Hochschule Ulm

University of Applied Sciences Ulm

Operating Systems

Linux Firewalls

using *'iptables'*

Lucas Mahler

May 19, 2018

Contents

1 Introduction **2**

2 'iptables' **3**

2.1 Packet Filter: netfilter . . . 3

2.1.1 Netfilter Architecture . . . 3

2.2 iptables: Chains and Tables . . . 4

2.3 Data Structures of iptables . . . 5

3 Usage of iptables **6**

3.1 iptables Basics . . . 6

3.2 Defining Rules . . . 7

3.3 Stateful Packet Inspection . . . 8

3.4 Logging . . . 9

3.5 Firewall Scripts . . . 10

4 References **13**

1 Introduction

The term *firewall* historically referred to an actual physical wall which should contain fires within certain areas of a building. This word, over time, transferred to other areas of commerce, like the automotive industry, where it stood for a metal shield in the engine compartment of a vehicle, in place to protect the interior space of it from engine fires. With raising poularity of computing, the term eventually spilled into the territory of computer networking and is nowadays mostly known for this purpose.

In general, *A firewall is a network security device that monitors incoming and outgoing network traffic and decides whether to allow or block specific traffic based on a defined set of security rules.*[1] The first generation of computer network firewalls, in the midst of the 1980s, used so called **packet filters** to implement the desired rules. Here, the packet filter looked at the network address and the port of a packet to determine if it matches a defined rule and if it matches, it is then dropped, rejected or allowed according to this rule. At this time, the *Transmission Control Protocol (TCP)* and the *User Datagram Protocol (UDP)* were the main network communication protocols and using the common ports of them, the packet filter was able to control the traffic on the network.

During the beginnings of the 1990s so called **stateful filters** emerged and introduced the second generation of firewalls. These stateful filters basically performed the work of packet filters but operated, in contrast to first generation packet filters, up to the fourth layer of the OSI refernece model, the transport layer. To achieve this, the stateful filter retains enough packets in its own memory until enough information is gathered to make an educated decision if the packet shall be allowed to pass, dropped or rejected.This storage of packets not only provided more information about incoming packets and thus more security, but also posed a reasonable security risk to *Denial of Service (DoS)* attacks, where the network is flooded with fake connections until the memory of the stateful filter is full and hence denies service.

The third generation of firewalls started to evolve in the midst of the 1990s and expanded the reach of the filters into the application layer , the last layer of the ISO/OSI reference model. This **application layer firewall** "understands" protocols and applications like *HTTP, FTP, SSH etc.* and is thus able to detect if an unwanted application or process tries to bypass the firewall using a protocol on an allowed port.

Next generation firewalls just deepen their knowledge of applications and most often also include an intrusion prevention system, user-ID management, web application firewalls and more.

[1] https://www.cisco.com/c/en/us/products/security/firewalls/what-is-a-firewall.html

2 'iptables'

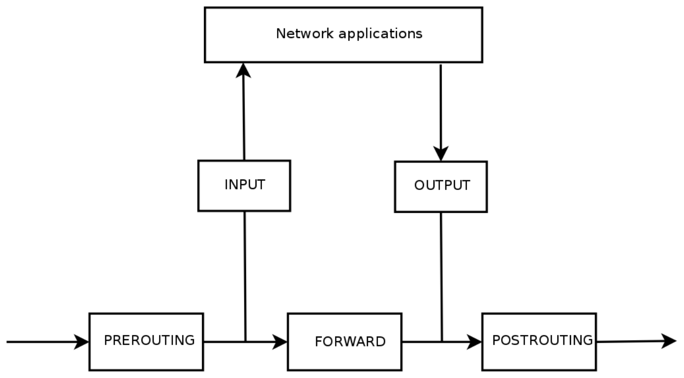
To realize a firewall in a Linux operating system, one has to interfere with the kernel, where the incoming and outgoing packets are processed. This kernel programming is very hard and tedious and needs to be done in C/C++, furthermore, each time one wants to change a certain rule, the complete kernel needs to be recompiled and then run. Thus, nowadays almost all Linux distributions include the *iptables* framework, like Ubuntu, fedora, freeBSD, etc. Iptables is an interface to the packet filter *netfilter*, which is integrated into the Linux kernel and is used to create, administer and inspect IP packet filter rules.

