Design and Implementation of Self-Securing Network Interface Applications

Stanley M. Bielski

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania
December 2005

Design and Implementation of Self-Securing Network Interface Applications

Approved by:

____

Professor Gregory R. Ganger

____

Professor Dawn Song

Date Approved ______________________

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Gregory R. Ganger for his kindness and mentorship throughout my stay at Carnegie Mellon. Greg's tireless work ethic and inexhaustible enthusiasm have been a continual source of inspiration for me and have helped shape me as a researcher and a person. Without his tremendous faith, support, and tutelage this work would never have been possible.

I would like to thank the staff, students and faculty of the Parallel Data Lab: you all have contributed to making this environment a great place to work and learn. Specifically, I would like to thank William Courtright for helping me pursue my degree. I would also like to thank Karen Lindenfelser for all the work she does behind the scenes; I can't imagine the PDL without Karen! This thesis also wouldn't have been possible without Gregg Economou's work on the Siphon kernel and Chris Long's work on Castellan: it was a pleasure working with both of you. I would also like to thank Dawn Song for volunteering to be the second reader of this thesis. I would also like to thank the members and companies of the PDL Consortium (including American Power Conversion, EMC, EqualLogic, Hewlett-Packard Labs, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate and Sun) for their interest, insights, feedback, and support.

Lastly, I would like to thank my mother, my father, and my brother: I love yinz guyz and trying to make you proud brings out the best in me. Thank you all!

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **iii**

I INTRODUCTION **1**

II TOWARDS PER-MACHINE PERIMETERS **4**

2.1 Threat model 4

2.2 Self-Securing NIs 5

2.3 Self-securing NI features 7

2.4 Self-securing NI Administrative Features 8

2.5 Costs, limitations, and weaknesses 9

III SELF-SECURING NI SOFTWARE DESIGN **12**

3.1 Goals 12

3.2 Basic design achieving these goals 14

3.3 Discussion 18

IV IMPLEMENTATION **19**

4.1 A self-securing NI prototype 19

4.2 Prototype implementation 19

4.3 Basic overheads 25

V EXAMPLE APPLICATIONS **26**

5.1 Detecting IP-based propagation 26

5.2 Detecting claim-and-hold DoS attacks 29

5.3 Detecting TTL misuse 31

5.4 Detecting IP fragmentation misuse 32

5.5 Other scanners 34

VI RELATED WORK **36**

VII CONCLUSION **38**

LIST OF FIGURES

1 Self-securing network interfaces. 5

2 Self-securing NI software architecture 15

3 Example of a `siphon_conf` 22

4 Example of a `scanner_conf` 22

LIST OF TABLES

1 Network API exported to scanner applications. 16

2 Base performance of the self-securing NI prototype. 25

CHAPTER I

INTRODUCTION

As the Internet grows, and as Internet applications become more ubiquitous and complex, networks intrusions will similarly increase in scale, frequency, and sophistication. The task of properly securing a network against this evolving threat requires a jumble of disparate technological solutions (e.g. firewalls, proxies, network intrusion detecton systems, anti-virus software) and administrative efforts (e.g. software patching, policy enforcement, network monitoring). Unfortunately, even a network with a state-of-the-art security infrastructure and a team of expert administrators is still vulnerable to attacks that exploit the lack of cohesion and adaptability inherent in this security model. Since centralized security infrastructure is often shared across a set of many networked hosts, it often lacks the resources and multiple vantage points needed to diagnose and contain threats that fly under the radar of traditional security models. For example, traditional network security leaves the protected intranet unprotected from an intruder who gets past the outer defenses. Moreover, often there is no centralized infrastructure for information-sharing and coordination among security components; the administrator must manually correlate separate log files and independently configure separate components. These time-consuming and error-prone tasks increase the time required to respond to a security incident, opening the window for additional attacks.

In response to these concerns, this thesis presents a novel security platform that narrows the architectural gaps between traditional network security perimeters in a highly scalable and fault-isolated manner while providing administrators with a simple and powerful interface for configuration and coordination of security policies across multiple network components. The heart of this platform is the concept of *self-securing network interfaces* (SS-NIs) [15, 16].

