EXPERIMENTAL EVALUATIONS OF EMBEDDED DISTRIBUTED FIREWALLS: PERFORMANCE AND POLICY

BY

MICHAEL A. IHDE

B.S., University of Iowa, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

To my parents, for always giving me unconditional support;
my wife, for her love and understanding;
and Gwen, for providing the best motivation without even knowing it.

ACKNOWLEDGMENTS

Many people have contributed their support, guidance, and time during the preparation of this thesis. I am grateful to all of you.

First and foremost, I would like to express my sincere gratitude to Professor William H. Sanders. The opportunities and challenges I have enjoyed while working with him and his group are the direct result of his support. Professor Sanders has been a great friend, role model, and technical adviser. His dedication to all of the students in his group is exemplary; I am forever thankful for his guidance.

I would especially like to thank my close friends and colleagues: Tod Courtney, Sankalp Singh, Fabrice Stevens, Ryan Lefever, James Lyons, and Michael McQuinn. Collaborating with each of these guys has been enlightening and enjoyable. They are all brilliant researchers who have helped me through many technical problems and have also been great friends outside the office.

All of the members of the PERFORM group were critical to my success. I would be negligent if I did not recognize the other members for their willingness to listen to my ideas, offer guidance, and participate in much-needed coffee breaks. Thanks to Adnan Agbaria, Kaustubh Joshi, Salem Derisavi, David Daly, Vinh Lam, Hari Ramasamy, Shravan Gaonkar, Mark Griffith, Eric Rozier, Jason Martin, Mouna Seri, and Michel Cukier.

Jenny Applequist is perhaps the most important person in the PERFORM group. Jenny was always patient, helpful, and insightful. I thank her for all she has done for me. She has graciously read drafts of this thesis, handled countless volumes of paperwork on my behalf, and been a general point of reference for all things.

I am grateful to the entire DPASA team for their contributions to this work, especially Charlie Payne and Dick O'Brien from Adventium Labs, and Partha Pal, Michael Atigetchi, Franklin Webber, and Rico Valdez from BBN. These talented individuals contributed useful experimental ideas and verified a portion of the results presented in this thesis.

To my parents, I cannot adequately express the indebtedness I feel for receiving the support and love you have provided throughout my life. You have unselfishly provided me with the resources to follow my interests and the encouragement to succeed.

To Marianne, you have been so supportive and loving during this entire process. I can never repay you for the many times you read my thesis, and endured the stressful final months before the thesis deposit, or the many other wonderful things you have done for me.

Finally, I would like to thank DARPA, particularly Jay Lala and Lee Badger, for partially funding this research under contract number F30602-02-C-0134.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADF Autonomic Distributed Firewall

AP access proxy

DC downstream controller

DMZ demilitarized zone

DPASA Designing Protection and Adaptation into a Survivability Architecture

EFW Embedded Firewall

FPGA field-programmable gate array

GUI graphical user interface

HTTP Hyper-Text Transfer Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

IT-JBI Intrusion-Tolerant Joint Battlespace Infosphere

JBI Joint Battlespace Infosphere

LAN local area network

NIC network interface card

PSQ publish, subscribe, and Query

RFC request for comments

RISC reduced instruction set computer

RTT round trip time

SM system Manager

SSH secure shell

TCP transmission control protocol

UDP user datagram protocol

VLSI very large scale integration

VPG virtual private group

VPN virtual private network

WAN wide area network

XML extensible markup language

XSLT extensible stylesheet language transformation

CHAPTER 1

INTRODUCTION

1.1 Introduction

With the increasing popularity of the Internet, the threat of cyber-attacks has become a significant problem. Many of the old maxims of network security are no longer effective against modern threats. Recent experience with worms, such as MyDoom and Sobig, have shown that standard firewalls provide inadequate protection for threats that bypass the perimeter protection either through allowed communications, like e-mail, or through mobile hosts that temporarily leave the safety of the firewall only to bring the worm into the network behind the firewall.

The effect of these worms, which bypass the external firewall and then spread unchecked behind the firewall, has made it clear that security enforcement must be pushed to the network end-points. The use of traditional firewalls at the network edge is the first line of defense against external attacks and, in many situations, will also limit the outgoing communications. However, the firewall is helpless once an attacker or worm is behind the firewall.

Distributed firewalls can help mitigate many "insider" attacks by applying the security enforcement at the end-points instead of at the network edge. Section 1.3 describes the detailed operation of distributed firewalls. Although there is little doubt that traditional perimeter firewalls will continue to play an important role in network security, distributed firewalls provide complementary protection at the end-points when configured with appropriate policies, which in turn may preventing a worm outbreak.

Although distributed firewalls provide enhanced network protection, it is a dangerous proposition to simply trust the implementation of either the distributed firewall, or the policies that the firewall enforces. Security mechanisms, especially relatively new ones, can harbor vulnerabilities that an attacker may exploit, thus negating the usefulness of any

additional security gained. Before an administrator places trust in a security mechanism, a reasonable effort must be made to ensure that the device itself cannot be compromised or used to create attacks.

Such vulnerabilities do not exist only in complex systems. Even a simple security mechanism can harbor hidden vulnerabilities. Consider the following policy. An administrator decides that user accounts should be locked after three failed login attempts in order to prevent brute-force password-guessing attacks. Although the policy will succeed in preventing password guessing, it inadvertently creates a denial-of-service vulnerability. An attacker can lock the account by incorrectly guessing the password three times, thus preventing the valid user from using their account. Unfortunately, the mechanism itself had a hidden vulnerability. Although it succeeded in preventing brute-force attacks it allowed an attacker a simple denial-of-service attack. The discovery of that weakness then begs the question of what other security mechanisms also inadvertently create vulnerabilities.

Firewalls are not immune to hidden vulnerabilities. Many documented exploits are available for all types of firewalls. Even when the firewall itself is free of vulnerabilities, the performance overhead of a correctly operating firewall may be problematic. As incoming packets arrive to the firewall, they are placed in a queue and then compared to the rule-set before being forwarded or denied. If the rule-set is too large, the incoming packet rate is too high, or the firewall performance is insufficient, the firewall will drop packets when the incoming queue becomes full. If enough packets are dropped, a denial-of-service will occur, preventing users from communicating to any host on the other side of the firewall. Therefore, it is important to evaluate the performance of the firewall before deploying it on a network. Firewalls that cannot process incoming packets fast enough need to be upgraded or protected, lest an attacker flood them.

This thesis examines two embedded distributed firewall devices, the Embedded Firewall (EFW) from 3COM and the Autonomic Distributed Firewall (ADF) developed by Secure Computing under funding from DARPA. Both devices provide similar functionality and share a common ancestral code-base. Further discussion of these devices can be found in Section 1.4.

Although the research described in this thesis was specifically looking for denial-of-service vulnerabilities in the EFW and ADF, the experiments, presented in Chapter 2, also measured the performance of both devices as a function of rule-set size. The results show that, unlike common software-based firewalls, both the EFW and ADF had significant impact on network performance, even when enforcing small rule-sets. At its worst, when the firewall network interface card (NIC) was configured to enforce the maximum-size rule-set, the available bandwidth on a 100 Mbps network was reduced to 50 Mbps for the EFW, and 33 Mbps for

the ADF. Surprisingly, even when the firewall NIC was enforcing the smallest rule-set, which was the "default allow" policy, a successful denial-of-service was created using a packet flood that consumed less than 30% of the network frames.

Validating the firewall NICs themselves, however, does not guarantee that they will provide sufficient security. It is also important to validate the policies that are being enforced by the firewalls. An errant policy creates a false sense of security, potentially increasing the risk of attack as other security measures will be ignored if administrators assumed that the firewall alone would protect the network. To find errors in the policy, it is necessary to perform a network audit. With traditional perimeter firewalls, there is a clear division between the internal and external networks; thus, audits can be performed without significant disruption to the network. With distributed firewalls, enforcement is provided by the sender, receiver, and any intermediary firewalls. Thus, the enforced policy is the union of all policies along the path the packet traveled. This implies that policy audits cannot be performed from just one host. Instead, each host with a distributed firewall must participate in the audit. Chapter 3 provides the methodology developed in this thesis work for experimentally auditing a network with distributed firewalls. This methodology was applied to a portion of the Designing Protection and Adaptation into a Survivability Architecture (DPASA) project to evaluate its usefulness.

Firewalls have received considerable attention for many years as they have offered seemingly easy-to-use, effective security. Although firewalls are no longer thought to be the final word in network security, they still garner considerable research. The following section provides an overview of other related work as it applies to the contents of this thesis.

1.2 Related Work

1.2.1 Traditional firewalls

Traditional firewalls have become the primary line of defense against network attacks for many organizations. Although considerable debate exists on whether this is a good practice, there is little doubt that traditional firewalls will continue to play a dominant role in network security. The operation and design of traditional firewalls and their policies are well-documented in many texts [1], [2], [3], yet firewall management has been, and continues to be, somewhat of an art. Firewall administrators must use their personal experience, a set of standard practices, and corporate policy to successfully create a firewall rule-set. Lacking a formal framework to aid administrators with rule-set construction, even skilled administrators find difficulty implementing error-free complex firewall policies.

Empirical evidence, by Wool [4], indicates that policy errors in firewalls are unfortunately relatively common. For example, over 86% of the surveyed firewall policies contained a rule that allowed the *any* destination for outbound traffic. The presence of this rule is considered an error because it "...gives internal users free access to the servers in the demilitarized zone (DMZ). Worse, it often allows the DMZ servers free access to the internal network ..." [4]. Wool's survey of in-use firewall policies indicated that if the complexity of the firewall policy increases, more errors will inevitably creep into the policy. However, simple was not always good. Some simple policies contained multiple errors. Adding distributed firewalls into the network significantly increases the complexity, so it seems that the difficulty of creating error-free policies will only increase unless new methods and tools are made available.

It is important to differentiate among the various types of errors that may exist in firewalls. Some errors, like those in [4], are simply the violation of some standard that is commonly deemed to be good practice. Those violations do reduce the security posture of the firewall, and therefore should be removed. Once a list of potential violations is available, it's easy to find errors in the firewall rule-set. The difficulty lies in the formation of the list of violations. Second, policy anomalies are errors that exist when the combinations of rules within or between firewalls overlap. Those errors are further discussed in Section 1.2.6. Finally, some errors only exist when the policy is deployed. Unanticipated or superfluous communication paths, the deviation from the intended policy, are examples of such errors. Unlike the first two types of errors, which can be detected via off-line analysis, the last class of errors requires on-line policy auditing for detection. Policy auditing is further discussed in Section 1.2.5 and Chapter 3.

1.2.2 Distributed firewalls

The distributed firewall concept, first introduced by Bellovin in 1999 [5], was a major innovation in network security. Distributed firewalls provide policy enforcement at the end-point of the network via a centrally defined policy, unlike traditional firewalls, which provide perimeter protection. Many benefits are gained by enforcing a global policy at the endpoints: it reduces internal threats, it is topology-independent, it provides fine-grained access control, and it reduces global performance bottlenecks. Section 1.3 provides an overview of basic distributed firewall operation.

Distributed firewalls are available as software or hardware solutions. Early software-based distributed firewall implementations existed as research projects; a preliminary OpenBSD implementation [6] based on Bellovin's original concept, and latter the StrongMan [7] frame-

work, are two examples. There also exist a few commercial software implementations [8], [9]. The only known hardware solution (which are the focus of Chapter 2) are the 3Com EFW, which is commercially available, and the ADF, a research project based on the EFW design.

Although distributed firewalls provide enhanced network security, broad acceptance and use of distributed firewalls have not yet been achieved. Currently, three factors are preventing distributed firewalls from being deployed in large numbers: policy management is too complex for large networks, accurate policy validation is difficult, and administrators are slow to adopt new technologies that have not been fully tested for potential vulnerabilities and side effects. This thesis aims to address the last two issues.

1.2.3 EFW/ADF

The EFW and ADF are hardware-based distributed firewalls that enforce the rule-set on the NIC [10], [11], [12]. Both implementations share a common ancestral code-base and similar underlying hardware. The EFW was developed first, providing stateless packet filtering and a central policy server. The ADF later added the ability to create encrypted communication channels, called a virtual private group (VPG) [12], [13], [14], which provides confidentiality, integrity, and sender authentication. Section 1.4 provides the details of the construction and operation of the EFW and ADF.

1.2.4 Firewall performance testing

Before deploying a firewall, it is essential to evaluate the impact it will have on network performance. Poor performance impacts the users of the network, and may also provide an attacker with a denial-of-service vulnerability. Performance measurements for the two most common open-source firewall software packages, `iptables` and BSD's `pf`, are available for comparison with the results presented in this thesis [15], [16], [17]. When comparing software firewalls, either to other software firewalls or to NIC-based firewalls, it is important to consider the processing power of the host. Unlike NIC-based firewalls, software firewalls will perform better with higher-performance host processors. Therefore, `iptables` on an older computer will have much lower performance than it would on a modern computer. Firewall performance gains due to increases in the underlying host performance must be discounted for accurate comparisons.

Two request for comments (RFC) papers [18], [19] provide recommendations for analyzing network interconnect devices and firewalls. Whenever possible we attempted to follow the guidelines in each RFC paper, deviating only when the particular nature of the EFW and

ADF firewalls demanded such modifications. The performance measurements for the EFW and ADF are available in Chapter 2.

1.2.5 Auditing firewall policies

Auditing firewall policies is the process of testing and validating the firewall configuration before and after deployment. "The goal is to ensure that the firewall is enforcing what you expect it to. This is done by scanning every network segment from every other network segment" [20]. For traditional firewalls, that process is often accomplished via `nmap` [21] or `firewalk` [22] scans of the firewall.

Both `nmap` and `firewalk` perform partial scans, in the sense that they both detect the action the firewall takes on packets from the perspective of the scanning machine. A complete audit would require the scan to detect the firewall action for all possible packet sources, a daunting task (IPv4 supports 2^{32} hosts).

The challenges of auditing traditional firewalls are only compounded if the network also contains distributed firewalls. Chapter 3 contains an examination of these challenges. Because of the limited deployment of distributed firewalls, no set of common "best practices" like those available for traditional firewalls has been established to guide administrators in performing audits. As distributed firewalls increase in popularity, the need for distributed firewall auditing techniques will be established.

To date, little research has been done in the realm of distributed firewall auditing. At the same time this thesis work was in development, Wheeler [23] implemented an auditing method similar to that described in this thesis. Wheeler's method requires a set of *Probers* to be located on both ends of the tested communication path, one on the sender and one on the receiver, to detect the firewall policy. Using this technique increases the speed of the scan, as there is almost no ambiguity during the test; either the packet reaches the listening host or it does not. Some errors may go undetected, though, as the scan is not necessarily complete. For example, suppose an unknown host exists on the network. The probe method would be unable to detect any allowed communication to this host, as the administrator clearly would not have installed the required prober software on it. Similarly, networked devices, such as printers, would not be detectable using a prober unless prober software could be constructed for the particular device, an unlikely situation. To provide a complete view of the network, the scans must be carried out without the help of the targeted machine.

Auditing provides a view of the policy being enforced on the network, instead of a model of the policy. Auditing is thus useful in finding a wide variety of errors that may creep into the security policy. They include policy design errors, network topology side effects, and

deployment misconfiguration. For defense-in-depth systems, in which the policy is enforced at multiple layers, proper auditing can find those places where the defense is less than expected. This is not to say that design-time policy- checking tools, such as firewall conflict detection, are not useful; they certainly aid administrators with complex policies, but they lack the ability to make any claims about the *actual* enforcement of the policy.

1.2.6 Conflict detection

The complexity of creating correct firewall rule-sets is, in part, due to the interaction between rules within a single firewall. When two rules overlap, they may create a conflict in the policy that can violate the original intentions of the administrator. If those anomalies are detected and resolved, there is less chance of an attacker finding an accidentally opened communications path through the firewall. Firewall anomalies can be detected through direct analysis of the policy, allowing the rule-set to be tested before deployment. A considerable body of recent research exists for both inter- and intra-firewall rule-set conflict detection [24]-[29].

In [29] the three properties of an conflict free rule-set are defined. A rule-set must be consistent, complete, and compact.

Consistent: All rules are ordered correctly.

Complete: Every possible packet will match at least one rule.

Compact: No redundant rules exist.

Conflicts within a firewall rule-set can be classified into five different classes: Shadowing, Correlation, Generalization, Redundancy, and Irrelevance [26]. The first three are conflicts that can affect the security of the network. The last two allow the administrator to optimize the rule-set, as the removal of the offending rule does not alter the security policy. Optimization, however, does decrease the threat of a denial-of-service attack due to the reduction of the rule-set length. It is unclear whether the inter-firewall anomalies [26] that can be detected in traditional firewalls are also applicable to distributed firewalls.

Conflict detection is complementary to auditing. Performance of one of them does not decrease the benefit of the other. Currently, conflict detection is rarely used, as no vendors provide such tools for their product line and no known generic tool supports the wide variety of policy languages used by each vendor.

1.3 Distributed Firewall Overview

By definition [5], a *distributed firewall* requires a centralized management system that distributes policies via a secure authenticated system to the end points for enforcement. The actual enforcement may be done in hardware or software using any suitable policy language. The distributed firewall envisioned in [5] and implemented in [6] uses digital certificates to provide sender authentication and application-level filtering. It is also acceptable for a distributed firewall to perform only simple packet filtering, as the EFW does. The benefit derived from using distributed firewalls is primarily due to the separation of policy and network topology rather than the exact policy enforcement method.

When a traditional firewall is used it creates an obvious "inside" and "outside." Communications are only regulated between the two zones; all communication inside a zone is assumed to be trusted. This arrangement is useful in protecting against external threats, but does little to counteract any attacks that originate internally. It also affords no protection to mobile hosts that may temporarily be on the "outside" of the firewall. Both of those situations represent major weaknesses in a network's defensive posture.

When internal traffic is unfiltered, a self-propagating worm, once behind the firewall, will quickly infect all vulnerable hosts. Commonly, the worm bypasses the firewall and enters the network via an e-mail attachment, web browser exploit, or infected laptop. (Usually the laptops become infected while outside the safety provided by the traditional firewall.) By *de-perimeterizing* the network security through the use of distributed firewalls, each host can be protected on the "inside" and the "outside." When the distributed firewall is a hardware solution, the security is enhanced due to the fact that it becomes more difficult to circumvent the policy enforcement.

