Finding and containing enemies within the walls with self-securing network interfaces

Gregory R. Ganger, Gregg Economou, Stanley M. Bielski

January 2003
CMU-CS-03-109 3

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

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 how self-securing network interfaces can help administrators to identify and contain compromised machines within their intranet. By shadowing host state, self-securing NIs can better identify suspicious traffic originating from that host, including many explicitly designed to defeat network intrusion detection systems. With normalization and detection-triggered throttling, self-securing NIs can reduce the ability of compromised hosts to launch attacks on other systems inside (or outside) the intranet. We describe a prototype self-securing NI and example scanners for detecting such things as TTL abuse, fragmentation abuse, "SYN bomb" attacks, and random-propagation worms like Code-Red.

1 Introduction

Traditional network security focuses on the boundary between a protected intranet and the Internet, with the assumption being that attacks come from the outside. The common architecture uses network intrusion detection systems (NIDSs) and firewalls to watch for and block suspicious network traffic trying to enter the protected intranet. This is a very appropriate first line of defense, since many digital attacks do indeed come from the wild. But, it leaves the protected intranet unprotected from an intruder who gets past the outer defenses.[1]

It is increasingly important to detect and contain compromised hosts, in addition to attempting to prevent initial compromise. Such containment offers a degree of damage control within the intranet, preventing one compromised system from infecting the others unchecked (NIDSs and firewalls are rarely used on the bulk of internal traffic). In addition, such containment is needed to reduce liability—as attack traceback techniques improve, increasing numbers of attacks may be traced to their apparent sources. Dealing with the aftermath of being blamed for attacks can be expensive (in money and time), even if the source machine in question is subsequently found to have been compromised.

A mechanism for identifying compromised hosts must have two features: (1) it must be able to observe potentially-malicious activities, and (2) it must not be trivially susceptible to being disabled. Host-based IDSs have the first feature but not the second; they are highly vulnerable to exactly the software whose soundness is being questioned, a fact illustrated by the many rootkits that disable host IDS checks. A NIDS at the edge of the intranet has the second feature but lacks much of the first; it can observe attacks launched outward but does not see attacks on other systems in the intranet. To also contain problem machines, a mechanism must have a third feature: (3) control over their access to the network.

One promising approach is to extend the various network interfaces (NIs) within the intranet to look for and perhaps contain compromised hosts [15]. By "NI", we mean some component that sits between a host system and the rest of the intranet, such as a network interface card (NIC) or a local switch port. These NIs are isolated from the host OS, running distinct software on separate hardware, and are in the path of a host's network traffic. Thus, they have all three features above. In addition, because they are in their host's path to the LAN, such NIs will see every packet, can fail closed, can isolate their host if necessary, and can actively normalize [16, 21] the traffic. We refer to NIs extended with intrusion detection and containment functionality as *self-securing network interfaces*.

Self-securing NIs enjoy the scalability and coverage benefits of recent distributed firewall systems [14, 19, 1]. They also offer an excellent vantage point for looking inward at a host and watching for misbehavior. In particular, many of the difficulties faced by NIDSs [30] are avoided: there are no topology or congestion vagaries on an NI's view of packets moved to/from its host and there are no packets missed because of NI overload. This allows the NI to more accurately shadow important host OS structures (e.g., IP route information, DNS caches, and TCP connection states) and thereby more definitively identify suspicious behavior.

This paper details several examples of how the NI's view highlights host misbehavior. First, many of the NIDS attacks described by Ptacek and Newsham [30] involve abusing TTLs, frag-

[1]Our focus in this work is on network intruders who successfully gain access to an internal machine, as opposed to insiders who can exploit physical access to the hardware.

mentation, or TCP sequencing to give the NIDS a different view of sent data than the target host. From its local vantage, a self-securing NI can often tell when use of these networking features are legitimate. For example, IP fragmentation is important, but is only used by a well-behaved packet source in certain ways; a self-securing NI can shadow state and identify these. The IP Time-To-Live (TTL) field may vary among packets seen by a NIDS at an intranet edge, because packets may traverse varied paths, but should not vary in the original packets sent by a given host for a single TCP stream. After taking away most deception options, a self-securing NI (or other NIDS) can use traditional NIDS functionality to spot known attacks [16].

Second, state-holding DoS attacks will be more visible, since the NI sees exactly what the host receives and sends. For example, the NI can easily tell if a host is ignoring SYN/ACK packets, as would be the case if it is participating in a "SYN bomb" DoS attack. Third, the random propagation approach used by the recent Code-Red worm [8] (and follow-ons [7, 9, 35]) can be readily identified by the abnormal behavior of contacting large numbers of randomly-chosen IP addresses with no corresponding DNS translations. To detect this, a self-securing NI can shadow its host's DNS cache and check the IP address of each new connection against it.

The main contributions of this paper are: (1) the description of an architecture, based on self-securing NIs, for addressing a two-stage attack model that can be expected to grow in importance as intranet perimeter strength grows; and (2) the description of several host misbehavior detectors, not previously seen by the authors, enabled by the placement of self-securing NIs at the host's LAN access point.

The remainder of this paper is organized as follows. Section 2 expands on the threat model being explored and how NI-embedded intrusion detection and containment functionality helps. Section 3 describes a prototype self-securing NI used for experimenting with the concepts. Section 4 describes several example detectors for self-securing NIs. Section 5 discusses related work.

