# The SuSE Firewall And Masquerading Setup

Version 2.0
February, 1999

Terrehon Bowden <terrehon@pacbell.net>
Bodo Bauer <bb@ricochet.net>

# Contents

# Chapter 1

# Introduction/Credits

This documentation is part of a soon to be released book published by IDG Books on the SuSE Linux distribution.

This document is work in progress, in specially it needs editing in terms of language issues. So please have this in mind when reading it. No need to report spelling errors. If you however find bugs or errors in the technical part, don't hestitate to email to `bb@ricochet.net`, and we will correct this as soon as possible and post an updated version.

A postcript version of this document can be downloaded right here.

As SuSE didn't include the scripts we're talking about, in the current 6.0 release, you will have to downlaod them. The script comming with SuSE work fine for 2.0.* kernels, but they will fail with 2.1.* and 2.2.* kernel versions.

An archive with the new versions is here. Just replace the scripts `/sbin/init.d/firewall` and `/sbin/init.d/masquerade` with the version in the archive.

Not very much cahnged in the functionality, but the internal structure of the scripts is quite different. You'll find documentation for the old version of the scripts in my Firewall Mini HowTo.

## 1.1 Legal Stuff

There is no warranty that the procedures given here work and do no harm. So, if you encounter problems even after following instructions, do not blame us. However, feedback is more then welcome and we'll do my very best to fix bugs and add new features for future releases.

# Chapter 2

# Firewall and Masquerading

Not so long ago, a networked computer was something special. Most machines were standalone. Only the government, big enterprises and universities could afford to build up networks of small computers. This picture has changed drastical in the last years. A computer without any access to a network, in special the Internet became a rarity. It seems like everybody has a email address these days, and a connection to the internet even on his home machine.

This development however has not only positive aspects. There is one technical backdrop, which is that the range of available IP addresses is limited to $2^{32}$ (apr. 4,300 million) valid IP numbers. This sounds like a lot, but free address space is becoming rare and it gets harder and harder to get officially registered IP addresses. The upcoming version 6 of the IP protocol suite (which is already supported by Linux) will extend the available address space. But as long as we are stuck with IPv4, we have limit ourself to use the available space more efficient. One way to do this is hiding whole networks with unregistered addresses behind one registered address. This technique is called masquerading.

The other problem is more a social one. The more people are online, the more abuse of the internet is done. The need to protect you from uninvited guests on your systems becomes important these days. Installing a Firewall is one way of protection. The Linux kernel supports package filtering, which can be used to implement a simple form of a firewall.

This chapter will give an introduction into the Linux package filter mechanisms, which can be used to do both, masquerade packages and building a firewall. SuSE Linux ships with a set of scripts who implement simple firewalling and masquerading using these kernel features. We will see how they work, and how to configure them.

## 2.1 The Kernel package filter

Package filtering is supported by Linux for quite a while. With version 2.2 of the kernel, there is a significant change in the structure of this service. The basic principles however remained the same. There are chains (or tables) of rules against the IP packages are matched. If a package matches a rule, this rule decides what should be done with the package. It can be accepted, denied, rejected, masqueraded or feed into another chain.

With this mechanism it's quite easy to build an stateless package filtering firewall. Stateless means that every packages is been looked independent on the TCP or UDP connection it belongs to. A state full firewall keeps track of connections rather that on single packages.

Linux also support several log and accounting features with it's firewall rules, and masquerading of packages.

### 2.1.1 IP chains

A firewall chain is nothing else than a set of rules, which are used to match a package in a certain order. As soon as a rule matches a package, the target of the rule determines what happens next. If the package doesn't match, the next rule in the chained will be taken. Is the end of the chain reached, than the default policy (the default target) is taken to proceed.

By default there are three predefined chains:

**Input**
: Rules in this chain regulate the acceptance of incoming IP packets. All packets coming in via one of the local network interfaces are checked against the input rules. When no matching rule is found, the default policy for the input chain is used.

**Output**
: These rules define the permissions for sending IP packets. All packets that are ready to be be sent via one of the local network interfaces are checked against the rules of the output chain. When no matching rule is found, the default policy for the output firewall is used.

**Forward**
: These rules define the permissions for forwarding IP packets. All packets sent by a remote host having another remote host as destination are checked against the forwarding chain. Again there is a default policy used for packages with no matching rule.

If we look at a machine acting as router, we can have three types of traffic. Packages sent to this machine, packages originated by this machine, and packages routed through the machine. Table 2.1 shows which chains packages are routed through which of the kernel chains.

