Security Quick−Start HOWTO for Red Hat Linux

Hal Burgiss

hal@foobox.net

v. 1.1, 2002−02−06

Revision History

Revision v. 1.1	2002−02−06	Revised by: hb
A few fixes, some additions and many touch−ups from the original.		
Revision v. 1.0	2001−11−07	Revised by: hb
Initial Release.		

This document is a an overview of the basic steps required to secure a Linux installation from intrusion. It is intended to be an introduction. This is a Red Hat specific version of this document.

Table of Contents

Table of Contents

1. Introduction

1.1. Why me?

Who should be reading this document and why should the average Linux user care about security? Those new to Linux, or unfamiliar with the inherent security issues of connecting a Linux system to large networks like Internet should be reading. "Security" is a broad subject with many facets, and is covered in much more depth in other documents, books, and on various sites on the Web. This document is intended to be an introduction to the most basic concepts as they relate to Red Hat Linux, and as a starting point only.

```
Iptables Weekly Log Summary from Jul 15 04:24:13 to Jul 22 04:06:00
Blocked Connection Attempts:

Rejected tcp packets by destination port

port                     count
111                      19
53                       12
21                       9
515                      9
27374                    8
443                      6
1080                     2
1138                     1


Rejected udp packets by destination port

port                     count
137                      34
22                       1
```

The above is real, live data from a one week period for my home LAN. Much of the above would seem to be specifically targeted at Linux systems. Many of the targeted "destination" ports are used by well known Linux and Unix services, and all may be installed, and possibly even running, on your system.

The focus here will be on threats that are shared by all Linux users, whether a dual boot home user, or large commercial site. And we will take a few, relatively quick and easy steps that will make a typical home Desktop system or small office system running Red Hat Linux reasonably safe from the majority of outside threats. For those responsible for Linux systems in a larger or more complex environment, you'd be well advised to read this, and then follow up with additional reading suitable to your particular situation. Actually, this is probably good advice for everybody.

We will assume the reader knows little about Linux, networking, TCP/IP, and the finer points of running a server Operating System like Linux. We will also assume, for the sake of this document, that all local users are "trusted" users, and won't address physical or local network security issues in any detail. Again, if this is not the case, further reading is strongly recommended.

The principles that will guide us in our quest are:

* There is no magic bullet. There is no one single thing we can do to make us secure. It is not that simple.
* Security is a process that requires maintenance, not an objective to be reached.
* There is no 100% safe program, package or distribution. Just varying degrees of insecurity.

The steps we will be taking to get there are:

* Step 1: Turn off, and perhaps uninstall, any and all unnecessary services.
* Step 2: Make sure that any services that are installed are updated and patched to the current, safe version −− *and then stay that way*. Every server application has potential exploits. Some have just not been found yet.
* Step 3: Limit connections to us from outside sources by implementing a firewall and/or other restrictive policies. The goal is to allow only the minimum traffic necessary for whatever our individual situation may be.
* Awareness. Know your system, and how to properly maintain and secure it. New vulnerabilities are found, and exploited, all the time. Today's secure system may have tomorrow's as yet unfound weaknesses.

If you don't have time to read everything, concentrate on Steps 1, 2, and 3. This is where the meat of the subject matter is. The Appendix has a lot of supporting information, which may be helpful, but may not be necessary for all readers.

1.2. Notes

This is a Red Hat specific version of this document. The included examples are compatible with Red Hat 7.0 and later. Actually, most examples should work with earlier versions of Red Hat as well. Also, this document should be applicable to other distributions that are Red Hat derivatives, such as Mandrake, Conectiva, etc.

Overwhelmingly, the content of this document is not peculiar to Red Hat. The same rules and methodologies apply to other Linuxes. And indeed, to other Operating Systems as well. But each may have their own way of doing things −− the file names and locations may differ, as may the system utilities that we rely on. It is these differences that make this document a "Red Hat" version.

1.3. Copyright

Security−Quickstart HOWTO for Red Hat Linux

Copyright © 2001 Hal Burgiss.