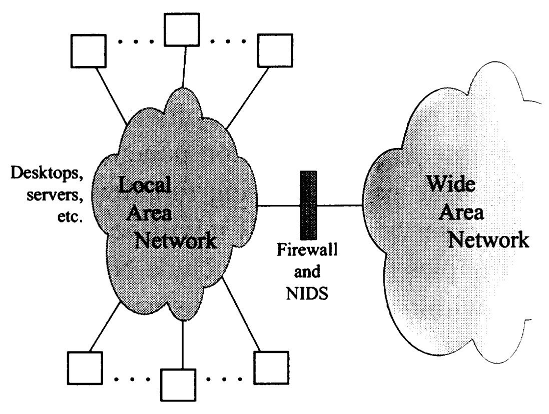
2 Towards per-machine perimeters

The common network security approach maintains an outer perimeter (perhaps with a firewall, some proxies, and a NIDS) around a protected intranet. This is a good first line of defense against network attacks. But, it leaves the entire intranet wide open to an attacker who gains control of any machine within the perimeter. This section expands on this threat model, self-securing NIs, how they help, and their weaknesses.

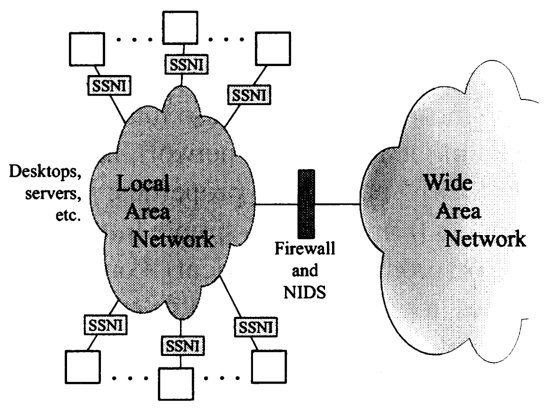
2.1 Threat model

The threat with which we are most concerned here is a multi-stage attack. In the first stage, the attacker compromises any host on the inside of the intranet perimeter. By "compromises," we mean that the attacker subverts its software system, gaining the ability to run arbitrary software on it with OS-level privileges. In the second stage, the attacker uses the internal machine to attack other machines within the intranet.

This form of two-stage attack is of concern because only the first stage need worry about intranet-perimeter defenses; actions taken in the second stage do not cross that perimeter. Worse, the first stage need not be technical at all; an attacker can use social engineering, bribery, a discovered modem on a desktop, or theft (e.g., of a password or insecure laptop) for the first stage.



(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 and a NIDS protect LAN systems from some WAN attacks. (b) shows the addition of self-securing NIs, one for each LAN system.

Finding a single hole is unlikely to be difficult in any sizable organization. Once internal access is gained, the second stage can use known, NIDS-detectable system vulnerabilities, since it does not enter the view of the perimeter defenses. In some environments, depending on their configuration, known attacks launched out of the organization may also proceed unhampered; this depends on whether the NIDS and firewall policies are equally restrictive in both directions.

Our main focus is on the second stage of such two-stage attacks. A key characteristic of the threat model described is that the attacker has software control over a machine inside the intranet, but does not have physical access to its hardware. We are not specifically trying to address insider attacks, in which the attacker would also have physical access to the hardware and its network connections. Clearly, for a self-securing NI to be effective, we must also assume that neither the administrative console nor the NI itself are compromised.

2.2 Self-securing network interfaces

A countermeasure to our two-stage attack scenario must have two properties: (1) it must be isolated from the system software of the first stage's target, since it would otherwise be highly vulnerable in exactly the situations we want it to function, and (2) it must be close to its host in the network path, or it will be unable to assist with intranet containment.[2] We focus on the network interface (NI) as a good place for countermeasures.

The role of the 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 network firewalls and NIDSs, a self-securing NI operates independently of host software; its checks and rules remain in effect even when the corresponding host OS is compromised. (Successful intruders and viruses commonly disable any host-based mechanisms

[2]Many of the schemes explored here could be also used at the edge to detect second-stage attacks from inside to out. Our goal of internal containment requires finer-grained perimeters. As discussed below, distributed firewalls can also protect intranet nodes from internal attacks, though they must be extended to what we call self-securing NIs in order to benefit from the containment and source-specific detection features described.

to hide their presence and bypass restrictions.) Immediate proximity also allows a self-securing NI to normalize and throttle its host's network traffic at the source.

In most systems, the base functionality of moving packets between the host software and the wire 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 host software. Examples of such NIs include NICs, leaf switches in a LAN, DSL or cable modems, and NI emulators within a virtual machine monitor [37]. The benefits of embedding detection and containment functionality in an NI applies to all of these.

Self-securing NIs enforce policies set by the network administrator, much like distributed firewalls [14, 19, 1]. In fact, administrators would configure and manage self-securing NIs over the network, since they must obviously be connected directly to it — this approach is necessary for an administrator to use the NI to protect the network from its host system; even the host OS and its most-privileged users must not be able to reconfigure or disable the NI's policies. Alerts about suspicious activity will be sent to administrative systems via the same secure channels. Prior work provides solid mechanisms for remote policy configuration of this sort, and recent research [4, 6, 14, 19] and practice [2, 22] clarifies their application to distributed firewall configuration.

2.3 Self-securing NI features

A self-securing NI performs intrusion detection on a host's network traffic, impedes communication when compromise is detected (if so configured), and normalizes odd traffic. For addressing the two-stage attack threat, self-securing NIs provide three main features.

