

Table of Contents

Part I Document Overview	2
Part II The Kernel	3
1 Drivers	3
2 Modules	3
3 Recompiling the Kernel	3
Part III X - Windows	4
Part IV The Shell	5
1 System Commands	5
2 Searching for a File in Subdirectories	6
3 Setting Environment Variables	7
Part V Installing Applications	8
1 Untar, Unzip, Compile, and Install	8
2 RedHat Packet Manager	9
Part VI File Permissions	10
1 General	10
2 Changing File Permissions	10
Part VII Daemons and Services	12
Part VIII Device Drivers	15
Part IX Configuring Network	16
Part X Iptables	17
Part XI Tips and Tricks	22
Index	0

1 Document Overview



Author: NETIKUS.NET ltd
Date: 18th Nov 2002
Revision: 2

From Windows NT to Linux

Title	From Windows NT to Linux
Summary	This document is intendend to be read by people familiar with Windows NT (or higher) looking into Linux. It covers basic differences and serves as a quick start for Linux.
Software	Any Linux OS
Skill Level	Beginner - Intermediate
Skills Required	- Basic understanding of Windows NT or higher - Basic understanding of networking
Download	http://www.netikus.net/ (guides section)

2 The Kernel

Whatever distribution you choose, whether it's RedHat, Debian, SuSe ... - the kernel is always a Linux kernel. The difference between the distributions is the installation routine, included applications and packages, directory structure, administration interfaces, update procedures, support and so on. So don't get confused and think that different distributions are a different Linux. You might want to look at the kernel revision though when you get a Linux distribution, make sure it's halfway recent.

2.1 Drivers

When coming from Windows we are used to installing drivers for new hardware, however you will not find traditional "drivers" under Linux (except for printers) like you do on Windows. Instead it's the kernel and its modules that support hardware devices.

2.2 Modules

While modules are technically not required they are very useful and can be somewhat compared to drivers. Due to their possibility of loading and unloading them on the fly they are, in my opinion, even more powerful on Linux than are drivers on Windows.

Modules, as their name implies, modularize the kernel and allow you to load only the support you need – without having to recompile the kernel. In the old days, when you needed additional hardware support for a new device, you would have to recompile the entire kernel (even that is quite easy these days) to add support for that particular device. This is no longer necessary! Just compile the module, load it and you should be set without requiring a reboot in most cases.

A quick note, according to many sources modules run with no slowdown at all, so including something in the core kernel because you think that it's faster does not seem to make much sense.

2.3 Recompiling the Kernel

A friend of mine used to (and probably still does) make of Linux – "Need to add a new mouse to Linux? Just recompile the kernel and you're all set!". Recompiling the kernel might have been difficult and tedious at some point, but it's really only typing in a few commands and rebooting.

The kernel is the core of Linux, the workhorse. Memory management, process scheduling (yes, multiple processes need to be scheduled!), file access and such are all things the kernel does.

Now why do we want to recompile the kernel? Most users actually might not want to recompile the kernel; due to the introduction of modules most installations ship a general core kernel (that supports most devices needed to boot) and provide the rest through modules. This means that you save memory since only the necessary modules are loaded and also means that you most likely will need to change or customize the kernel.

Compiling your own kernel can come in pretty handy though sometimes, old machines that have little memory will definitely benefit, exclude support for security sensitive features, use the latest stable version and so on.

So how do you compile the kernel? Navigate to <http://www.tldp.org> and look for the "kernel-howto" there.

3 X - Windows

X-Windows is the GUI for Linux. The interesting part about Linux is that it does not need a GUI to work, pretty much everything works from the command-line and it works well. So usually when I install server systems I don't even install the GUI – it saves quite a bit of memory and I can have multiple session open anyhow by pressing ALT+Fx...

If you install X-Windows you can choose from many different desktop environments. RedHat, for example, comes with GNOME by default, but you can also choose to setup KDE or others. Here is where the distributions come in, because some distributions provide more or less windows managers. Mandrake seems to support quite a few, I personally recommend "fluxbox" as a lightweight window manager.

Please note that specific desktop environments sometimes provide libraries (like DLLs) to programmers, so they don't have to program everything from scratch. This means that applications designed for one desktop environment (eg. GNOME) might not work with KDE and vice versa, unless you install the correct libraries (but this really depends on the applications and what it uses).

So is X-Windows all sunshine? It definitely has its shortcomings, especially if you compare it with the Windows GUI. It uses a lot of RAM (you should definitely consider ~128Mb of RAM if you use X-Windows, assuming you don't have any other memory consuming services running), troubleshooting and configuration (graphic drivers usually take longer since most vendors don't bother developing drivers for Linux) can be rather confusing, and the pool of applications out there is still a little bit frustrating. But – you can simply turn it off and forget X-Windows – try that with your "Windows" machine. By the way, if you want to disable the automatic startup of X-Windows, simply edit the file **/etc/inittab** and change the default run level from 5 to 3 there.

4 The Shell

4.1 System Commands

Let's compare Windows NT/DOS command line commandos their Linux counterpart, most of them are only slightly different:

copy	cp	copy files
rename	mv	rename a file
move	mv	move a file
delete	rm	delete a file
deltree	rm -rf	remove an entire directory
md	mkdir	make a directory
rd	rmdir	remove a directory
dir/w	ls	list directory contents
dir	ls -l	"
dir.	ls -la	"
dir /s	ls -lR	"
hostname	hostname	show the machine's hostname
cd..	cd ..	(yes, you need a space)

YOU HAVE SEVERAL DIFFERENT SHELLS AVAILABLE IN LINUX; YOU DON'T HAVE TO USE BASH. THERE IS ALSO THE C-SHELL (CSH), THE KORN SHELL (KSH) AND MANY MORE. MOST PEOPLE FIND THE BASH SHELL THE MOST USER FRIENDLY SINCE IT HAS THE MOST FEATURES AVAILABLE (ALIASING, HISTORY). YOU ARE FREE TO CONFIGURE YOUR USER ACCOUNT TO USE ANY INSTALLED SHELL BY EDITING THE /etc/passwd FILE. BUT – MAKE SURE THE SHELL SPECIFIED IN THE /etc/passwd FILE EXISTS; OTHERWISE YOU WILL NOT BE ABLE TO LOG IN.