This document is free; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You can get a copy of the GNU GPL at at http://www.gnu.org/copyleft/gpl.html.

1.4. Credits

Many thanks to those who helped with the production of this document.

- Bill Staehle, who has done a little bit of everything: ideas, editing, encouragement, and suggestions, many of which have been incorporated. Bill helped greatly with the content of this document.
- Others who have contributed in one way or another: Dave Wreski, Ian Jones, and Jacco de Leeuw.
- Various posters on comp.os.linux.security, a great place to learn about Linux and security.
- The Netfilter Development team for their work on iptables and connection tracking, state of the art tools with which to protect our systems.

1.5. Disclaimer

The author accepts no liability for the contents of this document. Use the concepts, examples and other content at your own risk. As this is a new document, there may be errors and inaccuracies. Hopefully these are few and far between. Corrections and suggestions are welcomed.

This document is intended to give the new user a starting point for securing their system while it is connected to the Internet. Please understand that there is no intention whatsoever of claiming that the contents of this document will necessarily result in an ultimately secure and worry−free computing environment. Security is a complex topic. This document just addresses some of the most basic issues that inexperienced users should be aware of.

The reader is encouraged to read other security related documentation and articles. And to stay abreast of security issues as they evolve. Security is not an objective, but an ongoing process.

1.6. New Versions and Changelog

The current official version can always be found at http://www.linuxdoc.org/HOWTO/Security−Quickstart−Redhat−HOWTO/. Pre−release versions can be found at http://feenix.burgiss.net/ldp/quickstart−rh/.

Other formats, including PDF, PS, single page HTML, may be found at the Linux Documentation HOWTO index page: http://linuxdoc.org/docs.html#howto.

Changelog:

Version 1.1: Various corrections, amplifications and numerous mostly small additions. Too many to list. Oh yea, learn to spell Red Hat correctly ;−)

Version 1.0: This is the initial release of this document. Comments welcomed.

1.7. Feedback

Any and all comments on this document are most welcomed. Please make sure you have the most current version before submitting corrections or suggestions! These can be sent to `<hal@foobox.net>`.

2. Foreword

Before getting into specifics, let's try to briefly answer some questions about why we need to be concerned about security in the first place.

It is easy to see why an e–commerce site, an on–line bank, or a government agency with sensitive documents would be concerned about security. But what about the average user? Why should even a Linux home Desktop user worry about security?

Anyone connected to the Internet is a target, plain and simple. It makes little difference whether you have a part–time dialup connection, or a full–time connection, though full–time connections make for bigger targets. Larger sites make for bigger targets too, but this does not let small users off the hook since the "small user" may be less skilled and thus an easier victim. Red Hat, and Red Hat based distributions, tend to make for bigger targets as well, since the installed user base is so large.

There are those out there that are scanning just for easy victims all the time. If you start logging unwanted connection attempts, you will see this soon enough. There is little doubt that many of these attempts are maliciously motivated and the attacker, in some cases, is looking for Linux boxes to crack. Does someone on the other side of the globe really want to borrow my printer?

What do they want? Often, they just may want your computer, your IP address, and your bandwidth. Then they use you to either attack others, or possibly commit crimes or mischief and are hiding their true identity behind you. This is an all too common scenario. Commercial and high–profile sites are targeted more directly and have bigger worries, but we all face this type of common threat.

With a few reasonable precautions, Red Hat Linux can be very secure, and with all the available tools, makes for a fantastically fun and powerful Internet connection or server. Most successful break–ins are the result of ignorance or carelessness.

The bottom line is:

- Do you want control of your own system or not?
- Do you want to unwittingly participate in criminal activity?
- Do you want to be used by someone else?
- Do you want to risk losing your Internet connection?
- Do you want to have to go through the time consuming steps of reclaiming your system?
- Do you want to chance the loss of data on your system?

These are all real possibilities, unless we take the appropriate precautions.

If you are reading this because you have already been broken into, or suspect that you have, you cannot trust any of your system utilities to provide reliable information. And the suggestions made in the next several sections will not help you recover your system. Please jump straight to the Have I been Hacked? section, and read that first.

