

Firewall Builder

Vadim Kurland

`vadim@fwbuilder.org`

The variety of firewall platforms currently available on the market creates problems for administrators who need to be proficient in many configuration languages and tools. In addition to that, different firewalls significantly deviate in their capabilities in terms of the parameters of the network packets they can inspect and the types of transformations that can be applied. This means administrators not only have to know how to configure different firewalls, but also should understand their differences very well in order to avoid mistakes. This paper presents Firewall Builder, an Open Source framework and the multi-platform firewall configuration tool, that allows the administrator to build a policy for an abstract firewall and then translate it into the actual configuration language. The abstract firewall supported by the Firewall Builder integrates many features found in different existing implementations. When a certain feature is missing on the target firewall, the tool tries to compensate using a workaround. Firewall Builder consists of the GUI frontend, a set of the policy compilers that produce code for different target firewalls, and the API library. Firewall Builder uses object-oriented approach to building firewall policy. The user creates objects representing firewalls, hosts and subnets, as well as IP, ICMP, TCP and UDP services using GUI. These objects can be used in the firewall policy rules; changes made to the object are immediately reflected in all rules of all firewalls using this object. Policy compilers transform high level rules composed in the GUI and produce the code in the configuration language of the target firewall. They can check the policy for common errors and even generate configurations that emulate rules or packet transformations not found in any existing firewall. Currently Firewall Builder can generate configuration files and scripts for the Open Source firewalls based on iptables, ipfilter, pf and ipfw. Its technology has also been used to develop a policy compiler for Cisco PIX. In this paper we will discuss the abstract firewall model supported by Firewall Builder, its properties and their mapping to some target firewall implementations, as well as Firewall Builder's design and implementation.

1 Model of the firewall created by Firewall Builder

Firewall Builder works with a syntetic firewall that combines useful features found in many existing implementations, as well as some new types of rules and packet transformations that can only be achieved through emulation. It represents policy in a high-level abstract terms and "hides" details and difficulties of the real target firewall software. This is somewhat similar

to programming in a high level programming languages versus programming in assembly. High level language allows one to concentrate on the problem, instead of spending lots of time working out details of coding. Our goal is to allow administrator to focus on the policy architecture and its high level design, rather than the language and low level requirements of the actual firewall.

For example, many firewalls require that rules be always associated with interfaces and the direction in which the packet crosses the interface be always explicitly specified. This is not so in Firewall Builder. It turns out, that even experienced administrators tend to make mistakes when choosing an interface and a direction for the given rule of a complex policy. Firewall Builder can use information encoded in the configuration of the firewall object to determine the interface the rule should be associated with. Firewall administrator can build the policy just using 'source', 'destination' and 'service' objects and compiler will generate the code suitable for the firewall platform in use. If the firewall can inspect the packet without information about its ingress and egress interface, then the compiler generates code like that. If the firewall requires information about interface and direction, then the compiler deducts these using configuration of the firewall object and generates the code appropriately. Firewall Builder also allows administrator to specify interface and direction manually. This is necessary because certain rules always require this information and it can not be determined automatically. One example of this is an anti-spoofing rule that matches inbound packets with source address that belongs to the firewall or one of the networks behind it. Rule like that can not be built without prior knowledge of the interface and direction, so Firewall Builder permits manual specification of these as well.

In Firewall Builder the choice whether the interface and direction should be used for every rule becomes a matter of design and personal preference of the administrator. Assigning rules to interfaces is only one of the many aspects in which Firewall Builder can assist the administrator.

Unfortunately some features can not be implemented with a relatively simple workaround if the underlying firewall platform does not provide at least basic set of features to support it. For example, iptables can include time and date into the matching criteria, while other Open Source firewalls supported by Firewall Builder do not provide this functionality. It would require running a daemon or a cron job on the firewall to emulate this feature. This is one of the areas of our future development and improvement.

1.1 Properties of the abstract firewall in Firewall Builder

Here is a brief list of properties of the abstract firewall in Firewall Builder:

- Firewall policy and NAT are represented as a set of rules in a format independent of the actual target firewall platform. Administrator can edit firewall policy by dragging and dropping network objects or services from the tree into rules. The policy rule consists of "Source", "Destination" and "Service" objects and has a parameter that defines its "Action". It can optionally be associated with firewall's interface and may have a parameter to define a direction in which the packet has to cross the interface in order to match.