And here some commands you won't find on Windows NT:

du	show directory usage (e.g. "du . -sk")
df	show free disk space (e.g. "df -k")
chgrp	change file group
chown	change file owner
chmod	change file permissions
id	current user
printenv	view all environment variables (they are very important especially when installing and compiling)
who -imH	list logged on users
history	show all previously typed commands (bash)
! !<command nr>	execute a command from the history (bash)
nice	modify priority of a process
ps	show running processes (e.g. "ps -ef")
rpm	redhat package manager (RedHat only ...)
startx	start x-windows
tar	untar and unzip archives
gzip,gunzip	(un)compress files
sort	sort text files
grep	search for a pattern in a file or standard input, Windows NT actually has a "find.exe" which does a similar thing but is less powerful unfortunately
find	find files

I will now take a closer look at some commandos and tricky important combinations (at least more important for beginners):

4.2 Searching for a File in Subdirectories

In Windows NT, if I search for a file in the command prompt, I would simply type:

```
dir /s c:/winnt/myfile*
```

In Linux however, you would use **find** and type:

```
find /somedir -name *myfile*
```

This will look in all subdirectories (starting in **/somedir**) for every file starting with **myfile**. Of course you can specify any directory (or **.** for the current directory) and any wildcard at the end. Make sure you use quotes when you use wildcards, otherwise the shell will expand it before "find" will handle it.

4.3 Setting Environment Variables

To set an environment variable in Windows NT (or DOS), you would just type

```
set PATH=C:\WINNT
```

and everybody would be happy. This will not work in Linux where you would have to type

```
export PATH="/usr/bin"
```

Please note that this only works if you are using the **bash** shell, different shells have different ways of declaring variables. In bash you could also type:

```
PATH="/usr/bin"
MYVAR="whuppy"
.... some code ...
export PATH
export MYVAR
```

which would yield the same result. To clear PATH again, we would type

```
unset PATH
```

To **extend** the PATH variable (which is something you will see quite often) and add the path **/opt**, simply type

```
export PATH=$PATH:/opt
```

Use **printenv** or **set** to view all environment variables. **\$PATH** simply evaluates to the current path string and **:/opt** is just being appended.

5 Installing Applications

I really learned to appreciate window's setup procedures after I had to do the same thing on Linux. It could be so easy, just run setup.exe, hit next a few times and we are done (aah, maybe a reboot first). We are spoiled children, coming from Windows. But what if you don't have a setup.exe? Well, Linux usually does not have a setup.exe, sorry (some applications do however, including Mozilla). But nevertheless, believe it or not, installing software on Linux can be easy – even easier than on Windows! That is the case if you have an rpm file – a RedHat package manager file. But more on that later.

So what if the latest apache, wu-ftp or mysql does not come as an .rpm file but instead as a .tar.gz file? Don't worry - there is a solution.

You will simply have to decompress, compile and install the software. This is actually where a lot of people are having problems – usually due to misconfigured systems, missing libraries and such. Okay, so we have two options:

a) Untar, unzip, compile and install

or

b) Use a package manager (RedHat uses RPM, other distributions (like Debian) use different package formats with similar functionality though)

5.1 Untar, Unzip, Compile, and Install

Let's assume we just downloaded the file `sylpheed-0.8.5.tar.gz` from the Internet and copied it to `/usr/local/src/sylpheed-0.8.5.tar.gz`. Here's what we'll do (after navigating to the download dir with `cd /usr/local/src`):

1. tar xvzf sylpheed-0.8.5.tar.gz

---> this untars, unzips and creates a directory `sylpheed-0.8.5` with all the files contained in the archive. This is what the options mean:

x	extract from archive
v	verbose
z	archive is compressed
f	filename follows

2. cd sylpheed-0.8.5

3. ./configure

---> Now we create the "makefile" which we need in order to compile the applications correctly. If this is successful (probably not – just kidding!) you type

4. make

---> And the source code will be compiled to executables. This will take the longest, depending on your CPU I guess. If this finishes without any error messages you can consider yourself a lucky folk and you quickly type

5. make install

---> Which will actually install (=copy the binaries) the application. Get yourself a treat.

You might come across files with the extension **.bz2**. These files are compressed with a different (and better) compression algorithm and will have to be treated a little differently. Instead of extracting them with one single step we will have to perform two steps:

1a. bunzip sylpheed-0.8.5.tar.bz2

This decompresses the file and creates the file sylpheed-0.8.5.tar

1b. tar xvf sylpheed-0.8.5.tar

As you can see we are omitting the "z" option this time since the archive has already been decompressed.

TAR? GZ? WHAT'S UP WITH THAT – WHY NOT SIMPLY ZIP? GZIP DOES NOT COMBINE MULTIPLE FILES INTO AN ARCHIVE – IT CAN COMPRESS ONE FILE AT A TIME. TAR, ON THE OTHER HAND, CAN COMBINE FILES WITHOUT COMPRESSING THEM. SO WHAT WE DO IS SIMPLY TAR THE REQUESTED FILES INTO A .TAR ARCHIVE AND THEN ZIP THE ENTIRE ARCHIVE WITH GZIP. THE LINUX VERSION OF TAR ALREADY DOES BOTH THINGS IN ONE STEP (THE `-z` ARGUMENT), MOST UNIX OPERATING SYSTEMS HOWEVER DON'T.

