Distributed Firewall Policy Validation

Kyle Wheeler

December 7, 2004

Abstract

With hacking attempts, the cost of security breaches, and the importance of defensive computer security in general all on the rise, strong firewalls are more relevant than ever. At the same time, demands for software diversity and increasingly complex network layouts make evaluating adherence to a unified security policy especially difficult. In this paper, I propose a method of uniformly validating firewall security policy in a heterogeneous network with a complex layout, using a hierarchical probing and test management system with very few host requirements and a simple security-policy language for easily mapping policy to evaluations for validation purposes. Preliminary tests indicate that such a system works well across a diverse network, even across the general Internet.

1 Introduction

The number of attacks on network-connected hosts has increased over the last several years, making the security of networks an increasingly important problem. One of the most common ways that networks are hardened against attack is to tightly control what kind of network traffic can enter and exit the network using a firewall. A firewall implements policy decisions at a categorical low-level, explicitly denying, allowing, or restricting broad categories of network packets, the underlying structure of network communication. As firewalls become more capable, to handle more complex policy decisions and to allow more flexibility in policy possibilities, the exact implementation of those policy decisions in a firewall becomes more complicated. As complexity increases, the possibility and probability of errors in implementation also increases.

The goal of this project is to create an easy way to test the implementation of policy, by testing the firewall rules. The object is to determine, in a fast, easy-to-use, and distributed fashion how the existing firewall (or firewalls) behave, and to compare that to policy. The target audience is a system administrator or Chief Security Officer (CSO) that wishes to verify that the policy that was mandated by their organization has been correctly implemented. Alternatively, the system could be used to determine what the existing policy appears to be, so that it can be used as a starting point for developing new policy. A general assumption of this system is that administrator of the firewalls within the network are not adversaries, and are not actively trying to hide faults from the primary administrator.

The general architecture of the system is as follows. The basic building block of the system is the "Prober" program, which sets itself up on a host, and can be directed to listen for and/or send packets from certain ports to other ports to or from any other host. These Probers can then be used by a coordinating "Manager" program to determine whether or not any firewalls are affecting (restricting) traffic in between the Probers. The Managers are directed and controlled by a central "Wizard" program. The Wizard program commands the Managers to perform connection attempts (network tests) in order to verify the policy rules laid out for the Wizard in the policy file by the administrator. Deviations from policy are reported, as is inability to test parts of the policy.

The difficulty is several-fold. First, Probers must be coordinated across many systems, even between disconnected subnets, to accurately determine how any firewalls in between them may be affecting the packet transmission between any of the probes. Second, Managers must be capable of collecting the data in a secure way using a minimum of network resources and guarantees. Third, the connectivity data must be somehow related to a unified policy declaration. Fourth, the policy declaration needs to be translatable into a human-readable form which can be meaningfully compared to an existing human-readable policy declaration, without needing to encode the complete network topology in the policy.

2 Related Work

As security itself becomes more and more important, it becomes more and more valuable for the level of security that can be provided to be quantified, so that businesses can directly match performance to investment [8]. Testing and validating firewalls has, of course, been a goal for quite some time [2, 14]. Security testing methodology in general can be useful for comparing techniques. Some of the major approaches to securing a system include threat-modeling, static vulnerability analysis, and regression testing.

The threat-modeling approach [15] takes a system and models it in terms of how it could and would be attacked; what are the entry points for an attacker, which parts of the system are vulnerable first, and what "surface" or interface does an attacker have to the system in which to find weaknesses. The threat-modeling approach may be useful when dealing with firewalls, but it is an approach that requires far more time and detailed knowledge of the network topology and the machines that are on the network than this project, which may make it unmanageable for all but the smallest systems.

The static vulnerability analysis approach, epitomized by projects such as splint [6], are essentially ways of validating that the implementation of a system does what you think it does—essentially, its goal is to avoid logical ambiguity and common loopholes. This approach is like threat-modeling in that it requires a detailed knowledge of the implementation details (and is thus very implementation specific) of all of the system's components, but is different in that it does not address how the systems interact. It would be possible, for example, to use a static vulnerability analysis program to analyze the policy description of a large network—it would be able to point out contradictions, incomplete coverage, and other errors, but would not be able to analyze multiple firewall rule-sets implemented in multiple disjoint languages (if the firewalls even use a descriptive language) and compare them to a policy document.

The regression testing approach seems to make the most sense in a multi-platform, implementation-independent sense because it allows one to treat the system as a black box, verifying it's input and output [7, 11, 1, 4], which is precisely what is desired in this situation. Existing projects that go about performing such tests, however, use simple approaches fit only for testing a single firewall, and which require root-access to the testing host in order to determine any useful information (by creating custom packets of various sorts and feeding them to the firewall being tested to see what happens). This design makes sense, as the original (and still very popular) design of a firewall was based around the concept of a single network entity that is connected to two (or more) networks and regulates traffic between hosts on those networks [14]. For generality, however, one may not know where firewalls are in a system, and in general security policies are often created that are independent of the particular implementation and network topology. In such a case, it is more desirable to consider the entire network as a black box; one that does not have as well-defined a network boundary as a single firewall would provide.

