An Open Source Solution for Testing NAT'd and Nested iptables Firewalls

Robert Marmorstein and Phil Kearns – The College of William and Mary

ABSTRACT

As firewalls have increased in power and flexibility, the complexity of configuring them correctly has grown significantly. An error in the firewall configuration can compromise the security of the system or interfere with normal network activity. The chance of an error increases when coordinating multiple firewalls, because the interaction between filters may hide errors more easily noticed on a single firewall. Firewalls on many networks use network address translation, which further increases the complexity of the firewall policy and creates additional opportunities for errors. Because errors in the firewall configuration are often extremely costly in time and security, system administrators need tools for verifying and debugging their firewall policy. ITVal is a tool for analyzing iptables-based firewalls that provides a plain English query language for simple firewall analysis. In this work, we describe extensions to ITVal that allow it to process network address translation rules and analyze multiple firewalls connected sequentially.

Introduction

Networks with a large number of hosts must defend against both external and internal intruders. While a perimeter firewall will block many external threats, it is useless against attacks from inside the network. With Trojan horses and viruses extremely prevalent, the problem of intrusions from internal hosts is growing rapidly [11]. To solve this problem, many system administrators complement the perimeter firewall with local firewalls on important internal hosts [10]. If the network is sufficiently large, the system administrator may also place additional firewalls between the perimeter firewall and groups of related hosts. The resulting architecture looks something like Figure 1, which depicts a network with a perimeter firewall, one unprotected host, two protected hosts, and a protected subnet. The protected hosts could be a mail server and a web server, while the protected subnet might consist of clients in an accounting department with financial information that must be secured. The perimeter firewall can mitigate denial of service and other external threats, while the firewall on each workstation secures services that must be protected from an inside intruder. Usually, most of the workstations have very similar filtering policies, which simplifies the distribution of changes to the policy, since the policy can be edited on a single administration host and then distributed across the network. One or more of the firewalls may also use network address translation(NAT) to further protect internal hosts or to work around the IPv4 address space problem.

The Linux firewall system, iptables, which provides NAT and stateful filtering, is well-suited for securing large networks of workstations and can be a

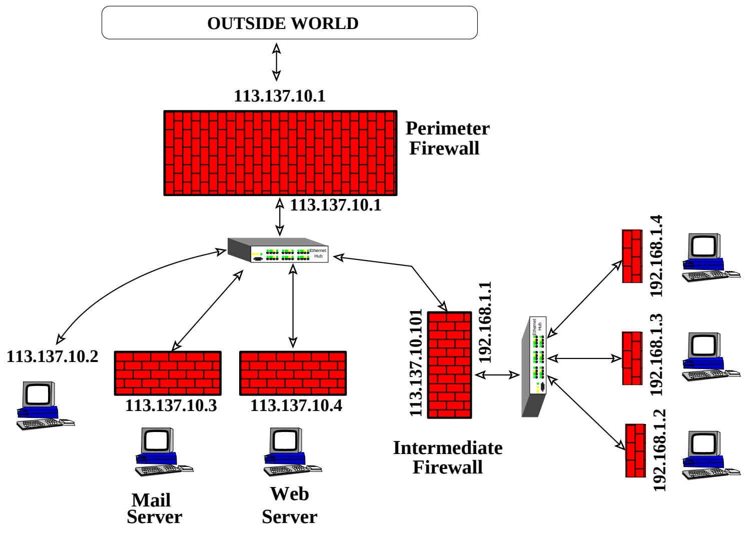


Figure 1: Common firewall architecture for defeating insider threats.

cheap solution for providing security against internal threats. Unfortunately, iptables rule sets are particularly difficult to understand and debug. Changes to the firewall may introduce subtle errors that disrupt normal traffic and can be difficult to diagnose.

The multiplicity of firewalls, in networks like the one described, greatly increases the difficulty of avoiding configuration errors. Removing a rule at the perimeter often means exposing hosts that are not sufficiently protected by their local firewalls. Incorrectly adding rules to a local or intermediate firewall can unintentionally block important network services. Firewalls with NAT are even more complex because a set of translation rules must be considered in addition to the filtering rules.

For example, consider the filtering chains given in Figure 2 that could be used to secure the network shown in Figure 1. The first chain is the forwarding chain of a perimeter firewall that protects a network 113.137.10.0/24 from intrusions on an insecure network 113.137.9.0/24. Rule 1 of the chain blocks traffic from the insecure network. Rule 2 protects the mail server by blocking all traffic from the outside world. The remaining rules secure various services that should be allowed to pass through the firewall. The second chain is the INPUT chain of the internal mail server, 113.137.10.3. It permits SMTP, secure IMAP, and SSH traffic, but blocks anything else.

There are several invariants that the administrator wishes to preserve about this network. First, web traffic from anywhere but the insecure network should always be allowed to host 113.137.10.4, the web server. Second, hosts from the outside world should never be able to SSH into the mail server. Third, no traffic should ever be permitted from the insecure network.