2.1. The Optimum Configuration

Ideally, we would want one computer as a dedicated firewall and router. This would be a bare bones installation, with *no* servers running, and only the required services and components installed. The rest of our systems would connect via this dedicated router/firewall system. If we wanted publicly accessible servers (web, mail, etc), these would be in a "DMZ" (De−militarized Zone). The router/firewall allows connections from outside to whatever services are running in the DMZ by "forwarding" these requests, but it is segregated from the rest of the internal network (aka LAN) otherwise. This leaves the rest of the internal network in fairly secure isolation, and relative safety. The "danger zone" is confined to the DMZ.

But not everyone has the hardware to dedicate to this kind of installation. This would require a minimum of two computers. Or three, if you would be running any publicly available servers (not a good idea initially). Or maybe you are just new to Linux, and don't know your way around well enough yet. So if we can't do the ideal installation, we will do the next best thing.

2.2. Before We Start

Before we get to the actual configuration sections, a couple of notes.

With Linux, there is always more than one way to perform any task. For the purposes of our discussion, we will have to use as generic set of tools as we can. Unfortunately, GUI tools don't lend themselves to this type of documentation. So we will be using text based, command line tools for the most part. Red Hat does provide various GUI utilities, feel free to substitute those in appropriate places.

The next several sections have been written such that you can perform the recommended procedures as you read along. This is the "Quick Start" in the document title!

To get ready, what you will need for the configuration sections below:

- A text editor. There are many available. If you use a file manager application like gmc or nautilus, it probably has a built in editor. This will be fine. **pico** and **mcedit** are two relatively easy to use editors if you don't already have a favorite. There is a quick guide to Text editors in the Appendix that might help you get started. It is always a good idea to make a back up copy, before editing system configuration files.
- For non−GUI editors and some of the commands, you will also need a terminal window opened. **xterm, rxvt,** and **gnome−terminal** all will work, as well as others.

We'll be using a hypothetical system here for examples with the hostname "bigcat". Bigcat is a Linux desktop with a fresh install of the latest/greatest Red Hat running. Bigcat has a full−time, direct Internet connection. Even if your installation is not so "fresh", don't be deterred. Better late than never.

---

3. Step 1: Which services do we really need?

In this section we will see which services are running on our freshly installed system, decide which we really need, and do away with the rest. If you are not familiar with how servers and TCP connections work, you may want to read the section on servers and ports in the Appendix first. If not familiar with the **netstat** utility, you may want to read a quick overview of it beforehand. There is also a section in the Appendix on ports, and corresponding services. You may want to look that over too.

Our goal is to turn off as many services as possible. If we can turn them all off, or at least off to outside connections, so much the better. Some rules of thumb we will use to guide us:

- It is perfectly possible to have a fully functional Internet connection with no servers running that are accessible to outside connections. Not only possible, but desirable in many cases. The principle here is that you will never be successfully broken into via a port that is not opened because no server is listening on it. No server == no port open == not vulnerable. At least to outside connections.
- If you don't recognize a particular service, chances are good you don't really need it. We will assume that and so we'll turn it off. This may sound dangerous, but is a good rule of thumb to go by.
- Some services are just not intended to be run over the Internet −− even if you decide it is something you really do need. We'll flag these as dangerous, and address these in later sections, should you decide you do really need them, and there is no good alternative.

3.1. System Audit

So what is really running on our system anyway? Let's not take anything for granted about what "should" be running, or what we "think" is running.

Which services get installed and started will vary greatly depending on which version of Red Hat, and which installation options were chosen. Earlier releases were very much prone to start many services and then let the user figure out which ones were needed, and which ones weren't. Recent versions are much more cautious. But this makes providing a ready made list of likely services impossible. Not to worry, as we shouldn't trust what is *supposed* to be running anyway. What we need to do is list for ourselves all running services.