To ensure maximum security, a distributed firewall must be noncircumventable. If the distributed firewall can be circumvented by malicious code (or users), then it can no longer protect the host. For software-only solutions, it is extremely difficult to create a 100% noncircumventable host resident firewall. A user who wishes to bypass the firewall could simply reboot the machine into an alternative operating system with removable media (if the machine has not been fully protected from this type of attack). There are also many ways the firewall software may be disabled, either directly by the user or through a vulnerability in the operating system. Compared to software-based distributed firewalls, the ADF and EFW provide a unique degree of protection against circumvention. Unless the NIC is physically removed and replaced, or an implementation flaw in the NIC firmware is found, the desired policy (including the fallback policy) will be enforced regardless of the software configuration.

1.4 EFW/ADF Overview

A typical EFW/ADF deployment consists of three main components: the EFW/ADF NIC for each protected host, the EFW/ADF device driver and helper application, and the EFW/ADF policy server (Figure 1.1 [10]). The EFW/ADF helper application is primarily responsible for generating heartbeats, which are sent to the policy server. The helper application is nonessential; the EFW/ADF NIC will enforce the policy even if the helper application is not running. The policy server provides a centralized management interface for up to 4000 EFW NICs. This functionality can be replicated on up to four hosts, providing some degree of fault tolerance.

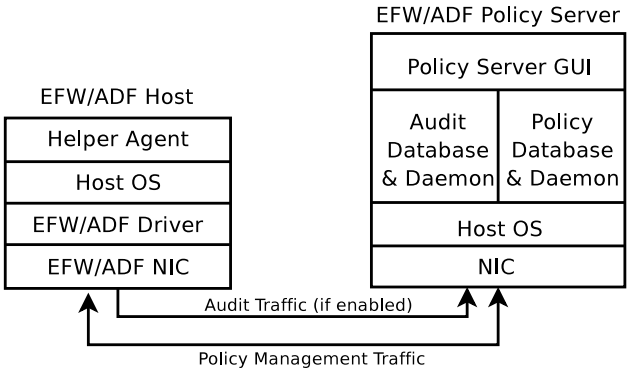


Figure 1.1 EFW/ADF Architecture

The policy server consists of a front-end graphical user interface (GUI) that drives a back-end policy and audit database. Using that GUI, the administrator can query the status of individual cards, create and modify rule-sets, and view the audit database. Normal traffic between the policy server and the NIC is minimal, except during policy updates.

When audits are enabled, a log of every packet processed by the NIC is sent to the policy server. The amount of audit traffic generated can cause an undesirable load on the network. Audits are generally only useful as an aid for policy generation, allowing administrators the ability to fine-tune a policy. The audit feature could potentially be used for intrusion detection, if it were not for the excessive overhead of sending audit data.

VPGs are an additional feature available with an ADF deployment. VPGs are basically a group virtual private network (VPN) for which the enforcement is performed on the NIC. The encryption of the packets is completely transparent to the host, allowing any application to benefit from the added protection. Key management and distribution are controlled by the policy server because of the limited processing capabilities of the NIC.

One of the primary goals of the EFW project was to remain cost-effective for large network deployment. To achieve this the device must be "fast, simple, and cheap" [10]. Because the EFW was implemented on top of an inexpensive existing network card (3CR990), the hardware costs were kept low enough for normal deployment. Although it would have been more expensive, hardware designed especially for packet filtering might have provided higher performance and possibly would have been able to withstand a packet flood attack, as shown later.

The basic EFW/ADF NIC architecture consists of [30]:

- A 100 MHz, 32-bit reduced instruction set computer (RISC) ARM9 processor,
- On board memory for storing packets and filter rules, and
- A very large scale integration (VLSI) dedicated crypto-processor.

1.5 Research Contributions

The research described in this thesis includes:

- A methodology for the performance evaluation of distributed firewalls.
- The implementation of a software-based packet flood generator to be used for performance and denial-of-service evaluations.
- An evaluation of the performance and denial-of-service tolerance for two embedded distributed firewalls: the EFW and the ADF.
- A methodology for distributed firewall policy auditing.
- The implementation of distributed firewall audit tools.
- The results of a distributed firewall audit performed on the DPASA network.

To the best of the author's knowledge, this thesis is the first publication to present performance metrics for either the EFW or ADF. As such, these results have not been verified nor confirmed by 3COM.

CHAPTER 2

EFW/ADF PERFORMANCE

2.1 Overview

Evaluating firewall performance is a critical aspect of validating its use as a security mechanism. Poorly performing firewalls, properly functional in all other respects, may create a denial-of-service vulnerability that attackers can easily exploit. The threat of this type of attack is clearly documented in RFC2647 [31]:

> Further, certain forms of attack may degrade performance. One common form of denial-of-service (DoS) attack bombards a firewall with so much rejected traffic that it cannot forward allowed traffic. DoS attacks do not always involve heavy loads; by definition, DoS describes any state in which a firewall is offered rejected traffic that prohibits it from forwarding some or all allowed traffic. Even a small amount of traffic may significantly degrade firewall performance, or stop the firewall altogether. Further, the safeguards in firewalls to guard against such attacks may have a significant negative impact on performance.

Determining whether or not a denial-of-service vulnerability exists, for a given configuration, is a trivial task. Simply attempting to use the service during a packet flood will provide the answer. This is a useful test and should be performed, yet it fails to explore the inherent interrelation between the chosen policy, performance, and flood tolerance.

It is well-known that firewall performance decreases as rule-set size increases; thus, the likelihood of creating a denial-of-service vulnerability increases as rule-set size increases. This chapter presents the methodology used to evaluate the performance of the EFW and ADF, along with the obtained results. A key contribution of this chapter is the experimental methodology itself, as it is not specific to the EFW or ADF and can be used to evaluate any distributed firewall solution.

Performance for the EFW and ADF was measured using four metrics: available bandwidth in the flood-free case, available bandwidth in relation to flood rate, the minimum flood rate required to create a denial-of-service, and Hyper-Text Transfer Protocol (HTTP) performance.

2.2 Experimental Methodology

2.2.1 Network configuration

The experimental network configuration was designed to eliminate as many potential sources of noise and error as possible. All experiments were performed on an isolated network, eliminating extraneous packets. Four hosts were connected via this isolated network: the policy server, flood generator (i.e., attacker), client, and target.

The hosts were connected through a standard 100 Mbps switch (3COM OfficeConnect 3C16734A). It was assumed that the Ethernet switch itself would not affect the results in any significant manner. In order to verify the assumption, identical tests were performed against a standard nonfiltering NIC (Intel EEPro 100). The performance loss measured with the standard nonfiltering NIC was attributed to the network switch and infrastructure. Any additional performance loss measured for the EFW and ADF is attributed to the NIC firewall itself.

Ideally, all potential side effects of the switch could be removed by using an Ethernet crossover cable to connect two hosts. This configuration was impossible to achieve as the test network required more than two hosts.

The network configuration is shown in Figure 2.1. The host configuration is located in Table 2.1. The EFW host used a 2.4 Linux kernel because of the lack of official support for the 2.6 Linux kernel. It was assumed that no major performance differences exist between the 2.4 and 2.6 kernels.

Table 2.1 Testbed Host Configurations

Host	Operating System	NIC	CPU/Memory
Policy Server	Windows 2000	EEPro 100	1 GHz Pentium III/256MB
Attacker	Fedora Core 2 (2.6 kernel)	EEPro 100	1 GHz Pentium III/256MB
Client	Fedora Core 2 (2.6 kernel)	ADF	1 GHz Pentium III/256MB
EFW Target	Redhat 7.2 (2.4 kernel)	EFW ver. 2.0	1 GHz Pentium III/256MB
ADF Target	Fedora Core 2 (2.6 kernel)	ADF	1 GHz Pentium III/256MB

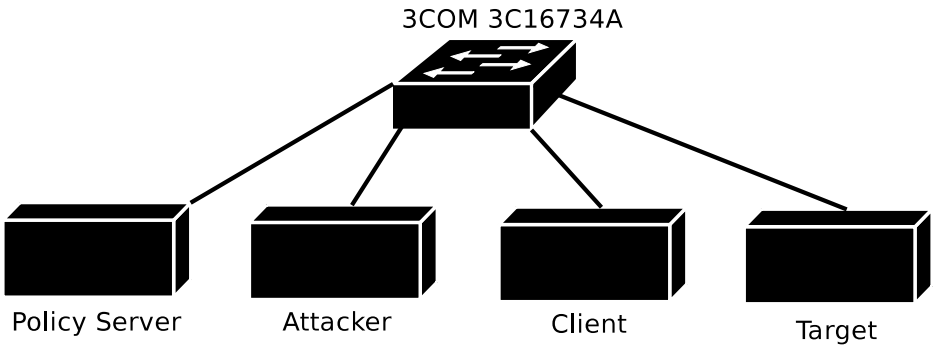


Figure 2.1 Testbed Network Configuration

2.2.2 Firewall rule-sets

The rule-sets used in the experiments were configured to act on the packets at a particular rule in the rule-set. Packets being processed by the firewall traverse a certain number of rules before the action is taken at the "action rule."

Sample eight-rule-traversal policies are are shown in Figures 2.2 and 2.3. For simplicity, each rule in the rule-set prior to the "action rule" denied all packets from any Internet Protocol (IP) address that was not used by any of the hosts in the test network. In the figures each row represents a single rule containing six fields (listed in order): rule number, protocol, source IP, source port, destination IP, destination port, and action.

```
1: any, 130.126.141.14, any, *.*.*.*, any, deny
2: any, 130.126.141.15, any, *.*.*.*, any, deny
3: any, 130.126.141.16, any, *.*.*.*, any, deny
4: any, 130.126.141.17, any, *.*.*.*, any, deny
5: any, 130.126.141.18, any, *.*.*.*, any, deny
6: any, 130.126.141.19, any, *.*.*.*, any, deny
7: any, 130.126.141.20, any, *.*.*.*, any, deny
8: default allow
```

Figure 2.2 Rule-Set Allowing Flood Packets at Eighth Rule

Evaluating VPG performance required slightly different rule-sets. For these experiments, the ADF was configured to have zero to three bidirectional VPGs that would not match the incoming packets from the client and one bidirectional VPG that does match the incoming packets. A full definition of a bidirectional VPG requires two rule slots, as seen in Figures 2.4 and 2.5. In the example VPG rule-sets, the device sets represent the hosts that are able

to send/receive as part of the VPG, and the EFW_IP represents the IP address of the NIC sending or receiving the packets.

```
1: any, 130.126.141.14, any, *.*.*.*, any, deny
2: any, 130.126.141.15, any, *.*.*.*, any, deny
3: any, 130.126.141.16, any, *.*.*.*, any, deny
4: any, 130.126.141.17, any, *.*.*.*, any, deny
5: any, 130.126.141.18, any, *.*.*.*, any, deny
6: any, 130.126.141.19, any, *.*.*.*, any, deny
7: any, 130.126.141.20, any, *.*.*.*, any, deny
8: any,    ATTACKER_IP, any, *.*.*.*, any, deny
9: default allow
```

Figure 2.3 Rule-Set Denying Flood Packets at Eighth Rule

```
1: enc, any, Client/Server Device Set, any, EFW_IP, any, allow
2: enc, any, EFW_IP, any, Client/Server Device Set, any, allow
3: default deny
```

Figure 2.4 Rule-Set With One VPG Between Client and Server

```
1: enc, any, Other Device Set, any, EFW_IP, any, allow
2: enc, any, EFW_IP, any, Other Device Set, any, allow
3: enc, any, Client/Server Device Set, any, EFW_IP, any, allow
4: enc, any, EFW_IP, any, Client/Server Device Set, any, allow
5: default deny
```

Figure 2.5 Rule-Set With One Unused VPG Followed By a VPG Between Client and Server

2.2.3 Measurement tools and techniques

Bandwidth between two hosts was measured using `iperf` [32], a cross-platform client-server software tool capable of measuring both transmission control protocol (TCP) and user datagram protocol (UDP) bandwidth. In order to measure available bandwidth, it was necessary for the firewall policy to allow communication between the `iperf` client and server, as seen in Figure 2.6.

HTTP load tests were performed using `http_load` [33] to repeatedly request a web page from an `apache2` web server. The web server was configured with the default Gentoo configuration. To achieve the goal of measuring performance loss, `http_load` was configured to use at most one connection at a time with an unlimited rate for 30 s. Alternatively, `http_load` could have been configured to measure the number of parallel connections supported by the server at a given connection rate.

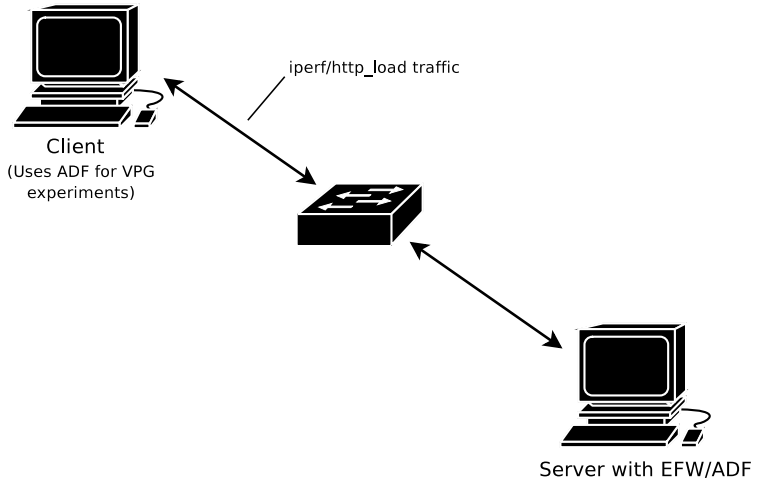


Figure 2.6 Experimental Methodology for Bandwidth/HTTP Tests

The flood tolerance of the EFW/ADF was tested using an additional machine as the hypothetical attacker, as seen in Figure 2.7. While floods were directed at the server from the attacker (using the flood generator described in Section 2.3), the bandwidth between the client and server was measured. If the flood was able to prevent the measurement from succeeding (i.e., 0 Mbps), then the denial-of-service attempt was deemed successful.

2.3 Implementing a Flood Generator

The implementation of the controlled packet flood generator was an important aspect of the experimental process. Unlike attack flood generation, which only needs to send at the fastest possible rate, a controlled flood requires packets to be sent at a specific rate, with a reasonable level of accuracy.

Achieving decent rate accuracy for very high flood rates is challenging in a software-only solution. Sending packets at the maximum rate for a 10 Mbps link would require the generation of one packet every 67 μs, which is far faster than current commodity non-real-time operating systems can manage. For high-speed links, even faster packet generation

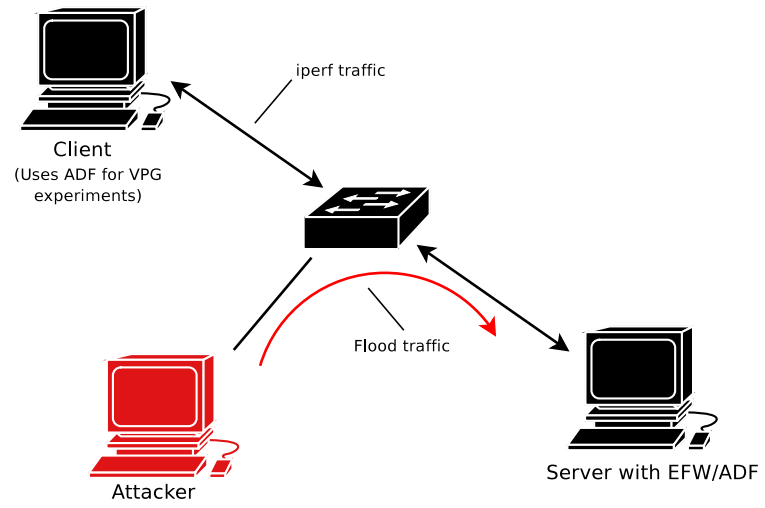


Figure 2.7 Experimental Methodology for Flood Tests

methods are required for successful flood tests. The maximum frame rates for common Ethernet bandwidths are shown in Table 2.2.

Controlled generation of high packet rates is possible with expensive dedicated hardware solutions. Lacking a hardware-based packet generator, the software flood generator sacrificed some degree of accuracy to achieve high frame rates. The most limiting factor in a software solution is the minimum scheduling period supported by the operating system. For Linux, it is 1 jiffie (1 ms by default in kernel version 2.6).

Table 2.2 Theoretical Maximum Frame Rates for Ethernet

Link Rate (Mbps)	Maximum Frame Rate (frames/s)
10	14 881
100	148 810
1000	1 488 100

To avoid overtaxing the host, the flood generator was constructed to use 2 ms timers to trigger packet transmission (500 packets/s). To achieve flood rates over 500 packets/s, the generator issued multiple packets in immediate succession. For example, the sending of 2000 packets/s was approximated by sending four packets every 2 ms. The packet flood generator could also be configured to generate an uncontrolled maximum-speed flood, similar to what would be expected from a brute-force attacker. This type of flood is not limited by the accuracy of the system timers and can therefore send at very high rates.

The maximum flood that can be generated by a single host is a function of many factors. Processor speed, network infrastructure, and scheduling overhead dominate over other minor factors. In the test network, the flood host was able to sustain approximately 45 000 packets per second if the destination host did not send replies in response to the flood packets. If the targeted host did send replies (i.e., reset packets) the maximum flood rate dropped to approximately 22 000 packets per second. This indicated that the underlying network infrastructure or host was limiting the overall packet rate to the sum of traffic in both directions.

The source code for the flood generator can be found in Appendix A. It is provided so that the experiments in this section can be validated by other researchers.

2.4 Results

2.4.1 Available bandwidth

Measuring the available bandwidth through the firewall gives an initial glimpse at the performance impact the firewall imposes on network traffic. Available bandwidth is an indirect measurement of maximum throughput, the fastest rate at which incoming packets can be processed by the firewall without loss. Generally speaking, available bandwidth is measured with the largest packet size allowable by the underlying medium. Doing so inflates the actual performance, as large packets are transmitted at a slower rate. Maximum throughput, on the other hand, uses the smallest packet size allowable; therefore, the two measures are not equivalent.