Network interfaces (NIs) are components that sit between a host system and the rest of the intranet, such as network interface cards (NIC) or local switch ports. 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.

Self-securing NIs have 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 those on the WAN. Second, a misbehaving host can be throttled at its source. As with firewalls, a self-securing NI operates independently of host software; its checks and rules remain in effect even when the corresponding host OS is compromised. 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. 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.

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 (which we refer to as *scanners*), particularly code that examines the application-level exchanges. As a result, well-designed system software is needed for self-securing NIs, both to simplify scanner implementations and to contain rogue scanners (whether buggy or compromised). This thesis describes a software architecture that addresses both issues.

One perceived drawback of using self-securing NI's is the additional administrative effort required to manage and configure them. We address this concern by describing our management infrastructure called *Castellan*. Castellan provides administrators with a GUI-based tool to easily perform most administrative tasks, such as installing and uninstalling SS-NI software, updating and managing SS-NI policy configurations, and viewing feedback from SS-NI's in the form of *alerts*. Additionally, Castellan has the capability to autonomically coordinate feedback among different devices and alter the network's security policy based on this feedback. By minimizing response time required for containment, Castellan can reduce windows of vulnerability in the network.

This thesis makes four main contributions: First, it makes a case for NI-embedded intrusion detection and containment functionality. Second, it describes the design of NI system software for supporting such functionality. Third, it discusses our implementation of NI system software and the Castellan administrative console. Fourth, it describes several promising applications for detecting

and containing network threats enabled by the placement of self-securing NIs at the host's LAN access point.

The remainder of this thesis is organized as follows. Section 2 discusses the concept of SS-NIs in further detail and describes related work. Section 3 describes the design and architecture of SS-NIs and Castellan. Section 4 discusses our prototype implementation of SS-NI system software and management software. Section 5 describes applications we have developed on the SS-NI platform. Section 6 discusses related work. Section 7 concludes this thesis by reflecting on our experiences and identifying areas of future work.

CHAPTER II

TOWARDS PER-MACHINE PERIMETERS

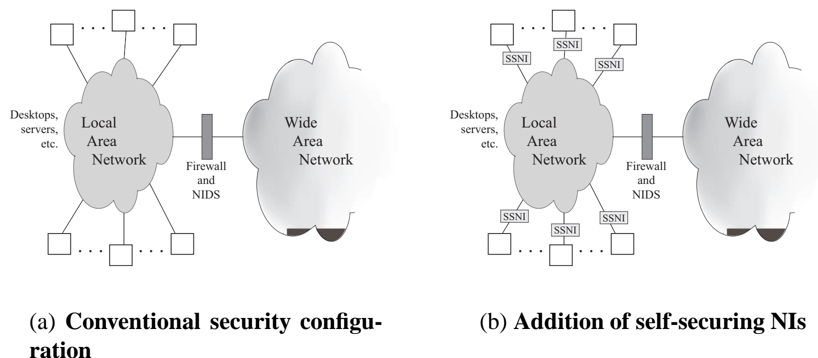
The common network security approach maintains an outer perimeter (perhaps with a firewall, some proxies, and a NIDS) around a protected intranet. This is a good first line of defense against network attacks. But, it leaves the entire intranet wide open to an attacker who gains control of any machine 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 that this thesis is most concerned with is a multi-stage attack. In the first stage, the attacker compromises any host on the inside of the intranet perimeter – 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 on the intranet or Internet.

This form of two-stage attack is of concern because only the first stage need infiltrate 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. Finding a single hole is unlikely to be difficult in any sizable organization. Once internal access is gained, the second stage is free to use known, NIDS-detectable system vulnerabilities, since it does not enter the view of the perimeter defenses. In some environments, known attacks launched out of the organization may also proceed unencumbered; this depends on whether the NIDS and firewall policies are equally restrictive in both directions.

The main focus of this work is on stopping 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. This thesis is not



(a) **Conventional security configuration**

(b) **Addition of self-securing NIs**

(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.