You see, that only the packages which come from another host and go to another host have to pass all three chains, packages destinated to the local machine only go though the input chain, packages originated by the local machine only through the output chain. This scheme leaves a lot of flexibility to set up rules for different kind of traffic.

Besides of the predefined chains, you can create new, user defined chains. Rules of the standard chains can forward packages to such a chain. This is useful to minimize the number of rules a package has to go through before a decision is made how this package will be treated. The user defined chains don't really add new functionality to the filter, but they are a nice way to make things easier.

In kernel versions prior to 2.2, a set of accounting rules was present. These have been integrated into the input and output chains with 2.2, and are no longer needed as separate rule set.

### 2.1.2 Firewall rules

We saw that the kernel provides chains of rules to filter IP traffic. We also saw which chain filters what kind of traffic. So the only thing missing are the rules itself.

Firewall rules are like all other rules, they have a condition when to apply them, and a so called target, where to send the package to when the the criterias defined by the rule match it. The target can be the name of a user defined chain, in which case the package will be fed into the beginning of this chain, or one of this special values:

**ACCEPT**
: This means to let the packet through. As soon as a package matches a rule with this target, processing will be stopped and the package can pass.

**DENY**
: This means to drop the packet on the floor. This package is dead and nobody will ever hear about it again, it's gone.

**REJECT**
: Rejecting a package means the same as to drop it, but is more polite and easier to debug, since an ICMP message is sent back to the sender indicating that the packet was dropped. (Note that `DENY` and `REJECT` are the same for ICMP packets).

| | to local host | to remote host |
|---|---|---|
| from local host | | Output |
| from remote host | Input | Input → Forward → Output |

Table 2.1: Packages and IP chains

**MASQ**

This target is only legal for the forward and user defined chains, and can only be used when the kernel is compiled with support for masquerading. With this, packets will be masqueraded as if they originated from the local host. Furthermore, reverse packets will be recognized as such and they will be demasqueraded automatically, bypassing the forwarding chain.

**REDIRECT**

Packages can only be redirected from the input and user-defined chains and can only be used when the Linux kernel is compiled with transparent proxy support. With this, packets will be redirected to a local socket, even if they were sent to a remote host.

**RETURN**

In user defined queues, this means that the processing of the package will be continued at next rule in the previous (calling) chain. In the standard chains this means that the default policy of this chain should be used for the package.

The condition statement is more complex than the targets. IP packages on the stack have a range of parameters which can be used to determine if the package matches a given rule or not. The rules contain a set of ranges or single values for these parameters. Parameters not specified by the rule match any value of this parameter in the package structure. These are the parameters which can be specified in a filter rule:

**Protocol**

The protocol of the packet to check. The specified protocol can be one of TCP, UPD, ICMP, `all` or it can be a numeric value, representing one of these protocols or a different one. Also a protocol name from `/etc/protocols` is allowed.

**Source**

Where does the package come from. The source information contains an IP address, or an address range by giving an address and a netmask It may include a port specification or ICMP type. This can either be a service name, a port number, a numeric ICMP type, or a ICMP type name.

**Destination**

Same values as seen in the source specification, but this time they specify where the package is headed to.

**Interface**

Name of the interface via which a packet is received, or via which is packet is going to be sent.

**Fragment**

This means that the rule only refers to second and further fragments of fragmented packets. Since there is no way to tell the source or destination ports of such a packet (or ICMP type), such a packet will not match any rules which specify them.

**SYN bit set**
Only match TCP packets with the SYN bit set and the ACK and FIN bits cleared. Such packets are used to request TCP connection initiation; for example, blocking such packets coming in an interface will event incoming TCP connections, but outgoing TCP connections will be unaffected. This option is only meaningful when the protocol type is set to TCP.

So, the next question is how to set up these rules, and how to monitor them? The command that inserts, deletes and lists the kernel firewall rules is `/sbin/ipchains`. The exact syntax of this command can be found in the man page `ipchains(8)`. We'll see it in use, when the masquerading script is discussed in the next section.

## 2.2 Masquerading

IP masquerading means, that a router replaces the source information of a IP packages with it's own address and port number before sending the package to it's destination. Reverse packages are recognized as beeing intended to be send to the original host, and will be forwarded to this host.

This means with IP masquerading, we can hide a whole network behind one IP address. This is totally transparent for the hosts inside the network. They won't even notice, that their connections are masqueraded when they leave the local area network and are sent i.e. over the internet to some other host. From the outside however, all traffic looks like it is originated by the masquerading host, as he source address of those packages is always the one of this router.