Ideally, maximum throughput would have been measured directly via the methods detailed in RFC2544 [18]. However, the methods recommended in RFC2544 are better suited to traditional firewalls, which use separate incoming and outgoing interfaces. Attempting to use the same measurement techniques for distributed firewalls would have required the EFW/ADF host to forward packets through a second interface, adding additional overhead and potential complications in the experiment. As an alternative, the methodology used for distributed firewalls did not require any additional interfaces or packet forwarding.

All bandwidth measurements were taken without any attack flood; all bandwidth loss was due to the additional processing overhead incurred by deeper rule-sets.

The results are presented in Figure 2.8. As expected, the EFW and ADF caused bandwidth loss as the rule-set grew in length. The amount of performance loss, however, was surprising. For comparison purposes, identical tests were performed against `iptables`.

Validating Hoffman et al.'s [15] results, `iptables` caused no bandwidth loss for rule-sets smaller than 64 rules with a 100 Mbps network.

The EFW and ADF lost 46% and 65%, respectively, of full bandwidth capacity when configured with the largest possible rule-set. The experiments indicated that rule-sets that contained less than 20 rules did not cause a significant amount of bandwidth loss. Therefore, it would be wise to move all rules that allow traffic for performance-sensitive applications early in the rule-set (the first 20 rules) to avoid the drastic bandwidth loss associated with rules deeper in the rule-set.

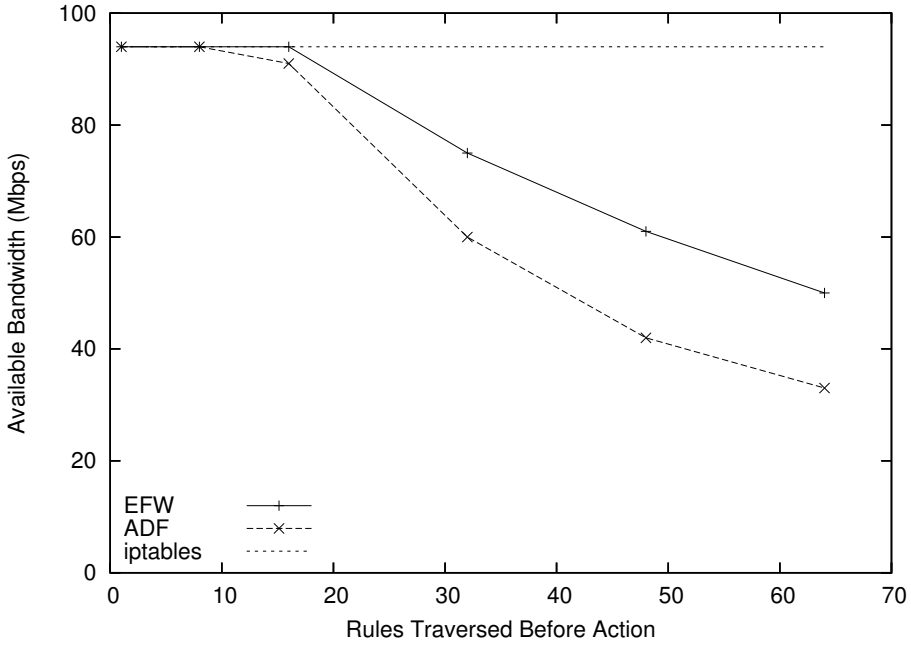


Figure 2.8 Bandwidth Loss as Rule-Set Depth Increases

When the ADF was configured to use VPGs, the performance drop was more significant, as seen in Figure 2.9. This was due to the encryption/decryption overhead for all VPG packets. Surprisingly, when additional nonmatching VPGs (those that did not match the packets of the client/server VPG) were inserted into the rule-set, performance did not decrease by any appreciable amount. That implies that the ADF was able to determine whether an incoming packet matched a VPG rule quickly without decrypting the packet.

2.4.2 Available bandwidth during floods

The poor performance of the EFW/ADF indicates that a flood of arbitrary packets may overload the EFW/ADF card. Bandwidth is related to frame rate by $BW = FrameRate *$

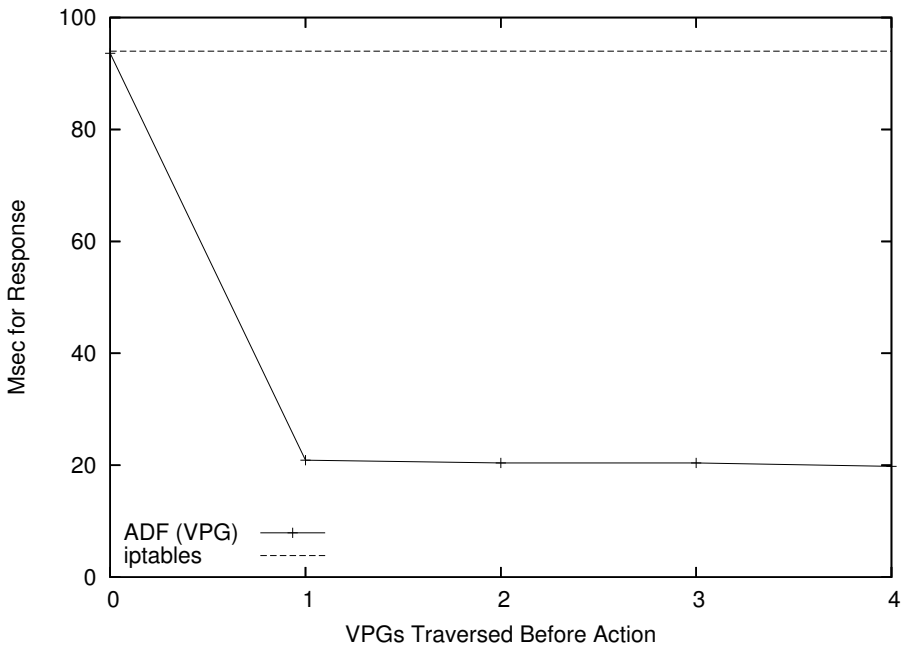


Figure 2.9 Bandwidth Loss as VPGs Are Added to the Rule-Set

FrameSize. During the available bandwidth tests the frames were the maximum size supported by Ethernet (1518-bytes); thus, the EFW/ADF was only able to process approximately 4100 packets/s when the policy contained 64 rules. For smaller policies, it was impossible to determine whether a smaller, higher-rate packet stream would overload the firewall card. With one rule the EFW/ADF was able to support the full network bandwidth, which sent large packets at a much lower rate than the theoretical maximum frame rate of the network (148 810 packets/s for a 100 Mbps network). Therefore, the maximum throughput could not be determined from the bandwidth experiments alone.

To measure the maximum throughput, another experiment was used. As before, this measurement was indirectly achieved through measurement of the available bandwidth while a flood of packets was being sent to the host with the `iperf` server. At each of nine flood rates, three bandwidth measurements were taken and averaged. These results are shown in Figure 2.10.

For the EFW/ADF, a major portion of bandwidth was lost with a flood of 16 000 packets per second. A flood with 20 000 packets per second caused the available bandwidth to drop to almost zero, thus creating a successful denial-of-service attack. The drastic bandwidth loss seen in the EFW/ADF did not occur for either the standard NIC or `iptables`, which both supported 77 Mbps when flooded with 20 000 packets per second. The only possible conclusion is that the EFW/ADF are alone responsible for the loss. The flood tolerance of

a single VPG was interesting due to the near-linear relation between bandwidth and flood rate.

The bandwidth loss associated with the EFW/ADF is due to packet loss caused by the flood. Packet loss invokes the TCP congestion control algorithm, which begins exponential back-off on the outgoing packet rate, drastically reducing bandwidth until packets are no longer lost. Placement of `iperf` in UDP mode revealed that packet loss was over 90% during successful packet floods. It is possible that some non-TCP protocols would be able to withstand high packet loss and thus continue to operate during a flood attack, but these protocols are not in common use.

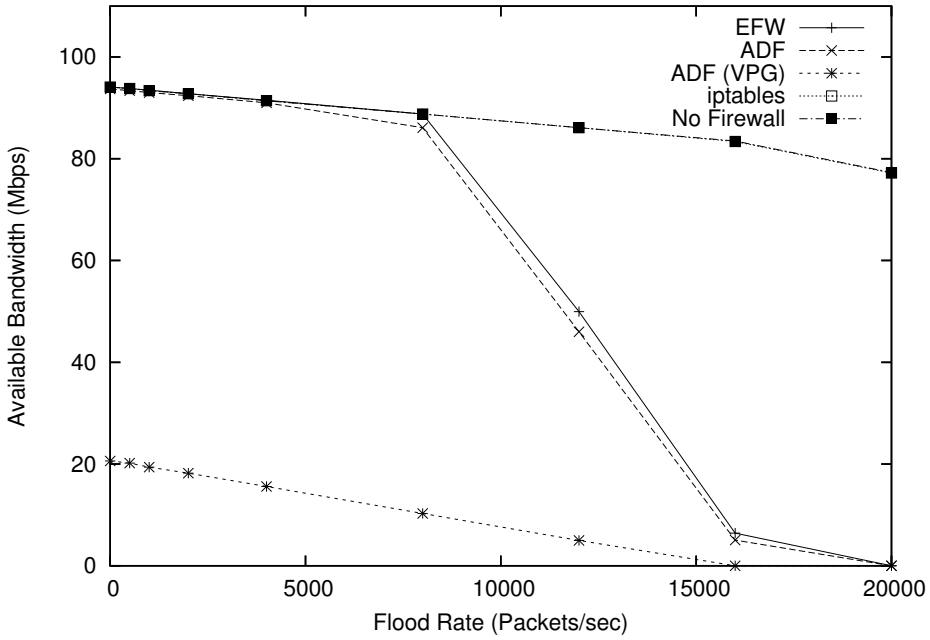


Figure 2.10 Available Bandwidth During TCP Packet Flood With Default Allow Rule

2.4.3 Minimum flood rate

All of the experiments described up to this point have used the most basic rule-sets, a single default allow rule or a single VPG. The experiments have shown that even a simple rule-set is vulnerable to denial-of-service attacks. However, it would be rare to find an EFW/ADF that was deployed with such simple rule-sets. It is important to determine the effect of additional rules as the increased rule depth lowers the minimum required flood rate.

The minimum flood rate an attacker must sustain to successfully cause a denial of service was measured by incrementally increasing the flood rate until the bandwidth fell to

approximately 0 Mbps. Two different rule-set classes were tested: one with the flood packets allowed and another with the flood packets denied. In each case, action was taken on the flood packets at rules 1, 8, 16, 32, and 64. Results are presented in Figure 2.11.

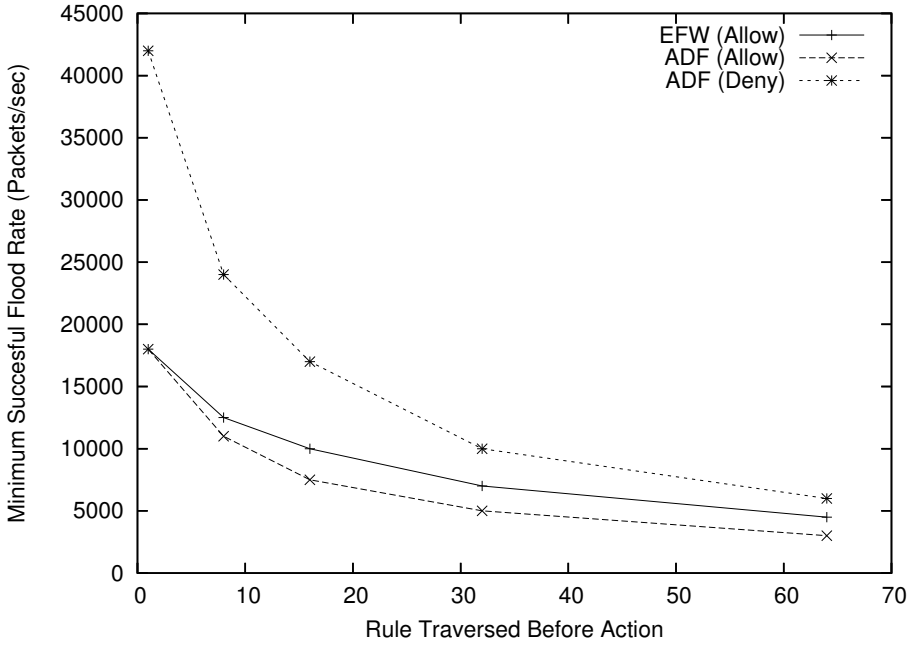


Figure 2.11 Minimum Flood Rate Required to Cause Denial-of-Service as Rule-set Depth Increases

The minimum required flood rate was also determined when VPGs were used, as shown in Figure 2.12. Oddly, there was a slight increase in flood tolerance when one VPG was used and the attack packets were denied. This is likely due to the NIC's ability to determine whether the flood packets do not match a VPG rule, and thus proceed to the next rule, quicker than it is able to check normal non-VPG rules.

With only eight rules the performance was low enough that an attacker on a 10 Mbps network could easily create a flood attack if the packets were allowed by the rule-set. When the largest rule-set was enforced, the attacker host only needed to generate 4500 packets/s to create a denial-of-service.

Some flood tolerance was gained when the attack packets were denied in the firewall rule-set. This effect was likely due to the lack of outgoing TCP responses normally generated by packets received by the host. When the attacker packets were dropped, the host would not receive the packet; thus, no outgoing response packets would be sent. As a result, total traffic through the firewall was halved, doubling the required flood rate.

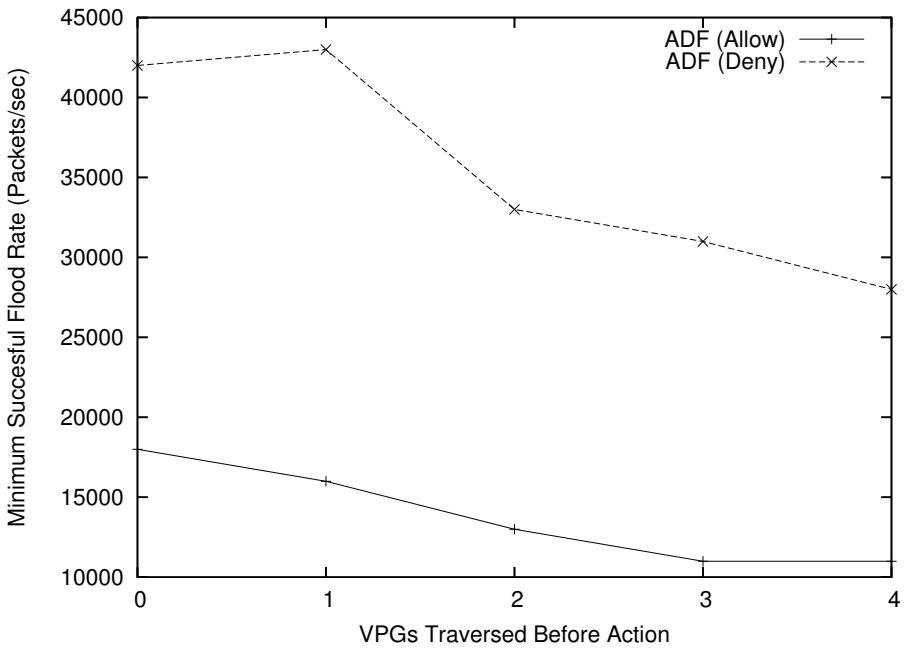


Figure 2.12 Minimum Flood Rate Required to Cause Denial-of-Service as VPG Depth Increases

In conflict with the earlier recommendation to place bandwidth sensitive services early in the rule-set, it is also important for the policy to deny any potential sources of attack early in the rule-set. However, early denial is only partially effective in preventing flood attacks, given the attacker's ability to spoof packets that will traverse deeper into the rule-set.

During the experiments it was not possible to capture any data for the EFW (Deny) case because the card halted and stopped processing packets when it was flooded with over 1000 packets/s. Restarting the firewall agent software restored functionality to the NIC until the next flood test. No solution was found.

As expected, `iptables` was able to withstand any packet flood attack directed at it. The `iptables` performance [15] has 22% network utilization for 100 rules on a 100 Mbps network (with 64-byte frames). This utilization translates to approximately 33 000 packets/s; thus, with only 64 rules it is unlikely that the flood generator was able to achieve a high enough rate to flood the firewall.

2.4.4 HTTP performance

The performance and denial-of-service experiments indicate that using the EFW/ADF will have a significant effect on application performance. Because there was no easy way to

convert raw packet performance to application-level performance, an additional experiment using HTTP was performed.

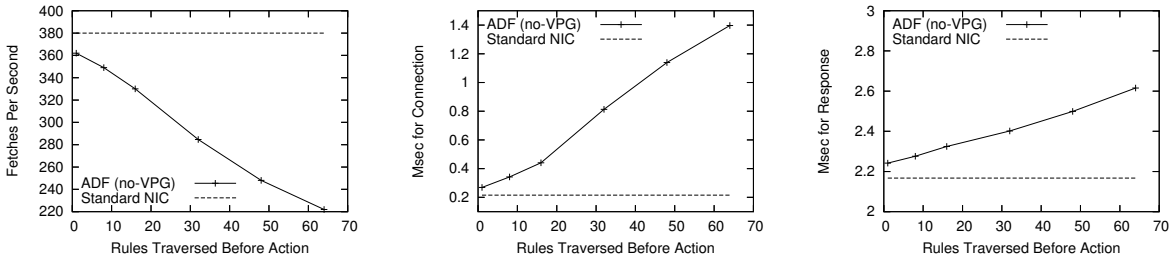
HTTP performance tests were run against an `apache2` web server. The measurements provided direct insight into the performance decrease associated with the firewall filtering. As anticipated, if the rule allowing HTTP traffic was placed deep in the rule-set, performance decreased.

Three measurements are provided by `http_load`: throughput, connection latency, and response latency. The throughput of the server, measured in page fetches per second, provides a rough estimate of how many users the server can support simultaneously. Connection latency is the time required to complete the 3-way TCP handshake. Response latency is the time required to complete the entire transfer of the requested web page.

