false alarms. Self-securing NIs can watch for suspicious traffic and generate alerts transparently. But, if they are configured to block or delay suspicious traffic, they may disrupt legitimate user activity.

2.2 Why self-securing NIs

The self-securing NI architecture described above has several features that combine to make it a compelling design point. This subsection highlights six. Two of them, scalability and full coverage, result from distributing the functionality among the endpoints [20]. Two, host independence and host containment, result from the NI being close to and yet separate from the vulnerable host software [17]. Two, less aggregation and more per-link resources, build on the scalability benefit but are worthy of independent mention.

Scalability. The work of checking network traffic is distributed among the endpoints. Each endpoint's NI is responsible for checking only the traffic to and from that one endpoint. The marginal cost of the required NI resources will be relatively low for common desktop network links. More importantly, the total available resources for examining traffic inherently scales with the number of nodes in the network, since each node should be connected to the network by its own self-securing NI. In comparison, the cost of equivalent aggregate resources in a centralized firewall configuration make them expensive (in throughput or dollars) or limit the checking that they can do. This argument is much like cost-effectiveness arguments for clusters over supercomputers [4].

Full coverage. Each host system is protected from all other machines by its self-securing NI, including those inside the same LAN. In contrast, a firewall placed at the LAN's edge protects local systems only from attackers outside the LAN. Thus, self-securing NIs can address some insider attacks in addition to Internet attacks, since only the one host system is inside the NI's boundary.

Host independence. Like network firewalls, self-securing NIs operate independently of vulnerable host software. Their policies are configured over the network, rather than by host software. So, even compromising the host OS will not allow an intruder to disable the self-securing NI functionality. (Successful intruders and viruses commonly disable any host-based mechanisms in an attempt to hide their presence.)

Host containment. Self-securing NIs offer a powerful control to network administrators: the ability to throttle network traffic at its sources. Thus, the LAN and its other nodes can be protected from a misbehaving host. For example, a host whose security status is questionable could have its network access slowed, filtered, or blocked entirely.

Less aggregation. Connected to only one host system, a self-securing NI investigates a relatively simple signal of network traffic. In comparison, a firewall at a network edge must deal with a noisier signal consisting of many aggregated communication channels. The clearer signal may allow a self-securing NI to more effectively notice strange network behavior.

More per-link resources. Because each self-securing NI focuses on only one host's traffic, more aggressive investigation of network traffic is feasible. Although this is really a consequence of the scalability feature, it is sufficiently important that we draw it out explicitly. Also, note that not all traffic must be examined in depth; for example, a self-securing NI could decide to examine e-mail and web traffic in depth while allowing NFS and Quake traffic to pass immediately.

Each feature alone is valuable, which is why most other network security configurations include one or more of them. Self-securing NIs are a compelling addition, because they offer all of these features. Switches and routers within an organization's networking infrastructure share many of these features, though some (e.g., scalability and host containment) degrade the further one gets from the end systems. The following subsection provides some concrete examples of the potential benefits of these features.

2.3 Spotting and containing attacks

The most obvious use of the self-securing NI is to enforce standard firewall filtering rules [10]. These rules typically examine a few fields of packet headers and allow only those for allowed network protocols to pass. Because the filtering occurs within an NI, it can also prevent basic spoofing (e.g., of IP addresses) and sniffing (e.g., by listening with the NI in "promiscuous mode") of network traffic. Previous researchers [17, 20] have made a strong case for distributing such rules among the endpoints, and at least one product [1] has been put on the market.

Traditional firewall rules, however, barely scratch the surface of what can be done with self-securing NIs. The reduced aggregation and reduced link rate/usage make it possible to analyze more deeply the traffic seen. Examples include reconstructing and examining application-level exchanges, shadowing protocol state and examining state transitions, and shadowing host state and correlating inter-protocol relationships. Several concrete examples of network attacks that can be discovered and mitigated are described below.

E-mail viruses. One commonly observed security problem is the rapidly-disseminated e-mail virus. Even after detecting the existence of a new virus, it often takes a significant amount of time to prevent its continued propagation. Ironically, the common approach to spreading the word about such a virus is via an e-mail message (e.g., "don't open unexpected e-mail that says 'here is the document you wanted'"). By the time a user reads the message, it is often too late. An alternative, enabled by self-securing NIs, is for the system administrator to immediately send a new rule to all NIs: check all in-bound and out-bound e-mail for the new virus's patterns.[1] E-mail exchanges generally conform to one of a small set of known protocols. Thus, a properly configured self-securing NI could scan attachments for known viruses before forwarding them. In many cases, this would immediately stop further spread of the virus within an intranet, as well as quickly identifying many of the infected systems. At the least, it would reduce the lifetimes of given e-mail viruses, which usually persist long after they are discovered and automated detection methods are available. Section 5.2 explores this example in greater depth.

Buffer overflow attacks. The most common technique used to break into computer systems over the network is the buffer overflow attack. A buffer overflow attack exploits a particular type of programming error: failing to perform bounds checks and consequently writing past the end of a finite array in memory. The memory beyond the array often holds variables of importance to subsequent program execution. If an attacker knows of such a programming error, he may be able to send to the software messages that exploit the error. Such attacks are particularly powerful when the overflowed buffer is allocated on the stack [29]. A clever attacker with full program knowledge can carefully craft a stack overflow to rewrite the return address of the current procedure to point further up the stack and place code he wishes to execute at that location. With such an attack, a malicious party can run their own code with the permissions of the compromised application process (often "Administrator" or "root" for network services).

The infamous "Internet worm" of 1988 [37] exploited a known such weakness (in the `fingerd` application) and the recent Code Red worms did the same (in Microsoft's IIS server). In each case, the particular overflow attack was well-known ahead of time, but the software fixes were slow to appear and administrators remained vulnerable until the software owners provided patches. By having a self-securing NI look for network service requests that would trigger such overflows, one can prevent them from reaching the system until patches are provided.

SYN bombs. Another frequently cited network attack, called a "SYN bomb," exploits a characteristic of the state transitions within the TCP protocol [31] to prevent new connections to the target. The attack consists of repeatedly initiating, but not completing, the three-packet handshake of initial TCP connection establishment, leaving the target with many partially completed sequences that take a long time to "time out". Specifically, an attacker sends only the first packet (a packet with the SYN flag set), ignoring the target's correct response (a second packet with the SYN and ACK flags set). This attack is difficult to deal with at the target machine, but a self-securing NI connected to the attacking machine (often a compromised host) can easily do so.