Recent research has finally begun to further explore the design space of firewalls. For example, firewalls can be designed, rather than wrapped around a single system "wall", as the sum of the firewalls placed on all individual hosts in a network [9]. Thus, far more complicated and in-depth security policies can be implemented, allowing for security decisions at multiple points rather than a single ingress/egress check at a network boundary. This makes it possible to regulate even communication on a fully connected network without forcing all internal communication through a single firewall bottleneck. Such new approaches to firewalls can allow for centralized policy implementation, making maintenance of a unified security policy for the entire network much easier, however it exposes the overall network to new methods of attack which are inherent to any centralized command system. That is, when the central firewall command center is compromised, due to attack or mistake by the administrator, the security of the entire network is compromised, unlike a situation where each firewall has a unique administrator and cannot be turned off en-masse. Also, as in any centralized command system, the commands can be attacked and spoofed, possibly allowing an attacker to subvert the firewall by interrupting or corrupting the command-chain. Additionally, a distributed firewall system is a layer of complexity on top of the primary defenses of the network, and requires all firewalls to be using the same system—making a single flaw in the implementation far more serious than a more diverse system. Nevertheless, the distributed firewall is an attractive option in terms of responsibility and from an ease-of-maintenance perspective, as it allows a single administrator to compare a single control interface with the official security policy to verify that it correctly implements the policy.

In accordance with the increased attention on security and the changing and more complex nature of firewall systems, security analysis capabilities have become more advanced in the way that they repre-

sent security and policy and allow administrators to reason about vulnerabilities [12, 3, 10]. However, the advances have been less in terms of firewalls and more in terms of complicated attack patterns—essentially, threat-modeling approaches tied to specific network topology and network participants. With that comes the same drawbacks of the threat-modeling approach: complicated to perform, and tied to a specific implementation, discouraging (though not ruling out) software diversity.

The specification of policy with a formalized language is also a heavy research area. Several languages, such as the Ponder project [5] are very rigorous. Such detailed policy language design is, however, out of the scope of this paper.

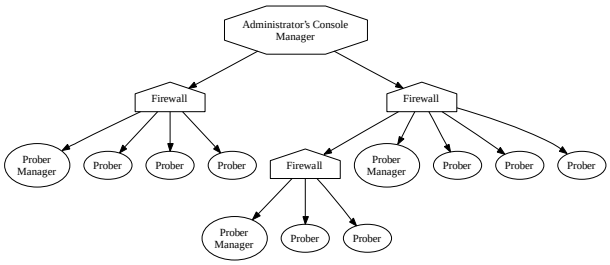

Figure 1: An example network

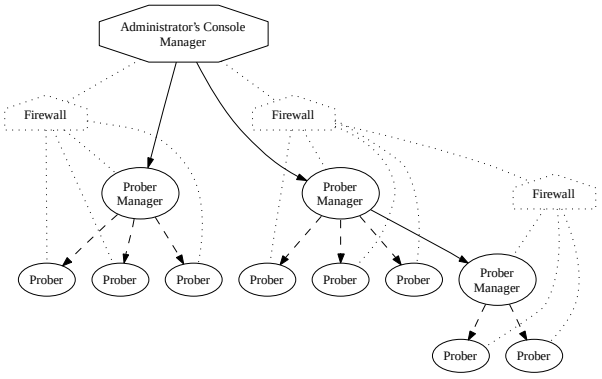

Figure 2: The example network with the policy validation control hierarchy emphasized

3 Architecture

The essential problem addressed by this project has four basic pieces. First, the establishment of coordinated network probing; second, the collection of data from the probes; third, the summarization of this data; fourth, the policy validation and communication with the user. The basic architecture of the system to address these basic pieces is a three-layer hierarchy of functionality with a virtually unlimited possible hierarchy of responsibility.

3.1 Nodes

The system consists of a network of programs on multiple hosts performing different duties. There are three kinds of programs, or three components to the system, as follows:

Prober nodes A Prober node is capable of answering very simple questions about network connectivity, which essentially has the structure of a simple destination or source question that results in a yes or no answer. The Prober node's capability can be expanded in the future to measure things like bandwidth with minimal disruption to the essential functionality of the node.

Manager nodes A Manager node coordinates probing between a subset of Prober nodes and other subordinate Manager nodes, and collects the data generated by the Prober nodes (or other Manager nodes) so that it can be used by the Wizard program. The Manager can spawn both subordinate Managers or Probers on any system it has access to, thus, from a single Manager node, an entire network of Probers and subordinate Managers can be spawned remotely, bending around any network topology restrictions to present a unified network view to the Wizard.

Wizard nodes The Wizard node is the interface to the system that the administrator has for verifying policy and interpreting test results. The Wizard's functionality is based around a basic policy syntax—a policy can be translated into this basic syntax, and the Wizard will interpret the policy in that form along with the information provided to it by the root Manager to answer questions about whether or not the policy has been violated or not, how it has been violated, and what parts of the policy cannot be tested with the existing Manager/Prober network.

This structure allows the system to be installed on virtually any network, independent of network topology. For example, on the network shown in Figure 1, the system provides a convenient unified network view, shown in Figure 2. This structure does, however, present some restrictions, which are discussed in Section 4.1.3.

3.1.1 Testing Completeness

When testing network connectivity, one of the most important questions to answer is what counts as "connected." A trivial solution is to simply consider a single successful TCP-handshake an indication of successful connection. However, the point of an automated system of testing as is presented in this paper is to find the "needle" mistake in the "haystack" of working policy implementation. A single working connection does not verify in a full enough sense that a given network can talk to another given network. On the other hand, a full test of all hosts, on all ports, to all other hosts on all other ports is a massive undertaking that will take an extremely long time, will use excessive amounts of resources, and in the end is entirely impractical. Because a full test of all combinations is not realistic, the final judgment must be one based on probabilities, and therefore, on a random collection of spot-checks. A sampling of the possible connection tests that can be done should be done, and a confidence level based on their collective success should reported.