Figure 2.13 shows that the ADF offered lower performance than a standard NIC in all configurations. As the action rule was placed deeper in the rule-set, web-server throughput was reduced. At its worst, the ADF was responsible for a 41% performance decrease compared to a standard NIC.

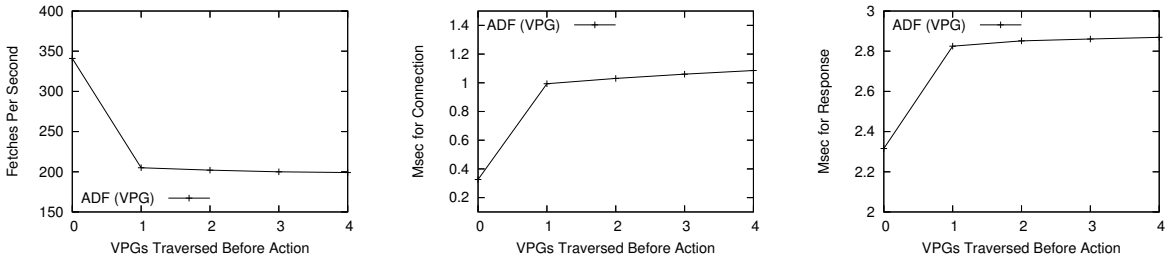
The connection time and response time are latency metrics that are important for interactive applications. Figure 2.13 shows that both latency measures increased as the rule-set size increased, but the additional delay was not excessive. Any additional latency would hardly be noticeable for Internet service, which typically has a latency greater than 50 ms. The additional latency might be noticeable for local area networks, but would only be problematic for the most demanding real-time applications.

Using VPGs also significantly affected HTTP performance. Figure 2.14 shows that the addition of a VPG dropped performance significantly, but that the insertion of other non-matching VPG rules did not alter the performance. This is similar to the effect seen for the available bandwidth experiments.



(a) Throughput of Apache (b) Connection Latency (c) Response Latency

Figure 2.13 HTTP Performance of ADF (no-VPG)



(a) Throughput of Apache (b) Connection Latency (c) Response Latency

Figure 2.14 HTTP Performance of ADF (VPG)

2.5 Discussion

2.5.1 Analysis of results

The experimental results show that the EFW and ADF do not perform well enough to be used on a 100 Mbps network in any configuration. This is quite problematic, given the proliferation of 100 Mbps networks. On a 10 Mbps network, the EFW/ADF can be safely used if the rule-set is kept to under eight rules. In general it is very difficult to provide a useful rule-set in under eight rules. For example, to protect an Oracle database server, 3COM recommends a rule-set that requires at least 31 rules to protect the appropriate ports (see Table 2.3) [34].

Table 2.3 Recommend Ports to Filter to Protect Oracle Database Server

TCP	UDP
66, 1521, 1525, 1527, 1529, 1571, 1575, 1630, 1748, 1754, 1808, 1809, 1830, 2481-2484, 3339, 7771-7777	66, 1521, 1525, 1527, 1529, 1571, 1575, 1630, 1748, 1754, 1808, 1809, 1830, 2005, 2481-2484

If access to the source code and hardware schematics had been available to us, it would have been possible to determine the location of the performance bottleneck. The slight increase in EFW performance is likely due to a change in the packet filtering code [35]. Unfortunately, without such access, it is only to possible to make conjectures about the exact implementation used on either the EFW or the ADF.

Publicly available information shows that the firewall NIC utilizes a single 100-MHz ARM9 RISC processor. Thus, the processing of both incoming and outgoing packets is done in a single loop, rather than with parallel processing for each direction. It is likely that the NIC operates with an algorithm similar to Algorithm 1. The loop period and time cost of each rule in the rule-set can therefore be estimated using the results of the minimum flood rate experiments, assuming that each individual rule evaluation will take approximately the same amount of time as all other rules to execute, regardless of the content of the rule. Given that both the EFW and ADF are simple stateless packet filters, that is likely the case.

Algorithm 1: Hypothesized Pseudo-code for EFW/ADF Filtering

```
(1)  if HasPackets(IncomingQueue)
(2)     packet ← Pop(IncomingQueue)
(3)     Filter(packet)
(4)  if HasPackets(OutgoingQueue)
(5)     packet ← Pop(OutgoingQueue)
(6)     Filter(packet)
(7)  goto 1
```

Assuming that during a successful flood nearly 100% of all packets being processed by the NIC are attack packets, the estimations will be fairly accurate. Using the minimum flood rate data for the deny case implies that there are no outgoing packets being processed; thus, the entire processing time is consumed with incoming flood packets. The results of the calculations are shown in Table 2.4.

Table 2.4 Calculating the Time-Cost Per Rule (ADF)

# of Rules	Min. Flood Rate (frame/s)	Period (μs)	Overhead (μs)	Per Rule (μs)
1	42 000	23.8	0.0	0.0
8	24 000	41.7	17.9	2.23
16	17 000	58.8	35.0	2.19
32	10 000	100.0	76.2	2.38
64	6000	167.0	143.0	2.23

2.5.2 Preventing packet flood attacks on distributed firewalls

It is possible to imagine many possible attack scenarios using the denial-of-service vulnerability found in the EFW/ADF. Any host that accepts incoming traffic from a high-bandwidth connection is vulnerable to flood attack. The EFW/ADF vulnerability also allows insider

attacks that can be difficult to trace. Finally, a virus or worm-based attack can be imagined, whereby a flood is initiated that is blocked as part of the filtering of outgoing packets. As the packets never reach the network, the user can be quite confused when the network connection ceases to operate.

Even though the EFW and ADF are susceptible to flood attacks, there is no reason to dismiss their use altogether. In many cases, the benefits of a hardware-based distributed firewall far outweigh the risk of a flood attack on any particular host. In the cases where flood attacks represent considerable risk, various mitigation strategies may be employed that either eliminate or reduce the threat of attack.

Prevention of external flood attacks will generally be the first priority, as outsider attacks represent the largest threat. In some situations, external flood attacks are prevented by the slow frame rate of the upstream wide area network (WAN) connection. Additionally, it is highly likely that any flood will be unintentionally throttled back to a safe level by intermediary Internet routers. In any situation, a perimeter border firewall should provide protection against floods via ingress or egress rate-limiting.

Ingress and egress rate-limiting would also be useful to protect against inside attacks. Through use of internal local area network (LAN) switches (with sufficient capabilities), the flood may be reduced to a safe level before reaching the host. Rate-limiting will prevent the flood attack but will simultaneously decrease the attack-free network performance. To prevent performance loss, rate-limiting should only be enabled if and when a flood attack is detected.

It may be very difficult to detect a packet flood, especially the source of the flood, depending on the network topology, available administrative tools, and firewall configurations. Unlike traditional firewalls, which store packet logs, the EFW and ADF do not store logs on the host directly for analysis. Although some degree of logging may be achieved if auditing is enabled, as the embedded firewall sends an audit packet to the policy server for each processed packet, it unfortunately will only exacerbate the problem by doubling the network traffic processed by the firewall. As the packets may be spoofed, the logs will not help direct the administrator to the correct source. Given those difficulties an administrator is faced with the challenging task of tracing the flood back through the various switches and routers to find the actual source. The process is difficult and time-consuming compared to mitigating or preparing for the attack in the first place and is not recommended.

If the policy is written to drop the incoming attack packets early in the rule-set, the minimum flood rate will be increased. However, the same optimizations that increase flood tolerance will adversely affect the performance of desired services (as the rule that allows the traffic is pushed deeper in the rule-set). Additionally, the addition of flood tolerance rules

will increase the complexity of the rule-set from the standard recommended *Allow Specifics - Deny All* to *Deny Specifics - Allow Specifics - Deny All*. It is safe to assume that an attacker will always be able to spoof packets that would be allowed, reducing the effectiveness of this mitigation.

The inability to protect against spoof attacks is a major weakness of stateless packet-filtering firewalls. Traditional firewalls are able to block some spoof attacks by considering the interface on which the incoming packets arrive, unlike a distributed firewall that has only one interface. In fact, one of the most important reasons for having a perimeter firewall is its ability to block spoofed packets from the Internet that appear to be from the internal network. Solving the spoofing problem, at least for distributed firewalls, can be achieved via encryption techniques. The original distributed firewall concept (by Bellovin [5]) and the VPG feature found on the ADFs both prevent such spoof attacks.

Each of the prevention and mitigation techniques described above may be used alone or in conjunction with others. In the end, the proper prevention technique for a given situation depends on the use of the network, the level of perceived risk, and the resources available to the administrator.

2.5.3 The future of NIC-based firewalls

Given the poor performance of the EFW and ADF, it is imperative to ask whether these performance results are indicative of NIC-based firewalls in general. Fundamental to the flood problem is the necessity for any NIC-based distributed firewall to be relatively inexpensive. If it were possible to eliminate cost as a primary factor, a NIC-based firewall could undoubtedly be constructed with sufficient performance.

Software-based distributed firewalls, on the other hand, are easier and cheaper to deploy. In some situations, using a software solution provides sufficient protection, but lacks the uncircumventablity that hardware-based firewalls offer. It may seem that software firewalls, such as `iptables`, provide vastly superior performance compared to the EFW and ADF. It is true that `iptables` performs better when it is run on modern processors, but when `iptables` is run on a processor whose capabilities are comparable to those of the ARM9, found on the EFW and ADF, the performance is actually much worse than that of the EFW and ADF. When `iptables` was run on a Pentium 166 MHz processor [16], the maximum throughput was only 4500 packets/s with a signal rule.

Constructing an EFW or ADF with a more powerful processor would surely increase the flood tolerance. Algorithmic changes may also yield a small gain, such as the one seen in the EFW firmware. Without a doubt, any performance increase is useful, but as the underlying

network speeds continue to increase above 1 Gbps, new technologies will be required in order to design NIC-based distributed firewalls.

One particular technique, using a field-programmable gate array (FPGA) as the firewall filtering engine, appears to hold significant promise. Processing packets in parallel via the FPGA's reprogrammable gates allows for extremely efficient evaluation of the rule-set. Initial research efforts have created FPGA packet classifiers suitable for network speeds 1 Gbps and above [36]. Using similar techniques, it should be possible to create a low-cost, high-performance distributed firewall.

CHAPTER 3

AUDITING DISTRIBUTED FIREWALL POLICIES

3.1 Overview

According to *The CERT Guide to System and Network Security Practices*, "The most common cause of firewall security breaches is a misconfiguration of your firewall system. Knowing this, you need to make thorough configuration testing ... one of your primary objectives "[2]. Firewalls that are not enforcing the correct policy give a false sense of security; a wall has been placed around the network but the front door has been left wide open.

Unfortunately, the current state of firewall testing tools lags behind the design and implementation of the firewalls themselves. Because it is not feasible to perform exhaustive tests of a firewall configuration [2] and few automatic tools exist to assist in the tests, usually only Idaho testing is performed. Ideally, advanced policy construction tools would alleviate the need for testing, as the policy is guaranteed to conform to a higher-level specification. Many efforts have been made to achieve this goal [37]-[40] but none can guarantee total compliance, as the actual network may be slightly different from that represented in the software. If the state of the network differs from the representation used during policy construction, errors may exist despite the use of high-level policy construction tools.

After policy construction, some errors can be detected and removed using conflict detection (as described in Section 1.2.6), but many other errors may exist in the deployed network and not in the theoretical representation of the network. This is the crux of the problem; the security of the network is a real property that depends on the actual network and the devices on it. Artificial representations are accurate only insofar as the model accurately represents the actual network. Given the difficulty of creating and verifying a model of a complex system, there will always be the need for direct, experimental testing.

This chapter presents the methodology used to audit a reasonably complex network, the implementation of a distributed firewall scanner, and the results from initial scans of the network.

3.2 Types of Errors Found by Auditing

A variety of errors may be detected by a firewall audit. Of the errors that may exist in the deployed network, auditing can detect both the errors that reduce the functionality of the network and, more importantly, the errors that reduce the security posture of the network. Auditing may also identify other minor problems in the network or policy that could be fixed.

Audits are able to detect the misconfigurations that inadvertently block traffic that the intended policy would have allowed. In other words, although the policy declares an allowed communication path between two hosts, the audit will reveal that the desired communication is blocked. Errors of this sort will reduce the functionality of the network. A multitude of reasons could exist for the inadvertent filtering of traffic that should have been allowed. For example, if during policy construction a particular network topology was assumed that differed from the actual network, then the policy could be incorrect.

Usually, errors that reduce functionality are quickly noticed by users when they attempt to use the service and find it unavailable. In some cases the error may go unnoticed if it affects a rarely used service. Such errors will sit dormant and will be detected only when the service is used. If the rare communication happens to be one that is critically important, like a failure alert, then the incorrect rule-sets created a significant problem. Performance of audits and detection of functionality errors reduce the likelihood that dormant errors will exist.

In contrast, errors that reduce the security posture of the network will rarely be caught unless an audit is performed. Without an audit, the detection of security errors is often too late; the attack itself is the only sign of the weakness. These errors generally manifest themselves as unintended or unnecessary communication paths. Unintended paths are those by which the client, instead of being denied, is able to connect successfully to a listening server. Unnecessary paths are those by which the client, instead of being denied, is able to pass packets to another host when no server was listening. Unnecessary paths, although harmless when no server is listening on the receiving host, are still very dangerous, as they may provide an attack vector if a server is ever started on the host, or may allow an attacker to use the communication path for hidden communications.

Both functionality and security errors arise from interactions in the network topology, network interconnection equipment, host software, and firewall policy. Due to the complexity of medium to large networks, these interactions are difficult to anticipate fully during policy construction. For example, the policy may have assumed that traffic between two hosts would have been restricted by the network topology, when in fact it was not.

Auditing is especially important when strong defense-in-depth is desired. Defense-in-depth is a design and operational philosophy by which multiple, sometimes redundant, layers of protection are used to increase security. It is this layered protection that can hide a potential misconfiguration in the defense layers. Errors at any level can be masked in this way during an audit. If an error is in a lower layer, the audit will be fooled when the upper layer blocks the communication. For example, Figure 3.1 shows how a hole in the EFW defenses could be masked by higher-level host protection mechanisms. The scan packets pass through the EFW but are blocked by higher-level defenses; from the scanner's perspective, the result is the same as if the EFW itself had blocked the packet. If, instead, the error is in an upper layer, the audit will only exercise the lowest layer of protection leaving the state of the upper layers hidden.

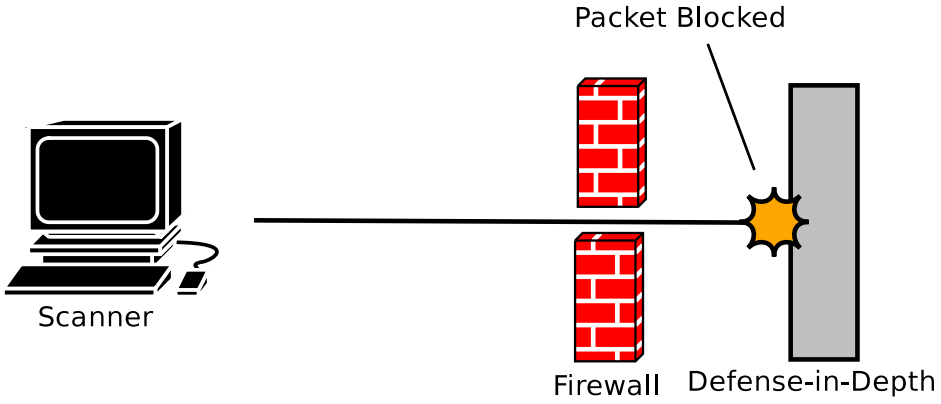


Figure 3.1 How Defense-In-Depth Errors May be Undetected

The only sure solution is to audit each layer independently *and* in conjunction with all other layers. The independent audits verify that each layer is enforcing the desired policy; the combined audit verifies that the defense layers do not interact in such a way as to create additional errors. If defense-in-depth misconfigurations go undetected, the system's effective "depth" of the defense will be reduced, increasing the risk of attack and providing a false sense of security.

3.3 Challenges in Auditing Distributed Firewalls

It is quite challenging to perform a complete audit of a single firewall, let alone an entire network. A significant time investment is needed to complete an audit of even a relatively small network. In general, it is infeasible to perform exhaustive tests of normal filtering firewalls [2]; adding defense-in-depth and/or distributed enforcement only exacerbates the problem. Typically, normal firewall audits use scanning hosts that are placed on both sides of the firewall (see Figure 3.2). Each host can act as either the scanner or the receiver, allowing audits of both the incoming and outgoing rule-sets. To test the rule-set, the scanning host sends a test packet to the firewall, which may or may not forward the packet, and then the receiving host monitors the outbound interface to determine the action the firewall took.

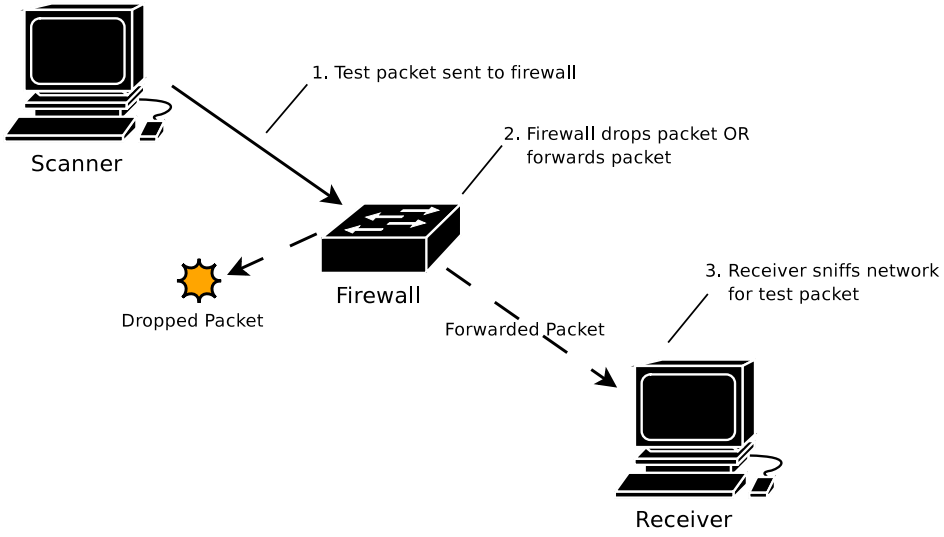


Figure 3.2 Typical Scanning Method Used for Auditing Traditional Firewalls

With distributed policy enforcement, there are no topological boundaries separating the inside and outside, so scanning cannot simply use two hosts as would be done with traditional firewalls. Furthermore, the actual enforced policy is the union of all policies along the path between the two hosts. If the network only uses distributed firewalls, then the policy is the union of the sender's and receiver's rule-sets. For that reason, each host that is protected by a distributed firewall must scan every other host to discover the complete global network policy.

