Simple stateful firewall

From ArchWiki

This page explains how to set up a stateful firewall using iptables. It also explains what the rules mean and why they are needed. For simplicity, it is split into two major sections. The first section deals with a firewall for a single machine, the second sets up a NAT gateway in addition to the firewall from the first section.

Warning: The rules are given in the order that they are executed. If you are logged into a remote machine, you may be locked out of the machine while setting up the rules. You should only follow the steps below while you are logged in locally.

The example config file (https://wiki.archlinux.org/index.php/Simple_Stateful_Firewall#Example_iptables.rules_file) can be used to get around this problem.

Contents

Prerequisites

Note: Your kernel needs to be compiled with iptables support. All stock Arch Linux kernels have iptables support.

First, install the userland utilities `iptables (https://www.archlinux.org/packages/?name=iptables)` or verify that they are already installed.

This article assumes that there are currently no iptables rules set. To check the current ruleset and verify that there are currently no rules run the following:

```
# iptables-save
```

```
# Generated by iptables-save v1.4.19.1 on Thu Aug  1 19:28:53 2013
*filter
:INPUT ACCEPT [50:3763]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [30:3472]
COMMIT
# Completed on Thu Aug  1 19:28:53 2013
```

or

```
# iptables -nvL --line-numbers
```

```
Chain INPUT (policy ACCEPT 156 packets, 12541 bytes)
num   pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
num   pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 82 packets, 8672 bytes)
num   pkts bytes target     prot opt in     out     source               destination
```

If there are rules, you may be able to reset the rules by loading a default rule set:

```
# iptables-restore < /etc/iptables/empty.rules
```

Otherwise, see Iptables#Resetting rules.

Firewall for a single machine

Note: Because iptables processes rules in linear order, from top to bottom within a chain, it is advised to put frequently-hit rules near the start of the chain. Of course there is a limit, depending on the logic that is being implemented. Also, rules have an associated runtime cost, so rules should not be reordered solely based upon empirical observations of the byte/packet counters.

Creating necessary chains

For this basic setup, we will create two user-defined chains that we will use to open up ports in the firewall.

```
# iptables -N TCP
# iptables -N UDP
```

The chains can of course have arbitrary names. We pick these just to match the protocols we want handle with them in the later rules, which are specified with the protocol options, e.g. `-p tcp`, always.

The FORWARD chain

If you want to set up your machine as a NAT gateway, please look at #Setting up a NAT gateway. For a single machine, however, we simply set the policy of the **FORWARD** chain to **DROP** and move on:

```
# iptables -P FORWARD DROP
```

The OUTPUT chain

We have no intention of filtering any outgoing traffic, as this would make the setup much more complicated and would require some extra thought. In this simple case, we set the **OUTPUT** policy to **ACCEPT**.

```
# iptables -P OUTPUT ACCEPT
```

The INPUT chain

Similar to the previous chains, we set the default policy for the **INPUT** chain to **DROP** in case something somehow slips by our rules. Dropping all traffic and specifying what is allowed is the best way to make a secure firewall.

Warning: If you are logged in via SSH, the following will immediately disconnect the SSH session. To avoid it: (1) add the first INPUT chain rule below (it will keep the session open), (2) add a regular rule to allow inbound SSH (to be able to reconnect in case of a connection drop) and (3) set the policy.

```
# iptables -P INPUT DROP
```

Every packet that is received by any network interface will pass the **INPUT** chain first, if it is destined for this machine. In this chain, we make sure that only the packets that we want are accepted.

The first rule added to the INPUT chain will allow traffic that belongs to established connections, or new valid traffic that is related to these connections such as ICMP errors, or echo replies (the packets a host returns when pinged). **ICMP** stands for **Internet Control Message Protocol**. Some ICMP messages are very important and help to manage congestion and MTU, and are accepted by this rule.