One of the drawbacks of large-scale testing is the amount of time it takes to perform all of the tests. Section 4.1.1 discusses the method for internode communication, which does not allow for immediate feedback, although it is well suited for multiple, parallel, ongoing tests. Sending out test requests and waiting for them to finish, however, is unavoidable, and the more tests there are to do, the longer they will take to be performed. As such, larger numbers of tests become infeasible for testing purposes, and a balance must be struck between the administrator's patience and the confidence level the administrator wishes to attain. Realistically, the best solution is one of widely deployed testers with near-continuous random testing of policy, so that the coverage is more complete, while at the same time providing the effect of constantly monitoring the network for problems.

3.1.2 Fault Tolerance

A key element to any large system is how well it handles the failure of a single element in the system. Where this policy validation system is concerned, there are two distinct kinds of failure.

Host Failure Any of the network hosts can fail at any time. Generally, a host failure may be difficult to distinguish from a network failure, from the perspective of the rest of the network. Recovery, however, is somewhat different. The things that a node needs to keep track of—subordinates, ongoing tests, previous test results, commands, the node ID, and so forth—do not change very quickly, and it is possible to store all of that information on disk. This way, when a node goes down, it can be restarted without loss of information, and from the perspective of the other nodes in the network there was simply a long network delay. Depending on the details of this storage, it may be possible that a command is lost or duplicated, however this is viewed as inconsequential as long as the Wizard can identify missing tests as not indicating anything about the connection it was supposed to have tested.

Network Failure The network can obviously fail at any time, or can simply not be laid out as expected. From this perspective, any command that gets lost can be viewed as an unexpected, failed network test. These can be ignored or reported to the root Manager in some way, as they indicate a network status that the administrator ought to be made aware of. The system ought to be tolerant of temporary network failure, however, and therefore attempts should be made to retry failed connection tests for some bounded length of time. Ideally, commands should be linked, such that a "cleanup" command terminates all previous command connection attempts. Where official connection tests are concerned, since at some point the Wizard will ask for all results, good or no, it is safe to simply continue attempting after failure. Expected network failures, or disconnected networks, require the administrator to intercede and transfer affected commands in an out-of-band manner. The key to bounding connection attempts for most network events is the "cleanup" command, which terminates all previous network activity, allowing the administrator to decide how long to wait for network activity to finish. The special case is the "cleanup" command itself, which must have it's own policy for how long to continue retries. Any policy will essentially be arbitrary and system-specific, so that should be left to the administrator to specify.

3.2 Policy Language

The policy language that the system uses is designed to be both flexible and simple. A complete policy description file has two parts, a preamble and a body. Comments are permitted, and are denoted either by a `#` or `//` symbol and extend to the end of the line, or they are enclosed, on a single line, within `/*` and `*/` symbols.

The policy language described here is rudimentary, but contains the basics necessary to express a simple policy and is easily extended to allow for

many different policy details. This does not mean that it has difficulty in expressing complex policies, within the limits of the kinds of connections it can describe. Figure 7 contains an example policy for the network described in Figure 4. Even as complicated a situation as that setup is, the policy to describe it is simplistic. Strictly speaking, for a full description of a set of n network entities (entities that have their own network policy attached to them, and can be either full subnets or single hosts), there must be n entries in the preamble of the policy, and at most $n^2 - n$ entries in the body of the policy. That many entries in the body can capture any set of connectivity requirements within the limitations of the expressiveness of connectivity statements in the body.

An example of an even more complex network than that in Figure 4 that can be described by this policy language is the full University of Notre Dame network. There are two main network categories, ResNet and the rest of the on-campus network. ResNet has some limitations in it's communication with the outside world, but no such restrictions on communication with the rest of campus. Within the on-campus network, multiple departments have their own networks. The Computer Science and Engineering Department, for example, contains several private computer clusters, including the ISS, Wombat, CVRL, and Striegel Research Cluster. Additionally, the department hosts a few Hydra nodes, the CSE networking lab, and an E-technology cell. Each of these clusters have their own firewalls and subnetworks. However, all of these networks can be described by even a rudimentary policy language like the one presented in this paper.

3.2.1 Preamble

The preamble of a policy description exists to set up convenient naming, to allow the document to be more human readable. The naming can be used to name both networks and single hosts. The named networks need not necessarily be connected to the outside world or to any one of the other named networks. For testing purposes, however, all networks should be accessible through some sequence of connections back to the root node unless test requests can be forwarded in some out-of-band way to a Manager/Prober hierarchy and the results of those tests can be returned to the Wizard's purview for evaluation. The named networks *should* be fully connected within the network. This does not place any specific requirements on the topology of the network—strictly speaking, only the Managers on the network *need* to be able to contact the Probers and subordinate Managers, while inter-Prober connectivity is unnecessary.

The preamble consists of naming statements in the form of the following:

```
network [name] [networkip] [netmask]
```

Names can consist of the letters a-z and A-Z, the numbers 0-9, the underscore, and the pipe (|), or, if they're enclosed in single quotes, can consist of anything but single quotes.

For example, to declare the "research" network as being the 192.168.57.x subnet, one would use the following line:

```
network research 192.168.57.0 255.255.255.0
```

A single host can be named either by using a full netmask or by using a host line. The following two lines are equivalent:

```
network webserver 192.168.1.1 255.255.255.255
host webserver 192.168.1.1
```

Neither Probers nor Managers are required to be on any of the named networks, this is merely for convenience, though Probers that are not on a named network will not be used. For lack of ambiguity, the named networks and hosts must not overlap. In the future, the Wizard may be able to distinguish between networks that share the same IP ranges by analyzing the addresses of the Managers and comparing that to the connection map established in the body of the policy description, however this is prone to error. Perfect resolution of overlapping or duplicated network addresses would require full network topology information being encoded in the policy.