There are two variants of the SYN bomb attack. In one variant, the attacker uses its true address in the source fields, and the target's response goes to the attacker but is ignored. To detect this, a self-securing NI can simply notice that its host is not responding with the necessary

[1] Many sites route e-mail through a dedicated mail server, which makes it a natural site for this kind of checking (as long as scalability is not an issue). In general, dedicated proxies are a good place to check the associated application-level exchanges. Self-securing NIs are a good place for such checks when dedicated proxies are not present, not checking, or not required and enforced (since an intruder does not have to conform to client configurations).

final packet of the handshake (an ACK of the target's SYN flag). Upon detecting the problem, the NI could prevent continued "bombing" or even repair the damage itself (e.g., by sending an appropriate packet with the RST flag set to eliminate the partial connection). In the second variant, the attacker forges false entries in the initial packet's source fields, so that the target's reply goes to some other machine. A self-securing NI can prevent such spoofing easily, even if the host OS has been compromised.

Random, exponential spread (Code Red). A highly-visible network attack in 2001 was the Code Red worm (and follow-ons) that propagated rapidly once started, hitting most susceptable machines in the Internet in less than a day [27]. Specifically, these worms spread exponentially by having each compromised machine target random 32-bit IP addresses. Extensions to this basic algorithm, such as hitlist scanning and local subnet scanning, can reduce the propagation time to less than an hour [41]. Looking deeply at the network traffic, however, reveals an abnormal signature: contacting new IP addresses without first performing DNS translations. Although done occasionally, such behavior is uncommon, particularly when repeated rapidly. To detect this, a self-securing NI can shadow its one host's DNS table and check the IP address of each new connection against it. The DNS table can be shadowed easily, in most systems, since the translations pass through the NI. Section 5.3 explores this example in greater depth.

2.4 Costs and Limitations

Self-securing NIs are promising, but there is no silver bullet for network security. They can only detect attacks that use the network and, like most intrusion detection systems [6], are susceptible to both false positives and false negatives. Containment of false positives yields denial of service, and failure to notice false negatives leaves intruders undetected. Also, until anomaly detection approaches solidify, only known attack patterns will be detected.

Beyond these fundamental limitations, there are also several practical costs and limitations. First, the NI, which is usually a commodity component, will require additional CPU and memory resources for most of the attack detection and containment examples above. Although the marginal cost for extra resources in a low-end component is small, it is non-zero. Providers and administrators will have to consider the trade-off between cost and security in choosing which scanners to employ. Second, additional administrative overheads are involved in configuring and managing self-securing NIs. The extra work should be small, given appropriate administrative tools, but again will be non-zero. Third, like any in-network mechanism, a self-securing NI cannot see inside encrypted traffic. While IP security functionality may be offloaded onto NI hardware in many systems, most application-level uses of encryption will make opaque some network traffic. If and when encryption becomes more widely utilized, it may reduce the set of attacks that can be identified from within the NI. Fourth, each self-securing NI inherently has only a local view of network activity, which prevents it from seeing patterns of access across systems. For example, probes and port scans that go from system to system are easier to see at aggregation points. Some such activities will show up at the administrative system when it receives similar alerts from multiple self-securing NIs. But, more global patterns are an example of why self-securing NIs should be viewed as complementary to edge-located protections. Fifth, for host-embedded NIs, a physical intruder can bypass self-securing NIs by simply replacing them (or plugging a new system into the network). The networking infrastructure itself does not share this problem, giving switches an advantage as homes for self-securing NI functionality.

3 Self-Securing NI Software Design

Self-securing NIs offer exciting possibilities for detecting and containing network security problems. But, the promise will only be realized if the required software can be embedded into network interfaces effectively. In particular, the proposed network traffic analyses will involve a substantial body of new software in the NI. Further, some of this software will need to be constructed and deployed rapidly in response to new network security threats. These characteristics will require an NI software system that simplifies the programming task and mitigates the dangers created by potentially buggy software running within the NI.

This section discusses design issues for self-securing NI system software. It expresses major goals, describes a software structure, and discusses its merits. The next section describes a system that implements this structure.

3.1 Goals

The overall goal of self-securing NIs is to improve system and network security. Clearly, therefore, they should not create more difficult security problems than they address. In addition, writing software to address new network security problems should not require excessive expertise. Other goals for NI-embedded software include minimizing the impact on end-to-end exchanges and minimizing the hardware resources required.

Containing compromised scanning code. As the codebase inside the NI increases, it will inevitably become more vulnerable to many of the same attacks as host systems, including resource exhaustion, buffer overflows, and so on. This fact is particularly true for code that scans application-level exchanges or responds to a new attack, since that code is less likely to be expertly implemented or extensively tested. Thus, a critical goal for self-securing NI software is to contain compromised scanning code. That is, the system software within the NI should be able to limit the damage that malicious scanning code can cause, working on the assumption that it may be possible for a network attacker to subvert it (e.g., by performing a buffer overflow attack).

Assuming that the scanning code decides whether the traffic it scans can be forwarded, malicious scanning code can certainly perform a denial-of-service attack on that traffic. Malicious scanning code also sees the traffic (by design) and will most likely be able to leak information about it via some covert channel. The largest concerns revolve around the potential for man-in-the-middle attacks and for effects on other traffic. Our main goal is to prevent malicious scanning code from executing these attacks: such code should not be able to replace the real stream with its own arbitrary messages and should not be able to read or filter traffic beyond that which it was originally permitted to control.

Reduced programming burden. We anticipate scanning code being written by non-experts (i.e., people who do not normally write NI software or other security-critical software). To assist programmers, the NI system software should provide services and interfaces that hide unnecessary details and reduce the burden. In the best case, programming new scanning code should be as easy as programming network applications with sockets.

Containing broken scanning code. Imperfect scanning code can fail in various ways. Beyond preventing security violations, it is also important to fault-isolate one such piece of code from the others. This goal devolves to the basic protection boundaries and bounded resource utilization commonly required in multi-programmed systems.

Transparency in common case. Although not a fundamental requirement, one of our goals is for self-securing NI functionality to not affect legitimate communicating parties. Detection can occur by passively observing network traffic as it flows from end to end. Active changes of traffic occur only when needed to enforce a containment policy.

Efficiency. Efficiency is always a concern when embedding new functionality into a system. In this case, the security benefits will be weighed against the cost of the additional CPU and memory resources needed in the NI. Thus, one of our goals is to avoid undue inefficiencies. In particular, non-scanned traffic should incur little to no overhead, and the system-induced overhead for scanned streams should be minimal. Comprehensive scanning code, on the other hand, can require as many resources as necessary to make their decisions — administrators can choose to employ such scanning code (or not) based on the associated trade-off between cost and security.

3.2 Basic design achieving these goals

This section describes a system software architecture for self-securing NIs that addresses the above goals. As illustrated in Figure 2, the architecture is much like any OS, with a trusted kernel and a collection of untrusted applications. The trusted NI kernel manages the real network interface resources, including the host and network links. The application processes, called scanners, use the network API offered by the NI kernel to get access to selected network traffic and to convey detection and containment decisions. Administrators configure access rights for scanners via a secure channel.