- The first rule that matches the packet's addresses and port numbers makes the decision about the action taken (accept or drop or reject) and stops processing. This is the default behavior on iptables and PIX; this requires using option "quick" with all generated rules for ipfilter and pf.
- The empty policy should block every packet. On iptables policy compiler uses option "-P" to set the default policy to "DROP"; on other platforms it adds explicit rule at the bottom of the ruleset to achieve this goal.
- Firewall performs address translation before applying the policy rules. This is the default behavior on all supported Open Source firewalls but it requires extra emulation for Cisco PIX.
- Firewall is assumed to be stateful. All supported real firewall platforms support state, but router ACLs don't and therefore require emulation in the compiler. Although we can not emulate real stateful inspection in such routers, we can at least automatically generate rules to match "reply" packets for known protocols.
- Policy rules may be associated with interfaces, but this is not mandatory.
- Negation is supported in Policy and NAT rules. It is possible to include multiple objects in the rule element (e.g. "Source") and apply negation. This usually requires a special processing in the compiler because platforms that support negation usually can do it properly only for a single address. For example, we use user defined chains to implement negation for iptables, while for ipfilter it is done with the help of the "skip" keyword. Policy compiler for OpenBSD pf scans all rules of the policy below the one with negation in order to determine whether the action should be "pass" or "block".
- The following types of the NAT rules are supported:
 - NAT rule that translates source address and optionally port
 - NAT rule that translates destination address and optionally port
 - NAT rule that translates both source and destination addresses and optionally port
 - NAT rule that only translates port numbers
 - NAT rule that does not perform any translation ("no nat" rule)

2 Errors caught by the policy compilers

Policy compilers not only convert high level definition of the policy rules into the target firewall code, they can also analyse it and find many common errors in the configuration of the objects and in the policy design. It is important to catch errors at policy preparation time rather than after it has been deployed or during deployment and Firewall Builder helps to do that.

Policy compilers recognize fatal and non-fatal errors. In a case of a fatal error compiler prints a message and stops processing the policy. In a case of a non-fatal error it prints a message and continues its job. In some cases compiler may implement a workaround to compensate

for an error, in other cases it may ignore certain elements, however it always makes its action clear in the error message it prints. Usually we follow conservative path and tend to treat most errors as fatal because we believe that the compiler must preserve a meaning of the policy and make as little changes in it as possible. One example of a situation that we treat errors as non-fatal is a use of a large address range object in the policy. If the target firewall platform does not support address ranges directly in its configuration language, the compiler must convert it into a set of CIDR address blocks. Although compiler tries to be “smart” about it and uses as little CIDR blocks as possible, this conversion may lead to too many individual rules in the resultant policy. In this case compiler issues a warning that suggests that redesigning the rule using network objects should help reduce its size. This is a non-fatal situation though because such conversion does not change the meaning of the policy, so the compiler continues processing it. Another example of a non-fatal error is when user defines a custom log prefix that exceeds the maximum length accepted by the target firewall. The logging prefix will be truncated, which obviously does not change the meaning of the policy rules. The compiler therefore issues only a warning.

Here are few examples of problems that policy compilers can detect:

- Network object misconfigurations, such as object groups recursively referencing themselves, etc, as well as various misconfigurations of the interfaces in the firewall object, such as illegal IP address / netmask combinations.
- Various situations with missing objects, such as interface of the firewall with missing IP address.
- NAT rules are checked for illegal configurations, such as non-contiguous address ranges used in a single translation rule, NAT rule using unnumbered interface, unsupported types of address and protocol translations, etc.
- “Rule shadowing” can be detected. Shadowing happens when a rule is a superset of a subsequent rule and any packets potentially matched by the subsequent rule have already been matched by the prior rule.