Let's assume that the administrator decides to allow SSH traffic through the perimeter firewall for hosts on subnet 113.137.8.0/24 by adding a rule to the

Chain FORWARD (Default Policy DROP)					
	target	prot	source	destination	flags
1	DROP	tcp	113.137.9.0/24	anywhere	
2	DROP	tcp	anywhere	113.137.10.3	
3	ACCEPT	tcp	anywhere	anywhere	TCP dpt:80
4	ACCEPT	tcp	anywhere	anywhere	TCP dpt:53

Rule Set on the Perimeter Firewall

Chain INPUT (Default Policy DROP)					
	target	prot	source	destination	flags
1	ACCEPT	tcp	anywhere	anywhere	TCP dpt:22
2	ACCEPT	tcp	anywhere	anywhere	TCP dpt:993
3	ACCEPT	tcp	anywhere	anywhere	TCP dpt:25

Rule Set on the Mail Server

Figure 2: Rule sets for an example network.

Chain FORWARD (Default Policy DROP)					
	target	prot	source	destination	flags
1	DROP	tcp	113.137.9.0/24	anywhere	
2	ACCEPT	tcp	113.137.8.0/24	anywhere	
3	DROP	tcp	anywhere	113.137.10.3	
4	ACCEPT	tcp	anywhere	anywhere	TCP dpt:80
5	ACCEPT	tcp	anywhere	anywhere	TCP dpt:53

Figure 3: An incorrect perimeter rule set.

Chain FORWARD (Default Policy DROP)					
	target	prot	source	destination	flags
1	DROP	tcp	113.137.9.0/24	anywhere	
2	DROP	tcp	anywhere	113.137.10.3	
3	ACCEPT	tcp	113.137.8.0/24	anywhere	
4	ACCEPT	tcp	anywhere	anywhere	TCP dpt:80
5	ACCEPT	tcp	anywhere	anywhere	TCP dpt:53

Figure 4: A correctly modified rule set.

forwarding chain of the perimeter firewall as shown in Figure 3. The first rule and the last three rules are the same as those in Figure 2, but the second rule is new. This change preserves the first and third invariant, but violates the second, because SSH traffic from the outside world can now reach the mail server. To correct the violation, she can either add restrictions to the filter on the mail server or switch the order of rules two and three in the perimeter filter.

A correct rule set for the perimeter firewall is shown in Figure 4. The new rule set allows HTTP and DNS traffic from 113.137.8.0/24, but preserves all three invariants.

Existing Tools

There is a fairly large body of tools available for testing firewalls. Port scanners, such as nmap [6], can be used to reveal open and closed ports on each host. Tools that perform general security audits, such as SATAN [5], Nessus [2], SARA [17], and ISS [9] also include components for testing firewalls by sending specially crafted packets to a host. In addition, there are a few tools, such as Ftester [3], designed specifically for analyzing stateful firewalls.

All of these tools are *active tools* that test the firewall by sending traffic through it. This has the disadvantage of consuming bandwidth and interfering with normal traffic. Furthermore, active tools are usually very inflexible. Rather than providing general functionality for investigating the firewall configuration, they are usually designed to test specific vulnerabilities. Also, they usually only simulate packets originating from a single host or small group of hosts. Incorrectly configured firewalls that allow packets from an untested host will pass the test even though an error exists. Some tools use address spoofing to mitigate this problem, but because of bandwidth constraints, no active tool can test every possible address that might originate a packet to the firewall.

More importantly, active tools do not work well with multiple firewalls and network address translation. Because packets dropped by one firewall are never seen by the second, it is often difficult for an active tool to generate a spoofed packet that will exploit configuration errors in both firewalls. Also, replies to packets with NAT'd source addresses may never been seen by the active analysis tool.

Because active tools have these drawbacks, *passive tools*, which perform an offline analysis of the firewall can be more practical. One such tool is the Lumeta firewall analyzer [18], a commercial product. Lumeta is based on Fang [1] and provides general query capability for Checkpoint and PIX firewalls. Because Lumeta provides an offline analysis of the firewall policy, it has many advantages over active tools. Unfortunately, Lumeta is not designed to work with iptables firewalls.

A few other groups have also done some work on passive analysis. A team at the University of Texas has developed a tool that uses SQL-like queries for firewall analysis [12]. They have also developed a system for improving the structure of the rule set using decision diagrams [7]. Another group has used decision diagrams to implement basic firewall queries [8, 16]. None of these tools are specifically targeted at iptables and they do not support NAT.

In previous work, we presented an open source tool, ITVal, for performing a passive analysis of a single iptables firewall. ITVal uses an efficient decision diagram library [13] to provide a plain-English query language that a system administrator can use to quickly test for vulnerabilities. ITVal is particularly useful for evaluating changes to the firewall configuration. A system administrator can perform an ITVal audit before and after each change to the firewall and examine the results to quickly determine if any of the important security invariants of the network have changed. As presented in [15], ITVal supported neither NAT nor analysis of multiple connected firewalls. In this work, we present extensions to ITVal that allow it to take these into account.

ITVal

ITVal implements a query engine for evaluating the configuration of a firewall. Some example queries are shown in Figure 5. The main components of each query are the keyword QUERY followed by an optional input chain, a subject, and a query condition. The input chain parameter determines whether the query should consider traffic inbound to the firewall, traffic outbound from the firewall, or traffic forwarded through the firewall.

The subject tells ITVal what information to report. For instance, the subject SADDY instructs ITVal to list the source addresses of packets that match the query. The condition is made up of primitives that can be combined with the logical operators AND, OR, and NOT to form complex queries. The available primitives are FROM, TO, ON, FOR, WITH, and IN, which specify source address, destination address, source port, destination port, TCP flag status, and connection state, respectively. There is also a special primitive, LOGGED, which matches packets for which the firewall contains a matching LOG rule. The reader is referred to [15] for the details of the query language. One feature of the query language we will exploit in our examples is the ability to name groups of hosts and services for use in multiple queries with the GROUP and SERVICE keywords, respectively.