Scanners. Non-trivial traffic scanning code is encapsulated into application processes called scanners. This allows the NI kernel to fault-isolate them, control their resource usage, and bound their access to network traffic. With a well-designed API, the NI kernel can also simplify the task of writing scanning code by hiding some unnecessary details and protocol reconstruction work. In this design, programming a scanner should be similar to programming a network application using sockets, both in terms of effort required and basic expertise required. (Of course, scanners that look at network protocols in detail, rather than application-level exchanges, will involve detailed knowledge of those protocols.)

Scanner interface. Table 1 lists the basic components of the network API exported by the NI kernel. With this interface, scanners can examine specific network traffic, alert administrators of potential problems, and prevent unacceptable traffic from reaching its target.

The interface has four main components. First, scanners can *subscribe* to particular network traffic, which asks the NI kernel for `read` and/or `contain` rights; the desired traffic is specified with a packet filter language [26]. The NI kernel grants access only if the administrator's configuration for the particularly scanner allows it. In addition to the basic packet capture mechanism, the interface should allow a scanner to subscribe to the data stream of TCP connections, hiding the stream reconstruction work in the NI kernel.

Second, scanners ask the NI kernel for more data via a *read* command. With each data item returned, the NI kernel also indicates whether it was sent by or to the host. Third, for subscriptions with `contain` rights, a decision for each data unit must be conveyed back to the kernel. The data unit can either be *pass*ed along (i.e., forwarded to its destination) or *cut* (i.e., dropped without forwarding). For a data stream subscription, *cut* and *pass* refer to data within the stream; in the base case, they refer to specific individual packets. For TCP connections, a scanner can also decide to *kill* the connection.

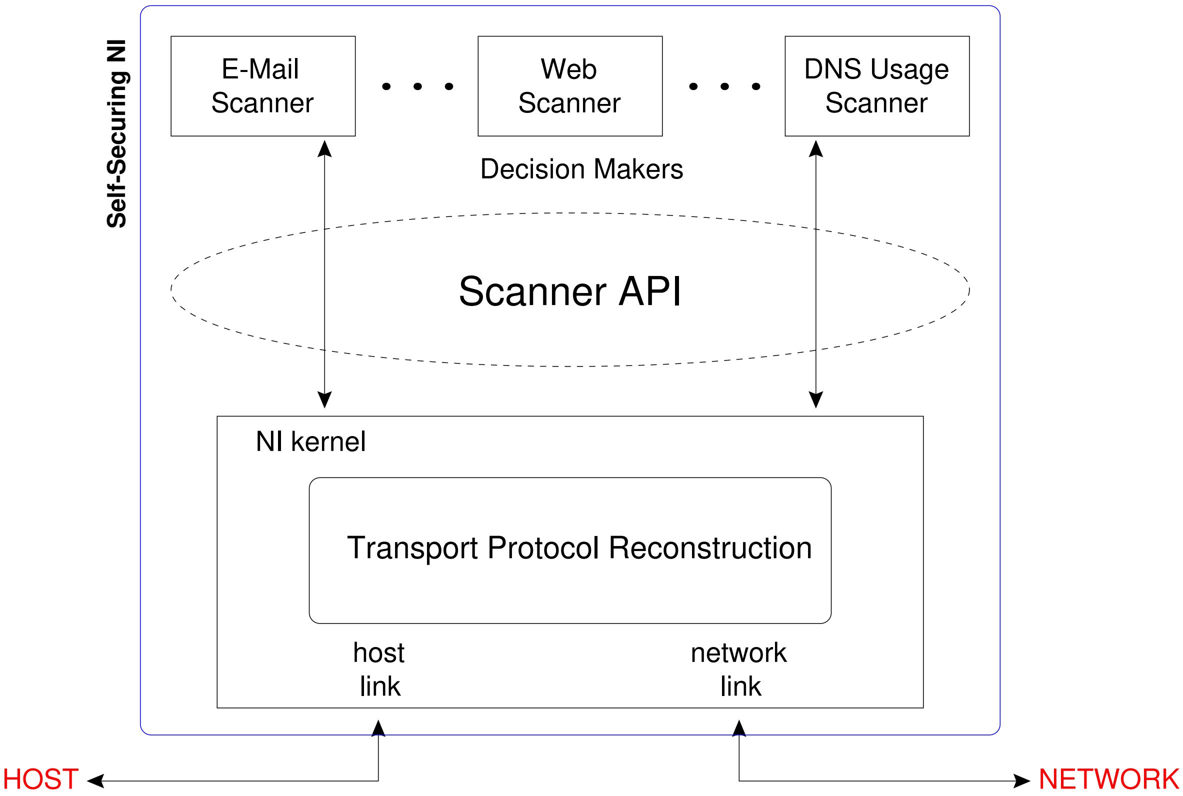


Figure 2: **Self-securing NI software architecture.** An "NI kernel" manages the host and network links. Scanners run as application processes. Scanner access to network traffic is limited to the API exported by the NI kernel.

Fourth, a scanner can *inject* pre-registered data into scanned communications, which may involve insertion into a TCP stream or generation of an individual packet. A scanner can also send an *alert*, coupled with arbitrary information or even copies of packets, to an administrative system.

The scanner interface simplifies programming, allows necessary powers, and yet restricts the damage a rogue scanner can do. A scanner can look at and gate the flow of traffic with a few simple commands, leaving the programmer's focus where it belongs: on the scanning algorithms. A scanner can ask for specific packets, but will only see what it is allowed to see. A scanner can decide what to pass or drop, but only for the traffic to which it has `contain` rights. A scanner can inject data into the stream, but only pre-configured data in its entirety. Combining *cut* and *inject* allows replacement of data in the stream, but the pre-configured *inject* data limits the power that this conveys. Alerts can contain arbitrary data, but they can only be sent to a pre-configured adminsitrative system.

NI Kernel. The NI kernel performs the core function of the network interface: moving packets between the host system and the network link. In addition, it implements the functionality necessary to support basic scanner (i.e., application) execution and the scanner API. As in most systems, the NI kernel owns all hardware resources and gates access to them. In particular, it bounds scanners' hardware usage and access to network traffic.

Packets arrive in NI buffers from both sides. As each packet arrives, the NI kernel examines its headers and determines whether any subscriptions cover it. If not, the packet is immediately forwarded to its destination. If there is a subscription, the packet is buffered and held for the

Command	Description
Subscribe	Ask to scan particular network data
Read	Retrieve more from subscribe buffers
Pass	Allow scanned data to be forwarded
Cut	Drop scanned data
Kill	Terminate the scanned session (if TCP)
Inject	Insert pre-registered data and forward
Alert	Send an alert message to administrator

Table 1: **Network API exported to scanner applications.** This interface allows an authorized scanner to examine and block specific traffic, but bounds the power gained by a rogue scanner.

appropriate scanners. After each `contain`-subscribing scanner conveys its decision on the packet, it is either dropped (if any say drop) or forwarded.