Figure 1: Self-securing network interfaces.

specifically trying to address insider attacks, in which the attacker would also have physical access to the hardware and its network connections.

2.2 Self-Securing NIs

A countermeasure to the 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.[1]

A host's network interfaces (NIs) have the two properties described above, so this work focuses on these components as good places to add a security perimeter. For the purposes of this thesis, components labeled as "NIs" will have three properties: (1) they will perform the base packet moving function, (2) they will do so from behind a simple interface with few additional services, and (3) they 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]. Leaf switches also have these properties for the hosts directly connected to them.

[1]Many of the schemes explored here could be also used at the edge to detect second-stage attacks from inside to out. The 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.

An NI is defined to be a *self-securing NI* (SS-NI) if it internally monitors and enforces policies on packets forwarded in each direction. With proper policies in place, a SS-NI can be an effective countermeasure to the two-stage attack scenario. Since a SS-NI operates independently of host software, its policies continue to be enforced even while the host OS is compromised. Through its ability to control host traffic at its source, a SS-NI can contain misbehaving hosts and mitigate the threat they pose to other hosts on the network. Moreover, 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 requires reconstruction within the SS-NI software. Although this work is redundant with respect to the host's network stack, it allows SS-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 [9, 10], 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, much like distributed firewalls [14, 20, 1]. Configuration and management of SS-NIs is performed over a secure channel in the network. Alerts about suspicious activity are sent to administrative systems via the same secure channels. These administrative systems can log alerts, construct an aggregate view of individual NI observations, notify administrators, and even distribute new policies in response to observations if so configured. In order to properly guard against the two-stage attack model, even the host OS and its most-privileged users must not be able to reconfigure or disable the NI's policies. Of course, for a SS-NI to be effective, one must assume that neither the administrative console nor the NI itself are compromised.

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.

2.3 Self-securing NI features

The SS-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 [14]. 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 is relatively low for common desktop network links, [14], particularly when their utilization is considered. More importantly, the total available resources for examining traffic necessarily scales with the number of nodes in the network, since each node should be connected to the network by its own SS-NI. Distributing such functionality among end-points avoids the bottleneck inherent in a centralized firewall configuration. Moreover, the cost of equivalent aggregate resources in a such a configuration makes them expensive (in throughput or dollars) or limits 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 SS-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, SS-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, SS-NIs operate independently of vulnerable host software. As such, their policies are exclusively configured by an adminstrative console over the network. So, even compromising the host OS will not allow an intruder to disable the SS-NI functionality. This property is important because 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, hosts on the LAN or WAN can be protected

from a misbehaving host operating behind a SS-NI. For example, a host whose security status is questionable could have its network access slowed, filtered, or blocked entirely. This feature distinguishes SS-NIs from distributed firewalls, which protect against internal machines by protecting targets rather than containing sources. While for some attack scenarios this strategy may be sufficient, the extra visibility and control offered at the NI vantage point of a compromised host can be exploited to reduce detection assumptions. For example, a SS-NI can see misuse of low-level networking features, as is the case in many deception attacks on NIDSs, and be used to normalize this problematic traffic. Doing this at the source leverages the effort involved in detecting NIDS deception attacks, since both SS-NIs and NIDSs require similar state tracking.

Less aggregation. Connected to only one host system, a SS-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 SS-NI to more effectively notice strange network behavior. Additionally, the NI's position in a single host's communication path means that it can fail closed without adversely affecting the communications of its intranet neighbors. 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 misbehaving host using this tactic on its self-securing NI is denying service only to itself.

More per-link resources. Because each SS-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 to draw out explicitly. Also, note that not all traffic must be examined in depth; for example, a SS-NI could decide to examine e-mail and web traffic in depth while allowing NFS and Quake traffic to pass immediately.

2.4 Self-securing NI Administrative Features

Using the NI as a security perimeter can also be a boon for administrators, allowing the implementation of powerful and desirable network management features. Since NIs require no changes to host software, SS-NIs provide administrators with a uniform platform for managing per-host

network security policies (such as firewalling and network intrusion detection rules) across a heterogeneous set of host operating systems. This is feature is particularly valuable for protecting or containing legacy hosts running deprecated operating systems that do not support the software needed to enforce desired security policies.