Now open an **xterm**, and **su** to root. You'll need to widen the window wide so the lines do not wrap. Use this command: `netstat -tap |grep LISTEN`. This will give us a list of all currently running servers as indicated by the keyword `LISTEN`, along with the "PID" and "Program Name" that started each particular service.

```
# netstat -tap |grep LISTEN
  *:exec               *:*        LISTEN    988/inetd
  *:login              *:*        LISTEN    988/inetd
  *:shell              *:*        LISTEN    988/inetd
  *:printer            *:*        LISTEN    988/inetd
  *:time               *:*        LISTEN    988/inetd
  *:x11                *:*        LISTEN    1462/X
  *:http               *:*        LISTEN    1078/httpd
  bigcat:domain        *:*        LISTEN    956/named
  bigcat:domain        *:*        LISTEN    956/named
  *:ssh                *:*        LISTEN    972/sshd
  *:auth               *:*        LISTEN    388/in.identd
```

```
*:telnet            *:*        LISTEN    988/inetd
*:finger            *:*        LISTEN    988/inetd
*:sunrpc            *:*        LISTEN    1290/portmap
*:ftp               *:*        LISTEN    988/inetd
*:smtp              *:*        LISTEN    1738/sendmail: accepting connections
*:1694              *:*        LISTEN    1319/rpc.mountd
*:netbios-ssn       *:*        LISTEN    422/smbd
```

Red Hat 7.x and Mandrake 8.x and later users will have `xinetd` in place of `inetd`. Note the first three columns are cropped above for readability. If your list is as long as the example, you have some work ahead of you! It is highly unlikely that you really need anywhere near this number of servers running.

Please be aware that the example above is just one of many, many possible system configurations. Yours probably does look very different.

You don't understand what any of this is telling you? Hopefully then, you've read the **netstat** tutorial in the Appendix, and understand how it works. Understanding exactly what each server is in the above example, and what it does, is beyond the scope of this document. You will have to check your system's documentation (e.g. Installation Guide, man pages, etc) if that service is important to you. For example, does "exec", "login", and "shell" sound important? Yes, but these are not what they may sound like. They are actually **rexec**, **rlogin**, and **rsh**, the "r" (for remote) commands. These are antiquated, unnecessary, and in fact, are very dangerous if exposed to the Internet.

Let's make a few quick assumptions about what is necessary and unnecessary, and therefore what goes and what stays on bigcat. Since we are running a desktop on bigcat, X11 of course needs to stay. If bigcat were a dedicated server of some kind, then X11 would be unnecessary. If there is a printer physically attached, the printer (lp) daemon should stay. Otherwise, it goes. Print servers may sound harmless, but are potential targets too since they can hold ports open. If we plan on logging *in to* bigcat *from* other hosts, sshd (Secure SHell Daemon) would be necessary. If we have Microsoft hosts on our LAN, we probably want Samba, so smbd should stay. Otherwise, it is completely unnecessary. Everything else in this example is optional and not required for a normally functioning system, and should probably go. See anything that you don't recognize? Not sure about? It goes!

To sum up: since bigcat is a desktop with a printer attached, we will need "x11", "printer". bigcat is on a LAN with MS hosts, and shares files and printing with them, so "netbios−ssn" (**smbd**) is desired. We will also need "ssh" so we can login from other machines. Everything else is unnecessary for this particular case.

Nervous about this? If you want, you can make notes of any changes you make or save the list of servers you got from **netstat**, with this command: `netstat −tap |grep LISTEN > ~/services.lst`. That will save it your home directory with the name of "services.lst" for future reference.

This is to not say that the ones we have decided to keep are inherently safe. Just that we probably need these. So we will have to deal with these via firewalling or other means (addressed below).

It is worth noting that the **telnet** and **ftp** daemons in the above example are *servers*, aka "listeners". These accept incoming connections to you. You do not need, or want, these just to use **ftp** or **telnet** *clients*. For instance, you can download files from an FTP site with just an **ftp** client. Running an ftp server on your end is not required at all, and has serious security implications.

There may be individual situations where it is desirable to make exceptions to the conclusions reached above. See below.