The internal representation of both the queries and the rule sets is handled by the FDDL [13] decision diagram library. FDDL provides a data structure called a Multi-way Decision Diagram(MDD) for representing large sets of vectors compactly. MDDs are particularly well suited for representing firewall rules. In fact, an MDD implementation of the iptables filtering algorithm showed significant performance gains over the existing implementation [4].

- QUERY SADDY TO 192.168.*;
  List all hosts with access to subnet 192.168.0.0/16.
- QUERY DPORT FROM 113.137.10.* AND NOT FOR TCP 993;
  List all destination ports, except the secure IMAP port(993), that can be accessed by hosts in the 113.137.10.0/24 subnet.
- QUERY SPORT NOT FROM 192.168.1.101 AND FOR 137.113.6.2;
  List all source ports open on host 137.113.6.2 to machines other than host 192.168.1.101.
- QUERY DADDY FOR TCP 25 AND (IN NEW OR IN ESTABLISHED);
  List all hosts that can receive packets on port 25 on a connection in the NEW or ESTABLISHED state.
- QUERY DADDY FROM 192.168.1.* AND (FOR TCP 25 OR FOR TCP 80 OR FOR TCP 110);
  List all hosts that can receive SSH, SMTP, or HTTP traffic from hosts on the 192.168.1.0/24 subnet.

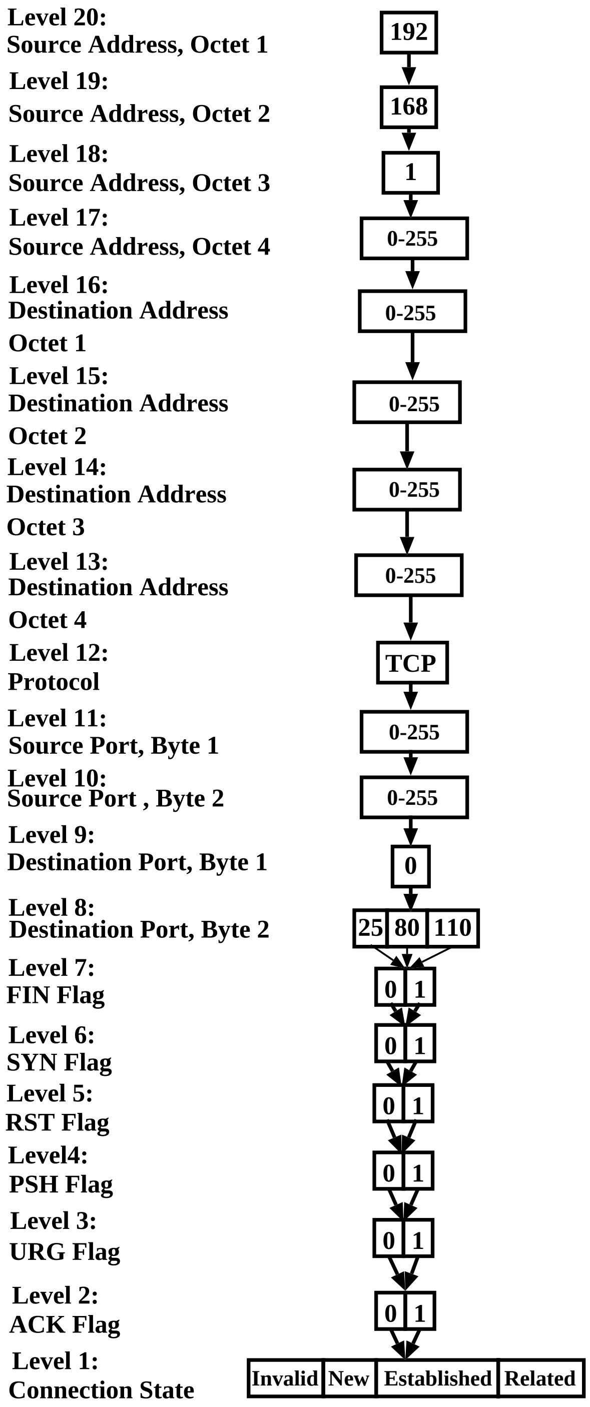
Figure 5: Some example ITVal queries.

Formally, an MDD is a directed acyclic graph in which the nodes are organized into levels and every arc from a node at a level $k > 0$ points to a node at level $k-1$. In ITVal, each level of the MDD corresponds to one attribute of a packet potentially seen by the firewall. Every node of the diagram represents a set of packets that share some common attributes. Each arc at level k represents a choice of value for attribute k. An example MDD for the last query of Figure 5 is shown in Figure 6. Because we do not allow duplicate nodes with all arcs pointing to the same descendants, this means that a path through the MDD represents exactly one packet.

We use MDDs to represent both the rule set of the firewall and a set of queries. To represent a rule set, we add a level of terminal nodes to the bottom of the MDD which correspond to the ultimate fate of the packet (accepted or dropped) as determined by the rule set. To represent the queries, we instead add a level of terminals that indicate whether the packet matches the query conditions or not.

