

Firewalls with IPTables

Jason Healy, Director of Networks and Systems

Last Updated Mar 18, 2008

Contents

1	Host-based Firewalls with IPTables	5
1.1	Introduction	5
1.2	Concepts	6
1.2.1	The Suffield Firewall Scripts	6
1.3	Installation of the Scripts	7
1.3.1	File Overview	7
1.3.2	Installing the Scripts	7
1.3.3	Configuration	7
1.3.4	Configuration Syntax	8
1.3.5	Variable Types	8
1.4	Protocol Rules	9
1.5	Usage	12

Chapter 1

Host-based Firewalls with IPTables

Last updated 2008/03/18

1.1 Introduction

Most organizations have a firewall which prevents machines on the Internet (or other untrusted networks) from accessing protected machines. While these firewalls form an important part of a network's security, they can't do everything. Specifically, they don't guard against access from so-called "trusted" networks (behind the firewall), which is not always safe.

Therefore, host-based firewalls fill the gap in network security. Host-based firewalls allow us to tailor the types of connections we will accept from all hosts (regardless of their location).

This document describes how to protect Linux machines using **iptables**. iptables have been a standard feature of the Linux kernel since 2.4, and provide a robust means for protecting hosts.

This document is heavily geared to the Debian distribution of Linux, though the concepts should readily apply to other flavors of Linux.

Additionally, this document describes **host-based** Linux firewalls; that is, we only discuss firewalls that protect a single machine. If you want to create a Linux firewall to protect multiple machines (and possibly perform other tasks such as NAT), you'll need to look elsewhere.

1.2 Concepts

iptables has the notion of several **tables** of rules which are organized into **chains**. A packet moving through the filter system is assigned to the appropriate table for the operation being performed on it (filtering, NAT, or manipulation), and each table contains chains based upon where the packet is going (in, out, through, etc).

Each table contains a few built-in chains (such as `INPUT` and `OUTPUT`) where rules can be created. Additionally, the user can create their own chains, and reference them from the built-in chains. In this way, complex rulesets can be created and reused.

All of this makes for a very flexible system that allows complex rulesets to be created. Unfortunately, flexibility and brevity are often competing goals when designing firewall rules. For host-based firewalls, this can cause frustration when the underlying task seems so simple (for example, "deny all traffic except for these few ports that I specify").

1.2.1 The Suffield Firewall Scripts

Creating firewall rulesets can be time-consuming and difficult. For host-based firewalls, we often have a simplified subset of requirements; traffic is allowed or denied based on its source host or network and destination port and protocol.

We've designed a small set of scripts that will automatically generate rulesets from a short configuration file. Most of the common tasks for host-based firewalls (setting a default policy, creating user-defined chains, clearing unused rules) can be performed by the scripts, and the user can simply worry about creating a few rules that are unique to the actual host.

We've designed the scripts to work closely with the Debian flavor of Linux. Debian has a framework in place to run scripts when the network state changes (an interface is brought up or down), and we exploit this framework to automatically apply firewall rules when interfaces are changed. If you don't use Debian, the scripts could easily be modified to work under a traditional init-script system (in fact, that's originally how they were authored).

Read on for specifics of how to use our scripts to quickly and easily create host-based firewall rulesets.

1.3 Installation of the Scripts

1.3.1 File Overview

Our firewall scripts are broken into several smaller scripts. The scripts and files are concentrated in two main directories:

- The `/etc/network/*` directories. Scripts in these directories are automatically invoked by Debian when a network interface changes state.
- The `/etc/default` directory, where configuration files are located. Files in this directory are unique to this host.

The scripts themselves are static, and should not normally require modification to run on a particular host. On the other hand, the configuration files are unique to a particular host, and must be customized for proper operation.

1.3.2 Installing the Scripts

The latest version of the scripts are located in our web repository:

[Suffield Firewall Scripts](#)

The files are laid out in a directory structure that mirrors that of the root Debian file system. Thus, the files in `etc/default` should be copied to `/etc/default/` on your host.

Copy each file to the correct relative location on the host you wish to protect.