NI kernels should also reconstruct transport-level streams for protocols like TCP, to both simplify and limit the power of scanners that focus on application-level exchanges. Such reconstruction requires an interesting network stack implementation that shadows the state of both endpoints based on the packets exchanged. Notice that such shadowing involves reconstructing two data streams: one in each direction. When a scanner needs more data than the TCP window allows, indicated by blocking *read*s from a scanner with pending decisions, the NI kernel must forge acknowledgement packets to trigger additional data sent from endpoints. In addition, when data is cut or injected into streams, all subsequent packets must have their sequence numbers adjusted appropriately.

Administrative interface. The NI's administrative interface serves two functions: receiving configuration information and sending alerts. (Although we group them here, the two functions could utilize different channels.)

The main configuration information is scanner code and associated access rights. For each scanner, provided access rights include allowed subscriptions (`read` and `contain`) and allowed injections. When the NI kernel starts a new scanner, it remembers both, preventing a scanner from subscribing to any other traffic or injecting arbitrary data (or even other scanners' allowed injections).

When requested by a scanner, the NI kernel will send an alert via the administrative interface. Overall, scanner transmissions are restricted to the allowed injections and alert information sent to pre-configured administrative systems.

3.3 Discussion

Implemented properly, we believe this design can meet the goals for self-securing NIs. Our experiences indicate that writing scanners is made relatively straightforward by the scanner API. Moreover, restricting scanners to this API bounds the damage they can do. Certainly a scanner with `contain` rights can prevent the flow of traffic that it scans, but its ability to prune other traffic is removed and its ability to manipulate the traffic it scans is reduced.

A scanner with `contain` rights can play a limited form of man-in-the-middle by selectively utilizing the *inject* and *cut* interfaces. The administrator can minimize the danger associated with *inject* by only allowing distinctive messages. (Recall that *inject* can only add pre-registered messages and in their entirety. Also, a scanner cannot *cut* portions of *inject*ed data.) In theory, the ability to transparently *cut* bytes from a TCP stream could allow a rogue scanner to rewrite the stream arbitrarily. Specifically, the scanner could clip bytes from the existing stream and keep just those that form the desired message. In practice, we do not expect this to be a problem; unless the stream is already close to the desired output, it will be difficult to construct the desired output without either breaking something or being obvious (e.g., the NI kernel can be extended to watch for such detailed clipping patterns). Still, small *cut*s (e.g., removing the right "not" from an e-mail message) could produce substantial changes that go undetected.

Although scanners still have some undesirable capabilities, we believe that the NI software architecture described is a significant improvement over unbounded access.

4 Implementation

This section describes a prototype implementation of the self-securing NI software architecture described in Section 3.

4.1 Overview

The prototype self-securing NI is actually an old PC (referred to below as the "NI machine") with two Ethernet cards, one connected to the real network and one connected point-to-point to the host machine's Ethernet link. Figure 3 illustrates the hardware setup. Clearly, the prototype hardware characteristics differ from real NIC hardware, but it does allow us to explore the system software issues that are our focus in this work.

Our software runs on the FreeBSD 4.4 operating system. Both network cards are put into "promiscuous mode," such that they grab copies of all frames on their Ethernet link; this configuration allows the host machine's real Ethernet address to be used for communication with the rest of the network. Our NI kernel, which we call *Siphon*, runs as an application process and uses BPF [23] to acquire copies of all relevant frames arriving on both network cards. Frames destined for the NI machine flow into its normal in-kernel network stack. Frames to or from the host machine go to Siphon. All other frames are dropped. Scanners also run as application processes, and they communicate with Siphon via named UNIX sockets. Datagram sockets are used for getting copies of frames, and stream sockets are used for reconstructed data streams. (In OS architecture terms, FreeBSD is being used as a microkernel with Siphon as an "NI kernel server.")

4.2 Scanner interface implementation

Functionally, our implemented scanner interface matches the one described in Section 3 and illustrated in Table 1. The structure of our prototype, however, pushes for a particular style in the interface. Scanners communicate with Siphon via sockets, receiving subscribed-to traffic via READ and passing control information via WRITE. This section details the interactions for both frame-level scanning and reconstructed-stream scanning.

Frame-level scanning interface. Scanners can see and make decisions on raw Ethernet frames via the frame-level scanning interface, which is a datagram socket connected to Siphon.

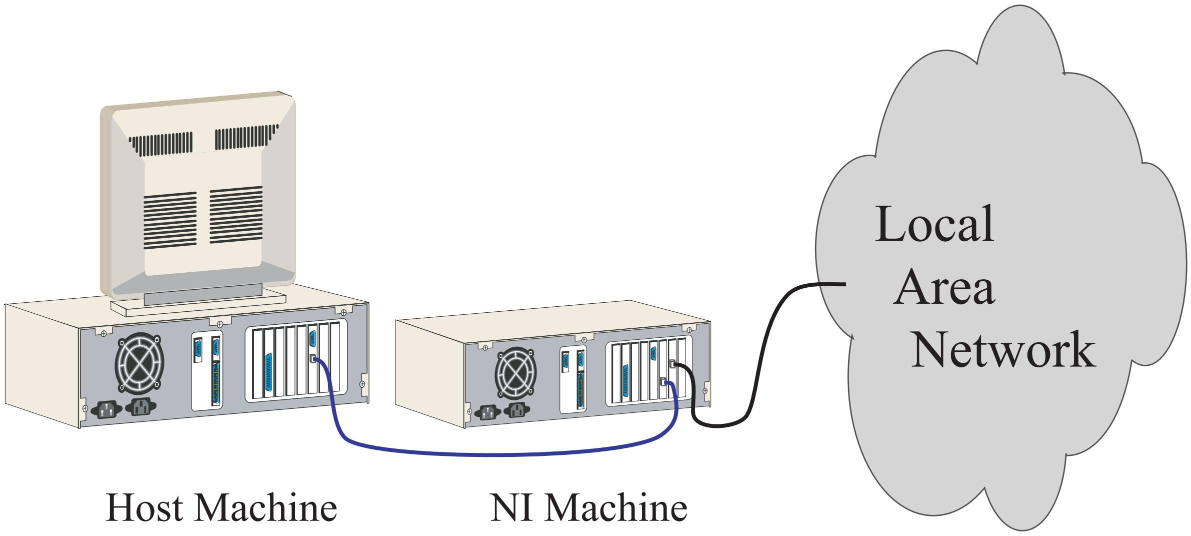


Figure 3: **Self-securing NI prototype setup.** The prototype self-securing NI is an old PC with two network cards, one connected directly to the host machine and one connected to the network.