3.2.2 Body

The body of a policy description exists to establish the state of the system as it *should* be—whether that be to assert that network components (hosts or networks) should or should not connect. In general, it is assumed that networks cannot connect unless stated otherwise, so cannot-connect statements are strictly speaking superfluous, however they are used in this system to differentiate between the assumption that network components cannot connect and the assertion that network components should not connect. In other words, network components that do not have an established relation are not *required* to connect, though they can, unbeknownst to the Wizard, because it does not affect policy compliance. On the other hand, components that have an explicitly stated "cannot connect" or "can connect" relationship offer a testable assertion. Undefined network policy is viewed as the policy-maker's problem.

Network component connectivity is set up with a series of statements in the form of the following:

```
[name1] -> [name2] {on [port]} {via [IP]}
[name1] -X [name2] {on [port]} {via [IP]}
```

The portions of the statement within braces are optional. The default port used for testing when one is not specified is port 3333 (this could be randomized in the future for more effective aberration detection). The `via` extension is to indicate a non-transparent firewall, such as a Linux-based firewall host with an IP address on two or more networks, performing standard network address translation. The meaning of each line is to establish a specific relationship between one network component (the first listed) and another (the second listed). If the `->` symbol is used, that indicates that the first network component listed ought to be able to connect to the second, possibly on a specific port or via some IP address. If the `-X` symbol is used the first network component must not be able to connect to the second on whatever port listed via the IP address listed.

For example, to declare the "sales" network can connect to the "research" network, but only on the AOL Instant Messenger port (5190), one would use the following line:

```
sales -> research on 5190
```

4 Implementation

The implementation of the system to demonstrate its feasibility comes in two basic parts. The first part of the system is the workhorse of the system: the Manager/Prober architecture, described in Section 4.1. This part can exist independently of the other as a good framework for distributing commands while working around network topology. The second part of the system can be considered the brains of the system: the Wizard, described in Section 4.2. The Wizard is the part that interprets the policy file (described in Section 3.2), and uses the Manager/Prober network for validating that policy. The Wizard could, theoretically, be swapped out for another policy-tester that could, for example, be more intelligent or more thorough about what tests to run, or could simply use the Manager/Prober architecture to do network mapping.

4.1 Managers & Probers

Both Managers and Probers use a similar implementation style. Both are implemented in `bash` [13]. Bash, an interpreted language, was chosen for two primary reasons. Firstly because it is a very common interpreted language and thus can be deployed with little fore-knowledge of the system it will be deployed on. Secondly because bash is very good at handling the interesting file redirection, threading, process monitoring, and signal handling tricks that are necessary for setting oneself up as a "server" to respond to queries and perform all of the necessary activities. Because of the use of `bash`, which is well-suited to file manipulation, and also because it makes crash-recovery possible, all status information about the servers is kept on disk. This includes the list of subordinates, any ongoing network tests, the server's ID, and so forth. In this prototype, this information is kept in a series of directories that stores key-value pairs by associating filenames with the first line of the contents of text files. For example, a Manager's list of subordinates is kept in a `subs/` directory. The files in this directory all are named with one of the internal names of the subordinates that belong to this Manager. The first line of their contents (subsequent lines are ignored at present) is the IP address of the subordinate. A similar scheme is used for storing all other status information.

4.1.1 Communication

The definition of a communication protocol allows multiple implementations of the Manager and Prober node software to work together, which enables possible a degree of device-independence and opens the door for avoiding a monoculture. Managers and Probers both use a communication protocol based on the same concept as Maildir [16] folders.

Maildir folders are structured as a set of three directories—`new`, `tmp`, and `cur`—within a parent directory. When a new mail must be delivered to the Maildir, a file is created with a unique name in the `tmp` directory. The data of the message is then written to the file. When the file is complete, the file is moved from the `tmp` directory into the `new` directory. On most filesystems, a move is an atomic action. When a mail client reads messages from the Maildir, it looks for files in the `new` and `cur` directories. In this way, the client avoids mistakenly reading incomplete mail messages. Also, no file locking is necessary when reading or delivering, allowing multiple deliveries and multiple mail clients simultaneous access to the directory. Simultaneous deliveries are made possible by the new-file naming scheme, which can involve the date, random numbers, the process ID, the file's inode number, and several other things in order to guarantee unique file names even under heavy load with multiple simultaneous deliveries. The exact name of the file is irrelevant to a mail client, of course.

There are many features of Maildirs that are desirable for the Manager/Prober communication mech-

anism, from the ability to avoid file locks, to the ability to allow multiple simultaneous commands, to the prevention of misinterpretation due to partial commands being received. The use of this mechanism for sending commands also places some restrictions on the Managers and Probers—namely, they require some local, writable disk space. But the most important feature of the Maildir mechanism in this context is the ability for the system to use many different mechanisms for command relay—any valid file transfer or creation mechanism will work—which avoids being reliant on specific network features for command transfer. An RPC mechanism is alternative method for relaying commands from Manager to Prober and back, but such an RPC mechanism would place network connectivity restrictions and requirements on the system that are rather inflexible. For example, the RPC mechanism requires an additional, specific, open port on the Prober that the Manager can communicate with. On the other hand, with a file-based command system, *any* mechanism for communication can be used to create such files. In the prototype implementation, SSH is used for not only spawning new nodes but also for file-transfer, but this is not a logical requirement of the communication mechanism. Commands and responses can even theoretically be relayed from one node to another by hand if necessary.

As they run, the Probers and Managers check the `command` directory, akin to the Maildir `new` directory, once every second for new commands. Commands are small, one-line files that are deleted as soon as they are interpreted. New commands can be placed in the command directories using any file transfer method, although the prototype Managers use `ssh` and `scp` to transfer and manipulate the files [17], allowing for security and unsupervised behavior with properly managed ssh keys. This system of command distribution is essentially a form of "message-passing" behavior implemented over SSH, which eliminates the need for the Managers and Probers to have a permanent network listener, and simplifies the implementation a great deal.