3 GUI

Firewall Builder uses object-oriented approach to the firewall policy design, that is all concepts administrator uses to build policy rules are represented as objects arranged in a tree-like hierarchy. Objects can be gathered in groups, that in turn can be used as objects in rules or other groups. Any object can be used in many places, and once the change is done to its configuration, it is immediately reflected in all rules or groups this object is a member of.

Here is the list of object types supported by Firewall Builder:

- Firewall
- Host

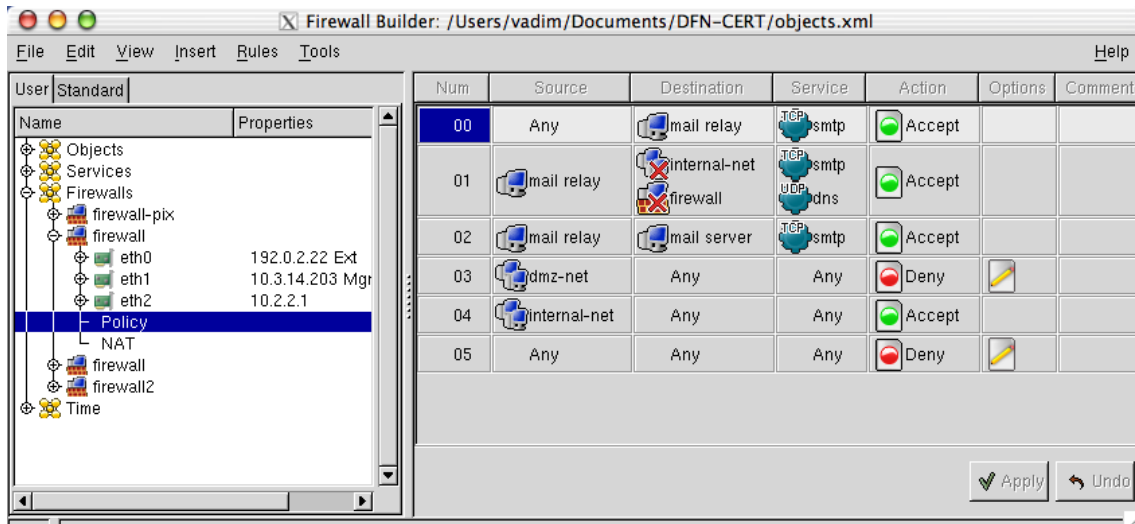


Figure 1: GUI screenshot

- Interface (may belong to a Host or a Firewall object)
- IP address
- MAC address
- Network
- Address range
- A group of hosts, firewalls, networks, address ranges
- Various services: IP, ICMP, TCP, UDP, custom
- A group of services
- Time interval

The administrator creates objects representing hosts, firewalls and networks and then puts together firewall policy using the GUI. Firewall Builder also comes with a few command line tools that help perform simple manipulations on the data files outside the GUI. Once all the objects are there and the policy has been created, administrator can call the policy compiler to generate configuration files or a script for the target firewall. Compiler can work either from within the GUI or as a standalone command line tool. The GUI represents objects and firewall policy rules visually (Fig. 1). The left panel of the GUI window shows the tree where all objects are arranged logically. Interfaces are located in the tree branches under corresponding host or firewall objects, IP addresses are in the branches under corresponding interfaces, and so on. Firewall policy and NAT are represented by a “leaf” objects hanging under their firewall objects. Clicking on the object in the tree opens it in the right panel of the window. User can edit the policy simply by dragging objects from the tree into the rule. The GUI also supports traditional “copy/paste” operation on objects. Hovering mouse cursor over the object in the policy brings up a tooltip which shows a brief summary of the object properties.

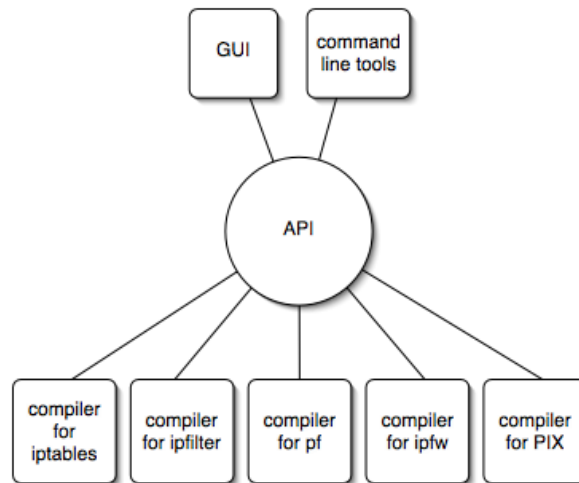


Figure 2: Architecture diagram

The GUI also integrates wizard dialogs that help create new objects, including a network crawler that can discover networks, hosts and firewalls automatically.