First, a self-securing NI sees the packets as sent and received by its host. This allows it to see misuse of low-level networking features, as is the case in many deception attacks on NIDSs. It also allows it to shadow host state and more clearly identify attack patterns than could a less-localized NIDS.

Second, a self-securing NI can slow, filter, or cut off communication to or from a host. For inbound traffic, this is a traditional firewall. For outbound traffic, this is a form of containment, which may be enacted if the host is determined to be compromised or otherwise misbehaving. Interestingly, the NI's position in the host's communication path means that it fails closed. Among other things, this addresses concerns of overloading attacks on its NIDS—such overloading slows down the NI, and thus the host, but does not cause it to miss packets. Thus, a compromised host using this tactic on its self-securing NI is denying service only to itself.

Third, a self-securing NI is in an ideal position to normalize outbound traffic. Doing it at the source leverages the effort involved in detecting NIDS deception attacks, since the two require similar state tracking.

Additional distributed firewall benefits In addition to host containment and local viewpoint, per-machine self-securing NIs share the benefits of previous distributed firewall systems. Previous researchers (e.g., [14, 19]) have made a strong case for distributing firewall and NIDS rules among the endpoints. In particular, distributing such functionality among end-points avoids a central bottleneck (scaling naturally with the number of machines on the network) and protects systems from intranet machines as well as the WAN.[3] These features do not require separation from the

[3]Distributed firewalls protect against internal machines by protecting targets rather than containing sources. It

host OS, so host-based IDS and firewall mechanisms will enjoy them.

The scalability benefit, in particular, impacts the cost consequences of a self-securing NI component. Each endpoint's NI is responsible for checking only the traffic to and from that one endpoint. Therefore, the marginal cost of the required NI resources will be low for common desktop network links [14], particularly when their utilization is considered. In comparison, the cost of equivalent aggregate resources in any centralized containment configuration would make it expensive (in throughput or dollars) or limit the checking that it can do, even if one were willing to suffer the wiring and performance costs of routing all LAN packets through a central point.

2.4 Costs, limitations, and weaknesses

Self-securing NIs (SSNIs) are promising, but there is no silver bullet for network security. SSNIs can only detect attacks that use the network and, like most intrusion detection systems [5], are susceptible to both false positives and false negatives. Containment responses to false positives yields denial of service, and failure to notice false negatives leaves intruders undetected.

Like any NIDS component, a self-securing NI is subject to a number of attacks [30]. Most insertion attacks are either detectable signals (when from the host) and/or subject to normalization [16, 21]. DoS attacks on the NI's detection capabilities are converted to DoS on the host; for attacks launched from the host, this is an ideal scenario.

As the codebase inside the NI increases, it will inevitably become more vulnerable to many of the same attacks as host systems, such as buffer overflows. Compromised scanning code sees the traffic it scans (by design) and will most likely be able to leak information about it via some covert channel. 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. The largest concerns, however, revolve around the potential for man-in-the-middle attacks and for effects on other traffic. In traditional passive NIDS components, such DoS and man-in-the-middle attacks are not a problem. Although we know of no way to completely prevent this, the software design of our prototype attempts to reduce the power of individual scanning programs.

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 some portion of network traffic opaque. 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,

is possible that this is enough. We believe, however, that the extra visibility and control offered by self-securing NIs are valuable. For example, the detectors described in Section 4 exploit the NI vantage point to reduce detection assumptions.

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 A self-securing NI prototype

This section describes our prototype self-securing NI. 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 or switch hardware, but it does allow us to explore the self-securing NI features that are our focus in this work.

Examining network traffic in detail will increase the codebase executing in an NI. 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). One goal of our prototype's design is to prevent malicious scanning code from executing arbitrary man-in-the-middle 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.

3.1 Internal software architecture

Our prototype's software architecture, illustrated in Figure 2, 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 an API offered by the NI kernel to access 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. (Of course, scanners that look at network protocols in detail, rather than application-level exchanges, will require detailed understanding of those protocols.)

Scanner interface. Table 1 lists the basic components of the scanner 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