Both programs also use additional folders as permanent data storage, for recording things like what are the process identifications of currently running `ttcp` sessions, what are the results of the connectivity tests, and where are all of a Manager's subordinates. This transparency allows for decentralized system setup—a single administrator need not be responsible for setting up Probers and Managers throughout a large network, but there can be some regional responsibility, with information about each network simply added to the relevant already running Manager. Managers do not even need, necessarily, to be aware of the full extent of their subordinate trees. Commands from a higher level contain full "routing" information, by virtual of the namingh scheme, allowing middle-Managers to store very little information—they need only know about their direct subordinates—to remain fully functioning.

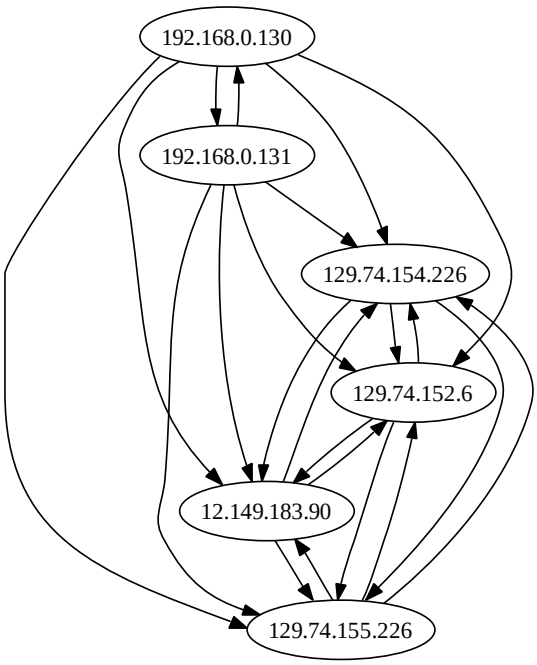

Figure 3: An example network connectivity graph

As part of initial testing, the Managers can even create a graph of the connectivity data they have access to. Figure 3 is a simple example of this graphing capability.

4.1.2 Internal Names

Managers and Probers have an internal set of names (IDs) to identify each other and the tests that are performed. These IDs are hierarchical, identifying the full path from the root Manager, and thus providing command routing information for any Manager further up the tree to forward commands to their eventual destination when the commands are given with an ID for a destination. Each new Manager gets its ID from its immediate superior in the hierarchy. The ID is constructed from the ID of the superior manager, a dot, the letter "m", and a unique number (within the scope of that superior Manager), in that order. A similar construction is used for generating Prober IDs, except the letter "p" is used instead of the letter "m". Tests are named with the ID of the requester of the test, with a period, the letter

"t", and a unique number appended to it. Probers, once they get a test command, identify which side of the test they were on by appending a period and the letters "l" (on the listening end) or "t" on the transmitting end. For the Wizard's purposes, the structure of the names is irrelevant, as long as that structure is maintained when issuing commands to the root Manager. An example of the IDs in practice can be seen in Figure 6.

4.1.3 Restrictions

As networks get larger, they tend to contain a broader and broader range of systems. Ideally, for a generalized network firewall policy validator, the fewer requirements the validator has and the fewer restrictions it places on network and system design, the broader appeal it will have. Many of the decisions in the design of the prototype system have been for the purposes of interoperability and keeping the requirements that the validator places on the overall network as low as possible. However, the implementation does have some requirements. Several of the requirements are implementation-specific, some are not.

Design Restrictions Requirements that are inherent to the design include the requirement of writable disk storage of some form. Without a writable disk, the Maildir mechanism breaks down. Also, while it is not a strict requirement, the design encourages some level of network connectivity between the hosts in the system—generally, a Manager must be able to connect to it's subordinates. Because commands may be relayed by hand by an administrator, this is not a strict requirement of the design. This connectivity requirement has some repercussions on the network topology that can be canvassed by this system—disconnected subnetworks, for example, can be tested, but must be tested by hand. Since there is no auto-detection of nearby Probers, very dynamic networks are very hard if not impossible to test.

There is also the issue of "users" to take into account with firewalls. One of the features of fancier host-based firewalls is to restrict or allow network communication based on what user is triggering the communication (by owning the process that is doing the communication, for example). While the design of this system does not specify that all Prober nodes must be running as particular user, it does not include any provisions for testing multiple users on a single host. This is not an insurmountable obstruction, however, on most POSIX-style systems, only the Administrator or root-user of a machine can pretend to be another user sufficiently in order to, for example, test the user-based firewall rules. Were additional provisions for testing connections based on users to be added to the system, this would require the nodes doing the testing to have such Administrator or root access, which opens the system up to further security concerns.

Similarly, because the Prober nodes may not necessarily have root access to their hosts, some policy rules, such as those involving ports under 1024, may be untestable.

Implementation Restrictions The implementation places additional restrictions on the systems. The prototype implementation requires, specifically, SSH [17] access between Managers and their subordinates. Commands may be relayed by hand, but the implementation contains no provision for notifying the administrator to relay commands. Of course, communication via SSH requires hosts to be running a standard SSH server. The prototype implementation also requires, for unattended, remote operation, that the SSH access be possible without requiring a password be typed in for every command. This can be achieved with an SSH authorization key, but the security of this key and of the accounts that may be accessible with it is left to the administrator. The use of SSH has the benefit that all command communication is encrypted, and the use of the key makes it more difficult for an attacker to attempt to send arbitrary commands.