3.2. The Danger Zone (or r00t m3 pl34s3)

The following is a list of services that should *not* be run over the Internet. Either disable these (see below), uninstall, or if you really do need these services running locally, make sure they are the current, patched versions *and* that they are effectively firewalled. And if you don't have a firewall in place now, turn them off until it is up and verified to be working properly. These are potentially insecure by their very nature, and as such are prime cracker targets.

- NFS (Network File System) and related services, including nfsd, lockd, mountd, statd, portmapper, etc. NFS is the standard Unix service for sharing file systems across a network. Great system for LAN usage, but dangerous over the Internet. And its completely unnecessary on a stand alone system.
- rpc.* services, Remote Procedure Call.*, typically NFS and NIS related (see above).
- Printer services (lpd).
- The so−called r* (for "remote", i.e. Remote SHell) services: rsh, rlogin, rexec, rcp etc. Unnecessary, insecure and potentially dangerous, and better utilities are available if these capabilities are needed. ssh will do everything these command do, and in a much more sane way. See the man pages for each if curious. These will probably show in **netstat** output without the "r": **rlogin** will be just "login", etc.
- telnet server. There is no reason for this anymore. Use sshd instead.
- ftp server. There are better, safer ways for most systems to exchange files like **scp** or via **http** (see below). ftp is a proper protocol only for someone who is running a dedicated ftp server, and who has the time and skill to keep it buttoned down. For everyone else, it is potentially big trouble.
- BIND (**named**), DNS server package. With some work, this can be done without great risk, but is not necessary in many situations, and requires special handling no matter how you do it. See the sections on Exceptions and special handling for individual applications.
- Mail Transport Agent, aka "MTA" (sendmail, exim, postfix, qmail). Most installations on single computers will not really need this. If you are not going to be directly receiving mail from Internet hosts (as a designated MX box), but will rather use the POP server of your ISP, then it is not needed. You may however need this if you are receiving mail *directly* from other hosts on your LAN, but initially it's safer to disable this. Later, you can enable it over the local interface once your firewall and access policies have been implemented.

This is not necessarily a definitive list. Just some common services that are sometimes started on default Red Hat installations. And conversely, this does not imply that other services are inherently safe.

3.3. Stopping Services

The next step is to find where each server on our kill list is being started. If it is not obvious from the **netstat** output, use **ps**, **find**, **grep** or **locate** to find more information from the "Program name" or "PID" info in the last column. There is examples of this in the Process Owner section in the **netstat** Tutorial of the Appendix. If the service name or port number do not look familiar to you, you might get a real brief explanation in your `/etc/services` file.

chkconfig is a very useful command for controlling services that are started via init scripts (see example below). Also, where xinetd is used, it can control those services as well. **chkconfig** can tell us what services the system is configured to run, but not necessarily all services that are indeed actually running. Or what services may be started by other means, e.g. from `rc.local`. It is a configuration tool, more than a real time system auditing too.

Skeptical that we are going to break your system, and the pieces won't go back together again? If so, take this approach: turn off everything listed above in "The Danger Zone", and run your system for a while. OK? Try stopping one of the ones we found to be "unnecessary" above. Then, run the system for a while. Keep repeating this process, until you get to the bare minimum. If this works, then make the changes permanent (see below).

The ultimate objective is not just to stop the service now, but to make sure it is stopped permanently! So whatever steps you take here, be sure to check after your next reboot.

There are various places and ways to start system services. Let's look at the most common ways this is done, and is probably how your system works. System services are typically either started by "init" scripts, or by **inetd** (or its replacement **xinetd**) on most distributions.

3.3.1. Stopping Init Services

Init services are typically started automatically during the boot process, or during a runlevel change. There is a naming scheme that uses symlinks to determine which services are to be started, or stopped, at any given runlevel. The scripts themselves should be in `/etc/init.d/` (or possibly `/etc/rc.d/init.d/` for older versions of Red Hat).