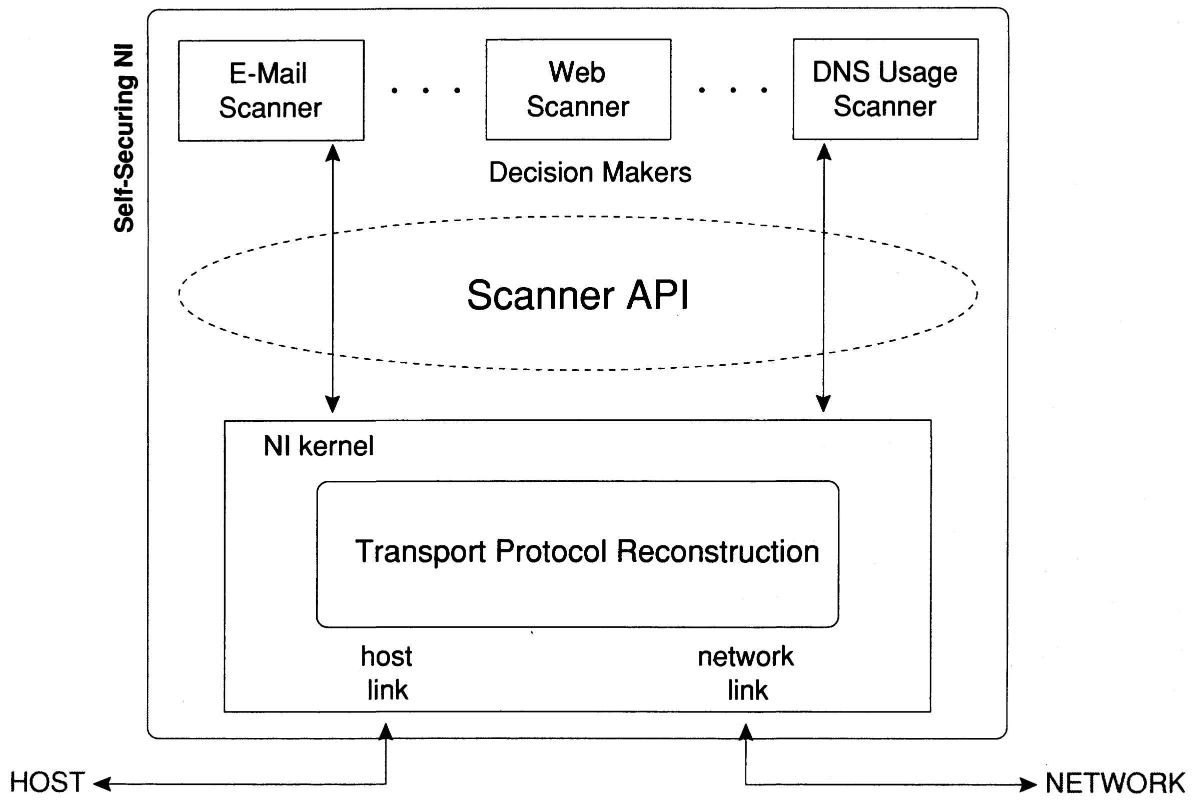


Figure 2: **Self-securing NI software architecture.** A "NI kernel" manages the host and network links. Scanners run as application processes. Scanner access to network traffic is limited to the API exported by the NI kernel.

with a packet filter language [25]. The NI kernel grants access only if the administrator's configuration for the particular scanner allows it. In addition to the basic packet capture mechanism, the interface allows 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.

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.

This scanner API simplifies programming, allows necessary powers, and yet restricts the damage a rogue scanner can do. 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

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. Pass, cut, kill, and inject can only be used by scanners with both read and contain rights.

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 the host and the network link. 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 appropriate scanners. After each `contain`-subscribing scanner conveys its decision on the packet, it is either dropped (if any say drop) or forwarded. For scanners that examine full packets (rather than raw frames), reconstitution of fragmented packets is done by the NI kernel; if not *cut*, normalized fragments are then forwarded.

The NI kernel reconstructs TCP streams to both simplify and limit the power of scanners that focus on application-level exchanges. As with any NIDS, 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. So that data seen by a scanner matches that seen by the destination, a normalized version is forwarded.

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.

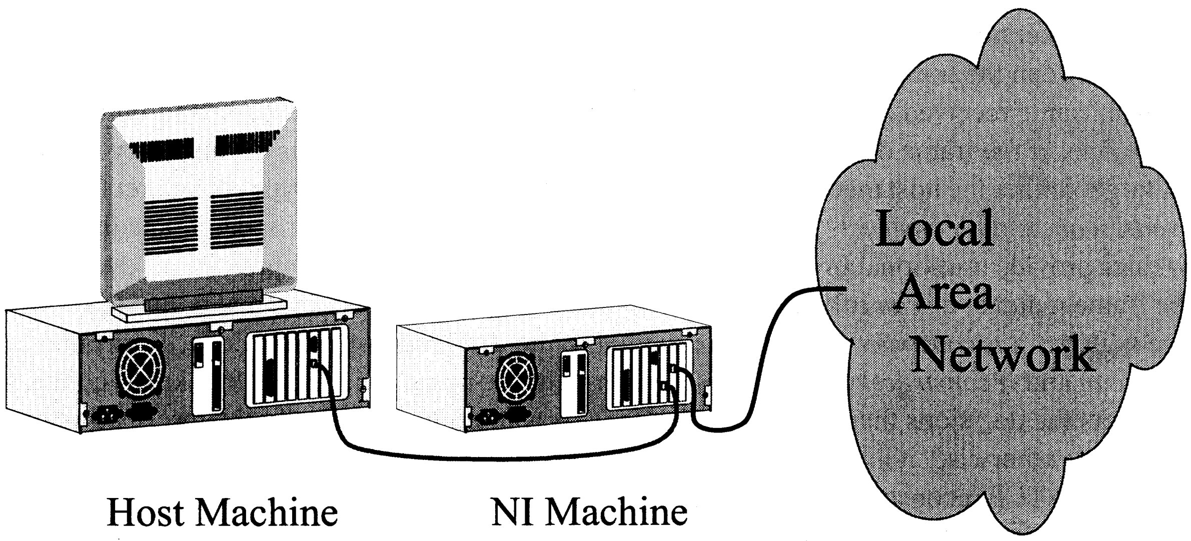


Figure 3: **Self-securing NI prototype setup.** The prototype self-securing NI is an old PC with two network cards, one connected directly to the host machine and one connected to the network.

3.2 Prototype implementation

Our software runs on the NetBSD 1.4.3 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*, sits inside the NetBSD kernel and taps into the relevant device drivers to acquire copies of all relevant frames arriving on both network cards. Frames destined for the NI machine are allowed to flow into NetBSD's normal in-kernel network stack. Frames to or from the host machine go to Siphon. All other frames are dropped.