Because the prototype Manager and Prober node servers are written in the `bash` language [13], hosts to be used for the purpose of being a Manager or a Prober must have `bash` installed on them. This limits the possible machines that can host Manager or Prober servers to machines that are running POSIX-compliant operating systems. Additionally, the network connection testing is built around the network testing tool `ttcp`, which must also be installed on the machines that host Prober servers.

4.1.4 Prober Commands

Probers understand a very small set of commands, relating to their small range of capabilities. The commands are as follows:

nop Just like the CPU instruction, this means "do nothing"

flush This instructs the Prober to terminate all ongoing tests (`ttcp` control threads) and generate results for them.

cleanup This instructs the Prober to do a "flush", and then to exit.

l [port] {u|t} [testid] This instructs the Prober to start a `ttcp` control thread to listen on the specified port, via either TCP or UDP, and to store the result in a file with the specified test identifier.

t [ip] [port] {u|t} [testid] This instructs the Prober to start a `ttcp` control thread to attempt a connection to the specified IP address on the specified port, via either TCP or UDP, and to store the result in a file with the specified test identifier.

4.1.5 Manager Commands

Managers understand a larger set of commands than the Probers do, in a large part because of their expanded responsibility in the network. Managers are responsible for not only coordinating tests, but also passing along commands to subordinate Managers, collecting data, and even finding out their full subordinate tree if necessary. The commands the Manager modules understand are as follows:

nop Just like the CPU instruction, this means "do nothing"

cleanup This instructs the Manager to send a "cleanup" signal to all of its subordinates (both Managers and Probers), and then exit

iptest [proberip] [proberip] [port] {u} This instructs the Manager to test a connection between the Probers on the specified IP addresses on the port specified, via either TCP or UDP, assuming that both of the IP addresses specified belong to direct subordinates of the Manager.

test [proberid] [proberid] [port] {u} This is the same as the "iptest" command, but uses IDs rather than IPs. Because it uses IDs, the Probers may not be direct subordinates, but may belong to a subordinate Manager somewhere in the Manager's subordinate tree. The ID contains the ancestral information necessary to route the command to the proper Managers.

prober [ip] This spawns a new Prober at the specified IP address and sets it up as a subordinate of this Manager.

testport [port] This does an all-pairs test of all subordinate Probers the Manager knows about, on the specified port.

graph This fetches the connection test data from the Probers, and creates a connection graph, like the one in Figure 3.

manager [ip] This spawns a new Manager at the specified IP address and sets it up as a subordinate of this Manager.

managercommand [id] [command] args This sends the specified command and all its arguments to the specified Manager. This ID must be a subordinate of this Manager.

subordinates This collects the IDs and IP addresses of the full subordinate tree of this Manager, and puts them in files, one per ID, in the "`./subs/`" directory.

proberlisten [id] [port] {u|t} [testid] This sends an "l" command to the specified Prober for the specified port and TCP/UDP combination, and tells the Prober to identify it with the specified test ID. The specified Prober must be a direct subordinate of this Manager.

probersend [id] [port] {u|t} [testid] This sends a "t" command to the specified Prober for the specified IP address, port, and TCP/UDP combination, and tells the Prober to identify it with the specified test ID. The specified Prober must be a direct subordinate of this Manager.

flushprobers This informs the Manager to tell all of its subordinate Probers to finish their tests immediately, and collects the results.

4.2 Wizard

The Wizard is written in C, to allow for more complicated memory management and faster "reasoning" about test results than would be easily accomplished by using `bash`. In combination with C, the Wizard uses a Lex/Yacc scanner/parser system to interpret the policy file, allowing for a more complicated language—although the current language is not very complex, there is much room for expansion.

The Wizard is meant to be run from within the folder containing the root Manager—the Manager at the root of the hierarchy, generally referred to as the "root" Manager. This root Manager should have been issued a "subordinates" command before the Wizard is run, so that the Wizard can know what Prober nodes are available.

As the policy file, as defined in Section 3.2, is interpreted the Wizard builds up several cross-referenced lists of what networks are where, which should connect to which, and on what ports. Then the Wizard collects the information about what Probers are available (as reported by the root Manager's "subordinates" call) from the files in the "`./subs/`" directory. Each Prober is associated with a network on the basis of its IP address—this is the primary reason that disconnected networks cannot currently share an IP range: because of the difficulty in associating a Prober with one of those networks without more information about the network topology embedded in the policy (this is considered an open problem). The Wizard runs through the list of connectivity assertions for each network, determines whether or not there are any Probers available to test the assertion.

The determination of Prober availability is based on IP address and connection type. Generally, connections merely need at least one Prober on both the source and destination networks, however the "via" keyword allows for more interesting setups. Because the "via" keyword indicates an opaque firewall, to test the specified connectivity the Wizard needs at least three Probers; one on the source network, one on the "via" IP address, and one on the target network. The assertion is only determined to be upheld if all chosen Probers on the source network can communicate with the Prober on the "via" IP address *and* the Prober on the "via" IP address can communicate with the chosen Probers on the destination network *and* the Probers on the source network cannot communicate with the Probers on the destination network. If there are sufficient Probers to test an assertion, the test pairings are encoded as "test" commands, and are added to a list in memory.

If there are insufficient Probers available for any one of the policy assertions, the Wizard informs the administrator which networks lack Probers and which policy assertions cannot be tested without them, and asks if the administrator would like to test the remaining testable assertions. If there are sufficient Probers, or if the administrator gives the Wizard permission to test with insufficient Probers, the Wizard then fills the root Manager's command directory with all of the test commands it stored in memory.