2.1 Packet Filter: netfilter

Netfilter is a packet mangling framework including three main parts. Firstly, so called **hooks** , which are well-defined steps in the packet's traversal of the protocol stacks, are specified. At each hook, the netfilter framework is called via the packet and the hook number. Then, in the second part, specific kernel modules can listen for different hooks for each protocol, thus if a packet is passed to netfilter, it gets checked if anybody registered for that protocol / hook combination. If yes, the packet is examined and then discarded *NF_DROP*, allowed to pass *NF_ACCEPT*, forgotten by netfilter *NF_STOLEN*, or queued for userspace *NF_QUEUE*. Finally, the third and last part is asynchronously collecting the queued packets for sending them to the userspace.

2.1.1 Netfilter Architecture

The Netfilter architecture is basically just a series of hooks in several points in a protocol stack. The packet enters on the left side and is firstly checked for



simple sanity, like is the IP checksum ok, is the packet not truncated and not a promiscuous receive, and then the packet is passed to the *prerouting hook.* Now the packet enters the routing code and is checked if the packet belongs to a local process or to another interface. If the packet is unroutable it may be dropped at this point, but if it's destined to this machine, the *input hook* of the netfilter is called before passing it to the process, if the packet shall be passed to another interface, the netfilter *forward hook* is called instead. Finally, the packet has to pass the final *postrouting hook* before being further communicated. The *output hook* is only called for locally created packets.

Kernel modules can listen to any of these hooks. The function of such a kernel module must specify it's priority, if a hook it listens to occurs. If that netfilter hook is then called from the kernel, each registered module is then called after it's assigned priorities and can freely manipulate the packet at hand. One of these five actions can then be specified by the kernel module to be executed by netfilter: [2]

- **NF_ACCEPT**: continue traversal as normal
- **NF_DROP**: drop the packet; don't continue traversal
- **NF_STOLEN**: I've taken over the packet; don't continue traversal
- **NF_QUEUE**: queue the packet (usually for userspace handling)
- **NF_REPEAT**: call this hook again

2.2 iptables: Chains and Tables

The packet selection system *iptables* has been built on top of the previously discussed netfilter and is the successor of the deprecated *ipchains* which descended from *ipfwadm*. Iptables uses tables to organize it's rules and within these tables, rules are classified according to the decision to be made. Using this selection method, iptables implements a packet filtering table, the *filter table*, a Network Address Translation table, the *NAT table* and a general pre-route packet mangling table, called *mangling table* and others. Inside each table, further organization is done by so called *chaines*, these chains represent the netfilter hooks which are triggered. Similar to netfilter hooks, chains represent the point of interference in a packets path to its destination, that is when a certain rule comes in action.

The **filter table's** main purpose is to filter packets. It allows to match and filter packets in any desireable way, one can look at the contents of a packet and take action accordingly, like *DROPPING, ACCEPTING* or *REJECTING* it. In this filter table most of the filtering work is done.

Inside the **mangling table** the actual change of information in the packet takes place. For example, one may change the time-to-live (TTL) of packet or other information of the IP-header. Furthermore, the mangling table may also internally mark packets to be further processed by, for example other tables or

[2] https://netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO-3.html#ss3.1

other networking software. When a packet is marked, it is not directly altered in any way but rather its representation by the kernel.

The **NAT table** takes packets from the pre-routing and from the post-routing hooks abd is able to alter the destination and source address of these non-local packets. To alter the destination of local packets, the hooks local-out and local-in are used. The main difference to the packet table is, that only the first packet is passed through the table, for all consecutive packets the result of this first rule is then applied as well.