4 Standard Objects

Firewall Builder comes with a collection of predefined objects that represent standard and often used networks (such as those defined in RFC1918), as well as over a hundred of typical services. This library of standard objects is constantly expanded. Future versions of Firewall Builder will allow users to create their own libraries of objects, save them to external XML file and then load them when necessary.

5 Design and Implementation

The Firewall Builder toolkit consists of the API library, the GUI and a set of the policy compilers (Fig. 2).

All components use the same API library.

5.1 API Architecture

All access to objects in the Network Object Database is done via C++ API. The data file format is open and is defined in the Firewall Builder XML DTD published on the project's web site. Other developers use Firewall Builder XML DTD for their projects [1], [2].

All API classes are organized into several sub-modules:

1. fwbuilder - Base API classes providing access to objects in the database as well as a few utility classes.
2. fwcompiler - Policy Compiler classes. Provides common classes used to construct Policy Compilers for supported firewall platforms.
3. XML storage and manipulation - provides classes for loading and saving XML files. There is special facility that performs automatic data format upgrades using XSLT transformations which we discuss in the next chapter. ([3]).
4. DNS - a collection of classes that provide methods to resolve host names into IP addresses (both for a single queries and bulk queries), transfer and parse DNS zones. All operations are thread-safe.
5. SNMP - a simple C++ wrapper for SNMP operations. Special classes exist for a high level queries such as extracting information about interfaces, ARP tables, routing tables. All operations are thread-safe.
6. Network Crawler - a sophisticated network discovery process. Given a 'seed' host it finds other hosts and networks and creates corresponding objects. Crawler can be restricted to a single subnet; other restrictions can be applied as well. Network discovery simplifies the task of entering hosts and networks into the network object database.

5.2 Autoupgrades

Firewall Builder stores objects tree, firewall configurations and policies in XML format defined by the Firewall Builder DTD. As project evolves, we make changes to the DTD once in a while, adding new objects or attributes. When such change is made to the DTD, the API library is changed too to provide necessary interfaces to the new objects or attributes. Changes in DTD make data files created with different versions of the program incompatible; to work around this problem we have developed an original mechanism of the automatic upgrades. Classes of the API that load the data file check its version and if it is older than the current version of the API, then the file is processed by the series of XSLT transformations that “upgrade” it. Each transformation makes changes and “upgrades” the file from one version to the next one. If the file has been created in the very old version of the product, then the API will sequentially run the file through multiple transformations to “catch up”. We include suitable XSLT script with every new version of the API that we release to ensure seamless upgrade path for all users, even those who upgrade from the very old versions of the program. The autoupgrade technique is explained in details in [3].

5.3 Policy compiler design

The policy compiler needs to transform abstract policy rules and object definitions into the language of the target firewall platform. This process takes into account specific features of the target firewall, its capabilities and limitations. Sometimes the original high level rule should be split onto many rules in the target firewall language, sometimes objects in the rule

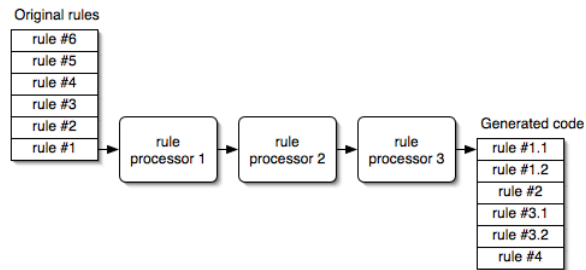


Figure 3: Policy Compiler Architecture

elements (e.g. "source" or "destination") need to be processed and replaced with others. All this makes writing policy compilers a non-trivial task. In order to simplify the process and provide for a way of reusing existing code, we have developed the architecture of the policy compiler shown in Fig. 3.