Once the test commands have been issued, the Wizard again prompts the administrator for how long to wait for the tests to complete. This length of time is entirely dependent upon network topology and how many tests were issued. In the future the policy assertions may be associated with time, requiring an even longer wait for results. Future work may include an active polling by each Manager to determine the outcome of any tests it has issued, so that the root Manager may actually be able to simply wait for the tests to complete (with a maximum outer bound, of course). Once the administrator has determined that enough time has passed, the Wizard sends the root Manager a "flushprobers" command to collect all test data. Finally, the results can be examined by the Wizard, allowing it to, on a line-by-line or on-the-whole basis, report back to the administrator whether or not the policy appears to be correctly implemented over the whole network.

4.3 Setup

As with any system that is designed for the "real" world, some attention must be paid to how easy it is to install and prepare for use. The prototype is slightly more difficult than a production-ready system would be, but is still fairly easy. The installation comes in three steps: first, designing the hierarchy around network topology, second, preparing the hosts, and third, spawning servers and connecting them together.

The first of the three steps requires, obviously, knowledge of the network topography. Generally, fully-connected networks need a single Manager and as many Probers as is practical. From there, arranging for subordinates should flow naturally from what is connected to what. Relatively invisible network topography, such as hubs and switches, can most likely be ignored, though some of the more advanced switches may require that subordinate Managers be used to get around them.

Once the duties of the various hosts in the system have been determined, the hosts must be prepared for the servers. Essentially, connectivity must be checked, an account for the Probers must be set up (one with an impossible-to-guess password is preferable), the SSH key must be placed in the correct location (usually `~/.ssh/authorized_keys`), and both `bash` and `ttcp` must be installed on the systems to be used as Probers (most likely, all of them).

Finally, the servers should be spawned and informed of each other. An easy way to do this, if SSH connectivity is working through the entire hierarchy, is to run the root Manager and tell it what subordinate servers to spawn. The Managers can spawn Prober and Manager nodes on remote hosts and automatically assign them unique names. For disconnected networks, or for administrators who don't necessarily have access to the root Manager, Probers and Managers can be spawned by themselves, assigned names in the hierarchy, and their superiors notified of their existence by placing appropriate files

in the record-keeping folders of those superiors.

5 Evaluation

5.1 Results

The initial results indicate that the system performs as described. A preliminary test was performed over the relatively simple network outlined in Figure 4. There are three major networks: "brk", "cse", and "iss". Although the connections aren't drawn, any system in the graph that is connected to the "internet" can connect to any other system that is connected to the "internet".

Managers and Probers placed on the systems in the network as indicated in Figure 5, with the irrelevant network topography laid out in dotted lines. The layout of the Manager and Prober hierarchy is clarified in Figure 6. In both graphs, subordinate relations are indicated by the non-dotted edges—subordinate Managers are indicated by a solid edge, and subordinate Probers are indicated by a dashed line. Each large shape is a host on the network, with its IP address listed at the top and underneath is a list of the relevant programs running on the node.

In this test system, the Wizard successfully verified the policy—which was already known to reflect the networks' connectivity (reflected in Figure 4). This simple policy is given in Figure 7. When the policy was modified to specify that communication was desired between the "iss" network and the "cse" network, the Wizard correctly reported that the policy was being violated. Finally, when the test was attempted without the Probers in the "cse" network, the Wizard correctly demanded more Probers be installed in order to test the policy.

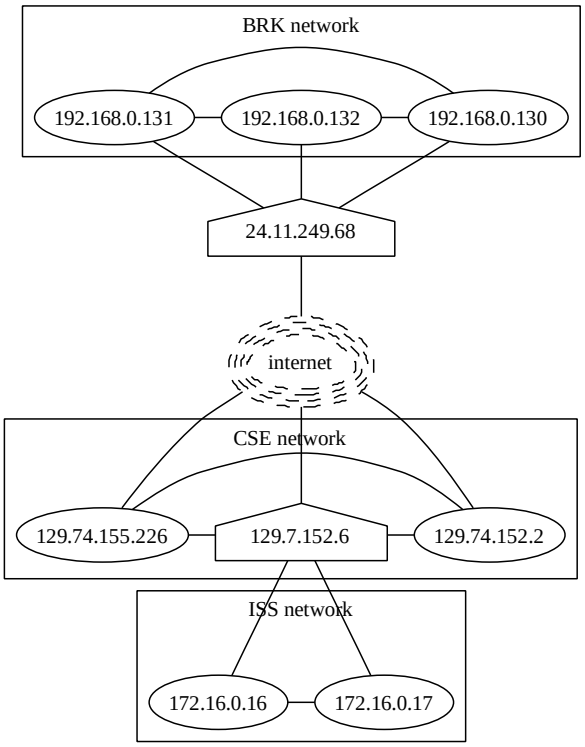

Figure 4: The testbed network

5.2 Lessons Learned

A platform independent hierarchical command structure is a highly successful method of controlling distributed network testing programs. While there were some difficulties designing the `bash` servers they seemed to stem primarily from `bash`'s lack of a pre-compiler and a lack of experience writing large programs in `bash`. Once the system was in place, it performed very well, working around the most common network topology restrictions with ease. A simple and rather straightforward Wizard implementation seems entirely sufficient for most networks, however the naive implementation has some restrictions that a more advanced implementation, with some modification to the policy language, could remove. This system