5.2 RedHat Packet Manager

The GUI version of the RPM pretty much explains itself; there is not much to add. For the command-line I list the most important commands and switches:

```
rpm -qa          List all installed packages
rpm -e          Remove (erase) an installed package
rpm -qi         View a package description
rpm -i          Install a package
```

Once you install a package, there is nothing more to do except configure the actual application. All the files are already in their proper location. It's actually surprisingly easy.

When calling "rpm -e" you will need to specify the package name without the .rpm extension.

6 File Permissions

6.1 General

File permissions under Linux are very different and yet somewhat similar to the ones on Windows NT at the same time. You might be slightly disappointed when I tell you that I find Linux's file permissions less powerful than Windows NT's or Windows 2000's ACLs. This is because Linux's native file system, extfs2, doesn't support more granular permissions (Solaris 7 and 8 in contrast do support ACLs similar to the ones from NT – but their administration is not exactly – aeh – easy).

As you most likely know you can assign multiple users and groups to an object under Windows NT (and Windows 2000). If you ever read a Microsoft book or attended an official administration course you might recall that, and that is quite important to understand, you should apply only group permissions to objects, rather than assigning individual users. Microsoft also recommends that you put users into global groups, global groups into local groups, and assign local groups (and permissions) to objects. The next figure shows this a little more obvious

```
User1, User2, User3      -> Administration (global)
User4, User7, User8     -> Research (global)          -> Presentation (local)          -> object (read)
User10, User11          -> Management (global)
```

So in this example we have three **global** groups that will all be member of the one **local** group **Presentation**. This local group will have **read** permissions on the object. What is the message behind this? Point is, that with this concept you can get around with file permissions by only assigning permissions to one group. And that's the catch, under LINUX (Unix), you can have three types of permissions assigned to a file (there a few more but not relevant right now):

```
owner      rwx
group      rwx
everyone   rwx
```

Every file has an owner, so the **owner** permissions determine what kind of permissions the owner has. Then you assign a group to the file and the according permissions, just like extensively described above. Finally you can configure everybody else's rights to the file. RWX obviously stands for READ, WRITE and EXECUTE.

6.2 Changing File Permissions

There are a couple of utilities that can be used to change file permissions, but let's start with viewing the file permissions. You can see file permissions when you type **ls -l** (in RedHat you can just type 'll'). Let's take a look at this sample output:

```
drwx----- 2 Wizard Wizard 4096 Nov 14 11:49 Wizard
d--x--x--x 2 root root 4096 Oct 30 07:30 bin
d--x--x--x 3 root root 4096 Nov 30 22:42 etc
-rw-r--r-- 1 root root 0 Dec 1 16:52 home.txt
```

We have basically 7 columns here. The ones relevant for permissions are column **1,3 and 4** (conveniently marked in green ;-)

Column 1 Effective permissions
 Column 3 File owner
 Column 4 File group

So let's look at the fourth line, file **home.txt**. The permissions for this file are **rw-r--r--**. Never mind the first letter, it only indicates what type of object it is (d = directory, - = file and so on). Let's break up the permissions into three blocks:

```
-rw-r--r--    1 root    root                0 Dec  1 16:52 home.txt
```

Owner	Group	Everyone
root	root	
rw-	r--	r--

Hmm, I already revealed it. The first part are the permissions for the **owner**, **root** in this case (column 3), **root** has the right to **read** and **write** to the file. The group assigned to this file is also **root**, but only has **read** permissions to the file. Everybody else on the system also has **read** permissions on the file – obviously you should be careful with this permission. Now how do we change those permissions in traditional Unix style? Let's look at this table:

Letter	Permission	Value
r	read	4
w	write	2
x	execute	1

Highly interesting, isn't it? If you assign permissions, you simply add the values of the rights you want to assign to the particular user or group. In the above example, the owner has the right **6**, since r = 4 and w = 2 and the sum is 6. Group and everyone both have ... **4** !!! Wow, not as hard as it looks like. So how do we change them? We use the program **chmod** (change mode). To change the permissions of the file **home.txt** to "rw- --- ---" to allow only the root user to access the file, we type

```
chmod 600 home.txt
```

I think that's enough explanation for changing the permissions. To give everybody all permissions, we would therefore type

```
chmod 777 home.txt
```

We might want to change the owner of a file, which only **root** or the **current owner** can do. You use **chown** (change owner)

```
chown Wizard home.txt
```

to change the owner to the user Wizard. If you would change permissions of a directory and want every file and subdirectory to be affected of that change, you would type

```
chown -R Wizard etc
```

This would change the owner to Wizard in every file and subdirectory of **etc**. Good, one more thing to learn, that is how to change the group. It works exactly like **chown**, except that you use the command **chgrp** (change group – did you guess it?). One example

```
chgrp Wizards home.txt
```

Here you assign the group **Wizards** to the file **home.txt**. And that's it about permissions.

7 Daemons and Services

Windows NT uses services to manage applications that must be active all the time, even when a user is not logged in. Services can also be controlled by starting or stopping them (pause and resume are also supported rarely). Now Linux doesn't have "services", but instead has daemons, which is essentially the same. Daemons (or services) like apache or samba fall into this category. They are specially crafted applications that are able to run in the background, staying active at all times. The bad news about Linux's daemons is that they are more complicated to configure, the good news is that you can configure almost everything you want – and sometimes you even have to.

Daemons are started with shell scripts which almost always come with the respective daemons (samba, apache , ...). But don't worry, you will not have to understand shell scripts to configure daemons – it will be easier than that.