Intuitively, a rule set MDD represents the set of packets accepted by the firewall, while a query MDD represents the set of packets that satisfy the query conditions. When depicting MDDs graphically, we will often show only paths to the ACCEPT node or the MATCHES node. To save space, we also omit the terminal level.

To perform a query, we first represent the query and the rule set as MDDs. The MDD for the rule set is constructed from a rule set description generated using the "iptables -L -n" command. The MDD for the query is generated from a query file provided by the user. We then evaluate the query on the rule set by applying an MDD intersection operator to the two MDDs. The intersection of the two MDDs is the set of packets accepted by the firewall which match the conditions of the query.


Level 20: Source Address, Octet 1
192
Level 19: Source Address, Octet 2
168
Level 18: Source Address, Octet 3
1
Level 17: Source Address, Octet 4
0-255
Level 16: Destination Address Octet 1
0-255
Level 15: Destination Address Octet 2
0-255
Level 14: Destination Address Octet 3
0-255
Level 13: Destination Address Octet 4
0-255
Level 12: Protocol
TCP
Level 11: Source Port, Byte 1
0-255
Level 10: Source Port , Byte 2
0-255
Level 9: Destination Port, Byte 1
0
Level 8: Destination Port, Byte 2
25 80 110
Level 7: FIN Flag
0 1
Level 6: SYN Flag
0 1
Level 5: RST Flag
0 1
Level4: PSH Flag
0 1
Level 3: URG Flag
0 1
Level 2: ACK Flag
0 1
Level 1: Connection State
Invalid New Established Related
Figure 6: An example MDD.

In the following sections, we present the MDD-based infrastructure of the changes made to ITVal to provide support for nested firewalls and NAT. The system administrator should have access to such information in order to increase his understanding of, and trust in, the tool.

Composing Nested Firewalls

In order to extend ITVal to work with multiple firewalls, we introduce the concept of a *meta-firewall*. A meta-firewall is an imaginary firewall that represents a composition of the rule sets of two or more serially connected firewalls. To analyze a meta-firewall, the user passes the names of the rule set description files on the command line. The order of the filenames must reflect the topology of the firewalls, with the innermost filter first on the command line and the outermost filter last.

The meta-firewall has three filter chains analogous to the three built-in chains of a normal firewall. The FORWARD chain of the meta-firewall regulates traffic passing through all the firewalls in either direction. The INPUT chain of the meta-firewall regulates traffic inbound to the innermost firewall through all of the outer firewalls. The OUTPUT chain represents traffic generated by the innermost firewall that successfully passes through the outer firewalls to the outside world.

Queries are performed against the meta-firewall as if it were a single iptables firewall. For instance, the query

```
QUERY FORWARD DPORT FOR 192.168.*
                        AND IN NEW;
```

will list the destination ports of packets bound for the 192.168.0.0/16 subnet that pass through all the firewalls in the set.

To construct the meta-firewall, ITVal joins the MDD for each chain of the component filters using the MDD intersection operator described in [15]. The algorithm is recursive and uses caching to improve performance. Using the caches, we are guaranteed to consider each pair of nodes in the MDDs only once. Because the performance depends only on the number of nodes in each MDD and not on the number of rules, this algorithm scales well to extremely large rule sets and queries.

Pseudocode for generating the meta-firewall is shown in Figure 7. The INPUT chain of the meta-firewall is constructed by intersecting the FORWARD chains of the outer *n-1* firewalls and the INPUT chain of the innermost firewall. The OUTPUT chain is created by intersecting the OUTPUT chain of the innermost firewall with the FORWARD chains of the outer *n-1* firewalls. The FORWARD chain is the intersection of all *n* FORWARD chains.

An MDD depicting the meta-firewall for the rule sets in Figure 2 is shown in Figure 8. In order to save space, only the levels for source address, destination address, and destination port are shown.

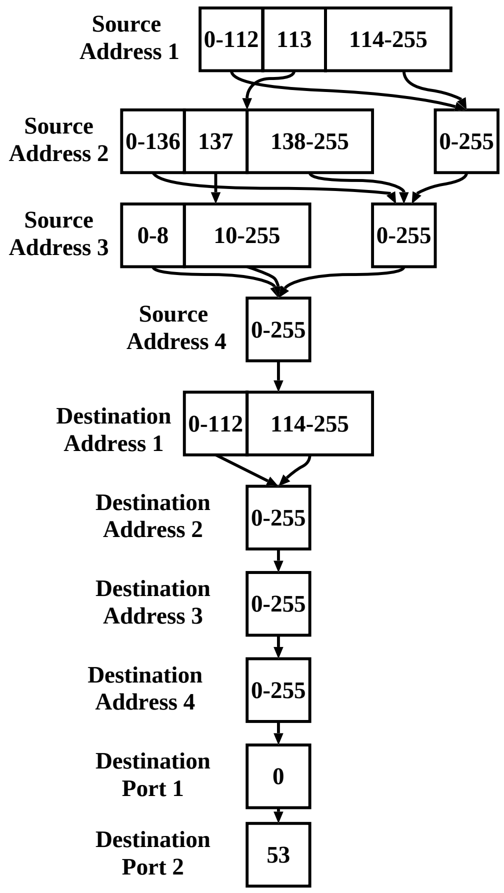


Figure 8: Combining two rule sets into a meta-firewall.