Scanners run as application processes. Scanners communicate with Siphon via named UNIX sockets, receiving subscribed-to traffic via READ and passing control information via WRITE. Datagram sockets are used for getting copies of frames, and stream sockets are used for reconstructed data streams.

Frame-level scanning interface. 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. *cut* and *pass* decisions are given in terms of this frame number. *inject* requests specify which pre-registered packet should be sent (via an index into a per-scanner table) and in which direction.

Reconstructed-stream scanning interface. For reconstructed-stream scanning, several sockets are required. One listens for new connections from Siphon. An ACCEPT on this 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. *cut* and *pass* decisions specify a byte offset and length within the stream in a particular direction. *inject* requests specify the byte offset at which the pre-registered data should be inserted into the stream (shifting everything after it forward by length bytes).

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.

Each frame received (from the host or from the LAN) 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. Upon seeing the host agree to a new TCP connection, 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*, packets are created, buffered, transmitted, and retransmitted as necessary. When the TCP connection closes, Siphon CLOSEs the corresponding stream socket. If a scanner asks for some data to be *cut* or *inject*ed, the sequence numbers and acknowledgements of subsequent packets must be adjusted accordingly. In addition, Siphon must send acknowledgements for the *cut* data once all bytes up to it have been acknowledged by the true receiver. *Kill* requests are handled by generating packets with the RST flag set and sending one to each end-point. Blocked *read* requests for `contain`ing stream scanners are a bit tricky. 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.

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. We plan to adopt one of the well-established cryptography-based protocols [2, 4, 6, 14, 19, 22] for remotely distributing policy updates and receiving alerts.

3.3 Discussion

Our prototype is still young, with the main goal of allowing us to experiment with NI-embedded scanners. Although it is too early to draw definitive conclusions, we believe that its software architecture is valuable. Our experiences indicate that the scanner API makes writing scanners relatively straightforward, though it could be made more so with Bro-like language support [28] at the scanner level. More importantly, 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 in their entirety. Also, a scanner cannot *cut* portions of *inject*ed data.) In theory, the ability

Configuration	Roundtrip	Bandwidth
No NI machine	0.16 ms	11.11 MB/s
No scanners	0.23 ms	11.11 MB/s
Frame scanner	0.23 ms	11.08 MB/s
Stream scanner	0.23 ms	10.69 MB/s

Table 2: **Base performance of the self-securing NI prototype.** 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. "Frame scanner" corresponds to copying all IP packets to a read-only scanner. "Stream scanner" corresponds to reconstructing the TCP stream for a read-only scanner.

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.

3.4 Basic overheads

Although performance is not our focus, it is useful to quantify Siphon's effect on NI throughput and latency. As found by previous researchers [14, 16, 28], we observe that NIDS and normalization functions can be made reasonably efficient for individual network links. Also, the internal protection boundary between scanners and the trusted base comes with a reasonable cost.

For all experiments in this paper, the NI machine 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 NIC or switch. The host machine runs SuSe Linux 2.4.7 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 four configurations: the host machine alone (with no NI machine), the NI machine with no scanners, the NI machine with a read-only frame-level scanner matching every packet, and the NI machine reconstructing all TCP streams for a read-only scanner. We observe a 47% increase in round-trip latency with the insertion of the NI machine into the host's path, but no additional increase with scanners. We observe minimal bandwidth difference among the four configurations, although reconstructing the TCP stream results in a 4% reduction.

4 Example detectors

This section describes and explores four examples of detectors that work particularly well with self-securing NIs. Each exploits the NI's proximity to the host and the corresponding ability to see exactly what it sends and receives. For each, we describe the attack, the scanner, relevant performance data, and associated issues.

4.1 Detecting IP-based propagation

A highly-visible network attack in 2001 was the Code-Red worm (and its follow-ons) that propagated rapidly once started, hitting most susceptable machines in the Internet in less than a day [26].

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. But, it exhibits an abnormal communication pattern. 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) [24]. 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, 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).[4] 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 destination 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 watched and contained in this way, the 14 hour propagation time of Code-Red (version 2) [26] would have grown to over a month (assuming the original scan rate was 10 per second per infected machine [35]).

[4]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.

Performance data: As expected, given the earlier roundtrip latency evaluation, the DNS scanner adds negligible latency to DNS translations and TCP connection establishment. 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.

It is interesting to consider the table size required for an aggregate table kept at an edge router. 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 [20], 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: We have not observed false positives in small-scale testing (a few hours) in front of a user desktop, though more experience is needed. The largest false positive 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 [36, 32, 42], 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[5] 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. So goes the escalation "game" of security.

An alternate containment strategy, blindly restricting the rate of connections to new destinations, has recently been proposed [40]. The proposed implementation (extending host-based firewall code) would not work in practice, since most worms would be able to disable it. But, a self-securing NI could use this approach, if further study revealed that it really would not impede legitimate work. Note that such rate throttling at the intranet edge may not be effective, because techniques like local subnet scanning [35] would allow a worm to parallelize external targetting.

Finally, it is worth noting that the Code-Red worms exploited a particular buffer overflow that was well-known ahead of time. A 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

[5]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.

the Nimda worm, that use random IP-based propagation but other security holes. Coincidentally, early information about the "SQL Slammer" worm [7] indicates that it would be caught by this same scanner.