Frames that match any of a scanner's frame-level subscriptions are written by Siphon to that scanner's socket. For each successful READ call on the socket, a scanner gets a small header and a received frame. The header indicates the frame's length and whether it came from the host or from the network. In addition, each frame is numbered according to how many previous frames the scanner has READ: the first frame read is #1, the second frame is #2, and so on.

A frame-level scanner conveys decisions on scanned frames via WRITEs to the socket, each consisting of a frame number and the decision (*cut* or *pass*). A scanner conveys its requests via this interface as well. *Inject* requests specify which pre-registered packet should be sent (via an index into a per-scanner table) and in which direction. *Alert* requests provide the message that should be sent to the administrative system. *Subscribe* requests ask for additional frames to be seen via the same socket; the desired frames are described via a sequence of <offset,value> pairs, much like most packet filter languages [26].

Reconstructed-stream scanning interface. Scanners use stream sockets to get reconstructed TCP streams from Siphon, with a similar interface to that described above. The differences are changes to some parameters, two new requests, and a way of attaching to new connections as they are established. The main parameter changes relate to how scanned data is identified: *cut* and *pass* decisions apply to a byte offset and length within the stream in a particular direction. As well, *inject* requests must specify the byte offset at which the pre-registered data should be inserted into the stream (shifting everything after it forward by length bytes). Finally, *subscribe* requests simply specify inbound and outbound TCP port numbers (or wildcards) on which to listen.

The two new requests are *kill*, which tells the NI kernel to terminate the connection being scanned, and *more*, which tells the NI kernel that more data is necessary before a decision can be made. The *more* request is needed because our application-level NI kernel cannot actually see a scanner blocking on a READ request, as it would if implemented as a true kernel. The NI kernel must know about the need for more data, since it may need to offer extra space in the TCP window to trigger additional data transmission.

For reconstructed-stream scanning, several sockets are required. One is used to convey subscription requests. A second listens for new connections from Siphon. An ACCEPT on this second connection creates a new socket that corresponds to one newly established TCP connection between the host machine and some other system. READs and WRITEs to such new connections receive data to be scanned and convey decisions and requests.

4.3 The NI kernel: Siphon

Siphon performs the basic function of a network interface, moving packets between the host and the network. It also exports the scanner API described above.

During initialization, Siphon sets up a BPF interface from which it can READ all frames sent to or from the host machine. Each frame is buffered and passed through a packet filter engine. If the frame does not match any of the packet filter rules, it is immediately forwarded to its target (either the host machine or the network link). There are three types of packet filter rules: *prevent*, *scan*, and *reconstruct*. If the frame matches a *prevent* rule, it is dropped immediately; *prevent* rules provide traditional firewall filtering without the overhead of an application-level scanner. If the frame matches a *scan* rule, it is written to the corresponding scanner's datagram socket. If the frame matches a *reconstruct* rule, it is forwarded to the TCP reconstruction code. For frames that match *scan* and *reconstruct* rules for subscriptions with `contain` rights, Siphon keeps copies and remembers the decisions that it needs. A frame is forwarded if and only if all subscribed scanners decide *pass*; otherwise, it is dropped.

Siphon's TCP reconstruction code translates raw Ethernet frames into the reconstructed-stream interface described above. While doing so, it tries to minimize the pertubation on an end-to-end exchange.

In the common case, Siphon can reconstruct TCP streams by just watching the packets that go by. Upon seeing the first packet of a new TCP connection that matches a subscription, Siphon creates two protocol control blocks, one to shadow the state of each end-point. Each new packet indicates a change to one end-point or the other. When the connection is fully established, Siphon opens and CONNECTs a stream socket to each subscribed scanner. When one side tries to send data to the other, that data is first given to subscribed scanners. If all such scanners with `contain` rights decide *pass*, the original packets are forwarded. When the TCP connection closes, Siphon CLOSEs the corresponding stream socket.

Once a scanner asks for an active change to the stream, Siphon can no longer just passively delay, reconstruct from, and then forward frames for the corresponding TCP connection; it must now modify some of them. If a scanner asks for some data to be *cut*, Siphon must prune that data from the original packets; doing so requires changes to several TCP and IP fields, and it may require splitting one packet into two. In addition, Siphon must send acknowledgements for the *cut* data once all bytes up to it have been acknowledged by the true receiver. Finally, the sequence numbers and acknowledgements of subsequent packets must be adjusted to account for the *cut* data.

A similar set of active changes are needed for *inject*. Siphon must create, forward, and buffer packets for the injected data, retransmitting as necessary. Subsequent packets must have their sequence numbers and acknowledgements adjusted, and one original packet may have to be split in two if its data spans the *inject* point.

More requests require less work. For small amounts of additional data, the TCP window can be opened further to get the sender to provide more data. Otherwise, Siphon must forge acknowledgements to the source and then handle retransmissions to the destination. In this case, Siphon must also drop redundant acknowledgements from the receiver.

Kill requests are handled by forging packets with the RST flag set and sending one to each end-point.

4.4 Issues

Our prototype focuses on exploring the scanner API, and it does set aside two important implementation issues on which we do not expect to innovate: administrative interface and bounded resource utilization. For a full implementation, one would employ well-established technologies for both issues. Although it is always a dangerous claim, we do not see either of these issues invalidating the experiences or results arising from use of the current prototype.

The administrative interface for the current prototype consists of a directly-connected terminal interface. Clearly, this is not appropriate for practical management of per-host self-securing NIs. Fortunately, well-established cryptography-based protocols [2, 5, 7, 17, 20, 24] exist for remotely distributing policy updates and receiving alerts.

The current prototype also does not preclude scanners from excessive resource utilization, instead relying on the underlying FreeBSD kernel to timeshare. A real NI kernel implementation would need to explicitly prevent any scanner from using too many resources. With each scanner as a single process with no I/O requirements, such resource management should be relatively straightforward for an OS kernel.

5 Evaluation

This section evaluates our scanner API and the Siphon prototype. It does so via two very different scanners, exploring how the interface supports their construction. The first scanner examines application-level exchanges for known problems (e-mail viruses). The second scanner looks for a particular suspicious activity (random IP-based propagation) at the network protocol level. Both are easily implemented given Siphon's scanner API.

5.1 Basic overheads

Although its support for scanners is our focus, it is useful to start with Siphon's effect on NI throughput and latency. For all experiments in this paper, the NI machine runs FreeBSD 4.4 and is equipped with a 300MHz Pentium II, 128MB of main memory, and two 100Mb/s Ethernet cards. After subtracting the CPU power used for packet management functions that could be expected to be hardware-based, we believe that this dated system is a reasonable approximation of a feasible NI. The host machine runs FreeBSD 4.4 and is equipped with a 1.4GHz Pentium III, 512MB of main memory, and a 100Mb/s Ethernet card. Although Siphon is operational, little tuning has been done.