Issuing a scan of every host from every host is time-consuming for the administrator and taxing on the network infrastructure. If the networks are in operational use, it is especially important not to interrupt normal services during the scans. The impact on the network

is usually tolerable if only a few scans are run simultaneously and the scanning is not too aggressive. Section 3.4.2 describes some of techniques to decrease the time spent during the `nmap` scan.

Simply distributing the scanning software to each host is another challenge. In some cases, it may be appropriate to install the software once and leave it on the host. If the host were attacked the attacker could utilize the scanning software for further attacks; thus, when security is extremely important, the scanning software should only be resident on the host for the shortest amount of time possible. This implies that the distributed scanning software should be easily installed and removed remotely. To compound the problem, heterogeneous networks, which are networks with a variety of different host configurations, require scanning software that is cross-platform and executable on all hosts on the network. Although distributing and creating cross-platform software is a challenge, it is possible to create adequate solutions with the correct design.

The next section describes the tools and methodology that were implemented to aid in distributed firewall auditing. The tools automate much of the process of distributing the scan software, performing the scans, collecting the results, and preparing the output.

3.4 Distributed Firewall Auditing Tools and Methodology

3.4.1 Scanning the network

The distributed firewall scanning tool was constructed using a set of existing tools and protocols. All network scans were performed by `nmap` [21], a mature, cross-platform, network-scanning software package. The port scan results from `nmap` detail the state of all TCP/UDP ports that were tested. A port can exist in one of three states, depending on the response to the probe packet:

Open: A response was received that indicated a successful connection,

Closed: A response was received the indicated the port was closed,

Filtered: No response was received.

Because distributed firewalls enforce policy on both the sender and receiver, the port may be filtered at either end. Figure 3.3 shows the various places traffic could be filtered between host A and host B. Either the outgoing packets or the response to them can be filtered, thus

blocking the establishment of full communication. A scan initiated from host A may have the probe packets blocked or the response blocked. From the perspective of the scanning host, there is no way to determine the location of the filtering. Usually the outgoing packets to the destination will be filtered at the sender, at the receiver, or at both hosts; thus, it is best to assume this behavior. Filtering only response packets, although possible, is a poor choice, as it allows the incoming packets to reach the server. Those packets may be able to exploit a server and thus compromise the host without requiring a response.

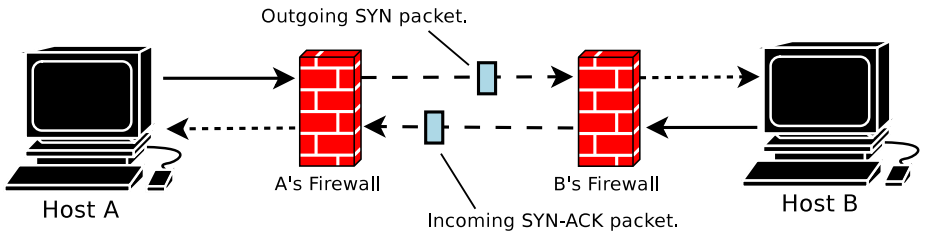


Figure 3.3 Various Locations to Filter Communication with Distributed Firewalls

In addition to the scans by each firewalled host, scans should also be performed by hosts without distributed firewalls. These additional hosts serve two primary purposes: (A) to find policy omissions due to sender-side filtering on protected hosts and (B) to determine whether the policy is overly loose. In the first case, the policy omissions would occur if the distributed firewall policy was filtering only on the sender's outbound traffic, and the receiver was accepting all inbound traffic. In the second case, the policy would be considered loose if it was correctly enforced on all hosts on the network, but allowed communications from hosts that were not part of the original network plan. Both of these errors would have the effect of allowing an attacker to connect to the network without a firewall and exploit the vulnerable machines.

The coordination of the distributed scans was handled with a centralized controller that used the secure shell (SSH) protocol for all interhost communications (see Figure 3.4). SSH was chosen for both its simplicity and cross-platform availability. Of course, this choice requires that a communication path be available for the controller to use to contact each host. If this communication is not already allowed by the policy, it must be temporarily added during the scans. As an alternative, extra network interface cards could be added to each host, providing a separate control network that is independent from the operational network. It would also be possible, although not desirable, to run each scan by hand.

Finally, network audits should be performed regularly to capture the changes in the network that will inevitably occur. The frequency of the audits depends upon the changes in

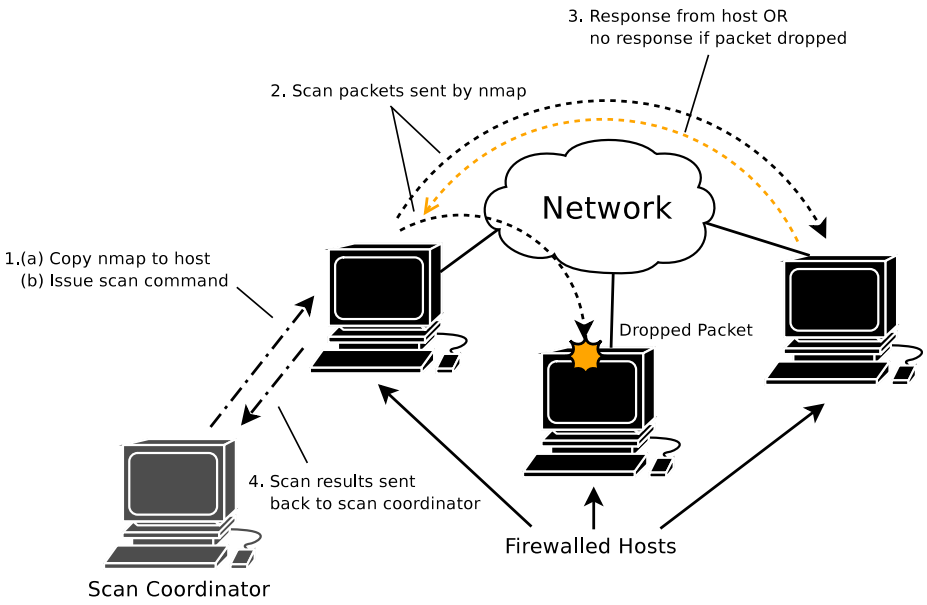


Figure 3.4 Scanning Method Used for Auditing Distributed Firewalls

the network, the demand for security, and the resources of the auditors. Schultz recommends once every three to six months [41]. Currently, there is no reason to believe that a network with distributed firewalls would require more frequent audits.

3.4.2 Optimizing scans

In some cases the `nmap` scans may take an excessively long time to complete, making an audit infeasible. The filtering performed by the firewalls is one potential reason for this slowdown. Results in [42] show that `nmap` scans are significantly slower for hosts that are filtered than for those that are unfiltered. There are three potential reasons why `nmap` would not receive a response to a probe packet it sent: the probe or the response got lost in the network, there is no host at the target address, or a firewall is blocking the probe or its response. To determine whether the packet is simply lost, `nmap` will resend the probe two times. If no response is received, the port is considered to be filtered, but it is possible that there is simply no host to respond to the probe. Usually dead host addresses are detected via Internet Control Message Protocol (ICMP) ping scans; once a host is known to be dead, no other scans need to be performed. If the firewall blocks ICMP pings, then TCP pings can be used only if a set of guaranteed unfiltered ports is known in advance. If no known ports are available for a TCP ping test then all host addresses, dead or alive, must be fully scanned to capture the entire network configuration.

In heavily filtered networks, there are a variety of ways to speed up `nmap` scans. If a known range of IP addresses is unused, then it may be excluded from the scan with the `--exclude` option. Exclusion of an IP address range should be used with caution, as it may cause open communication paths to any unknown hosts that reside in the excluded space to be missed. Another option is to set a reasonable maximum round trip time (RTT) to timeout filtered ports and dead hosts much sooner [42]. Setting the host timeout also will increase scan performance in sparse networks by halting scans for addresses that are responding to any probe packets. By default, `nmap` will not timeout any hosts, in order to prevent any inaccuracies from entering the scan. Finally, it has been shown that increasing the parallelism of the scans provides a linear gain in performance with little effect on accuracy [42].

For extremely large networks, the only feasible option may be partial scans of network segments over a period of time. Whether this strategy will provide accurate and useful results on a changing network remains an open question. Clearly, if the network can be dedicated to the audit and the policy is guaranteed not to change, then the concatenation of partial scans will provide results identical to those of a single scan.

3.4.3 Converting scans to a global view

After the network scan is complete, the scan results from each host are combined to create a global view of the network. The process is straightforward. Each individual scan is read as a set of communication paths from the scanning hosts to the hosts that are found. These communication paths and hosts are then inserted into the global view. The final result is an extensible markup language (XML) file containing a list of all the communication paths that were found in the system. Using an extensible stylesheet language transformation (XSLT) to transform the output, it is possible to compare that list to the original policy specification to find misconfigurations or unnecessary communication paths.

3.4.4 Policy language

A simple low-level policy language was created to represent the intended policy. By choosing a simple policy language representation the administrator can write the policy by hand or use automatic translation tools to convert other policy languages.

The desired policy is written as a list of tuples, which represent the allowed communication paths. Each tuple contains the source host, destination host, source port, and destination port. Generally speaking, the source port will be a random port in the ephemeral

range and is in most cases irrelevant. Figure 3.5 shows an example of what a policy would look like.

```
clientE, q1apext, Ephemeral, 20100
clientP, q1apext, Ephemeral, 20100
q1apext, clientE, Ephemeral, 9382
q1apext, clientP, Ephemeral, 9382
    q1dc, sm, Ephemeral, 1099
    q1dc, sm, Ephemeral, 20003
```

Figure 3.5 An Example of the Policy Language for the Audit Tool-Set

3.4.5 Comparing audit results to intended policy

It is easy to compare the audit results and the desired policy by locating the differences between the two files. If a communication path exists in the audit results, but does not exist in the desired policy, then the enforced policy lacks appropriate filters. If a communication path exists in the desired policy, but not in the audit results, then the enforced policy is overly restrictive. After viewing the results of the comparison, an administrator can investigate the root cause of the misconfiguration and take appropriate action to remedy the issue.

3.4.6 Creating a graph of the policy

The output from either the audit scan or the policy comparison can be converted to a graphical format for visualization. By viewing the graphical output, it is possible to observe additional policy rules that should have been included in the original policy. The graphical format also enables people without technical knowledge to comprehend and evaluate the results of the policy audit.

Figure 3.6 shows a sample output of the XML-to-dot conversion. The conversion is performed in two steps. First, the results of the audit which are in an custom XML format, are converted to dotML using an XSLT. Second, the dotML file is processed by another XSLT to create the *dot* file. The graphical output can be created from the *dot* file with any of the Graphviz [43] components.

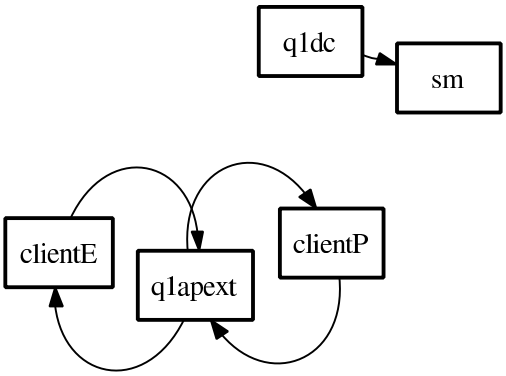


Figure 3.6 Sample Graphical Output of Network Scan

3.5 Case Study: DPASA

To verify that the audit tools and methodology work as described, they were used to audit a subset of the distributed firewall policies as part of the DPASA project. The focus of DPASA was to enhance the baseline Joint Battlespace Infosphere (JBI) with intrusion tolerance and defense [44].

As a general concept, a JBI is a platform that aims to integrate disparate command and control systems dynamically in an easy, secure manner. The Intrusion-Tolerant Joint Battlespace Infosphere (IT-JBI) uses a variety of security mechanisms, the ADF among them, to provide strong security. The next section contains a brief overview of the IT-JBI network infrastructure as currently implemented, followed by a discussion of the distributed firewall audit tools and the results of preliminary scans. In the context of this thesis, the detailed operation of the IT-JBI is inconsequential; however, a detailed description of the initial IT-JBI design can be found in [45].

3.5.1 Overview of the IT-JBI network

The IT-JBI design is separated into two distinct networks: the core and the clients. The core network is divided into four redundant quadrants, while the client network is divided into any number of individual client LANs as necessary. The basic architecture of a generic IT-JBI is shown in Figure 3.7.

Distributed firewalls are a critical component of the IT-JBI defense-in-depth strategy. Every host in DPASA is protected by an ADF that enforces a host-specific policy tailored to the exact communication needs of the host. Because only the minimal set of communications

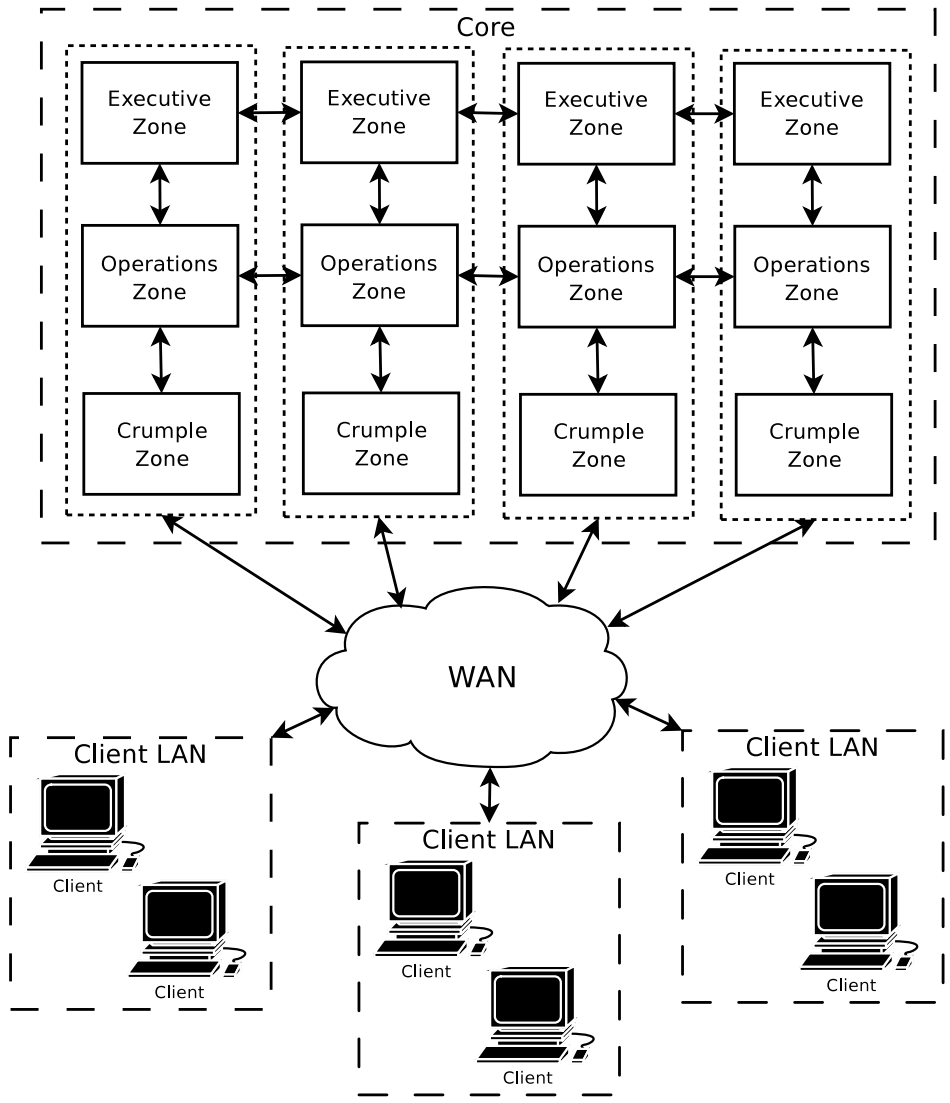


Figure 3.7 Generic IT-JBI Architecture

is allowed, security is enhanced. To protect hosts from spoofing or sniffing attacks, all traffic is protected by a VPG. Spoofing is an attack in which the attacker sends packets with incorrect source addresses. Sniffing is an attack in which the attacker silently intercepts traffic between the two hosts. With the VPG encryption, neither is possible. In response to the denial-of-service vulnerability found during the performance experiments described in Chapter 2, Cisco PIX firewalls were added to create a VPN in front of each access proxy (AP) separating it from the client LAN.

In addition, the four redundant quadrants in the core are separated by filtering switches. These filtering switches serve two purposes. During the attack-free case they restrict commu-

nication between quadrants to the minimum set required for operation. When one quadrant is deemed to be compromised, the switch can be automatically disabled, completely blocking all traffic between quadrants. As seen in Figure 3.7, the quadrants are logically separated into three zones: the crumple zone, the operations zone, and the executive zone.

The crumple zone is used to isolate the client and core networks, thus preventing direct communication between the clients and the core. Inside each crumple zone is a single AP host. The AP acts as a proxy firewall between the two networks, sanitizing incoming packets and passing the valid traffic between the two networks. The operations zone contains the hosts that perform the primary functions of the IT-JBI, specifically the publish, subscribe, and Query (PSQ) host and the downstream controller (DC) host. The executive zone contains the system Manager (SM) host, the only core host that is manned by an operator. The SM is responsible for controlling the IT-JBI.