4.2 Detecting claim-and-hold DoS attacks

Qie et al. [31] partition DoS attacks into two categories: busy attacks (e.g., overloading network links) and claim-and-hold attacks. In the latter, the attacker causes the victim to allocate a limited resource for an extended period of time. Examples include filling IP fragment tables (by sending many "first IP fragment" frames), filling TCP connection tables (via "SYN bombing"), and exhausting server connection limits (via very slow TCP communication [31]). A host doing such things can be identified by its self-securing NI, which sees what enters and leaves the host when. As a concrete example, this section describes a scanner for SYN bomb attacks.

What the scanner looks for: A SYN bomb attack exploits a characteristic of the state transitions within the TCP protocol [29] to prevent new connections to the victim. 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 (with the SYN flag set), ignoring the victim's correct response (a second packet with the SYN and ACK flags set). The scanner watches for instances of inbound SYN/ACK packets not receiving timely responses from the host. A well-behaved host should respond to a SYN/ACK with either an ACK packet (to complete the connection) or a RST packet (to terminate an undesired connection).

How the scanner works: The scanner watches all inbound SYN/ACK packets and all outbound ACK and RST packets. It works by maintaining a table of all SYN/ACKs destined to the host that have not yet been answered. Whenever a new SYN/ACK arrives, it is added to the 'waiting for reply' table with an associated timestamp and expiration time. Retransmitted SYN/ACKs do not change these values. If a corresponding RST packet is sent by the host, the entry is removed. If a corresponding ACK packet is sent, the entry is moved to a 'reply sent' cache, whose role is to identify retransmissions of answered SYN/ACK packets, which may not require responses; entries are kept in this cache until the connection closes or 240 seconds (the official TCP maximum roundtrip time) passes.

If no answer is received by the expiration time, then the scanner considers this to be an ignored SYN/ACK. Currently, the expiration time is hard-coded at 3 seconds. The current policy flags a problem if there are more than 2 ignored SYN/ACKs in a one minute period.

When an attack is detected: The SYN bomb scanner's current policy reacts to potential attacks only by sending an alert to the administrative system. Other possible responses include delaying or preventing future SYN packets to the observed victim (or all targets) or having Siphon forge RST packets to the host and its victim for the incomplete connection (thereby clearing the held connection state).

Performance data: The SYN bomb scanner maintains a histogram of the observed response latency of its host to SYN/ACK packets. Under a moderate network load, over a one hour period of time, a desktop host replied to SYN/ACKs in an average of 26 milliseconds, with the minimum being under 1 and the maximum being 946 milliseconds. Such data indicates that our current grace period of 3 seconds should result in few false positives.

Discussion: There are two variants of the SYN bomb attack, both of which can be handled by self-securing NIs on the attacking machine. In one variant, the attacker uses its true address in the source fields, and the victim's responses go to the attacker but are ignored. This is the variant targetted by this scanner. In the second variant, the attacker forges false entries in the SYN packets' source fields, so that the victim's replies go to other machines. A self-securing NI on the attacker machine can prevent such spoofing.

4.3 Detecting TTL misuse

Crafty attack tools can hide from NIDSs in a variety of ways. Among them are insertion attacks [30] based on misuse of the IP TTL field, which determines how many routers a packet may traverse before being dropped.[6] By sending packets with carefully chosen TTL values, an attacker can make a NIDS believe a given packet will reach the destination while knowing that it won't. As a concrete example, the SYN bomb scanner described above is vulnerable to such deception (ACKs could be sent with small TTL values). This section describes a scanner that detects attempts to misuse IP TTL values in this manner.

What the scanner looks for: The scanner looks for unexpected variation in the TTL values of IP packets originating from the host. Specifically, it looks for differing TTL values among packets of a single TCP session. Although TTL values may vary among inbound packets, because different packets may legitimately traverse different paths, such variation should not occur within a session.

How the scanner works: The scanner examines the TTL value for TCP packets originating from a host. The TTL value of the initial SYN packet (for outbound connections) or SYN/ACK packet (for inbound connections) is recorded in a table until the host side of the connection moves to the closed state. The TTL value of each subsequent packet for that connection is compared to the original. Any difference is flagged as TTL misuse, unless it is a RST with TTL=255 (the maximum value). Both Linux and NetBSD use the maximum TTL value for RST packets, presumably to maximize their chance of reaching the destination.

When an attack is detected: The current scanner's policy involves two things. The TTL fields are normalized to the original value, and an alert is generated.

Performance data: We applied the TTL scanner to the traffic of a Linux desktop engaged in typical network usage for over an hour. We observed only 2 different TTL values in packets originating from the desktop: 98.5% of the packets had a TTL of 64 and the remainder had a TTL of 255. All of the TCP packets were among those with TTL of 64, with one exception: a RST packet with TTL=255. The other packets with TTL of 255 were ICMP and other non-TCP traffic.

Discussion: This scanner's detection works well for detecting most NIDS insertion attacks in TCP streams, since there is no vagueness regarding network topology between a host and its NI. It can be extended in several ways. First, it should check for low initial TTL values, which might indicate a non-deterministic insertion attack given some routes being short enough and some not; detecting departure from observed system default values (e.g., 64 and 255) should be sufficient. Second, it should check TTL values for non-TCP packets. This will again rely on observed defaults, with one caveat: tools like traceroute legitimately use low and varying TTL values on non-TCP packets. An augmented scanner would have to understand the pattern exhibited by such tools in order to restrict the non-flagged TTL variation patterns.

[6]This should not be confused with the DNS TTL field used in the Code-Red scanner.