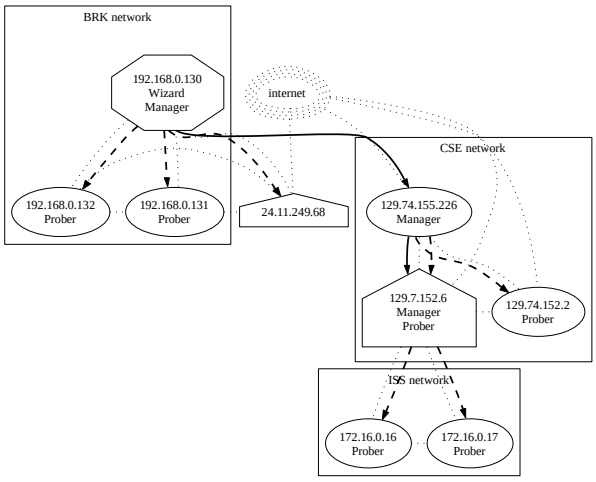

Figure 5: The testbed hierarchy with network layout

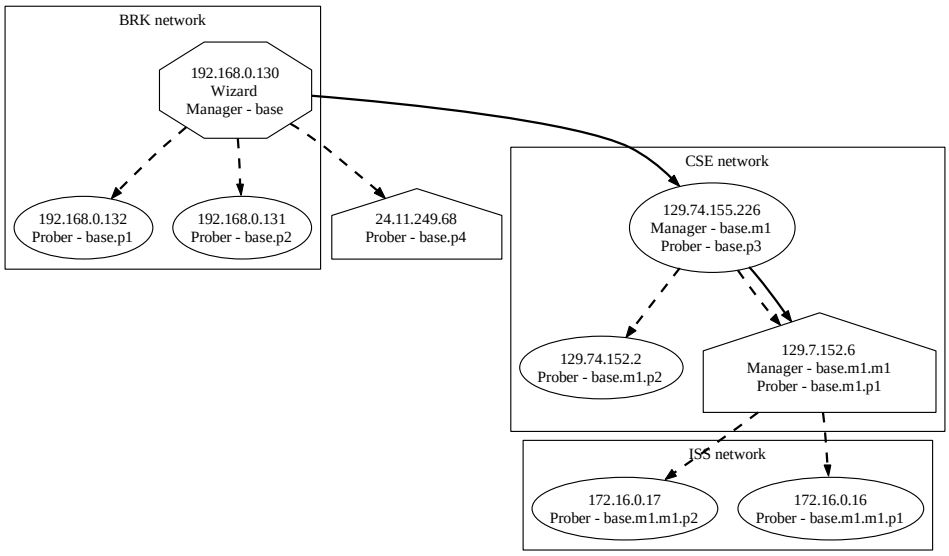

Figure 6: The testbed hierarchy

```
network iss 172.16.0.0 255.255.0.0
network cse 127.74.0.0 255.255.0.0
network brk 192.168.0.0 255.255.0.0

brk -> nd
brk -> iss via 129.74.152.6

cse -> brk via 24.11.249.68
cse -> iss via 129.74.152.6

iss -X nd
iss -X brk
```

Figure 7: The testbed policy

prototype, however, illustrates the potential of this architectural approach.

6 Conclusions

The job of a system administrator is a hard one, made ever more so by the increasing size, complexity, and importance of the systems that need to be administered. As the firewall is one of many valuable tools in waging the war against computer hackers, testing and evaluating the deployment of those firewalls is equally valuable. In particular, the ability to compare system-wide network security policy to system-wide firewall implementation. This paper presents an architecture for doing precisely that—using a large set of connection testers (Probers) and a tree-layout of command and communication links (Managers), with a single command-and-control point (Wizard) that understands a simple language for expressing security policy in a testable way. Inter-node communication is specified in a generic way, using the file-based Maildir algorithm, allowing for a large degree of adaptiveness to different network requirements.

Because an all-points to all-points test of the policy is infeasible, positive validation can only be done in terms of a confidence level, expressed as a percentage of successful network tests that have been completed compared to the total number of tests possible. Of course, the discovery of a single discrepancy (allowing for retries) indicates conclusively, with 100% confidence, that the policy is not being followed. The median used by the prototype is to simply continue testing until the administrator wants to know an answer, at which point the current confidence level is reported.

The prototype implementation of this architecture successfully accommodated a reasonably complex network and network communication policy easily, and verified that the test network's firewalls (and other network features) correctly implemented the policy.

References

[1] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul A. London. Incremental regression testing. pages 348–357.

[2] Khalid Al-Tawil and Ibrahim A. Al-Kaltham. Evaluation and testing of internet firewalls. *Int. J. Netw. Manag.*, 9(3):135–149, 1999.

[3] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 217–224. ACM Press, 2002.

[4] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *International Conference on Software Engineering*, pages 211–220, 1994.

[5] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–??, 2001.

[6] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[7] K. Fischer, F. Raji, and A. Chriscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference*, pages 1–6. IEEE Computer Society Press, 1981.

[8] Ronda R. Henning. Security service level agreements: quantifiable security for the enterprise? In *Proceedings of the 1999 workshop on New security paradigms*, pages 54–60. ACM Press, 2000.

[9] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199. ACM Press, 2000.

[10] Calvin Ko, Deborah A. Frincke, Terrance Goan, Jr., Todd Heberlein, Karl Levitt, Biswanath Mukherjee, and Christopher Wee. Analysis of an algorithm for distributed recognition and accountability. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 154–164. ACM Press, 1993.

[11] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[12] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*, pages 71–79. ACM Press, 1998.

[13] Chet Ramey. Bash, the Bourne-Again Shell. In *ROSE 94*. The Romanian UNIX User Group, 1994.

[14] Marcus J. Ranum. Thinking about firewalls. In *Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II)*, 1994.

[15] Frank Swiderski and Window Snyder. *Threat Modeling*. Microsoft Press, 2004.

[16] Sam Varshavchik. Benchmarking mbox versus maildir. http://www.courier-mta.org/mbox-vs-maildir/, March 2003.

[17] Tatu Ylönen. SSH – secure login connections of the internet. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, California, USA, July 1996. USENIX.