The connection state `ESTABLISHED` implies that either another rule previously allowed the initial (`--ctstate NEW`) connection attempt or the connection was already active (for example an active remote SSH connection) when setting the rule:

```
# iptables -A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

The second rule will accept all traffic from the "loopback" (lo) interface, which is necessary for many applications and services.

Note: You can add more trusted interfaces here such as "eth1" if you do not want/need the traffic filtered by the firewall, but be warned that if you have a NAT setup that redirects any kind of traffic to this interface from anywhere else in the network (let's say a router), it will get through, regardless of any other settings you may have.

```
# iptables -A INPUT -i lo -j ACCEPT
```

The third rule will drop all traffic with an "INVALID" state match. Traffic can fall into four "state" categories: NEW, ESTABLISHED, RELATED or INVALID and this is what makes this a "stateful" firewall rather than a less secure "stateless" one. States are tracked using the "nf_conntrack_*" kernel modules which are loaded

automatically by the kernel as you add rules.

Note:

- This rule will drop all packets with invalid headers or checksums, invalid TCP flags, invalid ICMP messages (such as a port unreachable when we did not send anything to the host), and out of sequence packets which can be caused by sequence prediction or other similar attacks. The "DROP" target will drop a packet without any response, contrary to REJECT which politely refuses the packet. We use DROP because there is no proper "REJECT" response to packets that are INVALID, and we do not want to acknowledge that we received these packets.
- ICMPv6 Neighbor Discovery packets remain untracked, and will always be classified "INVALID" though they are not corrupted or the like. Keep this in mind, and accept them before this rule! iptables -A INPUT -p 41 -j ACCEPT

```
# iptables -A INPUT -m conntrack --ctstate INVALID -j DROP
```

The next rule will accept all new incoming **ICMP echo requests**, also known as pings. Only the first packet will count as NEW, the rest will be handled by the RELATED,ESTABLISHED rule. Since the computer is not a router, no other ICMP traffic with state NEW needs to be allowed.

```
# iptables -A INPUT -p icmp --icmp-type 8 -m conntrack --ctstate NEW -j ACCEPT
```

Now we attach the TCP and UDP chains to the INPUT chain to handle all new incoming connections. Once a connection is accepted by either TCP or UDP chain, it is handled by the RELATED/ESTABLISHED traffic rule. The TCP and UDP chains will either accept new incoming connections, or politely reject them. New TCP connections must be started with SYN packets.

Note: NEW but not SYN is the only invalid TCP flag not covered by the INVALID state. The reason is because they are rarely malicious packets, and they should not just be dropped. Instead, we simply do not accept them, so they are rejected with a TCP RST by the next rule.

```
# iptables -A INPUT -p udp -m conntrack --ctstate NEW -j UDP
# iptables -A INPUT -p tcp --syn -m conntrack --ctstate NEW -j TCP
```

We reject TCP connections with TCP RST packets and UDP streams with ICMP port unreachable messages if the ports are not opened. This imitates default Linux behavior (RFC compliant), and it allows the sender to quickly close the connection and clean up.

```
# iptables -A INPUT -p udp -j REJECT --reject-with icmp-port-unreachable
# iptables -A INPUT -p tcp -j REJECT --reject-with tcp-rst
```

For other protocols, we add a final rule to the INPUT chain to reject all remaining incoming traffic with icmp protocol unreachable messages. This imitates Linux's default behavior.

```
# iptables -A INPUT -j REJECT --reject-with icmp-proto-unreachable
```

Example iptables.rules file

Example of `iptables.rules` file after running all the commands from above:

```
/etc/iptables/iptables.rules
```

```
# Generated by iptables-save v1.4.18 on Sun Mar 17 14:21:12 2013
*filter
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
:TCP - [0:0]
:UDP - [0:0]
-A INPUT -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m conntrack --ctstate INVALID -j DROP
-A INPUT -p icmp -m icmp --icmp-type 8 -m conntrack --ctstate NEW -j ACCEPT
-A INPUT -p udp -m conntrack --ctstate NEW -j UDP
-A INPUT -p tcp --tcp-flags FIN,SYN,RST,ACK SYN -m conntrack --ctstate NEW -j TCP
-A INPUT -p udp -j REJECT --reject-with icmp-port-unreachable
-A INPUT -p tcp -j REJECT --reject-with tcp-reset
-A INPUT -j REJECT --reject-with icmp-proto-unreachable
COMMIT
# Completed on Sun Mar 17 14:21:12 2013
```

This file can be generated with:

```
# iptables-save > /etc/iptables/iptables.rules
```

and can be used to continue with the following sections. If you are setting up the firewall remotely via SSH, append the following rule to allow new SSH connections before continuing (adjust port as required):

```
-A TCP -p tcp --dport 22 -j ACCEPT
```

The TCP and UDP chains

The TCP and UDP chains contain rules for accepting new incoming TCP connections and UDP streams to specific ports.

> **Note:** This is where you need to add rules to accept incoming connections, such as SSH, HTTP or other services that you want to access remotely.

Opening ports to incoming connections

To accept incoming TCP connections on port 80 for a web server:

```
# iptables -A TCP -p tcp --dport 80 -j ACCEPT
```

To accept incoming TCP connections on port 443 for a web server (HTTPS):

```
# iptables -A TCP -p tcp --dport 443 -j ACCEPT
```

To allow remote SSH connections (on port 22):

```
# iptables -A TCP -p tcp --dport 22 -j ACCEPT
```

To accept incoming UDP streams on port 53 for a DNS server:

```
# iptables -A UDP -p udp --dport 53 -j ACCEPT
```

See `man iptables` for more advanced rules, like matching multiple ports.

Port knocking

Port knocking is a method to externally open ports that, by default, the firewall keeps closed. It works by requiring connection attempts to a series of predefined closed ports. When the correct sequence of port "knocks" (connection attempts) is received, the firewall opens certain port(s) to allow a connection. See Port Knocking for more information.

Protection against spoofing attacks

> **Note:** `rp_filter` is currently set to `1` by default in `/usr/lib/sysctl.d/50-default.conf`, so the following step is not necessary.

Blocking reserved local addresses incoming from the internet or local network is normally done through setting `rp_filter` (Reverse Path Filter) in sysctl to 1. To do so, add the following line to your `/etc/sysctl.d/90-firewall.conf` file (see sysctl for details) to enable source address verification which is built into Linux kernel itself. The verification by the kernel will handle spoofing better than individual iptables

rules for each case.

```
net.ipv4.conf.all.rp_filter=1
```

This can be done with netfilter instead if statistics (and better logging) are desired:

```
# iptables -t raw -I PREROUTING -m rpfilter --invert -j DROP
```

Note: There is no reason to enable this in both places. The netfilter method is the modern choice and works with IPv6 too.

For niche setups where asynchronous routing is used, the `rp_filter=2` sysctl option needs to be used instead. Passing the `--loose` switch to the `rpfilter` module will accomplish the same thing with netfilter.

"Hide" your computer

If you are running a desktop machine, it might be a good idea to block some incoming requests.

Block ping request

A 'Ping' request is an ICMP packet sent to the destination address to ensure connectivity between the devices. If your network works well, you can safely block all ping requests. It is important to note that this *does not* actually hide your computer — any packet sent to you is rejected, so you will still show up in a simple nmap "ping scan" of an IP range.

This is rudimentary "protection" and makes life difficult when debugging issues in the future. You should only do this for education purposes.

To block echo requests, add the following line to your `/etc/sysctl.d/90-firewall.conf` file (see sysctl for details):

```
net.ipv4.icmp_echo_ignore_all = 1
```

Rate-limiting is a better way to control possible abuse. This first method implements a global limit (ie, only X packets per minute for all source addresses):

```
# iptables -A INPUT -p icmp --icmp-type echo-request -m limit --limit 30/min --limit-burst 8 -j ACCEPT
# iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
```

Or using the 'recent' module, you can impose a limit per source address:

```
# iptables -A INPUT -p icmp --icmp-type echo-request -m recent --name ping_limiter --set
# iptables -A INPUT -p icmp --icmp-type echo-request -m recent --name ping_limiter --update --hitcount 6
# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
```

If you choose to use either the rate limiting or the source limiting rules the PING rule that already exists in the INPUT chain needs to be deleted. This can be done as shown below, or alternatively do not use it in the first place.