You can use this for example if you have a dial up connection to your ISP, but you want to connect more than one host at a time to the Internet. As you usually get only one IP address from your ISP, and this address changes every time you connect, you can't go outside your LAN without having masquerading set up. But if you set up your dial up machine as masquerading router, and route all traffic of your other machines through this host, you'll be able to connect to the internet from every machine on your LAN without problems.

Another nice feature is, that the actual network topologie of your LAN is hidden to the outside world. Nobody sees how many machines are in your LAN and how the LAN is structured. If you use non registered IP addresses in your LAN (as specified in RFC1597) nobody will be able to connect directly to machines on your LAN even without a firewall, as those packages won't be routed to your gateway machine. These addresses are shown in table 2.2.

For masquerading it makes no difference what kinds of hosts you have in your LAN, as long as they use TCP/IP protocols to connect to the internet. These

| | | | |
|---|---|---|---|
| Class A net | 10.0.0.0 | → | 10.255.255.255 |
| Class B nets | 172.16.0.0 | → | 172.31.255.255 |
| Class C nets | 192.1680.0 | → | 192.168.255.255 |

Table 2.2: IP ranges for private use

may be Linux machines, but also every kind of UNIX, or Microsoft operating systems can be used in the LAN. Masquerading takes effect on the gateway machine and make no assumptions of the hosts on the LAN.

### 2.2.1 Manual setup

Figure 2.1 shows a simple network setup. One host used as gateway machine to the Internet. The LAN consists of one ethernet segment with a small number of hosts connected to it.

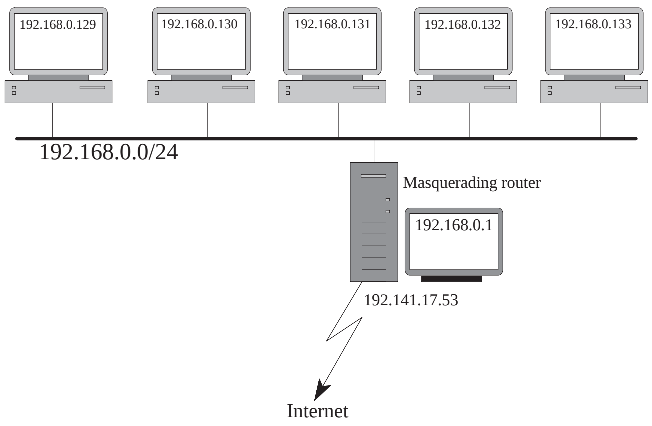


Figure 2.1: Example network for masquerading

All hosts have IP addresses in the class C net 192.168.0.0. The only host with a registered address (192.141.17.53) is the router. This is a typical setup for small offices or home users.

Now lets see how we set up masquerading for the gateway machine. The first to take care of is to ensure, that the kernel is configured for IP masquerading. The kernel configuration is handled by the SuSE manual. You'll find the masquerading flag in the menu for *Networking options*. Make sure that *IP: masquerading* is enabled.

Now we have to set up the firewall rules, to enable masquerading for all outgoing traffic, originated from the hosts on out LAN.

First we set up a new queue named `user_msq` and masquerade all traffic routed through this chain:

```
# /sbin/ipchains -N user_msq
# /sbin/ipchains -A user_msq -s 0/0 -d 0/0 -j MASQ
```

The parameter `-N` for `ipchains` is used to create a new chain. To add new rules to the end of a chain `-A` *chainname* is used. The source specification is made with `-s` and the destination with `-d`. These flags take an IP address as parameter, with an optional netmask and port. In the example we used `0/0`. This covers all possible IP addresses. The general format would be *ip/network address/netmask*. The netmask can be written in the usual format (i.e. 255.255.255.0) for an class C style network, or just by giving the number of bits set in the netmask, where `/24` is the same as `/255.255.255.0`. The target is of the rule is given with the `-j` flag. In this case we give `MASQ` as special target to have the matching packages masqueraded.

Now we need to forward the packages originated from our LAN hosts to this chain. As we have seen in sectionřefsec:ipchains all routed packages will be processed by the forward chain, so we add a rule to this chain, which puts all packages from the network `192.168.0.0/24` into the new `user_msq` chain. One more thing is to consider. As all packages from the local net leave the router on the outbound device, we want to monitor only packages from this device, and masquerade them with this IP address. The flag `-i` takes a device as argument, all other options are like in the statement we've seen a second ago:

```
# /sbin/ipchains -A forward -s 192.168.0.0/24 -d 0/0 -i ppp0 -j user_msq
```

This rule matches all packages beeing forwarded through device `ppp0` with source addresses from the class C network 192.168.0.0 destinated to any address and puts them in the chain `user_msq`.