Figure 9 illustrates how ITVal might be used to detect the errors described in section 1. We depict the results of three queries before and after the incorrect

```
Firewall* ConstructFirewall(int n, Firewall* fws)
    newFW = NewFirewall()
    newFW.forward = fws[0].forward
    newFW.input = fws[0].input
    newFW.output = fws[0].output
    for i in 1 to n-1:
        newFW.forward = IntersectMDD(K, newFW.forward, fws[i].forward).
        newFW.input = IntersectMDD(K, newFW.input, fws[i].forward).
        newFW.output = IntersectMDD(K, newFW.output, fws[i].forward).
    return newFW.
```

Figure 7: Algorithm for constructing a meta-firewall.

change. The original, valid, results are shown in Roman font, while the later query results are shown in bold.

Each query corresponds to one invariant that the administrator wishes to preserve. The first query asks which hosts, other than those on the insecure net, can access the web server. In the original results, we see that, as expected, any other host can access the web server. In the modified results, we see that this important invariant still holds.

The second query asks whether the SSH port on the mail server can be accessed from outside the firewall. In the original results, no external machine can reach it. After the modification, however, the results show that the SSH port can be accessed from outside the firewall. SSH traffic from 113.137.8.0/24 can now reach the mail server. By comparing these results, the administrator will realize that she has made a mistake and take steps to correct it.

The last query tests whether services on the mail server are available to the insecure network. In both cases, the answer is no.

Network Address Translation

In addition to filtering packets that pass through the firewall, iptables provides a mechanism for modifying the destination or source address of a packet before and after filtering, respectively. This network

```
>ITVal Example.fw mail.rs mail.nat perimeter.rs perimeter.nat

#First invariant: Web traffic not from insecure net
#can always reach the web server
GROUP insecure 113.137.9.*;
QUERY SADDY INPUT FOR 113.137.10.3 AND NOT FROM insecure AND FOR TCP 80;

# Addresses: [0-112].*.*.*  [114-255].*.*.*  113.[0-136].*.*
# 113.[138-255].*.*  113.137.[0-8].*  113.137.[10-255].*
# 4294967040 results.
```

**```
# Addresses: [0-112].*.*.* [114-255].*.*.* 113.[0-136].*.*
# 113.[138-255].*.* 113.137.[0-8].* 113.137.[10-255].*
# 4294967040 results.
```**

```
#Second invariant: External hosts should never be able to SSH to the mail
# server.
GROUP internal 113.137.10.*;
QUERY DPORT INPUT NOT FROM internal AND FOR TCP 22 AND TO 113.137.10.3;

# Ports:
# 0 results.

# Ports: 22
# 1 results.

#Third invariant: No traffic from the insecure network can reach
#the mail server.
QUERY DPORT INPUT FROM 113.137.9.*;

# Ports:
# 0 results.

# Ports:
# 0 results.
```

Figure 9: Query results before and **after** the change.

Chain PREROUTING (policy ACCEPT)					
	target	prot	source	destination	flags
1	DNAT	all	anywhere	113.137.10.101	TCP dpt:2002 to:192.168.1.2:22
2	DNAT	all	anywhere	113.137.10.101	TCP dpt:2003 to:192.168.1.3:22
3	DNAT	all	anywhere	113.137.10.101	TCP dpt:2004 to:192.168.1.4:22
4	DNAT	all	anywhere	113.137.10.101	TCP dpt:3000 to 192.168.1.2:9999

NAT rules for the intermediate firewall

Chain FORWARD (policy ACCEPT)					
	target	prot	source	destination	flags
1	DROP	tcp	113.137.10.4	192.168.1.0/24	

Filter rules for the intermediate firewall

Figure 10: Rule set of a NAT'ing firewall.
```

address translation (NAT) can also alter the destination and source ports of the packet. Properly handling NAT in the query engine is important, because the modified packet may be treated differently by the filtering rules than the original packet. In order for our queries to take NAT into account, we must modify the rule set MDD to reflect each of the NAT rules.

Like filtering rules, NAT rules are specified using chains. Destination NAT(DNAT) rules for incoming packets are specified in the PREROUTING chain, which is processed before filtering. In addition to the PREROUTING chain, iptables provides a POSTROUTING chain, processed after filtering, which is appropriate for source NAT(SNAT), and an OUTPUT chain for performing DNAT on locally generated packets.

An example rule set for a NAT'ing firewall, which might represent the intermediate firewall in Figure 1 is shown in Figure 10. This firewall protects an internal network 192.168.1.0/24 by hiding the addresses of hosts from the outside world. To access a host, an external system must connect to the NAT'ing firewall, which will forward the connection on the appropriate port. Each NAT rule has a domain and a range. The domain of the NAT rule specifies the set of packets that will be modified. In Figure 10, the domain of the first rule is "all TCP packets from 113.137.10.101 on port 2002". The range of the NAT rule specifies a set of new destination or source addresses and ports. In this case, the range of the rule is "for 192.168.1.2 on port 22". Because NAT can be used for primitive load balancing, the range may be a set of destinations rather than a single value. In this case, packets may be sent to any of the addresses in the range.

In the figure, the PREROUTING chain maps ports 2002-2004 to the SSH ports of internal hosts and also maps port 3000 to some proprietary database software running on port 9999 of machine 192.168.1.2. The FORWARD chain blocks traffic from the web server to those hosts.