An additional benefit is the ability to rapidly deploy bandaids for known vulnerabilities. The holes exploited by many network attacks are known and publicized well in advance of the actual attacks. For example, the infamous "Internet worm" of 1988 [36] exploited a known buffer overflow weakness (in the `fingerd` application), and the more 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 be deployed and administrators remained vulnerable until patches were constructed and installed. By creating SS-NI policies to look for network service requests that would trigger such overflows, one can prevent them from reaching the system until patches are provided. We think a SS-NI is an ideal place for such bandaids, since it requires no change to the host software, but a host-based mechanism would also work.

SS-NIs also give administrators additional vantage points on the network. This allows for a more complete view of traffic circulating in the intranet than does traditional infrastructure such as firewalls or NIDSs. The ability to correlate observations from multiple SS-NIs could prove to be an invaluable resource for a number of administrative tasks. For instance, if observations suggest a network host is acting suspicious, flexible policy management across SS-NIs would allow an administrator to dynamically deploy more fine-grained logging to capture that host's traffic patterns. This data could be useful in diagnosing the cause of the anomolous behavior. Moreover, if the host is actually compromised, logs from SS-NIs can be used in combination with logs from HIDSs or NIDSs as the basis for a forensic analysis of the security breach.

2.5 Costs, limitations, and weaknesses

SS-NIs are promising, but there is no silver bullet for network security. SS-NIs 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 SS-NI is subject to a number of attacks [31]. Most insertion attacks are either detectable signals (when from the host) and/or subject to normalization [17, 22]. 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 SS-NIs. The extra work should be small, given appropriate administrative tools, but again will be non-zero. Third, like any in-network mechanism, a SS-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 SS-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 SS-NIs. But, more global patterns are an example of why SS-NIs should be viewed as complementary to edge-located protections; it should

be stressed that SS-NIs are meant to augment existing network security infrastructure, not replace it. 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 SS-NI functionality.

CHAPTER III

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, flexibile support for wide-variety of 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 SS-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 SS-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 SS-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. The 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. One anticipates 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 worst 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 design goal is for SS-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 goal 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

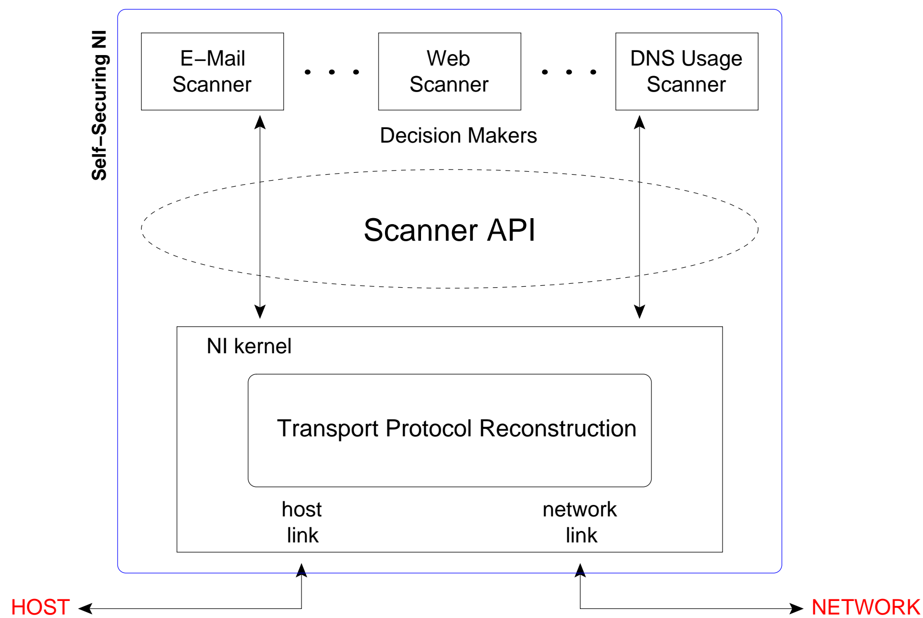
resources as necessary to make their decisions — administrators can choose to employ such scanning code (or not) based on the associated trade-off between cost and security.