You can get a listing of these scripts:

```
 # ls -l /etc/rc.d/init.d/ | less
```

To stop a running service now, as root:

```
# /etc/init.d/<$SERVICE_NAME> stop
```

Where "$SERVICE_NAME" is the name of the init script, which is often, but not always, the same as the service name itself. Older Red Hat versions may use the path `/etc/rc.d/init.d/` instead.

This only stops this particular service now. It will restart again on the next reboot, or runlevel change, unless additional steps are taken. So this is really a two step process for init type services.

chkconfig can be used to see what services are started at each runlevel, and to turn off any unneeded services. To view *all services* under its control, type this command in an xterm:

```
# chkconfig --list | less
```

To view only the ones that are "on":

```
# chkconfig --list | grep "\bon\b" | less
```

The first column is the service name, and the remaining columns are the various runlevels. We need generally only worry about runlevels 3 (boot to text console login) and 5 (boot straight to X11 login). xinetd services won't have columns, since that aspect would be controlled by xinetd itself.

Examples of commands to turn services "off":

```
# chkconfig portmapper off
# chkconfig nfs off
# chkconfig telnet off
# chkconfig rlogin off
```

Note that the last two are xinetd services. A very easy and nifty tool to use! Red Hat also includes **ntsysv** and **tksysv** (GUI) for runlevel and service configuration. See the man pages for additional command line options.

Another option here is to uninstall a package if you know you do not need it. This is a pretty sure−fire, permanent fix. This also alleviates the potential problem of keeping all installed packages updated and current (Step 2). RPM makes it very easy to re−install a package should you change your mind.

To uninstall packages with RPM:

```
# rpm −ev telnet−server  rsh  rsh−server
```

The above command would uninstall the "telnet server" package (but not telnet client!), "rsh" client and "rsh server" packages in one command. Red Hat also includes gnorpm, a GUI RPM management utility which can do this as well.

3.3.2. Inetd

Inetd is called a "super−daemon" because it is used to spawn sub−daemons. **inetd** itself will generally be started via init scripts, and will "listen" on the various ports as determined by which services are enable in its configuration file, `/etc/inetd.conf`. Any service listed here will be under the control of **inetd**. Likewise, any of the listening servers in **netstat** output that list "inetd" in the last column under "Program Name", will have been started by **inetd**. You will have to adjust the **inetd** configuration to stop these services. **xinetd** is an enhanced **inetd** replacement, and is configured differently (see next section below).

Below is a partial snippet from a typical `inetd.conf`. Any service with a "#" at the beginning of the line is "commented out", and thus ignored by **inetd**, and consequently disabled.

```
#
# inetd.conf  This file describes the services that will be available
#    through the INETD TCP/IP super server.  To re−configure
```

```
#    the running INETD process, edit this file, then send the
#    INETD process a SIGHUP signal.
#
# Version:  @(#)/etc/inetd.conf  3.10  05/27/93
#
# Authors:  Original taken from BSD UNIX 4.3/TAHOE.
#    Fred N. van Kempen, <waltje@uwalt.nl.mugnet.org>
#
# Modified for Debian Linux by Ian A. Murdock <imurdock@shell.portal.com>
#
# Echo, discard, daytime, and chargen are used primarily for testing.
#
# To re-read this file after changes, just do a 'killall -HUP inetd'
#
#echo  stream  tcp  nowait  root  internal
#echo  dgram  udp   wait    root  internal
#discard  stream  tcp  nowait  root  internal
#discard  dgram  udp   wait    root  internal
#daytime  stream tcp   nowait  root  internal
#daytime  dgram  udp   wait    root  internal
#chargen  stream tcp   nowait  root  internal
#chargen  dgram  udp   wait    root  internal
time  stream    tcp   nowait  root  internal
#
# These are standard services.
#
#ftp     stream  tcp   nowait  root  /usr/sbin/tcpd  in.ftpd -l -a
#telnet  stream  tcp   nowait  root  /usr/sbin/tcpd  in.telnetd
#
# Shell, login, exec, comsat and talk are BSD protocols.
#
#shell  stream  tcp  nowait  root  /usr/sbin/tcpd  in.rshd
#login  stream  tcp  nowait  root  /usr/sbin/tcpd  in.rlogind
#exec   stream  tcp  nowait  root  /usr/sbin/tcpd  in.rexecd
#comsat dgram   udp  wait    root  /usr/sbin/tcpd  in.comsat
#talk   dgram   udp  wait    root  /usr/sbin/tcpd  in.talkd
#ntalk  dgram   udp  wait    root  /usr/sbin/tcpd  in.ntalkd
#dtalk  stream  tcp  wait    nobody /usr/sbin/tcpd in.dtalkd
#
# Pop and imap mail services et al
#
#pop-2   stream  tcp     nowait  root    /usr/sbin/tcpd  ipop2d
pop-3    stream  tcp     nowait  root    /usr/sbin/tcpd  ipop3d
#imap    stream  tcp     nowait  root    /usr/sbin/tcpd  imapd
#
# The Internet UUCP service.
#
#uucp  stream tcp nowait uucp /usr/sbin/tcpd  /usr/lib/uucp/uucico -l
#

<snip>
```