The scripts that start the daemons are located in the directory **/etc/rc.d/init.d**. So if you want to know if a particular service can be controlled in that manner, this is the place to look for it. It is actually surprisingly easy to write your own startup script, assuming the daemon's requested environment isn't too complicated ... Now there is one thing that all of those scripts **must** have in common: They must support at least the command line arguments "start" and "stop". Sometimes they also support "restart" or "status", but "start" and "stop" are essential. Why? This is the way we (the penguin's servants) and Linux control the daemons.

But how does Linux know which services to start/stop at what time? The run levels and the **/etc/rc.d** directories are the answer. After the master process **init** has been spawned, Linux is always in a run level. If you don't use X then this would be run level 3, if you use X then you're running in runlevel 5. You don't tell Linux just to start a service, you tell it at **what run level to start** a service, and at what run level to **stop** a service. And on top of that you can tell it in what sequence to start (like the service dependencies on Windows) – e.g. you could start apache before samba and vice versa – just as an example. Now every used run level (I say used because some run levels are not really being used, such as run level 4) has its **/etc/rc.d** directory. For example run level three has the directory **/etc/rc.d/rc3.d**, run level 5 has **/etc/rc.d/rc5.d**.

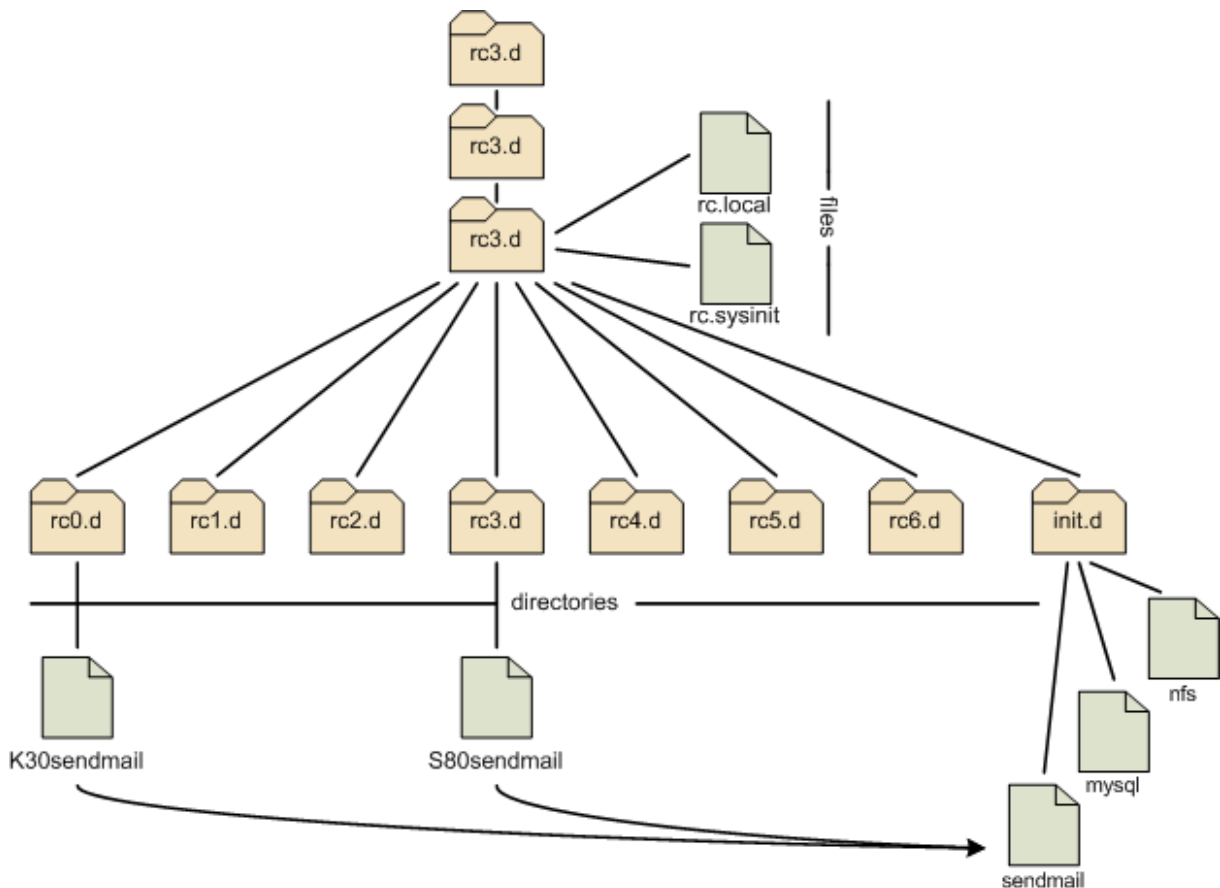
Let's assume that we have a script that starts up our beloved apache web server in the **/etc/rc.d/init.d** directory, called **httpd**. Just the simple presence of that script in the **init.d** directory won't make apache do anything upon boot. This would be like a service in Windows that is set to manual. You can start it manually, but upon boot it won't do anything. Let's consider this example: We want apache to start at run level 3 (and not at run level 5) – and shut down at run level 0 (shutdown) and run level 6 (reboot).