Reduced managemenent burden. If SS-NIs are to be embraced by network administrators, their management interface should have a minimal learning curve while being sophisticated enough to provide powerful features. For this purpose, a streamlined graphical user interace (GUI) that hides low-level management interactions is highly desirable. Network administrators should be able to painlessly install, remove, or update scanning code. Moreover, administrators should be able to easily define policies to contain compromised or malfunctioning scanning code, such as setting allowances for resource consumption or traffic permissions. Alerts produced by scanning code should be written in a well-understood and uniform format, and these alerts should be stored in a centralized repository to facilitate their perusal and correlation. One anticipates the need for rapid response capabilities (corresponding to the current level of paranoia, for instance), so administrators should be able to introduce bulk policy changes or updates with a single command. Well-designed management applications not only reduce the burden of administration, they also increase security by enabling an administrator to quickly anticipate, detect, and contain breaches.

3.2 Basic design achieving these goals

This section describes a system software architecture for SS-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. An additional application process, not visible in Figure 2, is the scanner manager. The scanner manager is responsible for launching scanners, monitoring them for failure or misbehavior, and enforcing their resource restrictions through interaction with the kernel. The administrative console communicates with the scanner manager over a secure channel to push policy changes and receive alerts from scanner applications.

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



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.

Figure 2: Self-securing NI software architecture

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 an 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 to 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

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

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.

Table 1: Network API exported to scanner applications.

with a packet filter language [26]. The NI kernel grants access only if the administrator's configuration for the particular scanner allows it. [1] 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.

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

[1] In practice, a better place for enforcing subscription access control may be in a separate, trusted application process, such as the scanner manager. In this model, a scanner asks the scanner manager for a subscription, which then forwards only permissible requests to the NI kernel. This is the approach used in the prototype and is further described in 4.

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 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.

Scanner Manager. The scanner manager is a trusted application process responsible for the coordination and monitoring of scanners and interfacing with the administrative console. Upon the instruction of the administrative console, the scanner manager will launch a set of scanners outlined by a configuration file provided by the administrator [2] Each scanner is associated with its own

[2]Thus, it as powerful as the NI kernel separated mainly for ease of programming.

configuration file that outlines its resource restrictions; the scanner manager interacts with the NI kernel to ensure that they resource restrictions are enforced. During the scanner operations, the scanner manager receives alerts from scanner applications and forwards them to the administrative console over a secure interface. It also monitors scanner applications and creates and sends its own alerts in cases of scanner failure or misbehavior.

3.3 Discussion

Implemented properly, this design should meet the design goals for SS-NIs. 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, this should not 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, the NI software architecture described is a significant improvement over unbounded access.

CHAPTER IV

IMPLEMENTATION

4.1 A self-securing NI prototype

This section describes our prototype self-securing NI. The prototype self-securing NI is actually a separate computer (referred to below as the "NI machine") with three Ethernet interfaces, one connected to the real network, one connected point-to-point to the host machine's Ethernet link, and one connected to the administrative subnet. [1] Clearly, the prototype hardware characteristics differ from real NIC or switch hardware, but it does allow us to explore the SS-NI features that are the focus in this work.

4.2 Prototype implementation

The SS-NI software runs on the NetBSD 1.6 operating system. Both network interfaces 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. The NI kernel, called *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.

[1] Alternatively, administration of self-securing NIs could occur over the protected host's LAN. The choice to use a separate subnet was made because it was expedient to do so.

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.

Scanner Manager. The Scanner Manager is implemented as an application process run as the root user. This process is launched after boot through the `/etc/rc.local` script and performs operations necessary to enable the Siphon and launch scanner applications. Upon start-up, the Scanner Manager queries the NI's resident DBMS (currently PostgreSQL 7.3.3) for the entry in the `running_siphon_conf` table. This entry corresponds to a binary large object (BLOB) holder containing an XML configuration file, which we refer to as a `siphon_conf`. An example `siphon_conf` is shown in Figure 3.