```
# iptables -D INPUT -p icmp --icmp-type 8 -m conntrack --ctstate NEW -j ACCEPT
```

Next you need to decide where you wish to place the rate limiting or source limiting rules. If you place the rules below the RELATED,ESTABLISHED rule then you will be counting and limiting new ping connections, not each ping sent to your machine. If you place them before the RELATED,ESTABLISHED rule then these rules will count and limit each ping sent to your machine, not each ping connection made.

More information is in the iptables man page, or reading the docs and examples on the webpage http://www.snowman.net/projects/ipt_recent/

Tricking port scanners

> **Note:** This opens you up to a form of DoS. An attack can send packets with spoofed IPs and get them blocked from connecting to your services.

Port scans are used by attackers to identify open ports on your computer. This allows them to identify and fingerprint your running services and possibly launch exploits against them.

The INVALID state rule will take care of every type of port scan except UDP, ACK and SYN scans (-sU, -sA and -sS in nmap respectively).

ACK scans are not used to identify open ports, but to identify ports filtered by a firewall. Due to the SYN check for all TCP connections with the state NEW, every single packet sent by an ACK scan will be correctly rejected by a TCP RST packet. Some firewalls drop these packets instead, and this allows an attacker to map out the firewall rules.

The recent module can be used to trick the remaining two types of port scans. The recent module is used to add hosts to a "recent" list which can be used to fingerprint and stop certain types of attacks. Current recent lists can be viewed in `/proc/net/xt_recent/` .

SYN scans

In a SYN scan, the port scanner sends SYN packet to every port. Closed ports return a TCP RST packet, or get dropped by a strict firewall. Open ports return a SYN ACK packet regardless of the presence of a firewall.

The recent module can be used to keep track of hosts with rejected connection attempts and return a TCP RST for any SYN packet they send to open ports as if the port was closed. If an open port is the first to be scanned, a SYN ACK will still be returned, so running applications such as ssh on non-standard ports is required for this to work consistently.

First, insert a rule at the top of the TCP chain. This rule responds with a TCP RST to any host that got onto the TCP-PORTSCAN list in the past sixty seconds. The `--update` switch causes the recent list to be updated, meaning the 60 second counter is reset.

```
# iptables -I TCP -p tcp -m recent --update --seconds 60 --name TCP-PORTSCAN -j REJECT --reject-with tcp-
```

Next, the rule for rejecting TCP packets need to be modified to add hosts with rejected packets to the TCP-PORTSCAN list.

```
# iptables -D INPUT -p tcp -j REJECT --reject-with tcp-rst
# iptables -A INPUT -p tcp -m recent --set --name TCP-PORTSCAN -j REJECT --reject-with tcp-rst
```

UDP scans

UDP port scans are similar to TCP SYN scans except that UDP is a "connectionless" protocol. There are no handshakes or acknowledgements. Instead, the scanner sends UDP packets to each UDP port. Closed ports should return ICMP port unreachable messages, and open ports do not return a response. Since UDP is not a "reliable" protocol, the scanner has no way of knowing if packets were lost, and has to do multiple checks for each port that does not return a response.

The Linux kernel sends out ICMP port unreachable messages very slowly, so a full UDP scan against a Linux machine would take over 10 hours. However, common ports could still be identified, so applying the same countermeasures against UDP scans as SYN scans is a good idea.

First, add a rule to reject packets from hosts on the UDP-PORTSCAN list to the top of the UDP chain.