4.4 Detecting IP fragmentation misuse

IP fragmentation can be abused for a variety of attacks. Given known bugs in target machines or NIDSs, IP fragmentation can be used to crash systems or avoid detection; tools like fragrouter [33] exist for testing or exploiting IP fragmentation corner cases. Similarly, different interpretations of overlapping fragments can be exploited to avoid detection. As well, incomplete fragment sets can be used as a capture-and-hold DoS attack.

What the scanner looks for: The scanner looks for five suspicious uses of IP fragmentation. First, overlapping IP fragments are not legitimate—a bug in the host software may cause overlapping, but should not have different data in the overlapping regions—so, the scanner looks for differing data in overlapping regions. Second, incomplete fragmented packets can only cause problems for the receiver, so the scanner looks for them. Third, fragments of a given IP packet should all have the same TTL value. Fourth, only a last fragment should ever be smaller than the minimum legal MTU of 68 bytes [18]; many NIDS evasion attacks violate this rule to hide TCP, UDP, or application frame headers from NIDSs that do not reconstitute fragmented packets. Fifth, IP fragmentation of TCP streams is suspicious. This last item is the least certain, but most TCP connections negotiate a "maximum segment size" (`mss`) during setup and modern TCP implementations will also adjust their `mss` field when an ICMP "fragmentation required" message is received.

How the scanner works: The scanner *subscribe*s (with `contain` rights) for all outbound IP packets that have either the "More Fragments" bit set or a non-zero value for the IP fragment offset. These two subscriptions capture all Ethernet frames that are part of fragmented IP packets. The first sequential fragmented packet has the "More Fragments" bit set and a zero offset. Fragments in between have the "More Fragments" bit set and a non-zero offset. The last fragment doesn't have the "More Fragments" bit set but it does have a non-zero offset.

The scanner tracks all pending fragments. Each received fragment is compared to held fragments to determine if it completes a full IP packet. If not, it is added to the cache. When all fragments for a packet are received at the NI, the scanner determines whether the IP fragmentation is acceptible. If the full packet is part of a TCP stream, it is flagged. If the fragments have different TTL values, it is flagged. If any fragment other than the last is smaller than 64 bytes, it is flagged. If the fragments overlap and the overlapping ranges contain different data, it is flagged. If nothing is flagged, the fragments are `pass`ed in ascending order.

Periodically, the fragment structure is checked to determine if an incomplete packet has been held for more than a timeout value (currently one second). If so, the pieces are `cut`. If more than two such timeouts occur in a second or ten in a minute, the host's actions are flagged.

When an attack is detected: There are five cases flagged, all of which result in an alert being generated. In addition, we have the following policies in place: overlapping fragments with mismatching data are dropped, under the assumption that either the host OS is buggy or one of the constructions is an attack; fragments with mismatching TTL fields are sent with all TTLs matching the highest value; incorrectly fragmented packets are dropped; timed out fragments are dropped (as described); fragmented TCP packets are currently passed (if the other rules are not violated).

Performance data: We ran the scanner against a desktop machine, but observed no IP fragmentation during normal operation. With test utilities sending 64KB UDP packets (over Ethernet), we measured the time delay between the first frame's arrival at the NI and the last. The average time before all fragments are received was 0.53ms, with values ranging from 0.46ms to 2.5ms.

These values indicate that our timeout period may be too generous.

Discussion: Flagging IP fragmentation of TCP streams is only reasonable for operating systems with modern networking stacks, which can be known by an administrator setting policies. Older systems may actually employ IP fragmentation rather than aggressive `mss` maintenance. Because of this and the possibility of fragmentation by intermediate routers, a rule like this would not be appropriate for a non-host-specific NIDS.

Our original IP fragmentation scanner also watched for out-of-order IP fragments, since this is another possible source of reconstitution bugs. In testing, however, we discovered that at least one OS (Linux) regularly sends its fragments in reverse order. The NI software, therefore, always waits until all fragments are sent and then propagates them in order.

We originally planned to detect unreasonable usage of fragmentation and undersized fragments by caching the MTU values observed (in ICMP "fragmentation required" messages) for various destinations. We encountered several difficulties. First, it was unclear how long to retain the values, since any replacement might cause a false alarm. Second, an external attacker could fill the MTU cache with generated messages, creating state management difficulties. Third, a conspiring external machine with the ability to spoof packets could easily generate the ICMP packets needed to fool the scanner. Since IP fragmentation is legal, we decided to focus on clear misuses of it.

As with most of the scanners described, the IP fragmentation scanner is susceptible to space exhaustion by the host. Specifically, a host could send large numbers of incomplete fragmented packets, filling the NIs buffer capacity. As noted earlier, however, such an attack mainly damages the host itself, denying it access to the network. This seems an acceptible trade-off given the machine's misbehavior. A similar analysis exists for the other scanners.

4.5 Other scanners

Of course, many other scanners are possible. Any traditional NIDS scanning algorithm fits, both inbound and outbound, and can be expected to work better (as described in [16, 21]) after the normalization of IP and TCP done by Siphon. For example, we have built several scanners for e-mail (virus scanning) and Web (buffer overflows, cookie poisoning, virus scanning) connections. As well, NIC-embedded prevention/detection of basic spoofing (e.g., of IP addresses) and sniffing (e.g., by listening with the NI in "promiscuous mode") are appropriate, as is done in 3Com's Embedded Firewall product [1].