Table 2 shows results for three configurations: the host machine alone (with no NI machine), the NI machine with no scanners, and the NI machine reconstructing all TCP streams but then immediately forwarding them. We observe minimal latency difference among the three configurations. But, the throughput difference between "No NI machine" and "No scanners" highlights a

Configuration	Roundtrip	Bandwidth
No NI machine	0.84 ms	7.73 MB/s
No scanners	0.88 ms	4.37 MB/s
Reconstruct	0.88 ms	4.30 MB/s

Table 2: **Base performance of our self-securing NI prototype.** The roundtrip latency is measured with 20,000 pings. Throughput is measured by RCPing 100MB. “No NI machine” corresponds to the host machine with no self-securing NI in front of it. “No scanners” corresponds to Siphon immediately passing on each packet. “Reconstruct” corresponds to reconstructing the TCP stream for scanning but then immediately forwarding it.

disappointing bottleneck in our prototype: the BPF interface from which Siphon gets its packets bounds its throughput far below the wire’s 100Mb/s bandwidth. A kernel-based implementation would not have this problem. Reconstructing the TCP stream results in only small additional overhead.

5.2 Example: e-mail virus scanner

A promising activity for self-securing NIs is to examine application-level exchanges for known problems, such as viruses or buffer overflows. As a concrete example, this section describes and evaluates a scanner that examines e-mail traffic.

What the scanner looks for: Commonly, viruses that propagate via e-mail do so in the form of infected attachments. For example, the attachment may be a malicious script or a complex file format (e.g., Excel or Word) with a malicious macro. Our scanner parses incoming and outgoing e-mail messages to identify attachments, which are then passed to virus checking code. By updating the virus checking code, an administrator can immediately identify subsequent attempts to propagate a known virus.

How the scanner works: There are several e-mail protocols. This scanner focuses on two common protocols: Post Office Protocol (POP) and Simple Mail Transport Protocol (SMTP). SMTP [32] is used to send e-mail from one machine to another. POP [33] is used to update a replica of one’s main mailbox on a second machine. Both protocols function such that e-mail messages are transfered in their entirety, including all attachments, over a TCP connection. Attachments are encoded in MIME format [16].

The e-mail scanner *subscribe*s with `contain` rights to reconstructed TCP streams corresponding to the default port numbers for these protocols (25 for SMTP and 110 for POP). Any communications other than e-mail messages are *pass*ed immediately. When the start of an e-mail message is detected, the scanner waits until it has the entire message before making its decision. If necessary, the scanner uses the *more* request to tell Siphon that it needs more data from the stream. The scanner then parses the message, decodes each attachment, and passes it to the virus checking code. If no viruses are detected, the scanner tells Siphon to *pass* the entire e-mail message.

The current scanner can use either of two virus checking mechanisms. The first is a simple table lookup, in which the MD5 hash is compared to a list of known malicious attachments. This check is time- and space-efficient, but it will only identify non-mutating e-mail viruses. The second is the Sophos Anti-Virus library [36]. This library uses modern scanning algorithms, with regular updates, to identify the wide variety of known viruses and even some virus-like signatures.

Configuration	Per-message latency
No scanner	22.4 ms
Null scanner	67.0 ms
Scanner w/MD5	107.1 ms
Scanner w/Sophos	109.9 ms

Table 3: **Message latency with the e-mail scanner.** The average per-message latency is for one pass through a month's worth of e-mail. Each value is an average of three iterations, and all standard deviations are less than 1% of the mean.

When a virus is detected: The scanner does not allow infected attachments to be forwarded. It tells Siphon to *cut* the range of bytes making up the attachment, to *inject* a replacement attachment at the original offset, and to *pass* the remainder of the e-mail message. The pre-registered replacement attachment (MIME-encoded as "text/plain") informs the recipient that an infected attachment was removed. In addition, the original message is sent to the administrative machine in an *alert*, for subsequent analysis. This also ensures that the message data is not lost, since it is possible that the attachment was flagged in error.

Performance data: We investigate the performance overhead involved with our e-mail scanner by using POP to transfer one month's worth of the first author's e-mail (1500 e-mail messages, with an average size of 8240 bytes and a maximum size of 776KB). Transferring the full set of messages at maximum speed, we measure the average time per message for four configurations: no scanner, a null scanner, the e-mail scanner using MD5, and the e-mail scanner using the Sophos library. Table 3 shows the results.

The results indicate that the scanner will delay e-mail messages. We observe substantial overhead for parsing the e-mail and exchanging information with Siphon. Much of this is due to an unoptimized scanner. As an anecdotal experiment in scanner programmability, this scanner was written by a recent B.S. graduate who learned POP, SMTP, and socket programming for this project. He observed that constructing an operational e-mail scanner was not difficult, and he reused functions for parsing e-mail, decoding MIME enclosures, and buffering data read from sockets.

Fortunately, the impact of slowed e-mail on user experiences should be minimal. Delaying e-mail delivery by a small amount is unlikely to be noticed. Throughput should also not be a problem, since even a popular user usually gets fewer than 100 messages in a day.

Discussion: Implementing an e-mail scanner for POP and SMTP was straightforward using the scanner API; the scanner simply watches for the beginning of an e-mail message and then examines that message. For more interactive mail exchange protocols, such as IMAP [12], additional effort will be required. In particular, IMAP transfers e-mail messages in pieces rather than as a whole, and those pieces are not self-identifying. An IMAP scanner will have to track the exchanges to identify when an attachment is being transferred so that it can invoke the virus scanner. Although this requires more application-level logic in the scanner, we do not expect it to be difficult.

5.3 Example: Code-Red scanner

Another promising activity for self-securing NIs is to look for suspicious activity at the network protocol level, such as unanswered SYN+ACK packets, incomplete IP fragments, and unexpected IP addresses. As a concrete example, this section describes and evaluates a scanner that watches for the Code-Red worm's abnormal network behavior.

What the scanner looks for: The Code-Red worm and follow-ons spread exponentially by having each compromized machine target random 32-bit IP addresses. This propagation approach is highly effective because the IP address space is densely populated and relatively small. Although done occasionally, it is uncommon for a host to connect to a new IP addresses without first performing a name translation via the Domain Name System (DNS) [25]. Our scanner watches DNS translations and checks the IP addresses of new connections against them. It flags any sudden rise in the count of "unknown" IP addresses as a potential problem.