The policy compiler in Firewall Builder consists of a set of small classes that we call "*Rule Processors*". Each Rule Processor performs a single elementary transformation of the rule or rule elements. For example, there is a class that processes negation in the "Source" element of the policy rule. There is another processor that deal with negation in "Destination". Yet another processor takes care of the rules that require logging, and so on. Few specially designed rule processors inspect policy for typical errors. Many rule processor classes perform common operations for all firewall platforms and are part of the Firewall Builder API. Each processor is derived from the same base class and inherits its methods. The compiler framework puts original rules in the queue and then calls the first rule processor, which takes one rule at a time from the queue, works with it, then puts the result into the output queue and calls the next processor. Some processors split the rule, in which case there will be more rules in the output queue than there were in the input one. Some processors may drop the rule, for example if it is a duplicate, in which case nothing is placed in the output queue.

The policy compiler for a given firewall platform consists of a collection of the rule processors and a framework classes that tie them together and provide a mechanism for passing rules from one to another. The scope of each processor is limited to a single elementary operation, which makes it easier to debug. We constantly expand our library of the standard processors that we reuse when the policy compiler for the new platform is developed.

6 Generated Code

All policy compilers place comments in the configuration file or a script they create to help associate generated commands with original rules defined in the GUI. This is especially important because more often than not, compilers generate several lines in the firewall configuration language for a single rule. Comments in the generated code help administrator figure out what commands have been added for a particular policy rule. Administrator can also attach text comments to rules in the GUI; these comments are placed in the output code as well. Some firewalls support custom logging prefixes (e.g. iptables and pf) which compilers can use to mark generated commands. This is the best way to "tie" log entries to the rules in the GUI,

which helps to debug the policy. Fig 4 demonstrates a fragment of a configuration generated for OpenBSD PF (long lines have been folded to make it readable).

```
#
# Rule 0(NAT)
nat on fxp1 proto {tcp udp icmp} from 192.168.1.0/24 to any -> 222.222.222.222
#
# Rule 2(NAT)
# Translation for the mail server
rdr proto tcp from any to 222.222.222.222 port 25 -> 192.168.1.10 port 25
#
# Rule 3(NAT)
# Redirect to Squid proxy
rdr proto tcp from any to any port 80 -> 127.0.0.1 port 3128
#
# Rule 0(fxp1)
# Automatically generated anti-spoofing rule
block in log quick on fxp1 inet from {222.222.222.222,192.168.1.1} to \
    any label "RULE 0 - DROP"
block in log quick on fxp1 inet from 192.168.1.0/24 to \
    any label "RULE 0 - DROP"
```

Figure 4: A Fragment of the Generated Configuration for OpenBSD PF

7 Examples of Rule Processors

7.1 Example 1: Converting Complex Rule to a Set of Atomic Rules







Num	Source	Destination	Service	Action
00	 netA  netB	 hostC	 http  ftp	 Accept

Figure 5: A Rule With Multiple Objects

Consider a policy rule that uses multiple objects in some of its elements (Fig. 5). If the target firewall does not support object grouping in its configuration language, this rule should be expanded as follows:

Src	Dst	Srv	Action
netA	hostC	http	Accept
netB	hostC	ftp	Accept
netA	hostC	http	Accept
netB	hostC	ftp	Accept

Table 1: Expanded Rule

Note that the rule processor that expands the policy rule as shown above works the same way regardless of what target firewall it compiles the rule for. This transformation happens on a purely abstract level and is going to be the same for a whole class of firewalls that do not support grouping of objects, such as iptables or ipfilter. On the other hand, OpenBSD pf supports object grouping which makes splitting of the rule like that unnecessary, so the policy compiler for pf simply does not use this rule processor.

7.2 Example 2: Translating a Rule With Negation





Num	Source	Destination	Service	Action
00	 netA netB	 hostC	 TCP http	 Accept

Figure 6: A Rule With Negation

Consider a policy rule that uses multiple objects and negation in "Source" (Fig. 6). Many firewalls support negation in their configuration languages, so one might convert this rule as shown in Tab. 2 (using "!" as a generic indication of negation).