```
# iptables -I UDP -p udp -m recent --update --seconds 60 --name UDP-PORTSCAN -j REJECT --reject-with icmp
```

Next, modify the reject packets rule for UDP:

```
# iptables -D INPUT -p udp -j REJECT --reject-with icmp-port-unreachable
# iptables -A INPUT -p udp -m recent --set --name UDP-PORTSCAN -j REJECT --reject-with icmp-port-unreacha
```

Restore the Final Rule

If either or both of the portscanning tricks above were used the final default rule is no longer the last rule in the INPUT chain. It needs to be the last rule otherwise it will intercept the trick port scanner rules you just added and they will never be used. Simply delete the rule (-D), then add it once again using append (-A) which will place it at the end of the chain.

```
# iptables -D INPUT -j REJECT --reject-with icmp-proto-unreachable
# iptables -A INPUT -j REJECT --reject-with icmp-proto-unreachable
```

Protection against other attacks

See the sysctl#TCP/IP stack hardening for relevant kernel parameters.

Bruteforce attacks

Unfortunately, bruteforce attacks on services accessible via an external IP address are common. One reason for this is that the attacks are easy to do with the many tools available. Fortunately, there are a number of ways to protect the services against them. One is the use of appropriate `iptables` rules which activate and blacklist an IP after a set number of packets attempt to initiate a connection. Another is the use of specialised daemons that monitor the logfiles for failed attempts and blacklist accordingly.

> **Warning:** Using an IP blacklist will stop trivial attacks but it relies on an additional daemon and successful logging (the partition containing /var can become full, especially if an attacker is pounding on the server). Additionally, if the attacker knows your IP address, they can send packets with a spoofed source header and get you locked out of the server. SSH keys provide an elegant solution to the problem of brute forcing without these problems.

Two packages that ban IPs after too many password failures are Fail2ban or, for `sshd` in particular, Sshguard. These two applications update iptables rules to reject future connections from blacklisted IP addresses.

The following rules give an example configuration to mitigate SSH bruteforce attacks using `iptables`.

```
# iptables -N IN_SSH
# iptables -A INPUT -p tcp --dport ssh -m conntrack --ctstate NEW -j IN_SSH
# iptables -A IN_SSH -m recent --name sshbf --rttl --rcheck --hitcount 3 --seconds 10 -j DROP
# iptables -A IN_SSH -m recent --name sshbf --rttl --rcheck --hitcount 4 --seconds 1800 -j DROP
# iptables -A IN_SSH -m recent --name sshbf --set -j ACCEPT
```

Most of the options should be self-explanatory, they allow for three connection packets in ten seconds. Further tries in that time will blacklist the IP. The next rule adds a quirk by allowing a total of four attempts in 30 minutes. This is done because some bruteforce attacks are actually performed slow and not in a burst of attempts. The rules employ a number of additional options. To read more about them, check the original reference for this example: compilefailure.blogspot.com (http://compilefailure.blogspot.com/2011/04/better-ssh-brute-force-prevention-with.html)

Using the above rules, now ensure that:

```
# iptables -A INPUT -p tcp --dport ssh -m conntrack --ctstate NEW -j IN_SSH
```

is in an appropriate position in the iptables.rules file.

This arrangement works for the IN_SSH rule if you followed this entire wiki so far:

```
*
-A INPUT -p icmp -m icmp --icmp-type 8 -m conntrack --ctstate NEW -j ACCEPT
-A INPUT -p tcp --dport 22 -m conntrack --ctstate NEW -j IN_SSH
-A INPUT -p udp -m conntrack --ctstate NEW -j UDP
*
```

The above rules can, of course, be used to protect any service, though protecting the SSH daemon is probably the most often required one.

> **Tip:** For self-testing the rules after setup, the actual blacklist happening can slow the test making it difficult to fine-tune parameters. One can watch the incoming attempts via `cat /proc/net/xt_recent/sshbf` . To unblock the own IP during testing, root is needed `# echo / > /proc/net/xt_recent/sshbf`

Saving the rules

The ruleset is now finished and should be saved to your hard drive so that it can be loaded on every boot.

The systemd unit file points to the location where the rule configuration will be saved:

```
iptables=/etc/iptables/iptables.rules
```

```
ip6tables=/etc/iptables/ip6tables.rules
```

Save the rules with this command:

```
# iptables-save > /etc/iptables/iptables.rules
```

and make sure your rules are loaded on boot enabling the **iptables** daemon.

Check that the rules load correctly by starting `iptables.service` and then checking the status of the service.

IPv6

If you do not use IPv6 (most ISPs do not support it), you should disable it.

Otherwise, you should enable the firewall rules for IPv6. After copying the IPv4 rules as a base:

```
# cp /etc/iptables/iptables.rules /etc/iptables/ip6tables.rules
```

the first step is to change IPs referenced in the rules from IPv4 format to IPv6 format.

Next, a few of the rules (built as example in this article for IPv4) have to be adapted. IPv6 obtained a new ICMPv6 protocol, replacing ICMP. Hence, the reject error return codes `--reject-with icmp-port-unreachable` and `--reject-with icmp-proto-unreachable` have to be converted to ICMPv6 codes.

The available ICMPv6 error codes are listed in RFC 4443 (https://tools.ietf.org/html/rfc4443#section-3.1), which specifies connection attempts blocked by a firewall rule should use `--reject-with icmp6-adm-prohibited`. Doing so will basically inform the remote system that the connection was rejected by a firewall, rather than a listening service.

If it is preferred not to explicitly inform about the existence of a firewall filter, the packet may also be rejected without the message:

```
-A INPUT -j REJECT
```

The above will reject with the default return error of `--reject-with icmp6-port-unreachable`. You should note though, that identifying a firewall is a basic feature of port scanning applications and most will identify it regardless.

In the next step make sure the protocol and extension are changed to be IPv6 appropriate for the rule regarding all new incoming ICMP echo requests (pings):

```
# ip6tables -A INPUT -p icmpv6 --icmpv6-type 128 -m conntrack --ctstate NEW -j ACCEPT
```

Netfilter conntrack does not appear to track ICMPv6 Neighbor Discovery Protocol (the IPv6 equivalent of ARP), so we need to allow ICMPv6 traffic regardless of state for all directly attached subnets. The following should be inserted after dropping `--ctstate INVALID`, but before any other DROP or REJECT targets, along with a corresponding line for each directly attached subnet:

```
# ip6tables -A INPUT -s fe80::/10 -p icmpv6 -j ACCEPT
```

Since there is no kernel reverse path filter for IPv6, you may want to enable one in *ip6tables* with the following:

```
# ip6tables -t raw -A PREROUTING -m rpfilter -j ACCEPT
# ip6tables -t raw -A PREROUTING -j DROP
```

After the configuration is done, enable the **ip6tables** service, it is meant to run in parallel to *iptables*.

Setting up a NAT gateway

This section of the guide deals with NAT gateways. It is assumed that you already read the first part of the guide and set up the **INPUT**, **OUTPUT**, **TCP** and **UDP** chains like described above. All rules so far have been created in the **filter** table. In this section, we will also have to use the **nat** table.

Setting up the filter table

Creating necessary chains

In our setup, we will use another two chains in the filter table, the **fw-interfaces** and **fw-open** chains. Create them with the commands

```
# iptables -N fw-interfaces
# iptables -N fw-open
```

Setting up the FORWARD chain

Setting up the **FORWARD** chain is similar to the **INPUT** chain in the first section.

Now we set up a rule with the **conntrack** match, identical to the one in the **INPUT** chain:

```
# iptables -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
```

The next step is to enable forwarding for trusted interfaces and to make all packets pass the **fw-open** chain.