The above example has two services enabled: **time** and **pop3**. To disable these, all we need is to open the file with a text editor, comment out the two services with a "#", save the file, and then restart **inetd** (as root):

```
  # /etc/rc.d/init.d/inetd restart
```

Check your logs for errors, and run **netstat** again to verify all went well.

A quicker way of getting the same information, using **grep**:

```
$ grep -v '^#' /etc/inetd.conf
time     stream  tcp     nowait  root   internal
pop-3    stream  tcp     nowait  root   /usr/sbin/tcpd  ipop3d
```

Again, do you see anything there that you don't know what it is? Then in all likelihood you are not using it, and it should be disabled.

Unlike the init services configuration, this is a lasting change so only the one step is required.

Let's expose one myth that gets tossed around: you shouldn't disable a service by commenting out, or removing, entries from `/etc/services`. This may have the desired effect in some cases, but is not the right way to do it, and may interfere with the normal operation of other system utilities.

3.3.3. Xinetd

xinetd is an inetd replacement with enhancements. Red Hat includes xinetd with 7.0 and later releases. It essentially serves the same purpose as inetd, but the configuration is different. The configuration can be in the file `/etc/xinetd.conf`, or individual files in the directory `/etc/xinetd.d/`. Configuration of individual services will be in the individual files under `/etc/xinetd.d/*`. Turning off xinetd services is done by either deleting the corresponding configuration section, or file. Or by using your text editor and simply setting `disable = yes` for the appropriate service. Or by using **chkconfig**. Then, xinetd will need to be restarted. See `man xinetd` and `man xinetd.conf` for syntax and configuration options. A sample **xinetd** configuration:

```
# default: on
# description: The wu-ftpd FTP server serves FTP connections. It uses \
#       normal, unencrypted usernames and passwords for authentication.
service ftp
{
       disable                 = no
       socket_type             = stream
       wait                    = no
       user                    = root
       server                  = /usr/sbin/in.ftpd
       server_args             = -l -a
       log_on_success          += DURATION USERID
       log_on_failure          += USERID
       nice                    = 10
}
```

You can get a quick list of enabled services:

```
$ grep disable /etc/xinetd.d/* |grep no
/etc/xinetd.d/finger:   disable = no
/etc/xinetd.d/rexec:    disable = no
```

```
/etc/xinetd.d/rlogin:   disable = no
/etc/xinetd.d/rsh:      disable = no
/etc/xinetd.d/telnet:   disable = no
/etc/xinetd.d/wu-ftpd:  disable = no
```

At this point, the above output should raise some red flags. In the overwhelming majority of systems, all the above can be disabled without any adverse impact. Not sure? Try it without that service. After disabling unnecessary services, then restart **xinetd**:

```
 # /etc/rc.d/init.d/xinetd restart
```

3.3.4. When All Else Fails

OK, if you can't find the "right" way to stop a service, or maybe a service is being started and you can't find how or where, you can "kill" the process. To do this, you will need to know the PID (Process I.D.). This can be found with **ps**, **top**, **fuser** or other system utilities. For **top** and **ps**, this will be the number in the first column. See the Port and Process Owner section in the Appendix for examples.