Src	Dst	Srv	Action
! netA	hostC	http	Accept
! netB	hostC	http	Accept

Table 2: Expanded Rule (Incorrect)

Unfortunately, this simple translation is incorrect. The original rule says that packets coming not from "netA" and not from "netB" to "hostC" using protocol "HTTP" should be permitted. The rule does not define any action for packets coming from either "netA" or "netB"; the action for these packets should be defined by policy rules below this one. The simple translation shown in Tab. 2 however permits packets coming from "netB". This contradicts original meaning of the rule shown in Fig. 6, the compiler must therefore use different approach.

Tab 3 shows how this rule is translated by the standard rule processor that deals with negation:

Src	Dst	Srv	Action
netA,netB	any	any	Continue
any	hostC	http	Accept

Table 3: Expanded Rule

This rule processor uses internal temporary action "Continue", which means it needs to generate code that will continue inspection of the packet in case source address matches "netA" or "netB". Another specialized rule processor generates code suitable for the target firewall for this action. Different firewall platforms offer various mechanisms that we can use to implement action "Continue", for example in iptables we use user-defined chain, while in ipfilter we use command "skip" to skip next rules. Some firewalls have no built-in mechanism we could use; in cases like that we can not use action "Continue" and have to scan the whole rule

set down from the rule with negation, trying to determine what action needs to be used in its place. OpenBSD PF is an example of such platform.

As in the previous example, this rule processor works on an abstract level and performs its transformation the same way for any firewall. It then passes resultant rules to the next rule processor in the pipeline, which performs its transformation and passes them further. The last processor generates the actual code for the target firewall. Examples of iptables and ipfilter code produced for the rule Fig 6 are shown in Fig 7 (using names for objects “netA”, “netB” and “hostC” instead of IP addresses for clarity):

```
iptables -N TMPC
iptables -A FORWARD -p tcp -d hostC --dport 80 -j TMPC
iptables -A TMPC -s netA -j RETURN
iptables -A TMPC -s netB -j RETURN
iptables -A TMPC -m state --state NEW -j ACCEPT

skip 2 in proto tcp from netA to any
skip 1 in proto tcp from netB to any
pass in quick proto tcp from any to hostC port = 80
```

Figure 7: Iptables and Ipfilter Code for the Rule With Negation

7.3 Example 3: Optimization

Num	Source	Destination	Service	Action
00	hostA hostB	net-1 net-2	TCP http ICMP ICMP	Accept

Figure 8: A Rule With Many Objects

Consider a policy rule that uses multiple objects in all rule elements (Fig. 8). Trivial translation of this rule for the firewall that does not support object grouping leads to $O(N^3)$ complexity (Tab 4).

Src	Dst	Srv	Action
hostA	net-1	http	Accept
hostA	net-1	icmp	Accept
hostA	net-2	http	Accept
hostA	net-2	icmp	Accept
hostB	net-1	http	Accept
hostB	net-1	icmp	Accept
hostB	net-2	http	Accept
hostB	net-2	icmp	Accept

Table 4: Expanded Rule (Too Long)

Rule with two objects in each element “source”, “destination” and “service” leads to eight commands in the generated firewall configuration. A rule with three objects in each rule element would generate 27 commands! To avoid such ineffective code, Firewall Builder applies

optimization to rules like that. The algorithm used for such optimization for iptables generates code as shown in Tab 5:

Chain	Src	Dst	Srv	Action
	hostA	any	any	C1
	hostB	any	any	C1
C1	any	net-1	any	C2
C1	any	net-2	any	C2
C2	any	any	http	Accept
C2	any	any	icmp	Accept

Table 5: Expanded Rule (Shorter)

The idea is that we match on every rule element separately and immediately abort comparison when we discover that the packet does not match any object in some rule element. In this example we compare source address of the packet with addresses of objects "hostA" and then "hostB". If the packet matches any of these, the control will pass to the user defined chain "C1", which will check destination address. If packet's source address does not match either "hostA" or "hostB", control will not pass to the chain "C1" and we continue with rules below those shown in Tab. 5. It only takes one or two address comparisons for the packet that does not match this rule to move on to the next one. Logically equivalent combination of rules shown in Tab. 4 would do 8 comparisons for the same packet, only to find out that it does not match.