The **raw table's** sole purpose is to mark packets to be able to exit connection tracking. This is useful since iptables supports stateful packet filtering, meaning evaluating packets in relation to their predecessors, which is usually applied right after the packet enters the network interface. Furthermore, a **security table** is included in iptables to internally mark packets with internal SELinux security contexts.

2.3 Data Structures of iptables

To provide meaningful content for these previously mentioned tables, a special data structure is needed, iptables provides this, for convenience reasons, with the same data structures for the userspace and for the kernel. Thus a rule in such a table consists of the following three parts:

- A **struct ipt_entry**
- 0 or more **struct ipt_entry_match** structures, each appended with variable amount of data
- A **struct ipt_entry_target** structure appended with variable amount of data

Due to the variable size of the structures, a lot of freedom is provided by iptables. But this freedom comes with a cost, one has to always be careful about alignment. It has to be ensured that *ipt_entry*, *ipt_entry_match* and *ipt_entry_target* structures are correctly sized and do not exceed the maximum alignment of the machine at hand. The *IPT_ALIGN()* function does exactly that.

A *struct ipt_entry* has a struct *ipt_ip part* which contains the IP header specifications to match, a bitfield indicating what the rule has examined called *nf_cache* and a field indicating the offset from the beginning of this rule where the *ipt_entry_target* structure begins, called *target_offset*. Furthermore, the struct ipt_entry contains also an *next_offset*, indicating the size of this rule, a *comefrom* field which is used for packet traversal tracking by the kernel and lastly, a counter counting the packets and bytes matched by this rule called *struct ipt_counters*. The other two structures, ipt_entry_match and ipt_entry_target, are very similar to this. They contain a field holding the length and name/-pointer combination.

A user basically has for options of interacting with iptables, he can read the current table, read the infom replace the table, and add new counters. The kernel on the other hand, starts the traversal at a location which is indicated by the particular hook. The rule at this point is then examined if the elements of the struct ipt_entry match, then each entry in the struct ipt_entry_match is verified. The iteration can then either stop at this point and can for example immediately drop the packet or continue until the end of the table and increment its counters. Then at the end of the table the struct ipt_entry_target is examined. If this target is standard, the verdict field is read specifying either the verdict or to jump to another point. But if the target is non-standard, the target function is called which returns a verdict since non-standard rules are not allowed to jump.

3 Usage of iptables

Now that we have a basic understanding on how iptables work, we want to look at an actual example. For this example a Linux machine is used, running Ubuntu 16.04 LTS, which includes iptables on default. Since iptables modifies the kernel, we need root privileges. In the following command line examples *sudo* is omitted for the sake of simplicity.

3.1 iptables Basics

A good starting point would be to get an overview of the present iptables configuration and the rules already in place. On a new machine this should usually be empty. The following command will do just that:

```
$ iptables -L

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Now you can see the three default chains of iptables, as discussed earlier, that is *INPUT*, *FOWARD* and *OUTPUT*. Furthermore we see the policy of each chain and the headers of the column of each table. No distinct rules are visible at this point.

Deleting a table can be done using the following command:

```
$ iptables -F
```

But it has to be considered that without parameterization only the filter table will be cleared. To delete the mangle, raw and nat tables they have to be specifically stated:

```
$ iptables -F <CHAIN> [-t <TABLE>]
```

If we want to set the default policy for a chain, the so called *clean up rule*, which is always in place if there was no previous rule that matched, this can be done as follows:

```
$ iptables -P <CHAIN> <TARGET>
```

Where TARGET defines the default behaviour. After restarting the system, all default policies are set to ACCEPT, thus all packets for which no rules exist are allowed to enter the system. But the default policy can only be set for pre-defined chains and not for user-defined ones.

To define a own chain one has to input the following:

```
$ iptables -N <CHAIN>
```

Deleting specific chains can be done equivalently using the options *-X* instead of *-N*.

For defining a new rule one has to enter:

```
$ iptables -A <CHAIN>
```

Here, the newly defined rule will be added to the end of the chain. Using the parameter *-I* one can specify a postition for the new rule in the chain, or without specifying a postition the rule will be added to the first postition of the chain:

```
$ iptables -I <CHAIN> [<POSITION>]
```

Furthermore, to delete a selected rule, the option *-D* is to be used combined with the full name of the rule or its position in the chain:

```
$ iptables -D <CHAIN> <POSITION/RULE>
```

3.2 Defining Rules

Rules are the elementary substance of the firewall, in iptables they consist of multiple filters which describe the packet and also the target, specifying what shall happen with the packet. Iptables provides the following filters:

-
  ```
  $ iptables -i / -o <INTERFACE>
  ```

  describes the input / output interface over which the packet shall be led. The option *-i* can only be used for prerouting input and forward, whereas *-o* can only be used for postrouting, output and forward.

-
  ```
  $ iptables -s / -d <IP-ADDRESS>
  ```

  specifies the source / destination.

-
  ```
  $ iptables -p <PROTOCOL>
  ```

  defines the protocol for the rule, like *tcp, udp, etc.*

-
  ```
  $ iptables --dport <PORT[:PORT]> / --sport <PORT[:PORT]>
  ```

  filters packets on source and destination ports. This is a module specific extension for UDP and TCP.

-
  ```
  $ iptables -m <MODULE>
  ```

  enables the loading of iptables extension modules.

-
  ```
  $ iptables -j <TARGET>
  ```

  defines the target of the rule, meaning that it names what shall happen with a matching target.

Now, packets pass one after the other these chains and since most targets directly interfere with the future of the packets, the order of these rules is of great importance.

As previously mentioned, targets define the future of the packet on which a rule holds, the following actions can be taken:

- *DROP* drops the packet immediately without any comment and leaves the requesting client with a timeout.
- *REJECT* also drops the packet immediately but sends an error message to the sender.
- *ACCEPT* lets the packet pass.

3.3 Stateful Packet Inspection

Stateful packet filtering / inspection (SPI) does not only check the source, destination and protocol of a packet but also the state of connection. To achieve this, it is tested if the packet is to be used to create a new connection or if it belongs to a new one. Therefore the kernel keeps track of existing connections via the *connection tracking* list. Using only the previously introduced rules, this wont be possible.

To use stateful packet filtering one has to import the module *state* into iptables. With the option *–state* the following parameters can be appended:

- *NEW* starts a new connection.
- *ESTABLISHED* states that the packet belongs to an existing connection.
- *RELATED* means that the packet is establishing a new connection but it does this on the basis of an already existing connection.

- *INVALID* says that the packet could not be categorized.

For a TCP connection, netfilter is easily capable of determining if the packet belongs to an existing connection, this is also easily done for UDP connections by examining for example the source, destination, port number or the time of allocation. Thus connection tracking can be used independently without knowing about the protocol what makes it very powerful. In real world applications stateful packet filtering is prevailing, not only for its convenience but also for the increased security.

3.4 Logging

One of the core functionalities of a firewall is the logging of events. This is very crucial for understanding what happens in the network and why certain connections work and others dont. Iptables uses the target *LOG* to achieve this functionality and since LOG is a non-termintating target, it does not change the path of packets. If a packet runs through a LOG filter, it is then handed over to *syslog*. The following example show how the target LOG passes over messages to the file */var/log/syslog*:

```
$ iptables -A OUTPUT -m state --state NEW -p tcp --dport \
 80 -j LOG

$ tail -1 /var/log/syslog
May 19 09:03:12 opsys kernel: [ 4938.400386] IN=
OUT=enp3s0 SRC=192.168.0.104 DST=192.121.140.177 LEN=60\
TOS=0x00 PREC=0x00 TTL=64 ID=43735 DF PROTO=TCP \
SPT=41196 DPT=80 WINDOW=29200 RES=0x00 SYN URGP=0
```

Since this log message is very complicated, one can simplify the output by appending the option *-log-prefix* which expects a string that is prefixed for each logged message:

```
$ iptables -A OUTPUT -m state --state NEW -p tcp \
--dport 80 -j LOG --log-prefix "tcp-outgoing:"
```

This enables the easy recognition and finding of specific rules in the log files. Generally, everything that could be useful for error-detection or detection of security risks should be logged, but since this is not very clear before an incident, usually every prevented connection is logged precationary what requires, depending on the size of the network, a lot of storage capacity over increasing amount of time, thus, if storage is a precious resource, the focus should lie on unauthorized outgoing connections. Hence, the number of outgoing connections should be fairly small, even in a larger, well maintained network. Furthermore, using the *limit* module, iptables is able to only examine a user-defined number of packets in a given timeframe:

```
$ iptables -A INPUT -m state -- state NEW -m limit \
--limit 20/minute -p tcp --dport ssh -j LOG \
--log-prefix "ssh_incoming:"
```

The previously defined rule only protocols 20 packets per minute but other timescales like second, hour or day are also possible. Furthermore, with the option *–limit-burst* it can be specified how many consecutive packets shall be protocolled before making a break, which prevents filling the limit already after a fraction of the desired timeframe, the standard without specifying this option is 5 packets.

3.5 Firewall Scripts

For organizing and storing rules, iptables uses scripts. This makes it very easy to load them on system start up.
As a practical example we want to set up a firewall script for a typical LAMP server. Thus we want to allow outgoing HTTPS, DNS, incoming and outgoing pings, SSH, loopback access, mySQL, IMAP and IMAPS, postfix or sendmail traffic. Furthermore we want to load balance the incoming web traffic, log dropped packets and prevent DDoS attacks.

```
#!/bin/bash

#network interface: eth0
#define variables for the different ip addresses
ADMIN_IP="213.160.70.160"
SERVER_IP1="213.160.72.161:443"
SERVER_IP2="213.160.72.162:443"
SERVER_IP3="213.160.72.163:443"


#first we set the default policies
iptables -P INPUT   DROP
iptables -P OUTPUT  DROP
iptables -P FORWARD DROP

#delete already existing rules
iptables -F

#enble loopback access
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT

#allow current connections
iptables -A INPUT -m state --state RELATED, ESTABLISHED -j ACCEPT
iptables -A OUTPUT -m state --state RELATED, ESTABLISHED -j ACCEPT

#allow HTTP and HTTPS connections
iptables -A OUTPUT -m state --state NEW -multiport -p tcp --dport
    http,https -j ACCEPT

#allow DNS reequests
iptables -A OUTPUT -m state --state NEW -p udp --dport DOMAIN -j
    ACCEPT
iptables -A OUTPUT -m state --state NEW -p tcp --dport DOMAIN -j
    ACCEPT
```

```

#allow incoming and outgoing ping
iptables -A OUTPUT -m state --state NEW -p icmp --icmp-type echo-request -j ACCEPT
iptables -A INPUT -m state --state NEW -p icmp --icmp-type echo-request -j ACCEPT

#allowing all incoming SSH connections
iptables -A INPUT -i eth0 -p tcp --dport 22 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 22 -m state --state ESTABLISHED -j ACCEPT

#allowing outgoing ssh connections
iptables -A OUTPUT -o eth0 -p tcp --dport 22 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A INPUT -i eth0 -p tcp --sport 22 -m state --state ESTABLISHED -j ACCEPT

#load balancing incoming web traffic
#using iptables nth module
iptables -A PREROUTING -i eth0 -p tcp --dport http -m state --state NEW -m nth --counter 0 --every 3 --packet 0 -j DNAT --to-destination SERVER_IP1
iptables -A PREROUTING -i eth0 -p tcp --dport http -m state --state NEW -m nth --counter 0 --every 3 --packet 1 -j DNAT --to-destination SERVER_IP2
iptables -A PREROUTING -i eth0 -p tcp --dport http -m state --state NEW -m nth --counter 0 --every 3 --packet 2 -j DNAT --to-destination SERVER_IP3