To accomplish this goal we have to use symbolic links. A symbolic link is like a shortcut under Windows – it's a file that points to another file. When you edit the symbolic link, you really edit the file it points to, but when you delete the symbolic link, you really only delete the symbolic link – fortunately. So now we create a symbolic link in the directory **/etc/rc.d/rc3.d** with the name S80httpd. Excuse me? S80httpd? Why not F234apache? The first letter of the symbolic link tells Linux how to control the service. An uppercase S (it must be UPERCASE) means to start the service (yes, and all that Linux really does is call the script with the previously explained "start" option, this is why it is essential to support "start" and "stop"), and an uppercase K means to stop (or kill) the service. Again, Linux simply calls the script with the "stop" argument. The number after S or K, 80 in this example, determines the start order. Linux begins executing the scripts with 0 until there is no more. You can use an existing number if you want to, I assume the alphabetical order will then determine which one to start first. The name after the number is really up to you, but to be halfway consistent you should really use the name of the script in the **init.d** directory. So here are the commands I would type to make this miracle happen:

```

cd /etc/rc.d/rc3.d
ln -s /etc/rc.d/init.d/httpd s80httpd
cd /etc/rc.d/rc0.d
ln -s /etc/rc.d/init.d/httpd k20httpd
cd /etc/rc.d/rc6.d
ln -s /etc/rc.d/init.d/httpd k20httpd
    
```

Of course I did more than would I had described above – the first two lines are enough to configure the daemon to start. The next four lines add the necessary symbolic links to shutdown the daemon properly when Linux is shutdown or restarted. I used 20 as the sequence number since httpd should be stopped in the beginning rather than in the end. Think about it, httpd depends on other network services that shouldn't be stopped before httpd! To round this chapter up I included a picture to make you visualize things a little easier:



Here you see the 7 directories for the different run levels (rc0.d ... rc6.d) and the symbolic links contained in them, K30sendmail and S80sendmail showed as examples. The **init.d** directory of course contains the original control files, such as **nfs**, **mysql** or **sendmail** serving as some examples here.

Please note the files **rc.local** and **rc.sysinit** here. They both serve an important purpose as well and you should be familiar with them. The **rc.sysinit** file is an integral part of Redhat since it sets up several components of Linux, networking for example. I recommend looking through the file in an editor, you will see a lot of interesting things.

The **rc.local** file is for Linux what autoexec.bat is (was ☺) for **Ms-Dos**. This shell script is executed

after all the init scripts and you can place your own initialization in there. You might want to change your login banner in there, just look for **/etc/issue**.

8 Device Drivers

Installing and administering drivers under Linux is definitely a big obstacle for most beginners, especially if you are used to Windows's easy of installation. Installing network or graphic adapter drivers under Windows is really just a clickedi-click, very rarely do you have to configure drivers with the registry. Now forget all this when you deal with Linux, here we have a different concept.

Originally Unix had all its drivers embedded (included) in the Kernel. This can of course be quite inefficient and inflexible; you would need a new kernel every time you install a hardware device whose driver is not included in the kernel. This is why most Unix systems now support modules, drivers that are loaded on demand. So the driver is either contained in the **kernel** or as a **module**.

Let's look at a very simple example, a 3com 3c509 network card. You have two choices to support this NIC in your Linux installation: Either its driver is **included in your kernel** (usually not the case by default) or it is **available as a module**. What is the difference and what is better? If we look at the difference we will see why one is better than the other for a particular situation.

Obviously you need to compile the kernel add a kernel level driver. But that also means that this driver is pretty much always loaded, no matter if our NIC is present or not. If you exchange the NIC you will have to recompile the kernel again, to remove the old driver and add support for the new one. I personally use kernel drivers since I don't change hardware very often and since new kernels come out all the time anyway – a change of hardware at least gives me an opportunity to compile a new kernel.

Modules. First you don't need to compile the complete kernel to add module support for the NIC. If the module is not already present (check ...) you will only have to compile the module, which takes significantly less time than compiling the kernel – a reboot is not required either. The good thing about modules is that they are only loaded when they are really required. If the system doesn't find a 3c509 then it won't load the module. The kernel stays small and the module is untouched.

So how do you compile a module? It's easier than you might think. When you configure your kernel with either "make menuconfig" or "make xconfig" then you have the option (for most drivers) to include it in the kernel or compile it as a module (M). Then you simply run

- `make dep`
- `make clean`
- `make modules`
- `make modules_install`

and the module(s) should be in their respective location.

9 Configuring Network

Configuring your ip addresses (I will only mention Ipv4 here, sorry) can be quite a pain also, especially if you don't use X windows. But, once you understand how Linux determines the configuration of your network cards, you will find it quite easy to configure your network card(s).

Please note that this chapter uses Redhat 7.x as an example, other distributions might store their network configuration files at different places.

Let's get back to our example, the 3c509 NIC. Let's assume that the driver has been loaded, either in the kernel or with a module. You can verify this by typing **dmesg** and looking for lines that mention 3com (or whatever your NIC's vendor is), eth0 and similar network related things.

How does our Redhat installation know that we want an ip address assigned to this card and which one? During boot up it looks for a specific file pattern in the **/etc/sysconfig/network-scripts/** directory. It looks for files starting with **ifcfg-ethX**, where **X** represents the interface number, usually 0 if you only have one NIC installed. So in our case we would need to create or edit the file **/etc/sysconfig/network-scripts/ifcfg-eth0** to configure our ip settings. This is how the file should look like this for setting a static ip address:

```
DEVICE=eth0
IPADDR=184.12.10.100
NETMASK=255.255.255.0
NETWORK=184.12.10.0
BROADCAST=184.12.10.255
ONBOOT=yes
```

Mighty easy, isn't it? Now let's see how we can tell Redhat to use a name server too. For this we need to look at two files. First we look at the file **/etc/nsswitch.conf**. This file tells the OS what services are used for a particular resolution type. One resolution type, e.g., is **hosts**; another one (which we won't discuss further) is **passwd**.