Chain FORWARD (policy ACCEPT)					
	target	prot	source	destination	flags
1	DROP	tcp	113.137.10.4	192.168.1.0/24	
2	DROP	tcp	113.137.10.3	113.137.10.101	TCP dpt:3000

Figure 11: Incorrectly configured filter on the NAT'ing firewall.

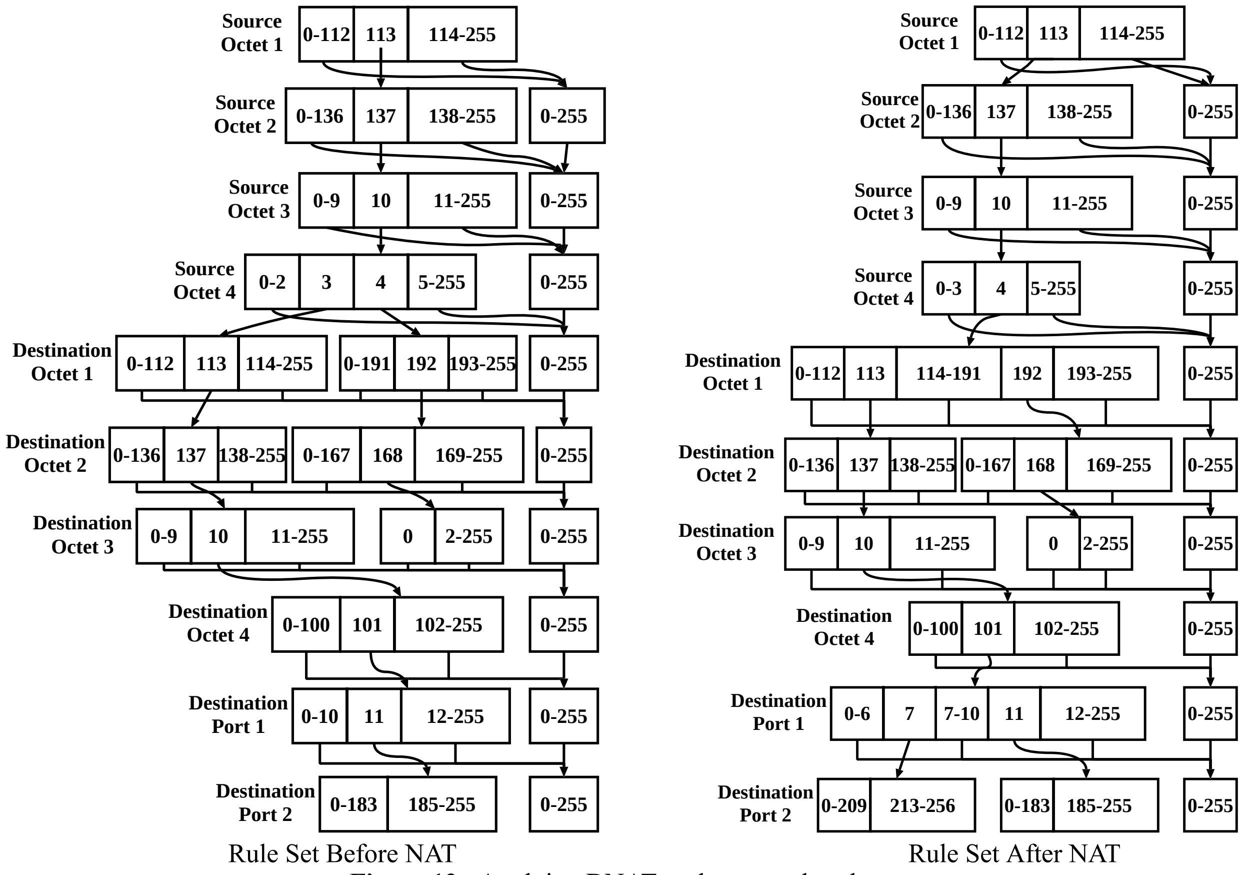


Figure 12: Applying DNAT to the example rule set.

To access host 192.168.1.2 in the example a user should connect to port 2002 on 113.137.10.101. The firewall will then replace the destination address of the packet with "192.168.1.2" and the destination port with "22" before passing it through the filter rules and sending it to the router.

NAT adds another layer of complexity to the configuration of a firewall. One common mistake is to add filtering rules for the original address rather than the NAT'd address. For instance, if the system administrator decides to further restrict access to the internal hosts, she might add rule 2 of Figure 11 to prevent the mail server from accessing the proprietary database. While this blocks connections to the firewall itself, it does not block forwarded connections to the internal hosts. Unfortunately, such a mistake is very subtle and difficult to catch.

To model network address translation using MDDs, we create an operator which takes as inputs a NAT rule and the MDD representation of a set of packets. It produces as output an MDD representing the NAT-modified set of packets. Figure 12 shows the filtering rule set of the example before and after application of the DNAT algorithm. Tracing the MDD from the root node along the path for a packet from address 113.137.10.3 to port 3000 of host 113.137.10.101, we find that the packet will be accepted by the firewall. Pseudocode for performing DNAT on the MDD representation of the rule set of a single firewall is shown in Figure 13. SNAT is implemented similarly.

The algorithm takes as parameters a NAT rule and two MDD nodes at level *k*. The first node represents a set of unmodified packets. The second node represents the same packets after NAT has been applied. Initially, both parameters point to the root node of the rule set MDD. Our goal is to create a new set of rules which map each original packet to the target of its NAT'd equivalent.