1.3.3 Configuration

All configuration takes place on the files in the `/etc/default/` directory. The firewall configuration scripts all start with the word `firewall`. Three files are included with the distribution:

1. `firewall-default-init`: default rules to apply to all interfaces when the firewall is initialized. Any rules that should apply to the machine as a whole go here.
2. `firewall-default-kill`: any special rules to apply when an interface is brought down. By default, all rules for a given interface are cleared (after all, the interface is down at this point, so you can't even get to it). If you want to "leave behind" a rule, you should add it to this file.

3. `firewall-sampleiface-init`: for machines with multiple interfaces, you may wish to override the default rules with ones for a particular interface. Make a copy of this file, changing the word "sampleiface" to be the name of the interface the rules are for (*e.g.*, `eth1` or `eth0:0`).

1.3.4 Configuration Syntax

In reality, the configuration files are simply shell script fragments that get evaluated by the main firewall script. However, it's best to work with them by creating variables that the main script will use as input to its ruleset generators.

The variables used by the script are whitespace-separated values. Thus, you should take care to only include spaces between multiple values, and not to have any extra space.

Also, when appending a value to an existing rule, you should use the shell syntax for variable expansion. For example, to add a new rule on to the end of the `$TCP_ACCEPT` ruleset, you would do the following:

```
TCP_ACCEPT="$TCP_ACCEPT <new_rule_definition>"
```

Compare with the following:

```
TCP_ACCEPT="<new_rule_definition>"
```

Because no expansion of the previous rule is included, all previous rules get obliterated and replaced with the single rule on the right-hand side of the assignment operator.

1.3.5 Variable Types

In general, there are four classes of variables that affect the creation of firewall rules:

1. "Evil" addresses: defines rules for hosts that should not be allowed to communicate with the host **at all**.
2. ICMP rules: rules specific to accepting, rejecting, or denying ICMP packets.
3. TCP rules: rules specific to accepting, rejecting, or denying TCP packets.
4. UDP rules: rules specific to accepting, rejecting, or denying UDP packets.

Evil Addresses

The `EVIL_IPS` variable contains whitespace-separated values of hosts or networks that should not be allowed to communicate with this machine **at all**. Examples include non-routable, malformed, or private netblocks (the default config contains several of these built-in).

To append a netblock to the ruleset, use a line like this:

```
EVIL_IPS="$EVIL_IPS 169.254.0.0/16"
```

That line appends a rule to block the entire Class B subnet starting with 169.254 (the `/16` syntax is CIDR notation for a Class B subnet). To block a single host, use a CIDR mask of `/32`; otherwise, use a CIDR mask appropriate for the netblock size.

Because these addresses are non-routable, we advise keeping these rules in the "default" ruleset file, so they apply to all interfaces on the machine.

1.4 Protocol Rules

In addition to the Evil address rules, three other classes of variables exist to process the major protocols involved in IPv4 communications: ICMP, TCP, and UDP.

For each of these three protocols, three variables exist to define rules. You should place rules in the variable that corresponds to the action that should take place if the packet matches. The packets are matched against the three actions in the order shown below:

1. **DROP**: defines rules for packets that should simply be discarded by the filter. No reply is sent to the host that sent the packets; thus, connection attempts that are dropped will cause the sending host to wait for some time before giving up on the connection. Usually, this is the best choice, as it reveals the smallest amount of information about your host. However, legitimate users will notice a long delay if they try to connect to a port they do not have access to.
2. **REJECT**: defines rules for packets that should not be accepted, but for which some reply should be sent to the host sending the packets. Sending hosts will see a "connection refused" or "port closed" message immediately, rather than waiting for the attempt to time out (as would happen with **DROP**). This is often used for ports like **ident**, which some machines will try to connect to as part of a legitimate query to the system. In these

cases, it's better to give an immediate rejection message, as the timeout associated with a DROP may cause a delay in other legitimate connections.

3. **ACCEPT**: defines rules for packets that should be accepted by the host, bypassing further inspection by the firewall.

Note that the firewall script includes a default action of **DROP** for any packets not explicitly matched by a rule. Therefore, you only need to create **DROP** rules when you wish to explicitly drop packets before processing other rules that would otherwise accept them. Additionally, you must create **REJECT** and **ACCEPT** rules for **all** packets that you do not wish to drop.

ICMP Rules

Three variables are available for ICMP processing: **ICMP_DROP**, **ICMP_REJECT**, and **ICMP_ACCEPT**.