The Scanner Manager then extracts this `siphon_conf` from the database and parses it. This file contains information such as the host's IP and MAC addresses and the initial set of scanner applications to be launched. The Scanner Manager then opens the siphon psuedo-device and issues `ioctl` commands on it to register the host's MAC address with the Siphon kernel and initialize its operation.

With the Siphon now running, the Scanner Manager proceeds to the task of launching scanner applications. Each scanner listed in the `siphon_conf` corresponds with an identifier for how that instance of the scanner should be configured. This configuration corresponds to two XML files stored as BLOBs in the database. The first is a `scanner_conf`, which contains a list of rights and restrictions for the execution of a particular scanner. The second is a `scanner_subconf`. This configuration file is used to pass arguments to the scanner application itself. Examples of a

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE siphon-conf SYSTEM "siphon.dtd">
<siphon-conf conf-name = "highly paranoid">
   <watched-host
         <host-ip-address>128.2.134.97</host-ip-address>
         <host-mac-address>00065BDCC62D</host-mac-address>
   </watched-host>
    <scanner name="filter-scanner" version="1.0" conf-name="port-firewall" priority="1"/>
    <scanner name="limiter-scanner" version="1.0" conf-name="paranoid-ssh" priority="2"/>
    <scanner name="limiter-scanner" version="1.0" conf-name="paranoid-ftp" priority="3"/>
</siphon-conf>
```

This figure presents an example of a `siphon_conf`. For this particular configuration, the siphon loads three scanners: a scanner to firewall certain ports, a throttling scanner for FTP connections, and a throttling scanner for SSH connections.

Figure 3: Example of a `siphon_conf`

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE scanner-conf SYSTEM "siphon.dtd">
<scanner-conf name="limiter-scanner" version="1.0" conf-name="paranoid-ssh">
        <scanner-conf-desc>Paranoid rate limiter for SSH traffic</scanner-conf-desc>
        <scanner-permissions>
                <scanner-permission>
                   <pcap-string>dst 22 and (tcp[tcpflags] &amp; tcp-syn!= 0)</pcap-string>
                   <subscription-type>packet</subscription-type>
                   <subscription-permissions>read-write</subscription-permissions>
                </scanner-permission>
        </scanner-permissions>
        <scanner-resources>
                <scanner-resource>
                    <resource-name>RLIMIT_AS</resource-name>
                    <resource-value>512000</resource-value>
                </scanner-resource>
        </scanner-resources>
</scanner-conf>
```

This figure presents an example of a `scanner_conf`. This particular scanner has `contain` permission on TCP SYNs for the SSH protocol. Its address space is constained to 256 Kilobytes.

Figure 4: Example of a `scanner_conf`

`scanner_conf` is shown in Figure 4.

To launch a scanner, the Scanner Manager first extracts the scanner's binary, its `scanner_conf`, and its `scanner_subconf`. From the `scanner_conf`, the Scanner Manager caches a scanner's traffic permissions and its resource limitations. The Scanner Manager then executes the `socketpair` command to reserve a communication channel between the itself and the scanner. It then executes a `fork`, leaving the child process to finish the launching process and the parent to launch additional scanners.

The child process then performs a series of commands to execute the scanner in a restricted environment. First, `chroot` is executed to confine the scanner's file system access. The scanner binary and the `scanner_subconf` are then placed in the `chroot`ed directory and appropriate

file permissions are set. A series of `setrlimit` commands are executed based on the resource limits defined in the `scanner_conf` file, the process is assigned a unique group id and user id via `setgid` and `setuid`, and the scanner is launched using the `execve` command with an argument containing the file id of the communication socket.

Upon launch, the scanner reads its XML configuration, opens sockets for communicating with the Siphon, and issues subscription requests to the Scanner Manager. If these subscription requests do not violate the traffic permissions outlined in the `scanner_conf` file, the Scanner Manager will generate a Berkeley Packet Filter (BPF) representation of the requested traffic and send it (along with the subscription type, permissions model, and path to the scanner's socket) to the Siphon through an `ioctl`. The Siphon then connects to the scanner's socket and begins forwarding it the subscribed traffic.