#allow mySQL connection only for the admin
iptables -A INPUT -i eth0 -p tcp -s ADMIN_IP --dport 3306 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 3306 -m state --state ESTABLISHED -j ACCEPT

#allow sendmail/postfix traffic
iptables -A INPUT -i eth0 -p tcp --dport 25 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 25 -m state --state ESTABLISHED -j ACCEPT

#allow IMAP traffic
iptables -A INPUT -i eth0 -p tcp --dport 143 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 143 -m state --state ESTABLISHED -j ACCEPT

#allow IMAPS traffic
iptables -A INPUT -i eth0 -p tcp --dport 993 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 993 -m state --state ESTABLISHED -j ACCEPT

#prevent DoS attacks on port 80
```

```
69 iptables -A INPUT -p tcp --dport 80 -m limit --limit 25/minute --
       limit-burst 100 -j ACCEPT
70
71 #logging all dropped packets
72 iptables -N LOGGING
73 iptables -A INPUT -j LOGGING
74 iptables -A LOGGING -m limit --limit 2/minute -j LOG --log-prefix "
       Dropped packages: " --log-level 7
75 iptables -A LOGGING -j DROP
```

In line 7-9 we set the default policy to DROP, that is not explicitly allowed packets are dropped. Then we delete previously defined and unwanted rules in line 13, this is done to prevent any unwanted behaviour of older rules. The next step is to allow packets of the loopback interface, which was disabled by our default policy. In lines 20 and 21 we allow already acknowledged connections, this should also be standard for all iptable scripts. Line 24 uses the multiport module, which enables the use of a comma separated list to specify *–dport* and *–sport* increasing the readability of the script, to allow all incoming and outgoing web traffic like HTTP and HTTPS. Next, we allow outgoing DNS requests and also incoming as well as outgoing ping requests. Now we first allow all incoming and already established SSH connections on the specified interface eth0. For the outgoing connections, in line 39 and 40 the rule slightly differs, here we use the *-o* option to specify the incoming ssh interface. For load balancing in lines 44-46, the rules get slightly mores sophisticated, we are using the *nth* module to load balance the traffic onto three different ip addresses, every 3th packet is load balanced to the appropriate server. We also want to provide access to the MySQL database only for the server admin for development purposes, thus we allow this connection in lines 53 and 54. In order to be able to receive emails, we also need to allow sendmail / postfix traffic, this is done by the lines 57 and 58. To also allow IMAP and IMAPS traffic we do this by the rules in the lines 61-66. Furthermore, we want to protect against simple denial of service attacks, although it is not the perfect solution but a very simple one, we use the *limit* module to limit the maximum connections to 25 per minute. To complete the firewall script we first create a new logging chain and make sure that all connections first pass through it, then we log the packets using a log prefix and then we drop these packets.
Now we can elevate the privileges of the script and make it executable and then run it using:

```
$ chmod +x firewall_script.sh
$ ./firewall_script.sh
```

4 References

https://help.ubuntu.com/community/IptablesHowTo Date of Access: 10.05.18
https://www.digitalocean.com/community/tutorials/how-to-set-up-a-firewall-using-iptables-on
Date of Access: 10.05.18
https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-ar
Date of Access: 10.05.18
https://netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.
html#toc4.2 Date of Access: 10.05.18
http://oceanpark.com/notes/firewall_example.html Date of Access: 14.05.18
https://www.howtoforge.com/bash-script-for-configuring-iptables-firewall
Date of Access: 14.05.18
http://www.linuceum.com/Server/srvFirewallEx.php Date of Access: 14.05.18
https://www.thegeekstuff.com/scripts/iptables-rules Date of Access:
14.05.18

Diemke, Dirk; Kania, Stefan; Khnast, Charly; van Soest, Daniel; Heinlein, Peter: Linux-Server - Das umfassende Handbuch, 3rd Edition, Bonn, Galileo Press, 2014. ISBN: 978-3-8362-3020-9