Several other examples of evasion and protocol abuse can be detected as well. For example, misbehaving hosts can increase the rate at which senders transmit data to them by sending early or partial ACKs [34]; sitting on the NI, a scanner could easily see such misbehavior. A TCP abuse of more concern is the use of overlapping TCP segments with different data, much like the overlapping IP fragment example above; usable for NIDS insertion attacks [30], such behavior is easily detected by a scanner looking for it.

Finally, we believe that the less aggregated and local view of traffic exhibited at the NI will help with more complex detection schemes, such as those for stepping stones [12, 41] or general anomaly detection of network traffic. This is an area for future study.

5 Related Work

Self-securing NIs build on much existing technology and borrow ideas from previous work, as discussed throughout the flow of this paper. Network intrusion detection, virus detection, and firewalls are well-established, commonly-used mechanisms [5, 10]. Also, many of the arguments for distributing firewall functions [14, 19, 27] and embedding them into network interface cards [1, 14] have been made in previous work. Notably, the 3Com Embedded Firewall product [1] extends NICs with firewall policies such as IP spoofing prevention, promiscuous mode prevention, and selective filtering of packets based on fields like IP address and port number. This and other previous work [2, 6, 22] also address the issue of remote policy configuration for such systems. These previous systems do not focus on host compromise detection and containment like self-securing NIs do. This paper extends previous work with examples of more detailed analysis of a host's traffic enabled by the location of NI-embedded NIDS functionality.

Many network intrusion detection systems exist. One well-described example is Bro [28], an extensible, real-time, passive network monitor. Bro provides a scripting language for reacting to pre-programmed network events. Our prototype's support for writing scanners could be improved by borrowing from Bro (and others). Embedding NIDS functionality into NIs instead of network taps creates the scanner containment issue but eliminates several of the challenges described by Paxson, such as overload attacks, cold starts, dropped packets, and crash attacks. Such embedding also addresses many of the NIDS attacks described by Ptacek and Newsham [30].

There is much ongoing research into addressing Distributed DoS (DDoS) attacks. Most counter-measures start from the victim, using traceback and throttling to get as close to sources as possible. The D-WARD system [23] instead attempts to detect outgoing attacks at source routers, using anomaly detection on traffic flows, and throttle them closer to home. The arguments for this approach bear similarity to those for self-securing NIs, though they focus on a different threat: outgoing DDoS attacks rather than two-stage attacks. The ideas are complementary, and pushing D-WARD all the way to the true sources (individual NIs) is an idea worth exploring.

A substantial body of research has examined the execution of application functionality by network cards [13, 17] and infrastructure components [3, 11, 38, 39]. 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.

6 Summary

Self-securing network interfaces are a promising addition to the network security arsenal. This paper describes their use for identifying and containing compromised hosts within the boundaries of managed network environments. It illustrates the potential of self-securing NIs with a prototype NI kernel and example scanners that address several high-profile network security problems: insertion and evasion efforts, state-holding DoS attacks, and Code-Red style worms.

Acknowledgments

We thank Mike Reiter for invaluable discussions about this work. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, 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.[7]

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] 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.

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

[6] 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.

[7] CERT. CERT Advisory CA-2003-04 MS-SQL Server Worm, January 25, 2003. http://www.cert.org/advisories/CA-2003-04.html.

[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.

[7]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.

[11] 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.

[12] David L. Donoho, Ana Georgina Flesia, Umesh Shankar, Vern Paxson, Jason Coit, and Stuart Staniford. Multiscale stepping-stone detection: detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. *RAID* (Zurich, Switzerland, 16–18 October 2002), 2002.

[13] 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.

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

[15] Gregory R. Ganger, Gregg Economou, and Stanley M. Bielski. *Self-securing network interfaces: what, why and how*. CMU-CS 02-144. August 2002.

[16] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. *USENIX Security Symposium* (Washington, DC, 13–17 August 2001), pages 115–131. USENIX Association, 2001.

[17] 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.

[18] Information Sciences Institute, USC. RFC 791 - DARPA Internet program protocol specification, September 1981.

[19] 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.

[20] 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.

[21] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. *IEEE INFOCOM* (Tel Aviv, Israel, 26–30 March 2000), pages 1381–1390. IEEE, 2000.

[22] 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.

[23] Jelena Mirkovic, Peter Reiher, and Greg Prier. Attacking DDoS at the source. *ICNP* (Paris, France, 12–15 November 2002), pages 312–321. IEEE, 2002.

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] Thomas H. Ptacek and Timothy N. Newsham. *Insertion, evasion, and denial of service: eluding network intrusion detection.* Technical report. Secure Networks Inc., January 1998.

[31] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: using an annotation toolkit to build dos-resistant software. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002), pages 45–60. USENIX Association, 2002.

[32] 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.

[33] SANS Institute. IP Fragmentation and Fragrouter, December 10, 2000. http://rr.sans.org/encryption/IP_frag.php.

[34] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM Computer Communications Review*, **29**(5):71–78. ACM, October 1999.

[35] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. *USENIX Security Symposium* (San Francisco, CA, 5–9 August 2001), 2002.

[36] 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.

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

[38] 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.

[39] 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.

[40] Matthew M. Williamson. *Throttling viruses: restricting propagation to defeat malicious mobile code*. HPL 2002-172R1. HP Labs, December 2002.

[41] Yin Zhang and Vern Paxson. Detecting stepping stones. *USENIX Security Symposium* (Denver, CO, 14–17 August 2000). USENIX Association, 2000.

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