```
# iptables -A FORWARD -j fw-interfaces
# iptables -A FORWARD -j fw-open
```

The remaining packets are denied with an **ICMP** message:

```
# iptables -A FORWARD -j REJECT --reject-with icmp-host-unreachable
# iptables -P FORWARD DROP
```

Setting up the fw-interfaces and fw-open chains

The meaning of the **fw-interfaces** and **fw-open** chains is explained later, when we deal with the **POSTROUTING** and **PREROUTING** chains in the **nat** table, respectively.

Setting up the nat table

All over this section, we assume that the outgoing interface (the one with the public internet IP) is **ppp0**. Keep in mind that you have to change the name in all following rules if your outgoing interface has another name.

Setting up the POSTROUTING chain

Now, we have to define who is allowed to connect to the internet. Let's assume we have the subnet **192.168.0.0/24** (which means all addresses that are of the form 192.168.0.*) on **eth0**. We first need to accept the machines on this interface in the FORWARD table, that is why we created the **fw-interfaces** chain above:

```
# iptables -A fw-interfaces -i eth0 -j ACCEPT
```

Now, we have to alter all outgoing packets so that they have our public IP address as the source address, instead of the local LAN address. To do this, we use the **MASQUERADE** target:

```
# iptables -t nat -A POSTROUTING -s 192.168.0.0/24 -o ppp0 -j MASQUERADE
```

Do not forget the **-o ppp0** parameter above. If you omit it, your network will be screwed up.

Let's assume we have another subnet, **10.3.0.0/16** (which means all addresses 10.3.*.*), on the interface **eth1**. We add the same rules as above again:

```
# iptables -A fw-interfaces -i eth1 -j ACCEPT
# iptables -t nat -A POSTROUTING -s 10.3.0.0/16 -o ppp0 -j MASQUERADE
```

The last step is to enable IP Forwarding (if it is not already enabled):

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Then edit the relevant line in `/etc/sysctl.d/90-firewall.conf` so it persists through reboot (see sysctl for details):

```
net.ipv4.ip_forward = 1
```

Machines from these subnets can now use your new NAT machine as their gateway. Note that you may want to set up a DNS and DHCP server like **dnsmasq** or a combination of **bind** and **dhcpd** to simplify network settings DNS resolution on the client machines. This is not the topic of this guide.

Setting up the PREROUTING chain

Sometimes, we want to change the address of an incoming packet from the gateway to a LAN machine. To do this, we use the **fw-open** chain defined above, as well as the **PREROUTING** chain in the **nat** table in the following two simple examples.

First, we want to change all incoming SSH packets (port 22) to the ssh server of the machine **192.168.0.5**:

```
# iptables -t nat -A PREROUTING -i ppp0 -p tcp --dport 22 -j DNAT --to 192.168.0.5
# iptables -A fw-open -d 192.168.0.5 -p tcp --dport 22 -j ACCEPT
```

The second example will show you how to change packets to a different port than the incoming port. We want to change any incoming connection on port **8000** to our web server on **192.168.0.6**, port **80**:

```
# iptables -t nat -A PREROUTING -i ppp0 -p tcp --dport 8000 -j DNAT --to 192.168.0.6:80
# iptables -A fw-open -d 192.168.0.6 -p tcp --dport 80 -j ACCEPT
```

The same setup also works with udp packets.

Saving the rules

Save the rules:

```
# iptables-save > /etc/iptables/iptables.rules
```

and make sure your rules are loaded when you boot enabling the **iptables** daemon.

See Also

- Internet sharing
- Router
- Firewalls
- Uncomplicated Firewall
- Methods to block SSH attacks (http://www.webhostingtalk.com/showthread.php?t=456571)
- Using iptables to block brute force attacks (http://www.ducea.com/2006/06/28/using-iptables-to-block-brute-force-attacks/)
- 20 Iptables Examples For New SysAdmins (http://linuxconfig.org/collection-of-basic-linux-firewall-iptables-rules)
- 25 Most Frequently Used Linux IPTables Rules Examples (http://www.thegeekstuff.com/2011/06/iptables-rules-examples/)