3.5.2 Results

At the time of this writing, the DPASA project is still under development. Due to the current status of the lab configurations, the hosts on the client network were unable to participate in a distributed firewall audit. Therefore, the distributed firewall audit was only performed for the core network using a subset of the core hosts. Of the 28 core hosts, 16 (NIDS, COR, and Solaris hosts) were unable to participate in the firewall audit due to the current status of the lab. Although the remaining 12 hosts represent a portion of the DPASA system, they are sufficient in number to allow verification of the audit tools. For reference, a diagram of the core network is shown in Figure 3.8.

All scans were performed without running the DPASA software; thus, most of the unfiltered ports were reported as closed, as they did not have a listening server attached to them. Had the DPASA software been running, these ports would have been expected to be open.

Two separate scans were performed on the core network: one for which no policies were enforced, and another for which the other with the ADF policies were enforced. The results can be seen in Figures 3.9 and 3.10. The host names that correspond to the IP address can be found in Appendix B. In the figure, the red lines indicate that open ports were found during the scan. Black lines indicate that closed, but unfiltered, ports were found. Blue lines ending with a flat head indicate that `nmap` reported the list of filtered ports, instead of the usual list of open and closed ports. This occurs when a minority of ports are filtered, and the remainder are closed. For clarity, port numbers are not shown on the diagram, but they can easily be read by an administrator from the textual audit results.

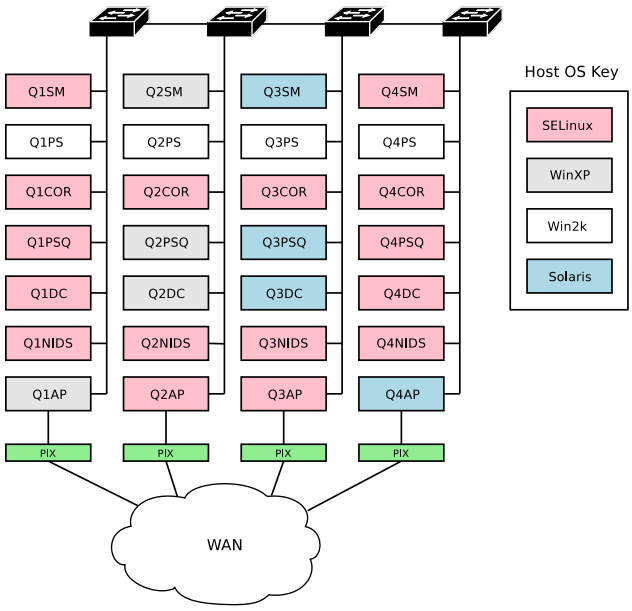


Figure 3.8 Network Topology of DPASA Core

The effect of the policy is readily apparent from the results of the two scans. Many of the interhost communication paths that had once been open were filtered, as those services were not intended to be accessible in DPASA. It is important to note that when the firewalls were enforcing, some open communication paths were found when a host was scanning itself. This was expected, and is due to the fact that the ADF only filters packets that traverse the NIC and cannot act on packets that are handled completely in the kernel.

Human analysis was required because with DPASA, the enforced policies were hand edited after being automatically generated. For that reason, there was no guarantee that the intended policy output, which was automatically generated, did not contain omissions or superfluous additions. Because the intended policy contained errors itself, the number of discrepancies between the intended policy and the audit results was inflated.

Prior to comparison, all *localhost* communication paths that existed in the audit results were removed, as they were not filtered through the distributed firewalls. The localhost communications are those between a host and itself; in other words, the source and destination are the same. Had an administrator assumed that intrahost communications would be filtered, these paths would have been identified as errors, alerting the administrator to the weakness. Without audits, such an assumption might have reduced system security.

After the *localhost* communication paths were removed from the audit results, the audit detected 133 potential errors and verified 54 communication paths. Of the 133 potential errors, 56 were classified as *extra* communication paths, as they existed in the audit but not

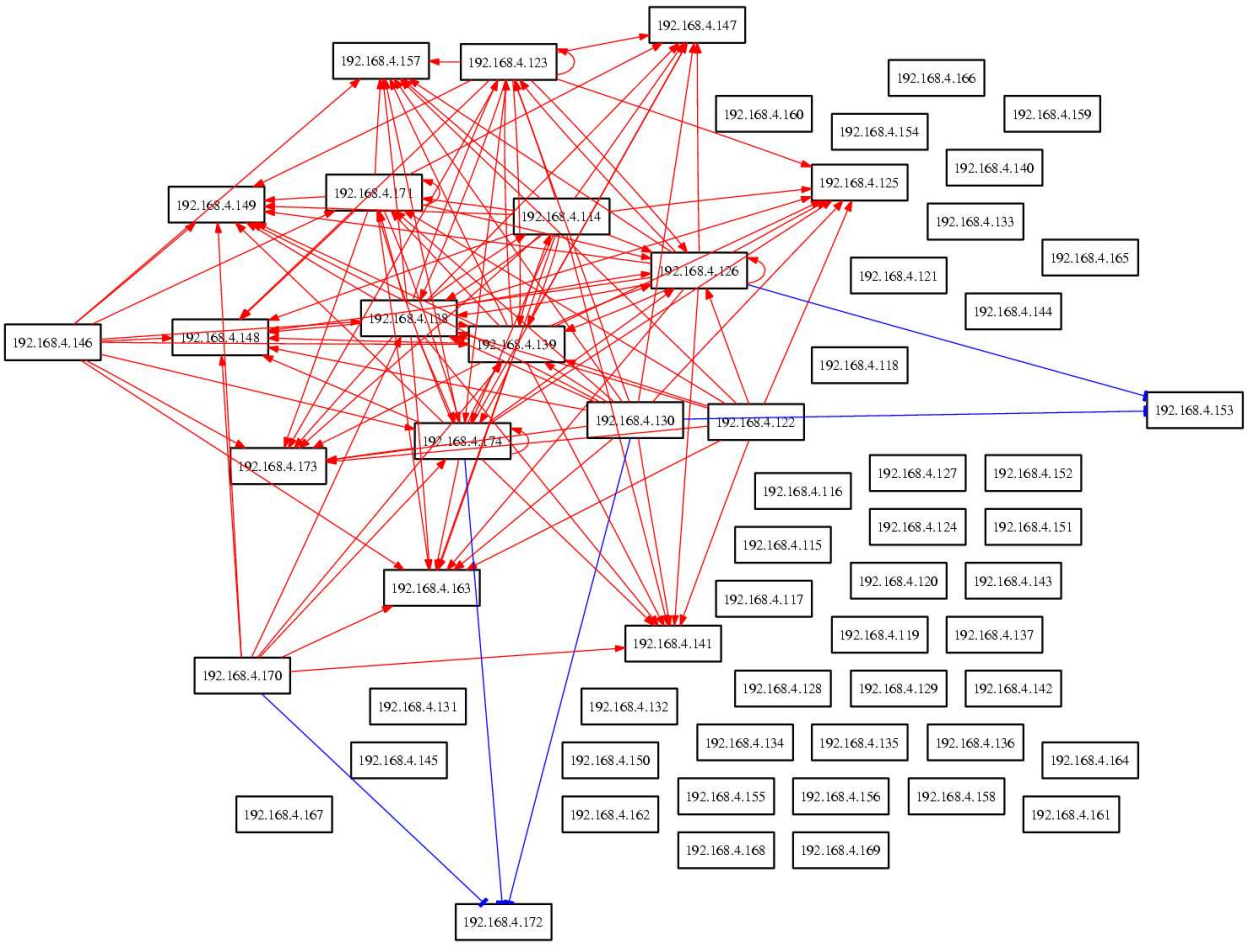


Figure 3.9 Scan Results of the Unenforced Network Configuration

the intended policy. The other 77 were classified as *blocked* communication paths, as they existed in the intended policy but not the audit.

Upon examination, many of the *extra* errors were found to be irrelevant, because the intended policy had omissions that resulted from the manual edits of the ADF policies. A few, however, were interesting, as they violated the original inter-quad communication paths that were not in the DPASA design plan. For example, Q1SM was able to communicate with Q2PS via unfiltered ports 2079 and 8567. Further examination revealed that these inter-quad paths were the side effect of manual edits, but were not added to the intended policy file. All of the *blocking* errors were false alarms caused by inaccuracies in the scan or hosts with known problems.

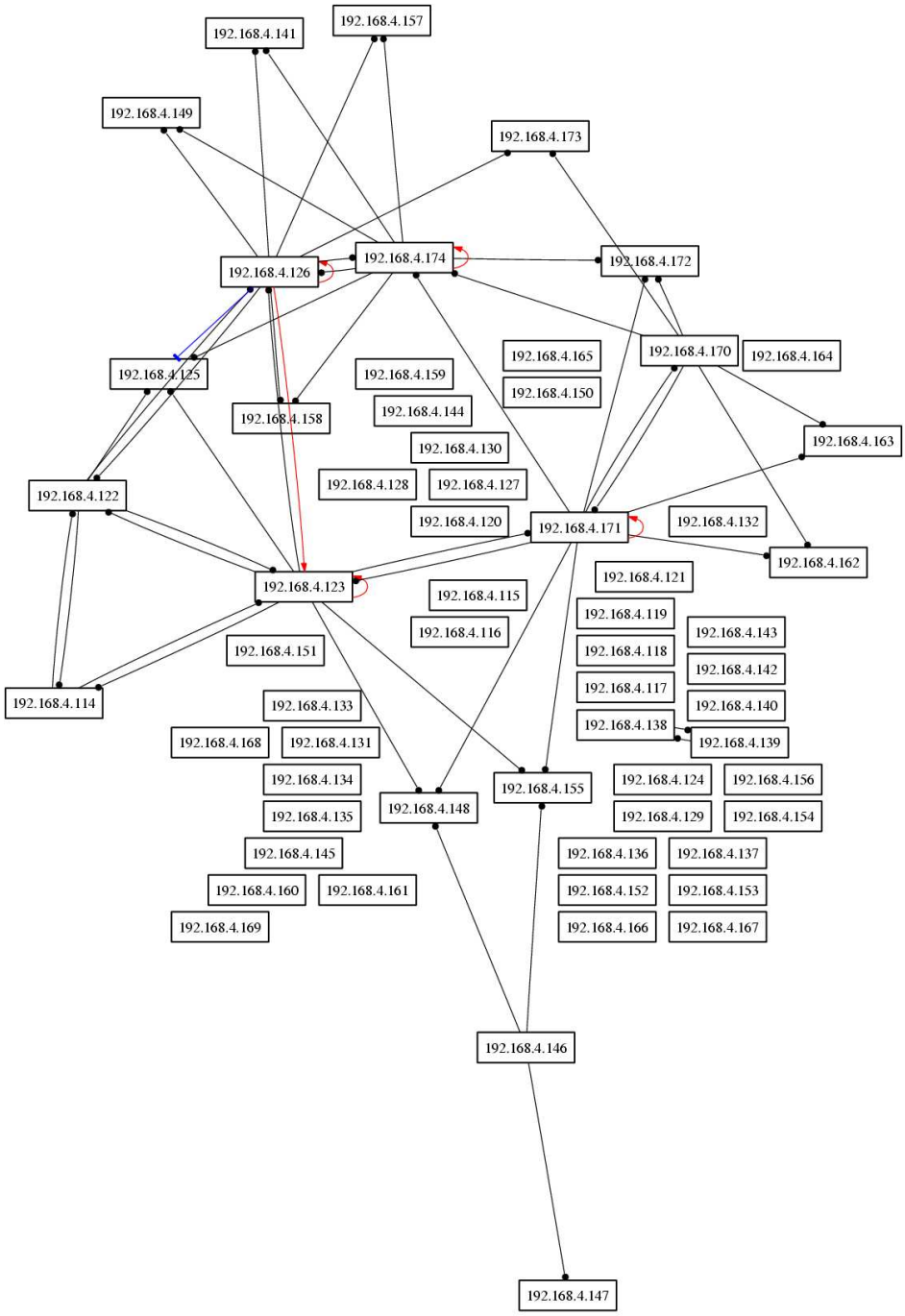


Figure 3.10 Scan Results of the Enforced Network Configuration

3.5.3 Discussion

By using a real networked system, i.e., DPASA, as a case study, it was possible to analyze the limitations of the audit tools. Although no errors were found in the DPASA policies, it

would be wrong to conclude that the audit was unnecessary. The primary motivation of an audit is the validation of the implementation. Without an audit, there is no evidence that the policies are actually being enforced correctly.

As previously mentioned, the excessive time required to complete the audit of a complex network can be prohibitive; DPASA was no exception. Through use of some optimization techniques, scan times were reduced to approximately two hours per host. When no optimizations were used a scan would sometimes take upwards of 10 h. The following optimizations were used for the DPASA scans:

- Scans were set to scan only the applicable address space (192.168.4.114-174) instead of the entire address space (192.168.4.1-254).
- The most aggressive `nmap` timing policy was selected (`-T5`).
- Port scans were parallelized (`--min_parallelism=25`).
- Host timeout was set to 1000 s (`--host_timeout=1000000`).
- Maximum RTT was set to 25 ms (`--max_rtt_timeout=25`).
- The scan manager parallelized eight scans simultaneously.

The particular values for the various optimizations were arrived at through trial-and-error. When the optimizations were set too high, the scan results were inaccurate and some unfiltered communication paths would not be detected. The appropriate optimization will be different for different situations; if there is doubt, one can use conservative values for the initial audit (guaranteeing accurate results) and then use more aggressive optimizations for later audits.

Although the process of comparing the intended policy to the audit results was automated, locating the source of each potential error required human analysis. That was the desired operation, as the audit should simply inform the administrator of the discrepancy, and then leave the resolution to the administrator. Automatic resolution may be possible, but only when the intended policy is formally proven to be free of errors.

Similarly, the graphical depiction of the scan results could be significantly improved. The current method, using Graphviz, suffers from two problems. First, the layout of the graph is automatically determined. For complex networks, the graphic results can therefore be convoluted and difficult to read. Second, the graphic is static, preventing an administrator from dynamically displaying or hiding information. Despite these limitations, the graphics can still be useful, as they provide a general feel for the degree of policy enforcement.

CHAPTER 4

FUTURE WORK AND CONCLUSIONS

4.1 Future Work

Future work can be classified into three main areas: performance evaluation, embedded distributed firewall design, and distributed firewall policy construction and validation.

The results in Chapter 2 indicate that current commercially available NIC-based distributed firewalls suffer from inadequate performance. As more vendors enter the distributed firewall market, it will be important to evaluate each device independently in a manner similar to that used in this thesis. However, it is hoped that this thesis has brought light to the problem, thus preventing future NIC-based firewalls from suffering the same performance limitations as the EFW or ADF. Thus, the second promising area of future research is the design and implementation of new embedded firewall architectures.

Clearly, new packet-filtering techniques must be developed to endow NIC-based firewalls with sufficient performance at a reasonable cost that assures widespread acceptance. If the performance can be increased, not only will the firewalls be safe from packet floods, but more advanced filtering techniques, such as stateful inspection, could be added to further increase the value of embedded distributed firewalls. Whether this performance gain will be the result of algorithmic changes or a completely new architecture is an open question. However, given that network speeds are continually increasing, it appears that a new architecture, like FPGA-based packet filters, will be needed to provide the requisite performance.

Finally, the effort required to validate and audit distributed firewall policies must be reduced to the point that it is usable widely by all administrators. This problem can be approached from many fronts. First, better policy construction tools must be created to aid administrators in creating correct policies *before* they are deployed. Although many efforts

have begun down this path, none has yet achieved a solution that has gained wide acceptance. Second, the policy audit tools presented in this thesis currently require a significant amount of user intervention to gather and interpret the results. Any efforts to remedy these problems would greatly increase the usefulness of audit tools.

4.2 Conclusions

This thesis presented the experimental evaluations of two NIC-based, embedded distributed firewalls: the EFW (from 3COM), and the ADF (developed by Secure Computing). The first group of experiments evaluated the performance and flood tolerance of the two NICs. The second experiment evaluated the network audit tool that was designed for this to audit networks with distributed firewalls, using DPASA as a case study.

Results of the performance experiments indicate that both firewall NICs had insufficient performance to be used safely without risking a denial-of-service attack. It was determined that on a 100 Mbps network an attacker could easily mount a denial-of-service attack against either firewall NIC. For example, the results show that an ADF enforcing the largest rule-set possible loses 65% of the full bandwidth capacity of a regular 100 Mbps Ethernet. From the attacker's perspective, this means a successful denial-of-service attack could be mounted against an ADF enforcing the largest rule-set with a flood of only 4500 packets/s. In addition to the risk of attack, the performance loss may significantly affect the services running on the network. At its worst, when compared to a standard NIC, the ADF was responsible for a 41% performance decrease in web server performance.

Still, in light of these shortcomings, the authors believe that either the EFW or ADF can be valuable pieces of the puzzle in forming a complete network security plan, as they provide additional depth to the defenses and noncircumventable protection. In fact, when combined with a solid defense-in-depth strategy and the mitigations presented in this thesis, the EFW and ADF can be safely used with reduced risk of flood attacks.

Results of the audit experiment, using DPASA as a case study, indicated that the methodology and tools developed in this thesis are suitable for use on a complex network with heavily filtered hosts. The primary goal of the audit tools is to aid administrators and developers in the validation of the enforced policies. By auditing the network, it is possible to identify all communication paths and then verify them against the desired security posture. Although the audit tools are usable, they lack an easy-to-use interface to control the scans. In addition, the audit itself can be time-consuming to configure and run. These limitations should be addressed in future efforts.

Finally, the methodologies used for both experiments are significant contributions of this thesis. Each methodology was designed to be useful beyond the EFW or ADF. The hope is that future research will continue to improve the methodologies, for embedded distributed firewalls appear to be important tools for achieving high levels of security for modern networks.

In general, the thesis results support the notion that effective security is not simply created by deploying a security mechanism. The importance of validating the device and auditing the configuration must not be underestimated. In the case of the EFW and ADF, the unvalidated belief that an embedded firewall must provide superior performance [46] was at least partially responsible for the denial-of-service vulnerability identified by the experiments. Security in future complex systems will be enhanced not just by new security mechanisms but also through experimental evaluation, formal methods, probabilistic modeling, and policy auditing.