The setup is almost complete. The IP chains are set up and all packages coming from our LAN machines are masqueraded. So what's missing? Some protocols have weird definitions, which makes it hard to recognize reverse packages. FTP for example is such a protocol. Connections are not only established from the client to the server, but the server also connects to the client. This means, that the Linux kernel has to recognize those connections as part of a conversation between a masqueraded client machine and the FTP server on the Internet, and forward these packages to the right machine on the LAN. This is not part of the standard masquerading in the kernel. However the kernel has modules to support a number of protocols which need special handling. Table 2.3 gives a list of these modules. The `ip_masq_user` module can be used to implement special handling in user space for protocols which are not (yet) supported by other modules.

If we want to use those protocols through the masquerading router, we need to load the referencing protocols as last step of the setup:

```
# /sbin/insmod ip_masq_cuseeme
# /sbin/insmod ip_masq_irc
# /sbin/insmod ip_masq_raudio
# /sbin/insmod ip_masq_vdolive
# /sbin/insmod ip_masq_ftp
# /sbin/insmod ip_masq_quake
```

Now we are done and the masquerading should work quite fine. You can control if all rules are set up OK, by listing them with `ipchains`:

```
# ipchains -L forward -n
Chain forward (policy ACCEPT):
target     prot opt     source               destination          ports
user_msq   all  ------  192.168.0.0/24       0.0.0.0/0            n/a

# ipchains -L user_msq -n
Chain user_msq (1 references):
target     prot opt     source               destination          ports
MASQ       all  ------  0.0.0.0/0            0.0.0.0/0            n/a
```

As you see, you can list a chain by using the flag `-L` with the chain name. Giving no name will list all present chains. The flag `-n` assigns `ipchains` to print the output in numerical values rather than resolving the IP addresses and printing names.

### 2.2.2 Using the SuSE masquerade script

If you have the need of using masquerading, you probably want to have it activated each time you machine is booted. So the obvious idea is to put the command we have seen in a rc-script and let them have executed at boot time. But as you're not the only one who wants to do this, SuSE has scripts to set up masquerading and a firewall in it's distribution. Both scripts are controlled by a set of variables in `/etc/rc.config`

The masquerading script `/sbin/init.d/masquerade` works exactly like we've seen it in section 2.2.1. It sets up a chain `user_msq` and forwards the traffic to be masqueraded in to this chain. It uses four variables to set up the rules:

| | |
|---|---|
| ip_masq_cuseeme | CuSeeMe protocol for video conferencing |
| ip_masq_irc | Internet Relay Chat |
| ip_masq_raudio | Real Audio |
| ip_masq_vdolive | VDO Live |
| ip_masq_ftp | File Transfer Protocol |
| ip_masq_quake | Quake, what else |
| ip_masq_user | User space masquerading control |

Table 2.3: Special masquerading modules

**MSQ_START**
Masquerading is only started at boot time if this variable is set to "yes". However you can start the script manual even if this is set to "no".

**MSQ_DEV**
Device where masquerading takes place. This is the device pointing to the outside network.

**MSQ_NETWORKS**
Space separated list of local networks that should be masqueraded. You can specify an arbitrary number of single IP addresses or networks here.

**MSQ_MODULES**
Modules that are needed for masquerading (see table 2.3).

To implement the same functionality like we had it in section 2.2.1 for the network shown in figure 2.1, the following settings in /etc/rc.config should be used:

```
#
# Masquerading settings
#
MSQ_START="yes"
MSQ_NETWORKS="192.168.0.0/24"
MSQ_DEV="ppp0"
MSQ_MODULES="ip_masq_cuseeme ip_masq_ftp ip_masq_irc \
             ip_masq_quake ip_masq_raudio ip_masq_vdolive"
```

The script supports the parameters start, stop and reload with the obvious functions. You also can get a list of the masqueraded connections by calling it with status. The output looks like this:

```
# /sbin/init.d/masquerade status
BB Masquerading v2.1
IP masquerading entries
prot expire   source               destination          ports
UDP  04:51.46 Netwinder.suse.com   norad-48.mcdn.net    1177 (61031) -> domain
UDP  04:36.02 Netwinder.suse.com   norad-48.mcdn.net    1175 (61028) -> domain
TCP  01:11.29 Netwinder.suse.com   www.apple.com        2153 (61027) -> telnet
TCP  01:56.27 Netwinder.suse.com   sfbay1.yahoo.com     2155 (61032) -> www
```

We see four masqueraded connections originated from the host Netwinder.suse.com. There are two DNS requests to norad-48.mcdn.net, one telnet session to www.apple.com and a HTTP connection to sfbay1.yahoo.com.