After all scanners are launched, the Scanner Manager is responsible for two important tasks. First, the Scanner Manager periodically receives heartbeats from each scanner. If a scanner misses a heartbeat, it is most likely hung, compromised, or terminated. In the event of a missed heartbeat, the Scanner Manager creates an Intrusion Detection Message Exchange Format (IDMEF) alert and sends it to the administrative console (Castellan, described below) over a secure channel. If the scanner is not terminated, the Scanner Manager will do so and perform basic cleanup.

The Scanner Manager's other responsibility is creating alerts and forwarding on the behalf of scanners. To do so, scanners simply send an alert message to the Scanner Manager, which will then create an IDMEF message containing the host name, the scanner name, and the scanner's configuration identifier along with the alert and forward it to Castellan. Making the Scanner Manager responsible for constructing and delivering a scanner's alerts reduces the trusted code base (TCB) in the scanner library and ensures a scanner cannot forge alerts from other scanners. Additionally, although not included in the current implementation, the Scanner Manager could be extended to consolidate alerts or throttle the alerts of a misconfigured or compromised scanner.

Castellan: Administrative Console. Castellan is the administrative console for managing the self-securing NI platform. It contains two distinct architectural components: a background daemon for automatically collecting, interpreting, and reacting to alerts, and a GUI for browsing alerts and performing manual policy changes. The background daemon maintains a connection over a secure

channel with all participating SS-NIs. It stores the alerts it receives in a centralized location and parses these alerts in an attempt to automatically determine the current state of the network. If desired, the background daemon may be configured to dynamically issue policy changes to multiple SS-NIs in reaction to the alerts it has accumulated. For example, if the network is perceived to be under attack, the background daemon could choose to instruct all SS-NIs to switch to a more paranoid `siphon_conf`, perhaps one which firewalls all non-essential ports, in an attempt to make the network more resilient. A sample set of SS-NI applications that rely on coordination from the background daemon is described in Section 5.

The other part of Castellan is its GUI interface. This interface provides the administrator with two main capabilities: browsing and filtering alerts, and browsing and managing SS-NI policies. Castellan was implemented in Java using the NetBeans IDE,

Castellan's GUI greatly simplifies the policy management of SS-NIs. An administrator can browse, delete, and install each NI's `siphon_conf`s, scanner applications, `scanner_conf`s, and `scanner_subconf`s. A number of checks are enforced to reduce the capacity for administrative error; notably all XML files are checked against their respective document definition types (DTDs) before being entered in an NI's database. Ordering is enforced on management operations to reduce the potential for invalid configuration caused by missing dependencies. For example, a `siphon_conf` that references an uninstalled scanner or `scanner_conf` cannot be installed. Similarly a scanner referenced by an installed `siphon_conf` cannot be deleted before that `siphon_conf` is deleted.

The administrator can also use Castellan to switch the running `scanner_conf` of a SS-NI. In order to do so, the administrator simply selects a SS-NI in Castellan, browses its collection of `siphon_conf`s, and clicks on a button to activate it. This causes Castellan to connect to the SS-NI's DBMS and alter the value of the `running_siphon_conf` table. The Scanner Manager sets a trigger on this table and is notified of all changes to it. After querying for the new value, the Scanner Manager kills all currently running scanners, performs garbage collection, and proceeds as described above to launch the new configuration.

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

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.

Table 2: Base performance of the self-securing NI prototype.

4.3 Basic overheads

Although performance is not the focus of this Thesis, it is useful to quantify Siphon's effect on NI throughput and latency. As found by previous researchers [14, 17, 29], one observes 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 266MHz 586, 128MB of main memory, and 100Mb/s Ethernet interfaces. After subtracting the CPU power used for packet management functions that could be expected to be hardware-based, this modest 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.

CHAPTER V

EXAMPLE APPLICATIONS

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, this work describes the attack, the scanner, relevant performance data, and associated issues.

5.1 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 [27].

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 address 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, 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).[1] 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.