APPENDIX A

FLOOD GENERATOR SOURCE CODE

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <assert.h>
5  #include <libnet.h>
6  #include <argp.h>
7  #include <unistd.h>
8  #include <netinet/tcp.h>
9  #include <netinet/ip.h>
10 #include <net/ethernet.h>
11 #include <sys/time.h>
12 #include <sys/resource.h>
13 #include <net/if_arp.h>
14
15 #ifdef DEBUG
16 #define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
17 #else
18 #define PDEBUG(fmt, args...) /* do nothing */
19 #endif
20
21 /*
       ================================================================
        */
22 /* Arugment Parsing Setup
                                                                  */
23 /*
       ================================================================
        */
24 const char *argp_program_version =
25   "bork_test 1.1";
26 const char *argp_program_bug_address =
27   "<ihde@uiuc.edu>";
28
29 /* Documentation */
30 static char doc[] =
31   "bork_adf - A set of attacks which can be used against an ADF NIC";
32
```

```
33 /* Argument description */
34 static char args_doc[] = "target_ip target_mac";
35
36 /* The available options */
37 static struct argp_option options[] = {
38         {"attack_type",         'a', "TYPE", 0, "The type of attack to
              run : (F)lood, (C)lam" },
39         {"rate",                'r', "RATE", 0, "Set the initial packet
               rate" },
40         {"frame_size",          'f', "FRAME_SIZE", 0, "Set the frame
              size used for the packets (>64)" },
41         {"out_interface",       'o', "INTERFACE", 0, "The outgoing
              interface to send from" },
42         {"use_tcp",             't', NULL, 0, "Use TCP packets instead
              of UDP" },
43         {"dst_port",            'p', "DSTPORT", 0, "Destination port
              for packets" },
44         {"spoof_vpg",           's', NULL, 0, "Make IP-Proto 174
              packets that appear to be VPG traffic"},
45         {"spoof_ip",            'i', "IP", 0, "Make the packets appear
              to be from IP"},
46         {"spoof_mac",           'm', "MAC", 0, "Make the packets appear
               to be from MAC"},
47         { 0 }
48 };
49
50 /* A structure to hold the parsed arguments */
51 struct arguments
52 {
53         char *target;           /* The target IP to test */
54         char *macdst;           /* The MAC of the target */
55         char *out_i;            /* The interface to send test packets
              from */
56         char *spoof_ip;
57         char *spoof_mac;
58         char attack_type;
59         int rate;               /* The rate at which to send packets */
60         int frame_size;         /* Set the frame size */
61         int dst_port;
62         int use_tcp;
63         int spoof_vpg;
64 };
65
66 /* The parsing function */
67 static error_t parse_opt (int key, char *arg, struct argp_state *state)
68 {
69         struct arguments *arguments = state->input;
70
71         switch (key)
72         {
73                 case 'a':
74                         arguments->attack_type = arg[0];
75                         switch (arguments->attack_type)
76                         {
77                                 case 'F':
```

```
78                                      PDEBUG("Attack type - Flood\n")
                                            ;
79                                      break;
80                              case 'C':
81                                      PDEBUG("Attack type - Clam\n");
82                                      break;
83                              default:
84                                      fprintf(stderr, "Unknown attack
                                            type %c specified\n",
                                           arguments->attack_type);
85                                      return ARGP_ERR_UNKNOWN;
86                                      break;
87                      }
88                      break;
89
90              case 'r':
91                      arguments->rate = atoi(arg);
92                      /* The best rate we can get is 10ms with itimer
                            ,
93                       * so the fastest packet rate is 100000
94                       */
95                      if (arguments->rate > 100000)
96                      {
97                              fprintf(stderr, "Rate higher than
                                    maximum timer resolution, setting to
                                     100000\n");
98                              arguments->rate = 100000;
99                      }
100                     PDEBUG("Packet rate set to %d\n", arguments->
                            rate);
101                     break;
102             case 'f':
103                     arguments->frame_size = atoi(arg);
104                     if (arguments->frame_size < 64)
105                     {
106                             fprintf(stderr, "Frame size too small,
                                    setting to 64 bytes");
107                             arguments->frame_size = 64;
108                     }
109                     PDEBUG("Frame size set to %d\n", arguments->
                            frame_size);
110                     break;
111             case 'o':
112                     arguments->out_i = arg;
113                     PDEBUG("Output interface set to %s\n",
                            arguments->out_i);
114                     break;
115             case 'i':
116                     arguments->spoof_ip = arg;
117                     PDEBUG("Spoof packets from IP %s\n", arguments
                            ->spoof_ip);
118                     break;
119             case 'm':
120                     arguments->spoof_mac = arg;
121                     PDEBUG("Spoof packets from MAC %s\n", arguments
                            ->spoof_mac);
```

```
122                         break;
123
124                 case 't':
125                         arguments->use_tcp = 1;
126                         PDEBUG("Using tcp packets instead of udp\n");
127                         break;
128                 case 'p':
129                         PDEBUG("Destination Port %s\n", arg);
130                         arguments->dst_port = atoi(arg);
131                         PDEBUG("Destination Port %d\n", arguments->
                               dst_port);
132                         break;
133                 case 's':
134                         PDEBUG("Spoofing VPG traffic\n");
135                         arguments->spoof_vpg = 1;
136                         break;
137
138                 case ARGP_KEY_ARG:
139                         if (state->arg_num >= 2)
140                                 argp_usage(state);
141                         else if (state->arg_num == 0)
142                         {
143                                 arguments->target = arg;
144                                 PDEBUG("Targeted host is %s\n",
                                       arguments->target);
145                         }
146                         else if (state->arg_num == 1)
147                         {
148                                 arguments->macdst = arg;
149                                 PDEBUG("Targeted host has MAC %s\n",
                                       arguments->macdst);
150                         }
151                         break;
152                 case ARGP_KEY_END:
153                         if (state->arg_num < 2)
154                                 argp_usage(state);
155                         break;
156                 default:
157                         return ARGP_ERR_UNKNOWN;
158         }
159         return 0;
160 };
161
162 /* The argument parser */
163 static struct argp argp= { options, parse_opt, args_doc, doc };
164
165 /*
        ================================================================
         */
166 /* Global Variables
                                                                       */
167 /*
        ================================================================
         */
168
169 static libnet_t *l = NULL; /* The libnet context */
```

```
170
171 static long packets_sent = 0;
172
173 static struct arguments arguments;
174
175 static int packets_per_signal = 1;
176
177 static struct timeval program_start_time;
178
179 /*
    ========================================================================
     */
180 /* Local Function Definitions
                                                    */
181 /*
    ========================================================================
     */
182 int begin_flood(char* target, char* macdst, int rate, int frame_size);
183 void stop_flood();
184 int send_test_packet(libnet_t *l, char *target, int frame_size);
185 void sigint(int signal);
186 void clam_adf(char* target, char* macdst);
187
188 /*
    ========================================================================
     */
189 /* Useful constants
                                                                  */
190 /*
    ========================================================================
     */
191 const int adf_dst_port = 2083;
192 const int adf_src_port = 2081;
193
194 /*
    ========================================================================
     */
195 /* Local Functions
                                                                   */
196 /*
    ========================================================================
     */
197
198 #define INJECTION_TYPE LIBNET_LINK
199
200 /**
201  * Initialize libnet
202  */
203 void initialize(char *out_i)
204 {
205         char errbuf[LIBNET_ERRBUF_SIZE];
206
207         assert(l == NULL);
208
209         /* Initialize the libnet library */
210         l = libnet_init(INJECTION_TYPE, out_i, errbuf);
```

```
211         if (l == NULL)
212         {
213                 fprintf(stderr, "libnet_init() failed: %s\n", errbuf);
214                 exit(-1);
215         }
216         PDEBUG("Libnet initialized\n");
217 }
218
219 /**
220  * Destroy libnet
221  */
222 void cleanup()
223 {
224         if (l != NULL)
225         {
226                 libnet_destroy(l);
227                 PDEBUG("Libnet destroyed\n");
228         }
229 }
230
231 /**
232  * The main function
233  */
234 int main(int argc, char* argv[])
235 {
236
237         /* Set the defaults for the arguments */
238         arguments.frame_size = 64;
239         arguments.rate = 0;
240         arguments.use_tcp = 0;
241         arguments.spoof_vpg = 0;
242         arguments.out_i = "lo";
243         arguments.spoof_ip = "";
244         arguments.spoof_mac = "";
245         arguments.attack_type = 0;
246         arguments.dst_port = adf_dst_port;
247
248         /* parse the command-line arguments */
249         argp_parse(&argp, argc, argv, 0, 0, &arguments);
250
251         /* Initialize the required libraries */
252         initialize(arguments.out_i);
253
254         /* Check the attack type, only the first character is
               significate */
255         if (arguments.attack_type == 'F')
256         {
257                 signal(SIGINT, sigint);
258                 begin_flood(arguments.target, arguments.macdst,
                        arguments.rate, arguments.frame_size);
259                 /* We should never reach here */
260                 assert(1 != 1);
261         }
262         else if (arguments.attack_type == 'C')
263         {
264                 clam_adf(arguments.target, arguments.macdst);
```

```
265         }
266
267         return 0;
268 }
269
270 /**
271  * This is the callback function when the timer expires, and is used
272  * to send a test packet.
273  * @param signal ???
274  */
275 void f (int signal)
276 {
277         int i;
278
279         PDEBUG("Caught SIGALRM\n");
280         for (i = 0; i < packets_per_signal; i++)
281         {
282                 send_test_packet(l, arguments.target, arguments.
                        frame_size);
283                 packets_sent++;
284         }
285 }
286
287 /**
288  * This function stops the experiment
289  */
290 void stop_flood()
291 {
292         struct itimerval value;
293         struct timeval program_end_time;
294         float flood_duration;
295         float tx_rate;
296
297         /* Stop sending packets */
298         value.it_interval.tv_sec = 0;
299         value.it_interval.tv_usec = 0;
300         value.it_value.tv_sec = 0;
301         value.it_value.tv_usec = 0;
302
303         /* Install signal handlers */
304         PDEBUG("Stoping timer\n");
305         setitimer(ITIMER_REAL, &value, NULL);
306
307         /* Get the time the flood was stopped */
308         if (gettimeofday(&program_end_time, NULL) == -1)
309         {
310                 fprintf(stderr, "gettimeofday(): failed\n");
311         }
312         flood_duration = (float)(program_end_time.tv_sec -
                program_start_time.tv_sec);
313         PDEBUG("Start: %d, %d\n", (int)program_start_time.tv_sec,
                program_start_time.tv_usec);
314         PDEBUG("End: %d, %d\n", (int)program_end_time.tv_sec,
                program_end_time.tv_usec);
315         if (program_end_time.tv_usec > program_start_time.tv_usec)
316         {
```

```
317                 flood_duration = flood_duration + ((float)(
                        program_end_time.tv_usec-program_start_time.tv_usec)
                        /1000000);
318         }
319         else
320         {
321                 flood_duration = flood_duration -1 +((float)(
                        program_start_time.tv_usec-program_end_time.tv_usec)
                        /1000000);
322         }
323         tx_rate = (float)packets_sent / flood_duration;
324
325         printf("Total Packets Sent : %ld\n", packets_sent);
326         printf("Flood Duration : %f\n", flood_duration);
327         printf("Packets Tx Rate : %f\n", tx_rate);
328
329         exit(0);
330 }
331
332 /**
333  * This is the callback when the user attempts to interrupt the program
        .
334  * @param signal ???
335  */
336 void sigint(int signal)
337 {
338         if (arguments.attack_type == 'F')
339         {
340                 PDEBUG("Caught SIGINT\n");
341                 stop_flood();
342         }
343 }
344
345 int begin_flood(char* target, char* macdst, int rate, int frame_size)
346 {
347         int r;
348         struct itimerval value;
349
350         /* Reset the number of packets sent */
351         packets_sent = 0;
352
353         printf("Starting flood of size %d\n", frame_size);
354
355         /* Capture the time for the start of the flood */
356         if (gettimeofday(&program_start_time, NULL) == -1)
357         {
358                 fprintf(stderr, "gettimeofday(): failed\n");
359         }
360
361         if (rate == 0)
362         {
363                 /* No rate given so lets fly as fast as we can */
364                 while (1 == 1)
365                 {
366                         send_test_packet(l, target, frame_size);
367                         packets_sent++;
```

```
368                     }
369             }
370             else
371             {
372                     /* Calculate the packet sending interval */
373                     long packet_interval_usec = (1.0 / rate) * 1000000;
374
375                     PDEBUG("interval %ld\n", packet_interval_usec);
376                     /* If the packet_interval_user > 1ms then we need to
                           send multiple packets
377                     * per ms, this is done to prevent using all the CPU as
                           a 1us time is the
378                     * max resolution...it would be possible to run every
                           500us */
379                     if (packet_interval_usec < 1999)
380                     {
381                             packet_interval_usec = 1999;
382                             packets_per_signal = rate / 500;
383                     }
384
385                     PDEBUG("interval %ld : per_sig %d\n",
                           packet_interval_usec, packets_per_signal);
386
387                     /* Prepare the timer */
388                     value.it_interval.tv_sec = 0;
389                     value.it_interval.tv_usec = packet_interval_usec;
390                     value.it_value.tv_sec = 0;
391                     value.it_value.tv_usec = packet_interval_usec;
392
393                     /* Install signal handlers */
394                     signal(SIGALRM, f);
395
396                     PDEBUG("Setting timer\n");
397                     r = setitimer(ITIMER_REAL, &value, NULL);
398
399                     while (1 == 1) { pause(); } /* Pause so we yeild CPU
                           time back */
400
401             }
402     }
403
404     /**
405      * Generates a payload to send in the test packet.  This contains a
           unique
406      * serial number and may eventually contain a timestamp.  The first
           byte
407      * in all test packet payloads is 0xDE, the next 8 bytes are the serial
408      * number, and the remainder of the bytes are filled with 0xAE.
409      *
410      * @param payload_len the size of the payload to create.
411      */
412     char* generate_payload(int payload_len)
413     {
414             char* payload = NULL;
415
```

```
416         /* Allocate memory for the payload, the caller is responsible
                for freeing it */
417         payload = calloc(payload_len, sizeof(char));
418         if (payload == NULL)
419         {
420                 fprintf(stderr, "Could not allocate payload\n");
421                 exit(0);
422         }
423         /* Set all the bytes in payload to 0xAE */
424         memset(payload, 0xAE, payload_len);
425
426         /* Set the first byte to DE for no particular reason :) */
427         memset(payload, 0xDE, 1);
428
429         return payload;
430 }
431
432
433 /**
434  * Sends a testpacket via libnet to target with a desired frame_size.
435  *
436  * @param l the libnet context.
437  * @param target the target ip or hostname.
438  * @param frame_size the desired ethernet frame size.
439  *
440  */
441 int send_test_packet(libnet_t *l, char *target, int frame_size)
442 {
443         static libnet_ptag_t tcp = 0;
444         static libnet_ptag_t udp = 0;
445         static libnet_ptag_t ip = 0;
446         static libnet_ptag_t eth = 0;
447
448         int c = 0;                      /* The result from writing the
                packet */
449
450         char *payload = NULL;
451         u_short payload_s = 0;
452
453         /* Who would do such a thing? */
454         assert(l != NULL);
455         assert(target != NULL);
456         /* The program should not allow frame sizes that are to small
                */
457         assert(frame_size > (LIBNET_TCP_H + LIBNET_IPV4_H +
                LIBNET_ETH_H));
458         assert(frame_size > (LIBNET_UDP_H + LIBNET_IPV4_H +
                LIBNET_ETH_H));
459
460         /* Calculate the payload length needed to create the desired
                frame size */
461         u_long length = 0;
462         /* Build the tcp contents, always referencing the last packet
                tag */
463         u_long src_prt = adf_src_port;
464         u_long dst_prt = arguments.dst_port;
```

```
465
466         if (arguments.use_tcp == 0)
467         {
468                 payload_s = frame_size - LIBNET_UDP_H - LIBNET_IPV4_H -
                        LIBNET_ETH_H;
469                 PDEBUG("Required payload length %d = %d - %d - %d\n",
470                         payload_s, frame_size, LIBNET_UDP_H,
                                LIBNET_IPV4_H);
471
472                 length = LIBNET_UDP_H + payload_s;
473
474                 /* Generate Payload */
475                 payload = generate_payload(payload_s);
476
477                 PDEBUG("Preparing to build UDP - src:%d, dst:%d, len:%d
                        \n",
478                         (int)src_prt, (int)dst_prt, (int)length);
479                 udp = libnet_build_udp(
480                         src_prt,
481                         dst_prt,
482                         length,
483                         0,
484                         payload,
485                         payload_s,
486                         l,
487                         udp
488                 );
489                 if (udp < 0)
490                 {
491                         fprintf(stderr, "libnet_build_udp() failed: %s\
                                n", libnet_geterror(l));
492                         free(payload);
493                         exit(-1);
494                 }
495                 free(payload);
496         }
497         else
498         {
499                 payload_s = frame_size - LIBNET_TCP_H - LIBNET_IPV4_H -
                        LIBNET_ETH_H;
500                 PDEBUG("Required payload length %d = %d - %d - %d\n",
501                         payload_s, frame_size, LIBNET_TCP_H,
                                LIBNET_IPV4_H);
502                 length = LIBNET_TCP_H + payload_s;
503
504                 /* Generate Payload */
505                 payload = generate_payload(payload_s);
506
507
508                 PDEBUG("Preparing to build TCP - src:%d, dst:%d, len:%d
                        \n",
509                         (int)src_prt, (int)dst_prt, (int)length);
510                 tcp = libnet_build_tcp(
511                         src_prt,
512                         dst_prt,
513                         0x01010101,
```