Lines 1 and 2 check for the base case condition. If we have passed the levels which correspond to the destination port, we don't need to do any more work as all the NAT related information is contained in the preceding levels. We simply return the node representing the destination of the NAT'd packets.

Lines 3 and 4 check to see if the result has already been computed. If so, it is retrieved from the cache and returned.

Line 5 creates the result node. If the recursion has not yet reached the destination address levels, lines 6-11 set its arcs to the result of the recursive call. Otherwise, we consider each value *i* of the current attribute. If *i* is in the domain of the NAT rule, line 17 creates an arc to child of the NAT'd node. Otherwise, we point arc *i* at the child of node *p*.

Finally, in lines 20 and 21, we remove the node if it duplicates another node. If it is a duplicate, the index of the original node is added to the cache. Otherwise, the index of the result is added to the cache.

Figure 14 provides a query that might be used to detect the error in Figure 11. The group "insecure" is a list of hosts that should be prevented from accessing the secure server. A correctly configured firewall should always return an empty result. After making the incorrect change to the filtering rules, the system administrator adds 113.137.10.3 to the group. Now, if she again runs the query, she will see the results in Figure 15 and detect the error.

```
node_idx DNAT(NAT_RULE nr, level k, node p, node q)
1   if k<DPORTLEVEL:
2      return q.
3   if r = (k,p,q) is in the cache:
4      return r.
5   r = NewNode(k).
6       If k>DADDYLEVEL:
7            for each arc i <k, p>:
8               <k, r>[i] = DNAT(nr, k-1, pchild, pchild).
9            r = RemoveDuplicates(r).
10            Add (k, p, q) = r to the cache.
11            return r.
12   For each arc i ∈ <k, p>:
13       pchild = <k, p>[i]
14       if i ∈ Range(nr[k]):
15           for each value j in nr[k](i):
16               qchild = DNAT(nr, k-1, pchild, <k, p>[j]).
17               <k,r>[i] = Union(<k,r>[i],
                        DNAT(nr, k-1, pchild, qchild)).
18       otherwise:
19           <k,r>[i] = pchild.
20   r = RemoveDuplicates(r).
21   Add (k,p,q) = r to the cache.
22   return r.
```

Figure 13: Algorithm for destination NAT of MDDs.

```
GROUP insecure 113.137.10.3;
QUERY SADDY INPUT FROM insecure
AND FOR TCP 3000;

# Addresses:
# 0 results.
```

Figure 14: Query for detecting errors in the NAT'ing firewall.

```
GROUP insecure 113.137.10.3
113.137.10.4;

QUERY SADDY INPUT FROM insecure
AND FOR TCP 3000;

# Addresses: 113.137.10.4
# 1 result.
```

Figure 15: Results for an incorrectly configured firewall.

Nested Composition with Network Address Translation

In order to analyze a network that contains nested and NAT'd firewalls, the user specifies the filenames of filter rule sets and NAT rule sets alternately on the command line starting with the innermost firewall and working toward the outside.

The pseudocode in Figure 16 combines NAT with analysis of multiple firewalls. The procedure DNAT_ALL applies the chain of DNAT rules pointed to by its first parameter to the MDD specified by the second parameter. The procedure SNAT_ALL works similarly for SNAT.

In order to correctly derive the output chain of the meta-firewall, we work from the outermost firewall toward the innermost firewall combining pairs of firewalls. We DNAT the outermost firewall, then enter a loop in which we intersect the result with the unNAT'd filter rules of the next firewall to be considered. In each iteration, of the loop, we perform SNAT on the result of the intersection using the SNAT rules of the first firewall. We then DNAT using the DNAT rules of the second firewall. This alternating behavior simulates the traversal of a packet first through the PREROUTING chain, then through the filtering rules, and finally through the POSTROUTING chain.

To derive the input and forward chains, we perform the same operations in reverse order, working from the innermost firewall to the outermost firewall.

Conclusions and Future Work

Although modifying rule sets can be a complicated and error-prone process, the use of passive analysis tools can greatly simplify the task. Tools like ITVal that identify firewall configuration errors can prevent subtle mistakes from compromising the long-term security of a network. With our modifications, ITVal can be successfully used to analyze NAT'd firewalls in a hierarchical network topology. While this works well for networks in which the internal workstations share a common policy, developing algorithms for processing a more general topology is desirable.