[1]Although not observed in CMU's network, DNS traffic can be passed on TCP port 53 as well. The current scanner will not see this, but could easily be extended to do so.

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) [27] would have grown to over a month (assuming the original scan rate was 10 per second per infected machine [37]).

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 [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: 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 [38, 33, 44], 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[2] 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 [42]. The proposed implementation (extending host-based firewall code)

[2]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.

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 [37] 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 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.

5.2 Detecting claim-and-hold DoS attacks

Qie et al. [32] 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 [32]). 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 [30] 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.

5.3 Detecting TTL misuse

Crafty attack tools can hide from NIDSs in a variety of ways. Among them are insertion attacks [31] based on misuse of the IP TTL field, which determines how many routers a packet may traverse before being dropped.[3] 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.

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

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.

5.4 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 [34] 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 [19]; 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.

5.5 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 [17, 22]) 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 [35]; 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 [31], 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, 43] or general anomaly detection of network traffic. This is an area for future study.

CHAPTER VI

RELATED WORK

Self-securing NIs build on much existing technology and borrow ideas from previous work, as discussed throughout the flow of this thesis. Network intrusion detection, virus detection, and firewalls are well-established, commonly-used mechanisms [5, 8]. Also, many of the arguments for distributing firewall functions [14, 20, 28] 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, 23] 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 [29], 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 [31].

There is much ongoing research into addressing Distributed DoS (DDoS) attacks. Most countermeasures start from the victim, using traceback and throttling to get as close to sources as possible. The D-WARD system [24] 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, 18] and infrastructure components [3, 11, 40, 41]. 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.

CHAPTER VII

CONCLUSION

Self-securing network interfaces are a promising addition to the network security arsenal. This thesis 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.

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. S. Alexander, K. G. Anagnostakis, W. A. Arbaugh, A. D. Keromytis, and J. 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] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, **15**(1):54–64, February 1995.

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

[9] E. Cooper, P. Steenkiste, R. Sansom, and B. Zill. Protocol implementation on the Nectar communication processor. *ACM SIGCOMM Conference* (Philadelphia, PA), September 1990.

[10] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, **7**(4):36–43, July 1993.

[11] D. S. Decasper, B. Plattner, G. M. Parulkar, S. Choi, J. D. DeHart, and T. Wolf. A scalable high-performance active network node. *IEEE Network*, **13**(1):8–19. IEEE, January–February 1999.

[12] D. L. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit, and S. 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] M. E. Fiuczynski, B. N. Bershad, R. P. Martin, and D. E. Culler. *SPINE: an operating system for intelligent network adaptors*. UW TR-98-08-01. 1998.

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

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

[16] G. R. Ganger, G. Economou, and S. M. Bielski. *Finding and Containing Enemies Within the Walls with Self-securing Network Interfaces*. Carnegie Mellon University Technical Report CMU-CS-03-109. January 2003.

[17] M. Handley, V. Paxson, and C. 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.

[18] D. Hitz, G. Harris, J. K. Lau, and A. 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.

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

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

[21] J. Jung, E. Sit, H. Balakrishnan, and R. 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] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. *IEEE INFOCOM* (Tel Aviv, Israel, 26–30 March 2000), pages 1381–1390. IEEE, 2000.

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

[24] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. *IEEE International Conference on Network Protocols* (los Angeles, CA, 12–15 November 2002), pages 312–321. IEEE, 2002.

[25] P. V. Mockapetris and K. 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] J. C. Mogul, R. F. Rashid, and M. 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] D. Nessett and P. Humenn. The multilayer firewall. *Symposium on Network and Distributed Systems Security* (San Diego, CA, 11–13 March 1998), 1998.

[29] V. 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.

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

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

[32] X. Qie, R. Pang, and L. 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). USENIX Association, 2002.

[33] A. Rowstron and P. 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.

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

[35] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *ACM Computer Communications Review*, **29**(5):71–78. ACM, October 1999.

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

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

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. 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] J. Sugerman, G. Venkitachalam, and B.-H. 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.

[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] D. 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.

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

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

[44] B. Y. Zhao, J. Kubiatowicz, and A. 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.