This transformation is also shorter since it produces only 6 commands for a rule with two objects in each rule element ("source", "destination" and "service") and 9 commands for a rule with three objects in each rule element. This is only $O(N)$ complexity.

Similar algorithm is used for ipfilter, too, but there we use "skip" command instead of user-defined chains.

7.4 Example 4: Complete Policy

This section represents relatively simple firewall ruleset (Fig. 9) intended for a firewall with three interfaces: the outside connection with IP address "192.0.2.22", internal interface with address "10.3.14.203" and DMZ interface with address "10.2.2.1". The rule #0 permits SMTP connections from anywhere to the mail relay on DMZ (address "10.2.2.100"), which in turn can connect via SMTP and send DNS queries to any host, except to the firewall and internal net (rule #1). Special rule permits SMTP connections from the mail relay to the mail server on internal LAN (rule #2). Mail relay is the only host on DMZ that is permitted to establish connections anywhere, other hosts on DMZ can not do it (rule #3). Internal net can connect to any host with any protocol (rule #4). Rule #1 in the example uses negation in "Destination", its meaning is as follows:

Permit connections on services "SMTP" and "DNS" from the host "mail relay" to anything except "internal-net" and "firewall".

Num	Source	Destination	Service	Action	Time	Options
00	Any	mail relay	TCP smtp	Accept	Any	
01	mail relay	internal-net firewall	TCP smtp UDP dns	Accept	Any	
02	mail relay	mail server	TCP smtp	Accept	Any	
03	dmz-net	Any	Any	Deny	Any	
04	internal-net	Any	Any	Accept	Any	
05	Any	Any	Any	Deny	Any	

Figure 9: Ruleset example in Firewall Builder GUI

Note that this rule does not specify any action in case “mail relay” tries to connect to hosts on “internal-net” or “firewall”. These connections simply do not match rule #1 and the decision should be made by the subsequent rules.

The policy built in the Firewall Builder GUI is shown in Fig 9 and generated code for iptables in Fig 10.

Rule #0 generated commands for both “OUTPUT” and “FORWARD” chains because compiler assumed that object “Any” in the “Source” rule element should match any address, including that of the firewall itself. In iptables, packets originated on the firewall are inspected in the “OUTPUT” chain, while packets crossing the firewall are inspected in the “FORWARD” chain. This is why compiler generated code for both “OUTPUT” and “FORWARD” chain. This behaviour is optional and can be turned off using a checkbox in the “Firewall” tab of the firewall object dialog in Firewall Builder GUI.

Rules #3, #4 and #5 generated code in “INPUT”, “OUTPUT” and “FORWARD” chains because of the same reason. These rules have “Any” in their destination and since compiler assumed that “Any” matched firewall’s address as well, it had to place code in the “INPUT” chain. Code was generated for the “OUTPUT” chain because object in the “Source” was either “Any” or matched one of the interfaces of the firewall.