## 2.3 The Firewall

Before go on by discussing how a firewall can be set up with the Linux filter rules, we have to think about what we want the firewall to do. The first thing of course is to protect our local area network against intruders from the internet. But we probably don't want to block all traffic from the outside. We may have services which should be accessible, like a WEB, FTP or mail server. There also may be trusted hosts, who should have access to our network. For example contractors, or maintenance accounts are possible visitors we don't want deny access. We may want to control who can access the Internet from the local workstations. I don't like this kind of restriction, but there are possible scenarios where only some workstations are allowed to access the outside world.

### 2.3.1 General thoughts

So the first issue is how to deny connections from the outside, while allowing traffic from the inside. Why is this a problem anyway? Well as the package filters are stateless, they don't know if the connection a package belongs to was initiated from a inside host, or a outside host, in fact they don't know anything about connections anyway. The filter rules look at each package as separate entity.

To distinguish between outbound and inbound traffic we need to know a little bit about the TCP/IP service structure. Lets take SMTP (Simple Mail Transport Protocol) as an example. SMTP is used by MTAs (Mail Transport Agents) to transport mail from one host to another. In order to do this the host (or the MUA at this host) who wants to deliver the mail opens a connection to port 25 on the target machine, knowing (or hoping) that a SMTP server is listening for incoming connections on this port. If so, the both mail agents will negotiate some parameters and the mail will be sent to the target machine. This means, if we block port 25 on our machine, nobody will be able to contact our mail server. Almost all TCP services work this way. There are special ports where a daemon waits for incoming connections, blocking this port will result in a denial of this service. This gives us a policy for selective protection of services for incoming traffic. But what about outgoing connections? Well, TCP/IP has port numbers from 0 to 65535. The first 1024 ports (0 to 1023) are reserved for system services. This means, that outgoing connections have port numbers higher than 1023, so all incoming packages who try to reach ports higher than 1023, are replies to connections initiated by internal requests.

So what we do is block all ports below 1024 and allow traffic to higher ports to pass. This is good as a general rule, but life is hard, and there are exceptions. Some sensible services are located at higher port numbers. The popular HTTP proxy i.e. uses port number 3128 by default. Database servers also tend to use high port numbers, *mysql* i.e. uses port 3306. So just blocking port 0 to 1023 may be not enough. You have to look what you have running on your system to make sure, that you get a complete list of all ports which may be a target for attacks from the outside and make sure to have them blocked.

### 2.3.2 The SuSE firewall script

This time we won't try to set up everything ourself. The firewall is much more complex as masquerading. The detailed description of the masquerading setup in section 2.2.1 was intended to provide a general understanding of how `ipchains` is used to set up firewall rules by giving a real world example. But as the real world is not always as easy as in this case, we'll just jump to the SuSE firewall script here. If you are interested, read the script. With the description given in this section, it is really not hard to understand what is going there. You even may find things you can improve yourself, the script is far from perfect and I am always interested in feedback.

To understand how it works, we'll have an example network again. It's shown in figure 2.2, and you notice, that it is a little bit more complex than the masquerading example.