The form of each rule is as follows:

```
ICMP_RULE="AAA.BBB.CCC.DDD/MM|icmp-type"
```

The first part is an IP address in dotted-quad notation, followed by a slash and the CIDR mask that should apply to the IP address. After the mask, a pipe symbol (|) and the ICMP type should be included.

As an example, suppose we wish to accept all ping requests from the 172.16.0.0/12 netblock:

```
ICMP_ACCEPT="$ICMP_ACCEPT 172.16.0.0/12|echo-request"
```

TCP Rules

Three variables are available for TCP processing: **TCP_DROP**, **TCP_REJECT**, and **TCP_ACCEPT**. **These rulesets only apply to initial TCP connection packets** (*e.g.*, those with the SYN flag set). By default, all "established" TCP connections are allowed, so we only allow the rules to affect new incoming connections. Thus, you don't need to worry about return traffic of outgoing TCP connections.

The form of each rule is as follows:

```
TCP_RULE="AAA.BBB.CCC.DDD/MM|src-port:src-port|dst-port:dst-port"
```

Again, the IP address and CIDR mask come first. Next, a range of source ports for the connection should be supplied (to allow connections originating from

any port, use 0:65535). Finally, a range of destination ports (or a single port) should be provided. You may use names from `/etc/services` or numeric ports. The address and port specifications are separated by pipe characters (`|`).

For example, to allow any host to connect to the web server running on the firewalled machine, use a rule like this:

```
TCP_ACCEPT="$TCP_ACCEPT 0/0|0:65535|www"
```

The 0/0 means "any host", the 0:65535 means that the client may initiate the connection from any port, and the `www` means that the connection must be destined for the "www" port on the firewalled machine ("www" in `/etc/services` resolves to port 80).

UDP Rules

Three variables are available for UDP processing: `UDP_DROP`, `UDP_REJECT`, and `UDP_ACCEPT`. Note that UDP is a stateless protocol (it doesn't have the notion of a "connection"), so you may need to create rules to allow return traffic to get back to your machine.

The form of each rule is as follows:

```
UDP_RULE="AAA.BBB.CCC.DDD/MM|src-port:src-port|dst-port:dst-port"
```

Again, the IP address and CIDR mask come first. Next, a range of source ports for the connection should be supplied (to allow connections originating from any port, use 0:65535). Finally, a range of destination ports (or a single port) should be provided. You may use names from `/etc/services` or numeric ports. The address and port specifications are separated by pipe characters (`|`).

For example, to allow any host to connect to the DNS server running on the firewalled machine, use a rule like this:

```
UDP_ACCEPT="$UDP_ACCEPT 0/0|1024:65535|domain"
```

The 0/0 means "any host", the 1024:65535 means that the client may initiate the connection from any *unprivileged* (≥ 1024) port, and the `domain` means that the connection must be destined for the "domain" port on the firewalled machine ("domain" in `/etc/services` resolves to port 53).

1.5 Usage

Under Debian Linux, usage of the scripts is automatic; their placement in the various subdirectories of `/etc/network/` means that the system will automatically call the appropriate firewall scripts when interfaces are brought up and down. Under Debian, this means whenever `ifup`, `ifdown`, or `/etc/init.d/network` are run.

There are four basic times that we should invoke the firewall scripts:

1. When we're bringing up the first network interface on the system
2. When we're bringing up any network interface
3. When we're shutting down any network interface
4. When we're shutting down the last network interface on the system

For the "any interface" cases, the scripts that are invoked simply look for rules specific to that interface. They clear out any existing rules for the interface, and (if defined) build new ones.

For the "first" and "last" cases, the script performs some additional housekeeping. Default policies are set for the input and output chains, and some "helper" rulesets (such as those for TCP state tracking) are created or destroyed.

To cover all these cases, we have four smaller scripts, which each invoke the larger `firewall-base` script. Each script corresponds to one of the cases listed above, and take the appropriate action (or passes the appropriate parameters to the main script).

In general, you should not need to invoke the scripts manually, though you can if something goes wrong with a particular script run. Note that the scripts assume that `iptables` is in a consistent state when they are run; care should be taken to always run the scripts in a full up/down cycle.