```
514                         0x02020202,
515                         TH_SYN,
516                         32767,
517                         0,
518                         10,
519                         length,
520                         payload,
521                         payload_s,
522                         l,
523                         tcp
524                 );
525                 if (tcp < 0)
526                 {
527                         fprintf(stderr, "libnet_build_tcp() failed: %s\
                            n", libnet_geterror(l));
528                         free(payload);
529                         exit(-1);
530                 }
531                 free(payload);
532         }
533
534         /* Build the IPV4 contents, only if needed */
535         if (ip == 0)
536         {
537                 u_long ipv4_len = LIBNET_IPV4_H + length;
538
539                 u_int32_t src_ip = libnet_get_ipaddr4(l); /* The
                            address of the device we intialized with */
540                 u_int32_t dst_ip = libnet_name2addr4(l, arguments.
                            target, LIBNET_RESOLVE);
541
542                 /* If we are to spoof the source IP, do it */
543                 if (strcmp(arguments.spoof_ip, "") != 0)
544                 {
545                         src_ip = libnet_name2addr4(l, arguments.
                                    spoof_ip, LIBNET_RESOLVE);
546                 }
547
548                 u_int8_t prot = 0;
549                 if (arguments.spoof_vpg == 1)
550                 {
551                         prot = 0xAE;
552                 }
553                 else
554                 {
555                         if (arguments.use_tcp == 0)
556                         {
557                                 prot = IPPROTO_UDP;
558                         }
559                         else
560                         {
561                                 prot = IPPROTO_TCP;
562                         }
563                 }
564                 PDEBUG("Building IPV4 len: %d\n", (int)ipv4_len);
565                 ip = libnet_build_ipv4(
```

```
566                     ipv4_len,
567                     0,
568                     242,
569                     0,
570                     64,
571                     prot,
572                     0,
573                     src_ip,
574                     dst_ip,
575                     NULL,
576                     0,
577                     1,
578                     0
579             );
580             if (ip < 0)
581             {
582                     fprintf(stderr, "libnet_build_ipv4() failed: %s
                           \n", libnet_geterror(l));
583                     exit(-1);
584             }
585     }
586     /* Build the ethernet frame, only if needed */
587     if ((INJECTION_TYPE == LIBNET_LINK) && (eth == 0))
588     {
589             PDEBUG("Building Ethernet Frame\n");
590             /* Get the hwaddress of our sender */
591             struct libnet_ether_addr *src_ether = libnet_get_hwaddr
                   (l);
592             int len = 0;
593
594             u_int8_t *enet_src = src_ether->ether_addr_octet;
595             if (strcmp(arguments.spoof_mac, "") != 0)
596             {
597                     enet_src = libnet_hex_aton((int8_t*)arguments.
                           spoof_mac, &len);
598             }
599             if (enet_src == NULL)
600             {
601                     fprintf(stderr, "libnet_hex_aton() failed: %s\n
                           ", libnet_geterror(l));
602             }
603
604             u_int8_t *enet_dst = libnet_hex_aton((int8_t*)arguments
                   .macdst, &len);
605             if (enet_dst == NULL)
606             {
607                     fprintf(stderr, "libnet_hex_aton() failed: %s\n
                           ", libnet_geterror(l));
608             }
609
610             eth = libnet_build_ethernet(
611                     enet_dst,
612                     enet_src,
613                     ETHERTYPE_IP,
614                     NULL,                   /* payload */
615                     0,                      /* payload size */
```

```
616                         l,                          /* libnet handle */
617                         0                           /* libnet id */
618                 );
619
620                 free(enet_dst);
621                 if (eth < 0)
622                 {
623                         fprintf(stderr, "libnet_build_ethernet() failed
                               : %s\n", libnet_geterror(l));
624                         exit(-1);
625                 }
626         }
627
628         c = libnet_write(l);
629         if (c < 0)
630         {
631                 fprintf(stderr, "libnet_write() faild: %s\n",
                           libnet_geterror(l));
632                 exit(-1);
633         }
634         return 0;
635 }
636
637 /**
638  * Generates a payload by reading the captured packet.  Currently is
        hardcoded
639  */
640 char* generate_clam_payload()
641 {
642         char* payload = NULL;
643         int payload_len = 76;
644         /* Eventually this should be input from command line, or better
                yet being sniffed */
645         char sample_payload[76] =
646                 {0x01, 0x05, 0x1f, 0x00, 0x00, 0x00,
647                  0x00, 0x00, 0x55, 0xaf, 0xce, 0x61, 0x0c, 0x4f, 0x65,
                            0x96, 0x5e, 0x47, 0x6b, 0xea, 0x6b, 0x19,
648                  0xff, 0x10, 0xc4, 0xd3, 0x70, 0x7c, 0x6e, 0xea, 0xc7,
                            0xd9, 0xbc, 0xb8, 0x99, 0x0e, 0x97, 0xf7,
649                  0x4c, 0x97, 0x49, 0x18, 0x18, 0x6d, 0xd6, 0x3b, 0x62,
                            0x5e, 0xf3, 0xdc, 0x0c, 0x3a, 0xd1, 0xa5,
650                  0xf5, 0x74, 0xf2, 0xbb, 0x76, 0x2f, 0xfe, 0x1a, 0x98,
                            0x17, 0x36, 0xac, 0x90, 0x21, 0x5d, 0x64,
651                  0x1e, 0xb1, 0x9f, 0xb1, 0xb2, 0x91};
652
653         /* Allocate memory for the payload, the caller is responsible
                for freeing it */
654         payload = calloc(payload_len, sizeof(char));
655         if (payload == NULL)
656         {
657                 fprintf(stderr, "Could not allocate payload\n");
658                 exit(0);
659         }
660         /* Set all the bytes in payload to 0x00 */
661         memset(payload, 0x00, payload_len);
662
```

```
663         /* Copy the sniffed bytes into the payload */
664         memcpy(payload, sample_payload, payload_len*sizeof(char));
665 
666         return payload;
667 }
668 /**
669  * Create a ADF UDP packet that is identical to the one sent from the
        policy
670  * server to block all traffic on the NIC.
671  */
672 void clam_adf(char* target, char* macdst)
673 {
674         static libnet_ptag_t udp = 0;
675         static libnet_ptag_t ip = 0;
676         static libnet_ptag_t eth = 0;
677 
678         int c = 0;                      /* The result from writing the
                packet */
679 
680         char *payload = NULL;           /* A pointer to the payload */
681         u_short payload_s = 0;          /* The length of the payload */
682 
683         /* Who would do such a thing? */
684         assert(l != NULL);
685         assert(target != NULL);
686         assert(macdst != NULL);
687 
688         /* Build the tcp contents, always referencing the last packet
                tag */
689         u_long src_prt = adf_src_port;
690         u_long dst_prt = adf_dst_port;
691 
692         payload_s = 76;
693         int length = LIBNET_UDP_H + payload_s;
694         /* Generate Payload */
695         payload = generate_clam_payload();
696 
697         PDEBUG("Preparing to build UDP - src:%d, dst:%d, len:%d\n",
698                 (int)src_prt, (int)dst_prt, (int)length);
699         udp = libnet_build_udp(
700                 src_prt,
701                 dst_prt,
702                 length,
703                 0,
704                 payload,
705                 payload_s,
706                 l,
707                 udp
708         );
709         if (udp < 0)
710         {
711                 fprintf(stderr, "libnet_build_udp() failed: %s\n",
                        libnet_geterror(l));
712                 free(payload);
713                 exit(-1);
714         }
```

```
715         free(payload);
716
717         /* Build the IPV4 contents, only if needed */
718         if (ip == 0)
719         {
720                 u_long ipv4_len = LIBNET_IPV4_H + length;
721                 u_int8_t prot = IPPROTO_UDP;
722                 u_int32_t src_ip = libnet_get_ipaddr4(l); /* The
                        address of the device we intialized with */
723                 u_int32_t dst_ip = libnet_name2addr4(l, target,
                        LIBNET_RESOLVE);
724
725                 if (strcmp(arguments.spoof_ip, "") != 0)
726                 {
727                         src_ip = libnet_name2addr4(l, arguments.
                                spoof_ip, LIBNET_RESOLVE);
728                 }
729
730                 PDEBUG("Building IPV4 len: %d\n", (int)ipv4_len);
731                 ip = libnet_build_ipv4(
732                         ipv4_len,
733                         0,
734                         242,
735                         0,
736                         64,
737                         prot,
738                         0,
739                         src_ip,
740                         dst_ip,
741                         NULL,
742                         0,
743                         l,
744                         0
745                 );
746                 if (ip < 0)
747                 {
748                         fprintf(stderr, "libnet_build_ipv4() failed: %s
                                \n", libnet_geterror(l));
749                         exit(-1);
750                 }
751         }
752         /* Build the ethernet frame, only if needed */
753         if ((INJECTION_TYPE == LIBNET_LINK) && (eth == 0))
754         {
755                 PDEBUG("Building Ethernet Frame\n");
756                 /* Get the hwaddress of our sender */
757                 int len = 0;
758                 struct libnet_ether_addr *src_ether = libnet_get_hwaddr
                        (l);
759
760                 u_int8_t *enet_src = src_ether->ether_addr_octet;
761                 if (strcmp(arguments.spoof_mac, "") != 0)
762                 {
763                         enet_src = libnet_hex_aton((int8_t*)arguments.
                                spoof_mac, &len);
764                 }
```

```
765                 if (enet_src == NULL)
766                 {
767                         fprintf(stderr, "libnet_hex_aton() failed: %s\n
                            ", libnet_geterror(l));
768                 }
769
770                 u_int8_t *enet_dst = libnet_hex_aton((int8_t*)macdst, &
                    len);
771                 if (enet_dst == NULL)
772                 {
773                         fprintf(stderr, "libnet_hex_aton() failed: %s\n
                            ", libnet_geterror(l));
774                 }
775
776                 eth = libnet_build_ethernet(
777                         enet_dst,
778                         enet_src,
779                         ETHERTYPE_IP,
780                         NULL,                   /* payload */
781                         0,                      /* payload size */
782                         l,                      /* libnet handle */
783                         0                       /* libnet id */
784                 );
785
786                 free(enet_dst);
787                 if (eth < 0)
788                 {
789                         fprintf(stderr, "libnet_build_ethernet() failed
                            : %s\n", libnet_geterror(l));
790                         exit(-1);
791                 }
792         }
793
794         c = libnet_write(l);
795         if (c < 0)
796         {
797                 fprintf(stderr, "libnet_write() faild: %s\n",
                        libnet_geterror(l));
798                 exit(-1);
799         }
800 }
```

APPENDIX B

DPASA IP TO HOST MAPPING

Table B.1 can be used to convert the IP addresses found in Figures 3.9 and 3.10 to the DPASA host names.

Table B.1: DPASA Host Name Mappings

Host	IP Address	Operating System
q1ap	192.168.4.114	Windows XP
q1nids	192.168.4.121	SELinux
q1dc	192.168.4.122	SELinux
q1psq	192.168.4.123	SELinux
q1cor	192.168.4.124	SELinux
q1ps	192.168.4.125	Windows 2000
q1sm	192.168.4.126	SELinux
q2ap	192.168.4.130	SELinux
q2nids	192.168.4.137	SELinux
q2dc	192.168.4.138	Windows XP
q2psq	192.168.4.139	Windows XP
q2cor	192.168.4.140	SELinux
q2ps	192.168.4.141	Windows 2000
q2sm	192.168.4.142	Windows XP
q3ap	192.168.4.146	SELinux
q3nids	192.168.4.153	SELinux
q3dc	192.168.4.154	Solaris
q3psq	192.168.4.155	Solaris
q3cor	192.168.4.156	SELinux

Table B.1: DPASA Host Name Mappings Continued

q3ps	192.168.4.157	Windows 2000
q3sm	192.168.4.158	Solaris
q4ap	192.168.4.162	Solaris
q4nids	192.168.4.169	SELinux
q4dc	192.168.4.170	SELinux
q4psq	192.168.4.171	SELinux
q4cor	192.168.4.172	SELinux
q4ps	192.168.4.173	Windows 2000
q4sm	192.168.4.174	SELinux

REFERENCES

[1] E. D. Zwicky, S. Cooper, and D. B. Chapman, *Building Internet Firewalls.* Sebastopol: O'Reilly & Associates, 2000.

[2] J. H. Allen, *The CERT Guide to System and Network Security Practices.* Boston: Addison-Wesley Professional, 2001.

[3] J. Wack, K. Cutler, and J. Pole, *Guidelines on Firewalls and Firewall Policy.* Gaithersberg: National Institute of Standards and Technology, 2002.

[4] A. Wool, "A quantitative study of firewall configuration errors," *IEEE Computer*, vol. 37, no. 6, pp. 62–67, June 2004.

[5] S. M. Bellovin, "Distributed firewalls," *;login:*, pp. 39–47, Nov 1999.

[6] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proceedings of the Seventh ACM Conference on Computer and Communications Security*, November 2000, pp. 190–199.

[7] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. Smith, "The strongman architecture," in *Proceedings of the Third DARPA Information Survivability Conference and Exposition*, vol. 1, April 2003, pp. 178–188.

[8] TheGreenBow, "TheGreenBow Distributed Firewall," June 2005, http://www.thegreenbow.com/fwd.html.

[9] F-Secure, "F-Secure Anti-Virus Client Security," June 2005, http://www.f-secure.com/products/anti-virus/fsavcs.

[10] C. Payne and T. Markham, "Architecture and applications for a distributed embedded firewall," in *Proceedings of the 17th Annual Computer Security Applications Conference*, December 2001, pp. 73–80.

[11] T. Markham and C. N. Payne, "Security at the network edge: A distributed firewall architecture," in *Proceedings of the Second DARPA Information Survivability Conference and Exposition*, June 2001, pp. 279–286.

[12] L. M. Meredith, "A summary of the autonomic distributed firewalls (ADF) project," in *Proceedings of the Third DARPA Information Survivability Conference and Exposition*, vol. 2, April 2003, pp. 260–265.

[13] M. Carney, R. O. Hanzlik, and T. R. Markham, "Virtual private groups," presented at Third Annual IEEE Information Assurance Workshop, West Point, New York, June 2002.

[14] T. Markham, L. Meredith, and C. Payne, "Distributed embedded firewalls with virtual private groups," in *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*, vol. 2, April 2003, pp. 81–83.

[15] D. Hoffman, D. Prabhakar, and P. Strooper, "Testing iptables," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, October 2003, pp. 80–91.

[16] D. Hartmeier, "Design and performance of the OpenBSD stateful packet filter (pf)," in *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 2002, pp. 171–180.

[17] M. R. Lyu and L. K. Y. Lau, "Firewall security: Policies, testing and performance evaluation," in *Proceedings of the 24th International Computer Software and Applications Conference*, October 2000, pp. 116–121.

[18] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices," Internet Engineering Task Force, RFC 2544, March 1999.

[19] B. Hickman, D. Newman, S. Tadjudin, and T. Martin, "Benchmarking methodology for firewall performance," Internet Engineering Task Force, RFC 3511, April 2003.

[20] L. Spitzner, "Auditing your firewall setup," December 2000, http://www.spitzner.net/audit.html.

[21] Fyodor, "Nmap - free security scanner for network exploration and security audits," June 2005, http://www.insecure.org/nmap/index.html.

[22] M. D. Schiffman, "Firewalk homepage," January 2003, http://www.packetfactory.net/projects/firewalk.

[23] K. Wheeler, "Distributed firewall policy validation," December 2004, http://cse.nd.edu/~dthain/courses/classconf/wowsys2004/papers/firewall.pdf.

[24] A. Hari, S. Suri, and G. M. Parulkar, "Detecting and resolving packet filter conflicts," in *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2000, pp. 1203–1212.

[25] F. Baboescu and G. Varghese, "Fast and scalable conflict detection for packet classifiers," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 42, no. 6, pp. 717–735, September 2003.

[26] E. Al–Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, March 2004, pp. 2605–2616.

[27] E. Al–Shaer and H. Hamed, "Management and translation of filtering security policies," in *Proceedings of the 38th IEEE International Conference on Communications*, May 2003, pp. 256–260.

[28] E. Al–Shaer and H. Hamed, "Firewall policy advisor for anomaly detection and rule editing," in *Proceedings of the IFIP/IEEE Eighth International Symposium on Integrated Network Management*, March 2003, pp. 17–30.

[29] M. G. Gouda and A. X. Liu, "Firewall design: Consistency, completeness, and compactness," in *Proceedings of the 24th International Conference on Distributed Computing Systems*, March 2004, pp. 320–327.

[30] 3Com Corporation, "3Com homepage," June 2005, http://www.3com.com.

[31] D. Newman, "Benchmarking terminology for firewall performance," Internet Engineering Task Force, RFC 2647, August 1999.

[32] NLANR/DAST, "Iperf 1.7.0 - the TCP/UDP bandwidth measurement tool," June 2005, http://dast.nlanr.net/Projects/Iperf/.

[33] ACME Software, "Homepage for http_load," June 2005, http://www.acme.com/software/http_load/.

[34] 3Com Corporation, "Cert coordination center threats and 3Com embedded firewall protection," May 2003, http://www.3com.com/other/pdfs/legacy/en_US/102053.pdf.

[35] C. Payne, (private communication), November 2004.

[36] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proceedings of the ACM/SIGDA 13th International Symposium of Field-Programmable Gate Arrays*, November 2005, pp. 238–245.

[37] J. D. Guttman, "Filtering postures: Local enforcement for global policies," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 1997, pp. 120–129.

[38] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," vol. 72, no. 4, pp. 381–420, 2004.

[39] D. Bradley and A. Josang, "Mesmerize: An open framework for enterprise security management," in *Proceedings of the Second Workshop on Australasian Information Security*, January 2004, pp. 37–42.

[40] T. E. Uribe and S. Cheung, "Automatic analysis of firewall and network intrustion detection system configurations," in *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, October 2004, pp. 66–74.

[41] E. E. Schultz, "How to perform effective firewall testing," *Computer Security Journal*, vol. 12, no. 1, pp. 47–54, 1996.

[42] R. Chandran and S. Pakala, "Reducing the time for port scans, an analysis of nmap," April 2004, http://www.paladion.net/papers/nmap_performance.pdf.

[43] AT&T Research, "Graphviz homepage," June 2005, http://www.graphviz.org.

[44] S. Singh, A. Agbaria, F. Stevens, T. Courtney, J. F. Meyer, W. H. Sanders, and P. Pal, "Validation of a survivable publish-subscribe system," *International Scientific Journal of Computing*, to appear.

[45] F. Stevens, "Validation of an intrusion-tolerant information system using probabilistic modeling," M.S. thesis, University of Illinois at Urbana-Champagin, 2004.

[46] E. Messmer, "Should you be using distributed firewalls?" June 2000, http://archives.cnn.com/2000/TECH/computing/06/21/distributed.firewalls.idg/.