Rule #1 implements negation in “Destination” using algorithm described in 7.2

Rules #3 and #5 require logging. In iptables, logging is implemented as a target. Since one iptables command can have only a single target, this command can either create a record in the log or make a decision whether to permit or deny the packet, but not both at the same time. Firewall Builder takes this into account and generates a special iptables command to do logging and then uses target “DROP” or “ACCEPT” in a separate command. If the rule has to be split onto several subrules using different chains, compiler automatically creates a temporary user defined chain to do logging and implement an action of the rule to reduce the size of the script it generates. Both rules #3 and #5 illustrate this, each rule generated code in a

```
#
# Rule 0(global)
iptables -A OUTPUT -p tcp -d 10.2.2.100 -dport 25 -m state --state NEW -j ACCEPT
iptables -A FORWARD -p tcp -d 10.2.2.100 -dport 25 -m state --state NEW -j ACCEPT
#
# Rule 1(global)
iptables -N C1.0
iptables -A INPUT -p tcp -s 10.2.2.100 -dport 25 -m state --state NEW -j C1.0
iptables -A INPUT -p udp -s 10.2.2.100 -dport 53 -m state --state NEW -j C1.0
iptables -A FORWARD -p tcp -s 10.2.2.100 -dport 25 -m state --state NEW -j C1.0
iptables -A FORWARD -p udp -s 10.2.2.100 -dport 53 -m state --state NEW -j C1.0
iptables -A C1.0 -d 192.0.2.22 -j RETURN
iptables -A C1.0 -d 10.3.14.203 -j RETURN
iptables -A C1.0 -d 10.2.2.1 -j RETURN
iptables -A C1.0 -d 10.3.14.0/24 -j RETURN
iptables -A C1.0 -m state --state NEW -j ACCEPT
#
# Rule 2(global)
iptables -A FORWARD -p tcp -s 10.2.2.100 -d 10.3.14.30 -dport 25 -m state --state NEW -j
ACCEPT
#
# Rule 3(global)
iptables -N RULE_3
iptables -A INPUT -s 10.2.2.0/24 -j RULE_3
iptables -A OUTPUT -s 10.2.2.0/24 -j RULE_3
iptables -A FORWARD -s 10.2.2.0/24 -j RULE_3
iptables -A RULE_3 -j LOG --log-level info --log-prefix "RULE 3 - DENY "
iptables -A RULE_3 -j DROP
#
# Rule 4(global)
iptables -A INPUT -s 10.3.14.0/24 -m state --state NEW -j ACCEPT
iptables -A OUTPUT -s 10.3.14.0/24 -m state --state NEW -j ACCEPT
iptables -A FORWARD -s 10.3.14.0/24 -m state --state NEW -j ACCEPT
#
# Rule 5(global)
iptables -N RULE_5
iptables -A OUTPUT -j RULE_5
iptables -A INPUT -j RULE_5
iptables -A FORWARD -j RULE_5
iptables -A RULE_5 -j LOG --log-level info --log-prefix "RULE 5 - DENY "
iptables -A RULE_5 -j DROP
```

Figure 10: Iptables Code Generated For Policy in Fig 9

several chains and to avoid duplication of the common commands compiler created additional temporary chain where it used targets "LOG" and "DROP".

Each command that creates a record in the log also uses custom logging prefix that helps identify original rule this command has been generated for. Custom logging prefix is defined in the GUI. Administrator can use special macros in it that are replaced with a rule number, interface name, action and some other parameters at compile time.

8 Related Work

Firewall Builder is not the only project that tries to simplify firewall administration and policy design tasks.

High Level Firewall Language [4] project came up with a universal language for firewalling rules. This language currently supports iptables, ipfilter, ipfw and Cisco ACL. There seems to be no support for the NAT rules though.

Quite a few products try to simplify configuration for a single firewall platform. For example FireHOL [5] is another high level firewall configuration language that supports only iptables. Another project, ISBA [6], helps configure and manage ipfilter firewall.

Neither HLFL, nor single-platform firewall management tools can analyse the policy and catch common problems besides the syntax errors in their respective languages.

The only commercial multi-platform security management platform is produced by Solsoft [7]. Although it really supports a variety of the commercial and Open Source firewalls and represents them to the administrator on a fairly high level of abstraction, the product is very expensive and is targeted at high profile customers.

Bibliography

- [1] cp2fwbuilder, data file converter from Checkpoint Firewall 1 to FwBuilder,
<http://cp2fwbuilder.sourceforge.net/>
- [2] fwboptimizer: Optimizer for fwbuilder
<http://gd.tuwien.ac.at/opsys/linux/sf/subcat/tim/fwboptimizer/>
- [3] Vadim Zaliva, Managing XML Document Versions and Upgrades with XSLT,
http://www.crocodile.org/lord/XML_XSLT_Configuration.html
- [4] High Level Firewall Language,
<http://www.hlfl.org/>
- [5] FireHOL, the iptables stateful packet filtering firewall builder.
<http://firehol.sourceforge.net/>
- [6] Ipfiler ruleset editor and remote management tool,
<http://inc2.com/isba/>
- [7] Solsoft NP, a suite of policy management solutions for network security,
<http://www.solsoft.com/>