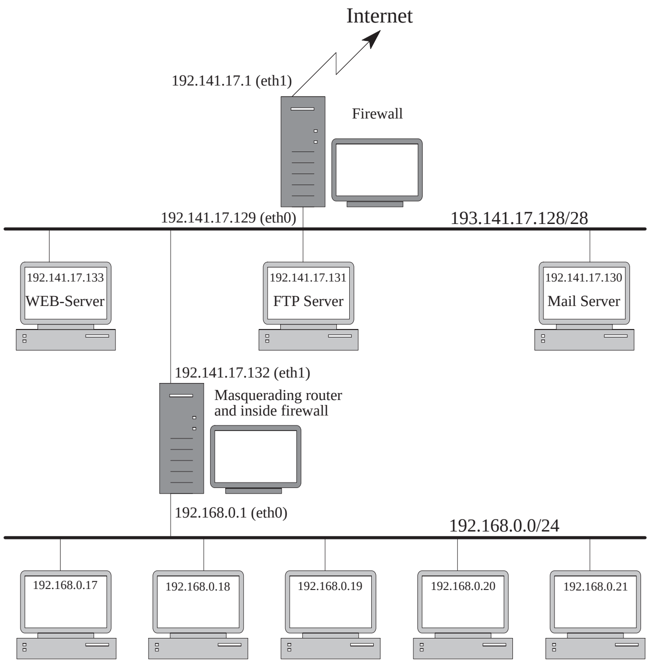


Figure 2.2: Example network for the firewall setup

Like you see, we have two segments this time. The innermost segment uses private IP addresses (192.168.0.0/24) and is connected to the other segment through a masquerading firewall. The outside segment uses registered addresses and has hosts connected to it, who provide services which can and should be accessible from the internet. The connection from this segment to the internet is realized by a firewalling gateway machine. This topologie has some advantages over putting all machines in a single segment. The local workstation are protected by two firewalls. If somebody manages to break through the first firewall, he has only access to machines who provide services accessible from the outside anyway. Of course he can do damage by destroying your WEB site, but the real sensible data is still protected by the second firewall. Under no circumstances should direct access from the outside segment to the innermost LAN be allowed. This would pervert the whole idea of this setup.

#### How does the script work?

With the given knowledge about TCP/IP services, it's easy to design a package filter setup satisfying the given conditions. We add rules to the input chain which forward all incoming packages coming from the device pointing to the outside world and going to an address we want to protect to the user defined chain `user_fw`. All packages coming from the local network and going to the outside will be routed to the user defined chain `user_out`.

Then we add rules to the chain `user_fw`. The first thing to do is to prevent spoofing. Spoofing means that somebody is pretending to be on the inside of the firewall, but he really is coming from the outside. This is usually done by faking the return address by replacing it with an address from our local network. Detecting these kind of packages is rather easy, as no packages from our LAN are coming from the device pointing to the outside, we can drop all packages with a local source address on this device by adding deny rules for these packages. Next thing is to accept packages from trusted hosts by rules with target `ACCEPT` who match packages with those source addresses. Be careful with this, as these packages can be spoofed. Then a list of accept rules follows for the different service such as FTP, HTTP, secure HTTP (SSL), SSH, SMTP, NNTP and DNS. We'll see this in the next paragraph where the configuration is explained. Then the given port ranges for are closed for UPD and TCP traffic. Remember, that the rules are matched in a sequential order. You can add port for services you want to allow here, as the legal packages have been detected by the rules in chain before the packages come to this point. The last thing is to accept everything which hasn't been denied so far.

When the chain for the incoming traffic is set up, rules for outgoing traffic are added to the chain `user_out`. This is only done if you want to monitor outgoing traffic at all. If this is the case, an accept rule is added for each host who should get access. Then a deny rule blocks traffic from all other hosts.

Options are set to accept and deny rules for logging if specified in the setup. Before anything is done, some plausibility checking ensures, that preconditions for a firewall setup (IP forwarding and kernel firewall support present) are satisfied by the present kernel.

### Transparent proxy support

One thing not mentioned so far is support for transparent proxying. As it's not directly related to the firewall problematic I postponed this to this point. The Linux kernel supports (if properly configured) the redirection of packages destinated to remote machines to local ports. This can be used for in- and outbound traffic. For example could you redirect all outgoing connections to port 80 (HTTP) to a local port, to filter them to a local proxy server. Rules for this are also added to the `user_fw` and `user_out` queue if this service is specified by the configuration.

### Configuration of the firewall

Like the masquerading script, the firewall is also configured by a set of variables in `/etc/rc.config`.

These variables have the prefix `FW_` and follow the same format. They contain a list of IP or network addresses with optional net masks, separated by blanks. Exceptions to this rule are described below. You have to use IP addresses, host names won't be resolved. During the setup of the firewall all network traffic is blocked, so no name server requests can be made to resolve names into addresses.

Instead of specifying an IP address you also can use the special string `IP@`*device*. This will be replaced by the IP address the given networking *device* has at the moment the firewall script is started. It's in specially useful if you have a dialup connection where your IP address changes every time you connect to your ISP. If you use PPP dialup, you can specify your outgoing interface as `ppp0` and put `IP@ppp0` in the list of your local networks. In this case however it's important, that you start the firewall after you powered up the connection to your ISP. This means you don't want to start the firewall at boot time, as there is no internet connection at this moment. You rather start the firewall from the script you use to dial up to your ISP.

After this short ex curse, lets see how to configure the firewall anyways. Most of the used variables have pretty obvious meanings if you keep the explanation of the script in mind:

**FW_START**
: The firewall is only started at boot time if this variable is set to `"yes"`. However you can start the script manual even if this is set to `"no"` (i.e. from your dial up script, like discussed above).