How the scanner works: The Code-Red scanner consists of two parts: shadowing the host machine's DNS table and checking new connections against it. Upon initialization, therefore, the scanner *subscribe*s to three types of frames. The first two specify UDP packets sent by the host to port 53 and sent by the network from port 53; port 53 is used for DNS traffic.[2] The third specifies TCP packets sent by the host machine with only the SYN flag set, which is the first packet of TCP's connection-setup handshake. Of these, only the third subscription includes `contain` rights.

Each DNS reply can provide several IP addresses, including the addresses of authoritative name servers. When it *read*s a DNS reply packet, the scanner parses it to identify all provided IP addresses and their associated times to live (TTLs). The TTL specifies for how long the given translation is valid. Each IP address is added to the scanner's table and kept at least until the TTL expires. Thus, the scanner's table should contain any valid translations that the host may have in its DNS cache. The scanner prunes expired entries only when it needs space, since host applications may utilize previous results from gethostbyname() even after the DNS translations expire.

The scanner checks the destination IP addresses of the host machine's TCP SYN packets against this table. If there is a match, the packet is *pass*ed. If not, the scanner considers it a "random" connection. The current policy flags a problem when there are more than two unique random connections in a second or ten in a minute.

When an attack is detected: The scanner's current policy reacts to potential attacks by sending an alert to the administrative system and slowing down excessive random connections. It stays in this mode for the next minute and then re-evaluates and repeats if necessary. The *alert* provides the number of random connections over the last minute and the most recent port to which a connection was opened. Random connections are slowed down by delaying decisions; in attack reaction mode, the scanner tells Siphon *pass* for one of the SYN packets every six seconds. This allows such connections to make progress, somewhat balancing the potential for false positives with the desire for containment. If all susceptible hosts were equipped with self-securing NIs, this policy would have increased the 14 hour propagation time of Code-Red (version 2) [27] to over a month (assuming the original scan rate was 10/second per infected machine [41]).

Two extensions to the current scanner are under consideration. First, the scanner can log the exchanges of random connections via the *alert* interface, allowing an administrator to study them at her convenience. Second, when the rate of random connections is very high, SYN packets can

[2] Although we see none in our networks, DNS traffic can be passed on TCP port 53 as well. Our current scanner will not see this, but could easily be extended to do so.

simply be dropped; this will prevent connections from being established, but can also cause harm given a false positive.

Performance data: We evaluate two aspects of Code-Red scanner performance: its effect on latency and the required DNS table size. To evaluate the scanner's effect on DNS translation latency, we measured the times for 100 different translations with and without the scanner. The results indicate that the scanner increases the translation latency by 99% (from 1.5ms to 2.9ms) compared to having no NI machine. Since DNS translations and SYN packets are minor parts of overall network activity, we believe that the increased latencies due to scanning would have negligible impact on performance.

We evaluate the table sizes needed for the Code-Red scanner by examining a trace of all DNS translations for 10 desktop machines in our research group over 2 days. Assuming translations are kept only until their TTL's expire, each machine's DNS cache would contain an average of 209 IP addresses. The maximum count observed was 293 addresses. At 16 bytes per entry (for the IP address, the TTL, and two pointers), the DNS table would require less than 5KB. (We also observed that the average latency over the 700,000 translations was 1.6ms, very close to our baseline "no NI machine" measurement above.)

It is interesting to consider the table size required for an aggregate table. As a partial answer, we observe that a combined table for the 10 desktops would require a maximum of 750 entries (average of 568) or 12KB. This matches the results of a recent DNS caching study [21], which finds that caches shared among 5 or more systems exhibit a 80–85% hit rate. They found that aggregating more client caches provides little additional benefit. Thus, one expects an 80–85% overlap among the caches, leaving 15–20% of the entries unique per cache. Thus, 10,000 systems with 250 entries each would yield approximately 375,000–500,000 unique entries (6MB–8MB) in a combined table.

Discussion: The largest danger of the Code-Red scanner is that other mechanisms could be used (legitimately) for name translation. There are numerous research proposals for such mechanisms [38, 34, 43], and even experimenting with them would trigger our scanner. Administrators who wish to allow such mechanisms in their environment would need to either disable this scanner or extend it to understand the new name translation mechanisms.

With a scanner like this in place, different tactics will be needed for worms to propagate without being detected quickly. One option is to slow the scan rate and "fly under the radar," but this dramatically reduces the propagation speed, as discussed above. Another approach is to use DNS's reverse lookup support to translate random IP addresses to names, which can then be forward translated to satisfy the scanner's checks. But, extending the scanner to identify such activity would be straightforward. Yet another approach would be to explore the DNS name space randomly[3] rather than the IP address space; this approach would not enjoy the relevant features of the IP address space (i.e., densely populated and relatively small). There are certain to be other approaches as well. The scanner described takes away a highly convenient and effective propagation mechanism; worm writers are thus forced to expend more effort and/or to produce less successful worms. Security is a "game" of escalation, and self-securing NIs arm those in the white hats.

Finally, it is worth noting that all of the Code-Red worms exploited a particular buffer overflow

[3]The DNS "zone transfer" request could short-circuit the random search by acquiring lists of valid names in each domain. Many domains disable this feature. Also, self-securing NIs could easily notice its use.

that was well-known ahead of time. An HTTP scanner could easily identify requests that attempt to exploit it and prevent or flag them. The DNS-based scanner, however, will also spot worms, such as the Nimda worm, that use random propagation but other security holes.

6 Related Work

Self-securing NIs build on much existing technology and borrow ideas from previous work. In particular, network intrusion detection, virus detection, and firewalls are well-established, commonly-used mechanisms [6, 10]. Also, many of the arguments for distributing firewall functions [17, 20, 28] and embedding them into network interface cards [1, 17] have been made in previous work. This previous work and others [2, 7, 24] also address the issue of remote policy configuration for such systems. We extend previous work with examples of more detailed traffic analysis and a system software structure for supporting them.

There are few examples of detailed network intrusion detection documented in the literature, though many system administrators create tools when the need arises. One well-described example is Bro [30], an extensible, real-time, passive network monitor. Bro provides a scripting language for reacting to pre-programmed network events, which works well for experts. Its clean framework replaced a collection of *ad hoc* scripts, which is how most environments examine network traffic. Our work builds on such previous work by providing a programming model that should accommodate less-expert scanner writers and contain broken scanners. As well, embedding scanning functionality into NIs instead of network taps eliminates several of the challenges described in Bro, such as overload attacks, dropped packets, and crash attacks.

Application proxies, particularly for e-mail and web traffic, can be used as intermediaries between vulnerable LAN systems and particular application services. Some such proxies examine the corresponding dataflows to identify and block dangerous data [22, 35]. Each such proxy addresses a single protocol, introduces a central bottleneck, and sometimes creates a visibility problem by making all requests appear to come from a single system. Self-securing NIs allow similar checking in a multi-purpose, scanner-constraining platform.