Linux gives us several options, and more important even an order, to resolve an ip address. If you enter **grep ^hosts /etc/nsswitch.conf** (the **^** character matches only if **hosts** appears at the beginning of the line) you will see something similar to this:

```
hosts: files nisplus nis dns
```

This shows me that when I resolve a hostname to an ip address, Linux will first use the **files** service (**/etc/hosts**) and then the **nisplus** or the **nis** service, and if it still can't find that hostname it will query the configured dns servers. You configure your dns servers in the file **/etc/resolv.conf**. A typical **resolv.conf** file looks like this:

```
nameserver xx.xx.xx.xx1
nameserver xx.xx.xx.xx2
search netikus.net
```

The **nameserver** directive obviously tells your system which name servers to contact for the hostname resolution. You can list multiple nameserver by adding multiple **nameserver** directives. Now the **search** directive is not absolutely required and kicks in when you don't specify a fqdn (fully qualified domain name, like www.linuxdocs.org), but instead only a hostname. In that case the OS simply appends the domainname that you specified to the hostname. So if you where to type **ping www**, the OS would query the configured dns servers for www.netikus.net. This is probably most useful in an intranet.

10 Iptables

I do not consider the Internet a lovely play ground with friendly people anymore, something it might have been 10 years ago. Instead it's more like an uncontrolled and compressed combination of good and evil. You have to get the good stuff yourself, but the evil stuff comes into your machine by itself. I look my door all the time, but when I imagine that every human being could arrive at my doorstep from any place in the world within seconds, that does sound slightly scary. But that's the way it is with the Internet, everybody out there can come in anytime without knocking, if you're not prepared. High bandwidth connections via cable or DSL are very convenient but naturally require security. I don't like desktop security applications and installing them on every machine on my network does not make sense anyhow. IP Masquerading is another nice thing when your stingy ISP only gives you one ip address for an already expensive DSL connection.

So what is the solution? Certainly commercial firewall solutions would be the best choice if you had a lot of money, but do you have a \$5000 in your yearly firewall budget? I don't, and that's why I take an old Pentium machine with 32MB RAM and a 1Gb HD and make it the masquerading firewall.

The Iptables can be a little bit confusing to understand and I certainly recommend some thorough understanding of TCP/IP before you read on. Check my website for networking books that I recommend. It took me quite a while until I had a grasp on the Iptables and I will try to pass that on to you. Don't let configuration files scare you, at first glance they look very confusing (like most things on Linux if you're not used to them) but turn out to be quite tame once you understand them. I will use a simple but hopefully helpful example to explain **iptables**. Some basics now:

Our beloved **iptables** use several so-called chains to configure its behavior: INPUT, OUTPUT, FORWARD and for NAT: PREROUTING and POSTROUTING.



The **input** chain will handle a packet that is destined for our host above, a packet that leaves via an interface will be handled by the **output** chain. Very simple, but what is the **forward** chain good for then? If our box is routing a packet, essentially meaning that it is not really being processed by any service at our then the **forward** chain comes into play. So forwarding really only makes sense if you use your Linux box as some kind of **router**. Please note that a packet only traverses one of the mentioned chains, it's incoming, outgoing or forwarded.

We will then apply rules to those chains to customize and control the behavior of our Linux Bastille.

Please note that, without mentioning it again in the examples later, I assume that all traffic has been disallowed by using `-F chain` and `-P chain DROP`, example:

```
iptables -F INPUT           # FLUSH THE INPUT CHAIN (remove all rules)
iptables -P INPUT DROP     # SET THE DEFAULT POLICY TO DROP
iptables -F OUTPUT         # FLUSH THE OUTPUT CHAIN
iptables -P OUTPUT DROP   # SET THE DEFAULT POLICY TO DROP
iptables -F FORWARD       # ...
iptables -P FORWARD DROP  # ...
```

The above lines need some explanation. As already stated we apply rules to the existing INPUT, OUTPUT or FORWARD chains. Whenever we configure our box I want to make sure that the chains are **empty** since rules will be followed step-by-step.

The **-F** switch accomplishes this by removing all rules of a specific chain, lines 1,3 and 5 are explained.

The **-P** switch on the other hand sets the default policy for a chain. Right after we flush the chain it is empty, packets will not match a single rule since there aren't any rules. A default policy of a chain can either be ACCEPT or DROP and I prefer DROP. This means that every packet that has arrived at the end of a chain will be matched against the default policy, DROP in the example above. It is usually more secure to deny everything that has not been explicitly allowed.

Additionally we want to set up environment variables to make our script(s) easier to maintain:

```
EXT_IFACE="eth1"
LAN_IP="172.24.10.0/24"
```

Example 1:

Let's imagine a simple example, we set up a host and only want it to accept http connections since this is the only service that should be accessed. Of course we already deactivated all services, but additionally we want to allow only **TCP** packets that were sent to the local port **80**. Clearly this will be configured on the input chain since we are talking about an incoming packet. Good, now since our remote http client would probably appreciate some kind of feedback, we also need to allow packets leaving the interface, via the **output** chain. Of course we don't want any packet to leave the **output** chain, since an intruder could use that to his advantage and send data from our host, no matter if that is a Trojan horse or a simple ftp client. The **iptables** have a very nice feature as they now support the statefulness of packets, by using `-m state -state ESTABLISHED, RELATED`. Sounds good, it sure is! If we could actually talk to **iptables** (maybe in the 4.14 kernel?) we would say:

- Allow all packets that are **TCP** and are sent to port **80** on this interface.
- Allow all packets that refer to a previous packet (**-ESTABLISHED**) or to an existing connection (like ICMP replies, FTP data sessions ..., **-RELATED**) to leave the interface.