**FW_WORLD_DEV**
: Device that should be protected. You can have enter a list of devices here, if you have more then one outgoing device (i.e. virtual devices for WEB servers). All traffic on these devices will be monitored by the firewall rules. If you have PPP dialup connection this may be the device `ppp0`. It may be an ISDN device if you dial out using an ISDN card.

**FW_LOCALNETS**
List of local networks. Only IP addresses listed in here that are protected. If you want to protect the firewall itself, the IP address of it's outbound device has to be listed here.

**FW_TCP_LOCKED_PORTS**
TCP port numbers that should be locked Here you may enter a range that consists of pairs of numbers separated by a colon.

Example: `"1:6 8:1023"`

The ports 1 to 6 and 8 to 1023 are locked. The default is to lock all ports up to 1023. Make sure that you catch all important ports here. Look at section 2.3.2 to understand the meaning of this parameter.

**FW_UDP_LOCKED_PORTS**
UDP port numbers that should be locked Syntax is the same as with the TCP ports. It is recommended to set this to 1:1023 so all reserved ports are locked.

**FW_INT_DEV**
Device for the internal network. Outbound traffic is monitored using this device. As with the world device, you may list more than one device here.

**FW_LOG_DENY**
If this is set to "yes" all violations of the firewall-deny rules are logged to `/var/log/messages`. This means that every attempt of breaking the firewall is logged.

**FW_LOG_ACCEPT**
If this is set to "yes" all packages that suit the firewall-accept rules are logged to `/var/log/messages`. This means that each packages that passes the firewall (allowed) is logged. be careful with this option, as it creates a lot of log entries.

**FW_FTPSERVER**
Addresses of FTP sites that are free accessible from the outside by everyone. This doesn't mean that all services of this machine are available. Only FTP traffic will be allowed, all other traffic will still be blocked (same for all other services covered by separate settings of the firewall).

**FW_WWWSERVER**
Addresses of WWW sites that are accessible from the outside. Same as for FTP, only HTTP connections may be made to this host.

**FW_SSLSERVER**
Addresses of Secure-Socket-Layer (SSL) WWW sites that are accessible from the outside. Requires, that the SSL port is specified in `FW_SSLPORT`.

**FW_SSLPORT**
Port where the SSL server expect requests. Here you may only enter one number.

**FW_MAILSERVER**
Addresses of SMTP sites that are accessible from the outside.

**FW_DNSSERVER**
Addresses of DNS sites that are accessible from the outside.

**FW_NNTPSERVER**
Addresses of NNTP sites that are accessible by news feeds (see below for news feeds).

**FW_NEWSFEED**
Addresses of news feeds that are allowed to connect to the NNTP servers. The two variables `FW_NNTPSERVER` and `FW_NEWSFEED` need both to be set. They are used to setup a set of rules, to allow every listed news feed to access all listed internal news server.

**FW_ROUTER**
Address of the Internet router. This should only be set if the routers address lies in the range given in `FW_LOCALNETS`, but is located on the unprotected side of the firewall. Figure 2.3 illustrates this constellation. You see that the connection to the internet is made with an external router device, connected to the firewall by a destinated ethernet segment (this is important, no other machine has to be in this segment).

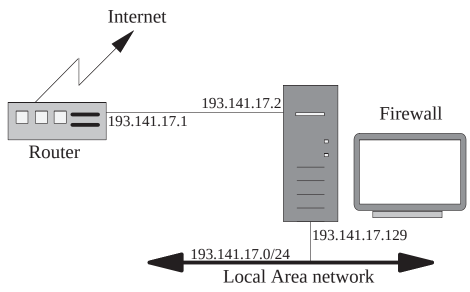


Figure 2.3: Setup with dedicated router

The important (and unfortunate) thing is, that the IP address of this router is within the network we want to protect, in this case the 193.141.17.0/24 network. Without taking special care, all traffic coming from the router would be dropped, as it looks like a spoofing attack (remember IP addresses of the protected net are not supposed to come from the outbound device). By setting the routers address in `FW_ROUTER`, IP packages with this source address can bypass the spoofing filter. But here you see why you should avoid a setup like this. If somebody actually spoofes the router address, he will be able to pass the firewall.

So try to avoid this, by i.e. using subnets to give your router an address that doesn't belong to the inside network. This parameter was added to the firewall setup to be able to handle those constellations in existing networks, where changing to a better topologie was not possible. If you design a network from scratch, don't use this feature!

**FW_INOUT**
If this is set to "yes" `/etc/fw-inout` is read. Otherwise every machine on the local network full access to the Internet (see below for more).