In addition to supporting destination and source NAT, iptables provides two special case NAT targets. The REDIRECT target rewrites the destination address of a packet so that it will be routed to the firewall itself. The MASQUERADE target rewrites the source address of a packet so that it appears to have been originated by the firewall. The REDIRECT and MASQUERADE targets are extremely useful for environments in which addresses are assigned dynamically, since the address of the original host need not be known apriori when designing the rule set. In order to represent REDIRECT and MASQUERADE rules, ITVal needs to lookup the IP address of the host and perform SNAT or DNAT using the host IP as the new IP address. Although this is not currently implemented in the tool, we plan to add it soon. In order to perform MASQUERADE and REDIRECT, ITVal needs to know the IP addresses of each network interface.

```
Firewall* NAT(int n, Firewall** FW)
     newFW = NewFirewall()
     newFW.forward = DNAT_ALL(fws[0].dnat, fws[0].forward).
     newFW.input = DNAT_ALL(fws[0].dnat, fws[0].input).
     newFW.output = DNAT_ALL(fws[n-1].nat, fws[n-1].output).
     for i in 1 to n-1:
        newFW.forward = IntersectMDD(K, newFW.forward, fws[i].forward).
        newFW.forward = SNAT_ALL(fws[i-1].snat, newFw.forward).
        newFW.forward = DNAT_ALL(fws[i].dnat, newFW.forward).
        newFW.input = IntersectMDD(K, newFW.input, fws[i].forward).
        newFW.input = SNAT_ALL(fws[i-1].snat, newFw.input).
        newFW.input = DNAT_ALL(fws[i].dnat, newFW.input).
        newFW.output = IntersectMDD(K, newFW.output, fws[(n-i)-1].forward).
        newFW.output = SNAT_ALL(fws[(n-i)].snat, newFw.output).
        newFW.output = DNAT_ALL(fws[(n-i)-1].output, newFW.output).
     newFW.forward = SNAT_ALL(fws[n-1].snat, newFw.forward).
     newFW.input = SNAT_ALL(fws[n-1].snat, newFw.input).
     newFW.output = SNAT_ALL(fws[0].snat, newFw.output).
     return newFW.
```

Figure 16: NAT with multiple firewalls.

Since this information cannot be determined from the rule set, we will need to create an input mechanism for specifying these addresses.

Improving the output mechanism could also greatly improve the tool. Queries that return a large number of results can sometimes generate "information overload." Investigating ways to present this information more concisely, perhaps through a graphical tool, would greatly improve the utility of the tool. While system administrators can use ITVal to catch configuration errors, it is still sometimes difficult to identify the particular rules that cause the problem. Labeling arcs in the rule set and result MDDs could address this issue. Identifying the rules that cause a configuration error would also make it possible to perform guided repair of the firewall. Future versions of ITVal may not only catch firewall errors, but give a system administrator suggestions about how to change those errors to satisfy safety and liveness invariants of the network.

The latest version of ITVal is available on the web at http://itval.sourceforge.net/ .

Author Information

Robert Marmorstein is a graduate student at the College of William and Mary and a hard-core free software geek. When he is not hacking away at firewall analysis tools, he can usually be found tinkering with his assortment of Linux and BSD based systems. His e-mail address is rmmarm@wm.edu .

References

[1] Wool, Avishai, Alain Mayer, and Elisha Ziskind, "Fang: A firewall analysis engine," *Proceedings of the IEEE Symposium on Security and Privacy*, May, 2000.

[2] Anderson, Harry, *Introduction to Nessus*, October, 2003.

[3] Barisani, Andrea, "Testing firewalls and ids with ftester," *TISC Insight*, Vol. 5, 2001.

[4] Christiansen, Mikkel and Emmanuel Fleury, "An mtidd based firewall using decision diagrams for packet filtering," *Telecommunication Systems*, Vol. 27:2-4, pp. 297-319, 2004.

[5] Farmer, Dan and Wietse Venema, *SATAN: Security Administrator's Tool for Analyzing Networks*, 1995.

[6] Fyodor, "The art of port scanning," *Phrack*, 7(51), September, 1997.

[7] Gouda, Mohamed G. and Alex X. Liu, "Firewall design: Consistency, completeness, and compactness," *Proceedings of the International Conference on Distributed Computing Systems*. IEEE Computer Society, March 2004.

[8] Hazelhurst, Scott, "A proposal for dynamic access lists for tcp/ip packet filtering," *Technical Report TR-Wits-CS-2001-2*, University of Witwatersrand, April, 2001.

[9] "Internet Security Systems," *Internet Scanner User Guide, Version 7.0 SP 2*, 2005.

[10] Jonah, Kevin, "Multiple firewalls defend against multiplying threats," *Washington Technology*, Vol. 18, Num. 8, July, 2003.

[11] Leon, Mark, "Inside the Firewall: Will Bigger Encryption Keys Keep Your BI Data Safe From Harm?" *Intelligent Enterprise*, May, 2005.

[12] Liu, Alex X., Mohamed G. Gouda, Huibo Heidi Ma, and Anne HH. Ngu, "Firewall queries," *Proc. of the 8th International Conference on Principles of Distributed Systems (OPODIS-04), LNCS 3544*, Springer-Verlag, December, 2004.

[13] Marmorstein, Robert, *Designing and implementing a user library for manipulation of multiway decision diagrams*, MS Project Report, Department of Computer Science, The College of William and Mary, http://www.cs.wm.edu/Pubs/710paper.pdf, 2004.

[14] Marmorstein, Robert, *ITVal Website*, http://itval.sourceforge.net/ , 2005.

[15] Marmorstein, Robert, and Phil Kearns, "A tool for automated iptables firewall analysis," *FREENIX/Open Source Track, 2005 USENIX Annual Technical Conference*, pages 71-82, April, 2005.

[16] Fatti, Anton, Scott Hazelhurst and Andrew Henwood, "Binary decision diagram representation of firewall and router access lists," *Technical Report TR-Wits-CS-1998-3*, University of Witwatersrand, October, 1998.

[17] Todd, Bob, *SARA man page*, http://www-arc.com/sara/sara8.html .

[18] Wool, Avishai, "Architecting the Lumeta Firewall Analyzer," *Proceedings of the 10th USENIX Security Symposium*, August, 2001.