All other packets would be denied since they were not specifically allowed. This is the "code":

```
iptables -A INPUT -p TCP --dport 80 -i eth0 -j ACCEPT
iptables -A OUTPUT -p TCP -o eth0 -m state --state ESTABLISHED -j ACCEPT
```

Line 1:

-A INPUT	means that we want to configure the input chain
-p TCP	refers to only TCP packets (and not UDP packets)
--dport 80	matches only packets that were sent to port 80
-i eth0	applies only to the eth0 interface
-j ACCEPT	jumps to the ACCEPT target, hence lets the packet pass

Line 2:

-A OUTPUT	packets leaving our interface
-p TCP	TCP only
-o eth0	packets leaving interface eth0
-m state	specify a state
--state ESTABLISHED	only related packets are allowed, not new connections
-j ACCEPT	let the packet go trough

Example 2:

This might be a little bit more complicated than the previous example. We just set up our new DSL line and are mighty happy, except for one thing: We are only supposed to use one computer to connect to the internet! Dang, how boring! Of course we have several computers around that all want to connect

to the Internet which at this point doesn't work. I always feel a little badly to fool my ISP who, in endless generosity, has granted me the privilege to connect to the Internet in such high speed. Yet I really only use one computer *at the time* to connect to the Internet, but from different machines.

In plaintext, we need a firewall that supports IP masquerading. No problem for the **iptables** since they support **connection tracking**. Now what is exactly that we want?

- All packets originating from our internal LAN should be sent to the correct host on the Internet; all packets coming from the Internet that relate to a connection that was initiated previously from the internal LAN should be allowed to enter our network.
- Additionally, the **ip headers** need to be modified by **iptables** so that our firewall appears to be the requesting and receiving host, instead of our machines in the LAN that probably have ip addresses not even valid on the Internet (10.0.0., 172.24.0.0, 192.168.0.0).

That's what we call masquerading. **iptables** accomplishes this by keeping track which host (from the LAN) requested which connection, and by subsequently, upon arrival of the response (from the Internet), sends the packet back to the internal host. Puh, cool stuff.

This is the first example where we need to look at another chain, the **FORWARD** and **POSTROUTING** chain. I found the **POSTROUTING** (and **PREROUTING**) chain a little confusing, so read the following if you find it confusing too.

Let's forget the **iptables** stuff for a while and think of our Linux box as a plain router. We have our little LAN, and we have the endless space of the Internet. To send packets from our internal LAN we need to route the packets through our Linux box. So if we request <http://www.netikus.net> then our machine from the LAN will send a packet to the Linux box (assuming that the Linux box is configured as the default router for that host), destined for the ip address of www.netikus.net. Let's also assume that the ip address of our host in the LAN is 172.24.10.50. Now without NATing, our Linux box would forward this packet to the next router on the path to www.netikus.net. The next router however would most likely discard the packet since 172.24.10.50 is not really a valid address on the Internet since it's part of the address space reserved for internal usage. So what did our router do wrong? It correctly accepted the packet and correctly sent it to its own default router, but that was simply not enough. For all this to work the source ip address of the packet we sent needs to be modified to something valid, in fact to the ip address of the external Linux host's NIC. This is why we need to use the **POSTROUTING** chain, the chain that modifies our source packet **after** the OS has made its routing decision. If we were to modify the packet **before** the routing decision would be made, then no routing decision would be made at all since the packet would appear to come from itself (remember, we change the source address of **our** packet to the address of the firewall's external interface) – not a good idea. So we need to make Linux make its routing decision and **then, after** routing (**POSTROUTING**), change the source ip address. So we write:

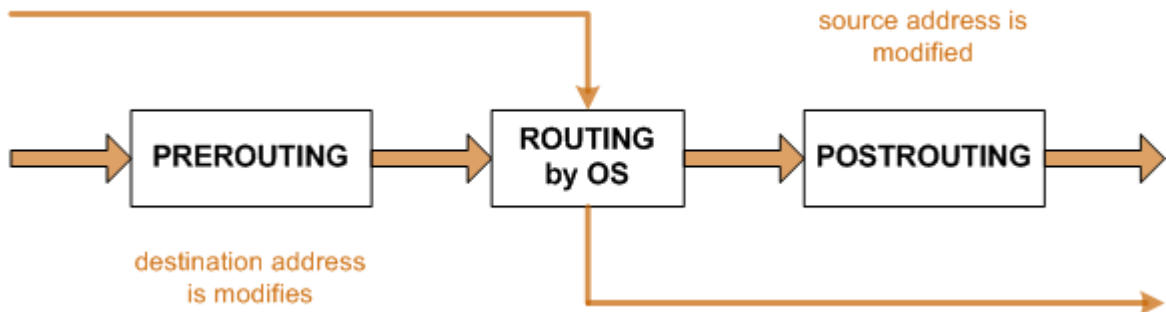
```
iptables -t nat -A POSTROUTING -o $EXT_IFACE -s $LAN_IP -j MASQUERADE
```

The "-t nat" specifies that we want to use the **nat** table, which enables us to use the **POSTROUTING** chain. If we don't specify a table with **-t**, then the default **filter** table is being used (which contains the already described **INPUT**, **OUTPUT** and **FORWARD** default chains). Then we want to masquerade packets as they leave the \$EXT_IFACE interface (this should be set to whatever your external interface is, like **eth1** or **ppp0**) and come from an address from our internal network. We haven't seen the **MASQUERADE** target before, that's because it's only available in the **nat** table. You should only use **MASQUERADE** if you are using a dial up connection or a connection that gets the ip address assigned dynamically. This is because all stored states of packets will be forgotten (erased) once the interface goes down. If you have a permanent ip address then use the **SNAT** target with the **-to-source ipaddress** option instead where *ipaddress* would be the ip address of your external interface. Good, with this one rule Linux modifies packets that:

- Originate from our internal networks ip address space

- Leave the firewall on its external interface

This figure might explain the **POSTROUTING** (and **PREROUTING**) chain better:



This may sound like this is all we want, but it's not really. Our firewall at this point (since previously only configured to drop everything) will not accept any packet. So what are we missing? Packets coming in on the internal interface need to be accepted, provided they only pass through, right? We need to apply another rule that utilizes the **FORWARD** chain, since packets are being forwarded. **iptables** must simply look at the destination ip address of the packet and then decide if it should use the **INPUT** or **FORWARD** chain. In **iptables** syntax this should look something like this:

```
iptables -A FORWARD -i $INT_IFACE -o $EXT_IFACE -s $LAN_IP -p TCP -j ACCEPT
```

Remember, **-i** is the incoming interface and **-o** is the outgoing interface. We already used a simpler rule above. Good, now packets are accept from the internal network and correctly masqueraded. Packets sent by an internal host will now arrive at the destination host on the Internet, which would think that our firewall had actually sent the packet. Then the Internet host would reply to the packet by sending it to the firewall. And yes, the firewall would drop the packet since it doesn't allow incoming packets on the external interface. So we write another rule, using the **-m state** option again.

```
iptables -A FORWARD -i $EXT_IFACE -o $INT_IFACE -p TCP -m state --state ESTABLISHED,RELATED -j ACCEPT
```

This is very similar to the first example, except that I added **RELATED** to the state flags. That's because I might end up enabling ICMP and ftp packets as well, that won't work without the **RELATED** flag. You might wonder why I didn't use **INPUT** here, since the packets coming from the Internet are certainly addressed to our firewall. That's a good question and I haven't found documentation about that yet. The only explanation that makes sense for me is that **iptables** recognize the incoming packet(s) to belong to the **nat** table and change the destination address immediately in some kind of **prerouting** chain, before the OS does it's routing decision.

Please note that you still need to allow UDP packets to port 53 to allow your internal hosts to resolve host names, I will leave it up to you to figure that rule out.

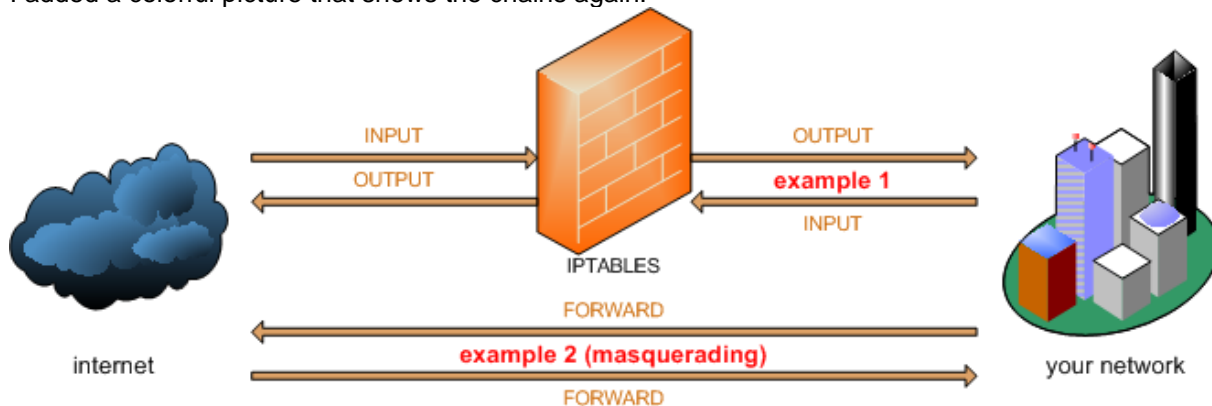
There is a lot more about the **iptables** and I suggest you visit netfilter.samba.org; they have a lot of good documents there. When you try to set up iptables rules, first picture where exactly the packets go, like:

- Where are the packets originating from
- Where are the packets going to
- What protocol will be used

Once you figured that out, go on by determining the chains that will be used:

- Will packets be sent to the firewall or will they be forwarded?
- If the packets will be modified (NAT), will they be changed **before** or **after** routing

I added a colorful picture that shows the chains again:



This chapter got a lot longer than I had anticipated but should give you a good understanding of the iptables.

11 Tips and Tricks

Here are some brief tips not worth dedicating chapters to:

*) To start an app from a terminal window in X-WINDOWS and being able to continue to type in the term window, type "<app-name> &", such as

```
netscape &
```

- Note that the launched app will terminate if you close the terminal window. If you want the application to stay active no matter what, you would have to type:

```
nohup netscape &
```

which detaches the application from the console.

*) To CD to your home directory, just type

```
cd ~
```

*) To change a keyboard language in X-Windows you can type "setxkbmap us" for an example. Other information can be found in

```
/usr/sbin/kbdconfig
```

*) To put a job into foreground or background type

```
fg job#           for foreground
bg job#           for background
```

*) You can run multiple commands in the command line by separating them with a semi-colon, such as

```
ls -la; more /etc/password
```

*) More ls switches you might like:

```
-a show all files (including hidden files)
-l long format
-h file size in kb
-R include subdirectories
-t sort by time
```

*) DNS. If you want to query all entries for a given domain you can enter

```
host -l domainname
```

*) If you want to display only certain columns of an output, let's say only the usernames of the /etc/passwd file, you simply execute cut with the number of column you need

```
cat /etc/passwd | cut -d: -f1,3
```

Here the file /etc/passwd is being passed to the cut command which separates the input by the : character (d = delimiter) and shows only the first and third column. There are more options, type man cut to see them

*) You can scroll in the console if you hit <SHIFT> together with <PGUP> or <PGDOWN> respectively

*) One can set aliases in UNIX to avoid typing a long command all over all the time. Typing

```
alias ls='ls -al'
```

always produces a nice ls output with all details. After setting the alias you only have to type ls instead of ls -al. However, if you would like to use the original command again, after having set the alias, you can precede the alias with a backslash and the command will execute without the alias. In the above case you would type `\ls` and the alias would be ignored.