**FW_TRANS_PROXY_IN**
Here you may enter a list of ports and IP addresses for redirecting incoming traffic on-the-fly to local ports. Each entry consists of source IP of the packages, destination IP and port, and the local port it should be redirected to, separated by commas:

`Source IP,Target IP,Target Port,Local Port`

This in particular means that any package that comes from a host with *Source IP* and is aimed for a machine `Target IP` to port *Target port* are redirected to the *Local Port*. I.e. if you want to redirect all traffic to any WEB server in your network to the local web server, you may want to set `FW_TRABS_PROXY_IN` to:

`0/0,193.141.17.0/0,80,80`

**FW_TRANS_PROXY_OUT**
Same as above but for outgoing connections. `FW_TRANSPROXY_OUT` has the exact same meaning for outgoing traffic as `FW_TRANSPROXY_IN` for incoming packages. The difference is that `_OUT` filters traffic on the device given in `FW_INT_DEV`, whereas `_IN` does the same on `FW_WORLD_DEV`.

**FW_FRIENDS**
If this is set to "yes" the file `/etc/friends` is read. Otherwise no machine on the Internet net has full access to the local network (see below for more).

**FW_SSH**
This for opening the SSH port (port 22) for those hosts listed in `/etc/fw-ssh` (see below for more).

Besides of these settings in `/etc/rc.config`, the firewall reads some files in the directory `/etc`, to get additional host lists:

**/etc/fw-friends**
Machines that have uncontrolled access to the local net are listed in this file. Enter each trusted machine's IP address on separate lines with one machine per line. Comments may be inserted on lines beginning with a

pound sign (#). This file will only be read if `FW_FRIENDS` is set to `yes`. Otherwise no machine from the outside has full access to the local net.

**/etc/fw-inout**
Only the hosts listed here have direct access to the Internet if `FW_INOUT` is set to `yes`. Every machine not included in this list will be blocked. Comments are marked with a pound sign (#) again. If `FW_INOUT` is set to `no` any machine of the local network may access the Internet.

**/etc/fw-ssh**
Just as above this file contains a list of IP addresses. If `FW_SSH` is set to `yes` all listed machines have access to port 22. This means they may access to the `sshd` (secure shell daemon) on the local net.

### Using the script

The example network shown in figure 2.2 has two machines set up as firewalls. Table 2.4 shows the variable settings for this two machines. You see that we don't need to use all the features for this simple example.

| Variable | inside Firewall | outside firewall |
|---|---|---|
| FW_START | yes | yes |
| FW_WORLD_DEV | eth1 | eth1 |
| FW_LOCALNETS | 192.168.0.0/24 | 193.141.17.128/28 |
| FW_FTPSERVER | | 193.141.17.131 |
| FW_WWWSERVER | | 193.141.17.133 |
| FW_SSLSERVER | | |
| FW_SSLPORT | | |
| FW_MAILSERVER | | 193.141.17.130 |
| FW_DNSSERVER | | |
| FW_NNTPSERVER | | |
| FW_NEWSFEED | | |
| FW_INT_DEV | eth0 | eth0 |
| FW_LOG_ACCEPT | no | no |
| FW_LOG_DENY | yes | yes |
| FW_ROUTER | | |
| FW_FRIENDS | no | no |
| FW_INOUT | no | no |
| FW_SSH | no | no |
| FW_TRANSPROXY_OUT | | |
| FW_TRANSPROXY_IN | | |
| FW_TCP_LOCKED_PORTS | 1:1023 | 1:1023 |
| FW_UDP_LOCKED_PORTS | 1:1023 | 1:1023 |

Table 2.4: Firewall setting for the example network

Most important is to get the FW_WORLD_DEV and FW_LOCALNETS correctly. These settings specify where the package filter is monitoring the traffic, and which IP addresses should be protected. Also, like mentioned earlier, the _LOCKED_PORTS variables are important, as they determinate which services are blocked by the filter rules.

The firewall script itself is a regular rc-script, containing the commands to set up the filter rules. It's structure is very similar to the masquerading script, discussed in section 2.2.1. The list of parameters contains no surprise:

**start**
: Start the firewall, by setting up the package filter rules and adding user defined chains like discussed in section 2.3.2.

**stop**
: Remove the user defined chains and the rules added to the standard chains.

**reload, restart**
: Same as first `stop`, and then `start`.

**status**
: Prints a list of the firewall rules on standard output.

So much about the firewall. To make the example complete you may want to add masquerading to the inside firewall. As the IP addresses are exactly the same like seen in the masquerading example, you can just use the settings as we had them there. The scripts for masquerading and firewall work fine with each other, they are totally independent and you can activate either one or both at the same time without any restrictions.