A substantial body of research has examined the execution of application functionality by network cards [15, 19] and infrastructure components [3, 14, 40, 42]. Although scanners are not fully trusted, they are also not submitted by untrusted clients. Nonetheless, this prior work lays solid groundwork for resource management within network components.

7 Summary

Self-securing network interfaces are a promising addition to the network security arsenal. This paper makes a case for them, identifies NI software design challenges, and describes an NI software architecture to address them. It illustrates the potential of self-securing NIs with a prototype NI kernel and example scanners that address two high-profile network security problems: e-mail viruses and Code-Red style worms.

References

[1] 3Com. *3Com Embedded Firewall Architecture for E-Business*. Technical Brief 100969-001. 3Com Corporation, April 2001.

[2] 3Com. *Administration Guide, Embedded Firewall Software*. Documentation. 3Com Corporation, August 2001.

[3] D. Scott Alexander, Kostas G. Anagnostakis, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. *The price of safety in an active network*. MS–CIS–99–04. Department of Computer and Information Science, University of Pennsylvania, 1999.

[4] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, **15**(1):54–64, February 1995.

[5] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. *IEEE Symposium on Security and Privacy* (Oakland, CA, 4–7 May 1997), pages 65–71. IEEE Computer Society Press, 1997.

[6] Stefan Axelsson. *Research in intrusion-detection systems: a survey*. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[7] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[8] CERT. CERT Advisory CA-2001-19 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL, July 19, 2001. http://www.cert.org/advisories/CA-2001-19.html.

[9] CERT. CERT Advisory CA-2001-26 Nimda Worm, September 18, 2001. http://www.cert.org/advisories/CA-2001-26.html.

[10] B. Cheswick and S. Bellovin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley, Reading, Mass. and London, 1994.

[11] Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol implementation on the Nectar communication processor. *ACM SIGCOMM Conference* (Philadelphia, PA), September 1990.

[12] M. Crispin. *Internet message access protocol – version 4rev1*, RFC–2060. Network Working Group, December 1996.

[13] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, **7**(4):36–43, July 1993.

[14] Dan S. Decasper, Bernhard Plattner, Guru M. Parulkar, Sumi Choi, John D. DeHart, and Tilman Wolf. A scalable high-performance active network node. *IEEE Network*, **13**(1):8–19. IEEE, January–February 1999.

[15] Marc E. Fiuczynski, Brian N. Bershad, Richard P. Martin, and David E. Culler. *SPINE: an operating system for intelligent network adaptors*. UW TR-98-08-01. 1998.

[16] N. Freed and N. Borenstein. *Multipurpose internet mail extensions (MIME) part one: format of internet message bodies*, RFC–2045. Network Working Group, November 1996.

[17] David Friedman and David Nagle. *Building Firewalls with Intelligent Network Interface Cards*. Technical Report CMU–CS–00–173. CMU, May 2001.

[18] Gregory R. Ganger and David F. Nagle. Better security via smarter devices. *Hot Topics in Operating Systems* (Elmau, Germany, 20–22 May 2001), pages 100–105. IEEE, 2001.

[19] David Hitz, Guy Harris, James K. Lau, and Allan M. Schwartz. Using Unix as one component of a lightweight distributed kernel for multiprocessor file servers. *Winter USENIX Technical Conference* (Washington, DC), 23-26 January 1990.

[20] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. *ACM Conference on Computer and Communications Security* (Athens, Greece, 1–4 November 2000), pages 190–199, 2000.

[21] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. *ACM SIGCOMM Workshop on Internet Measurement* (San Francisco, CA, 01–02 November 2001), pages 153–167. ACM Press, 2001.

[22] David M. Martin Jr, Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking java applets at the firewall. *Symposium on Network and Distributed Systems Security* (San Diego, CA, 10–11 February 1997), pages 16–26, 1997.

[23] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 259–269, January 1993.

[24] Mark Miller and Joe Morris. Centralized administration of distributed firewalls. *Systems Administration Conference* (Chicago, IL, 29 September – 4 October 1996), pages 19–23. USENIX, 1996.

[25] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. *ACM SIGCOMM Conference* (Stanford, CA, April 1988). Published as *ACM SIGCOMM Computer Communication Review*, **18**(4):123–133. ACM Press, 1988.

[26] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: an efficient mechanism for user-level network code. *ACM Symposium on Operating System Principles* (Austin, TX, 9–11 November 1987). Published as *Operating Systems Review*, **21**(5):39–51, 1987.

[27] D. Moore. The Spread of the Code-Red Worm (CRv2), 2001. http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml.

[28] Dan Nessett and Polar Humenn. The multilayer firewall. *Symposium on Network and Distributed Systems Security* (San Diego, CA, 11–13 March 1998), 1998.

[29] Aleph One. Smashing the Stack for Fun and Profit. *Phrack 49*, **7**(49), November 8, 1996.

[30] Vern Paxson. Bro: a system for detecting network intruders in real-time. *USENIX Security Symposium* (San Antonio, TX, 26–29 January 1998), pages 31–51. USENIX Association, 1998.

[31] J. Postel. *Transmission Control Protocol*, RFC–761. USC Information Sciences Institute, January 1980.

[32] Jonathan B. Postel. *Simple mail transfer protocol*, RFC–788. Network Working Group, November 1981.

[33] M. Rose. *Post office protocol – version 3*, RFC–1081. Network Working Group, November 1988.

[34] Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms* (Heidelberg, Germany, 12–16 November 2001), pages 329–350, 2001.

[35] Sophos. MailMonitor, 2002. http://www.sophos.com/products/software/mailmonitor/.

[36] Sophos. Sophos Anti-Virus, 2002. http://www.sophos.com/products/software/antivirus/.

[37] Eugene H. Spafford. The Internet worm: crisis and aftermath. *Communications of the ACM.*, **32**(6):678–687.

[38] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Conference* (San Diego, CA, 27–31 August 2001). Published as *Computer Communication Review*, **31**(4):149–160, 2001.

[39] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *Annual USENIX Technical Conference* (Boston, MA, 25–30 June 2001), pages 1–14. The USENIX Association, 2001.

[40] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications*, **35**(1):80–86, January 1997.

[41] N. Weaver. Warhol Worms: The Potential for Very Fast Internet Plagues, posted August 15, 2001. http://www.cs.berkeley.edu/∼nweaver/warhol.html.

[42] David Wetherall. Active network vision and reality: lessons from a capsule-based system. *Symposium on Operating Systems Principles* (Kiawah Island Resort, SC., 12–15 December 1999). Published as *Oper. Syst. Rev.*, **33**(5):64–79. ACM, 1999.

[43] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. *Tapestry: an infrastructure for fault-toleran wide-area location and routing*. UCB Technical Report UCB/CSD–01–1141. Computer Science Division (EECS) University of California, Berkeley, April 2001.