Example (as root):

```
# kill 1163
```

Then run **top** or **ps** again to verify that the process is gone. If not, then:

```
# kill -KILL 1163
```

Note the second "KILL" in there. This must be done either by the user who owns the process, or root. Now go find where and how this process got started ;−)

The `/proc` filesystem can also be used to find out more information about each process. Armed with the PID, we can find the path to a mysterious process:

```
$ /bin/ps ax|grep tcpgate
 921 ?   S    0:00        tcpgate
```

```
# ls -l /proc/921/exe
lrwxrwxrwx 1 root  root  0 July 21 12:11 /proc/921/exe -> /usr/local/bin/tcpgate
```

3.4. Exceptions

Above we used the criteria of turning off *all* unnecessary services. Sometimes that is not so obvious. And sometimes what may be required for one person's configuration is not the same for another's. Let's look at a few common services that fall in this category.

Again, our rule of thumb is if we don't need it, we won't run it. It's that simple. If we do need any of these, they are prime candidates for some kind of restrictive policies via firewall rules or other mechanisms (see below).

* identd − This is a protocol that has been around for ages, and is often installed and running by default. It is used to provide a minimal amount of information about who is connecting to a server. But, it is not necessary in many cases. Where might you need it? Most IRC servers require it. Many mail servers use it, but don't really require it. Try your mail setup without it. If identd is going to be a problem, it will be because there is a time out before before the server starts sending or receiving mail. So mail should work fine without it, but may be slower. A few ftp servers may require it. Most don't though. Older versions of Red Hat started identd via inetd. Recent versions start this via init scripts.

  If identd is required, there are some configuration options that can greatly reduce the information that is revealed:

  ```
  /usr/sbin/in.identd in.identd −l −e −o −n −N
  ```

  The `−o` flag tells identd to not reveal the operating system type it is run on and to instead always return "OTHER". The `−e` flag tells identd to always return "UNKNOWN−ERROR" instead of the "NO−USER" or "INVALID−PORT" errors. The `−n` flag tells identd to always return user numbers instead of user names, if you wish to keep the user names a secret. The `−N` flag makes identd check for the file `.noident` in the user's home directory for which the daemon is about to return a user name. It that file exists then the daemon will give the error "HIDDEN−USER" instead of the normal "USERID" response.
* Mail server (MTA's like sendmail, qmail, etc) − Often a fully functional mail server like sendmail is installed by default. The only time that this is actually required is if you are hosting a domain, and receiving incoming mail directly. Or possibly, for exchanging mail on a LAN, in which case it does not need Internet exposure and can be safely firewalled. For your ISP's POP mail access, you don't need it even though this is a common configuration. One alternative here is to use fetchmail for POP mail retrieval with the `−m` option to specify a local delivery agent: `fetchmail −m procmail` for instance works with no sendmail daemon running at all. Sendmail, can be handy to have running, but the point is, it is not required in many situations, and can be disabled, or firewalled safely.
* BIND (named) − This often is installed by default, but is only really needed if you are an authoritative name server for a domain. If you are not sure what this means, then you definitely don't need it. BIND is probably the number one crack target on the Internet. BIND is often used though in a "caching" only mode. This can be quite useful, but does not require full exposure to the Internet. In other words, it should be restricted or firewalled. See special handling of individual applications below.

3.5. Summary and Conclusions for Step 1

In this section we learned how to identify which services are running on our system, and were given some tips on how to determine which services may be necessary. Then we learned how to find where the services were being started, and how to stop them. If this has not made sense, now is a good time to re−read the above.

Hopefully you've already taken the above steps. Be sure to test your results with **netstat** again, just to verify the desired end has been achieved, and only the services that are really required are running.

It would also be wise to do this after the next reboot, anytime you upgrade a package (to make sure a new configuration does not sneak in), and after every system upgrade or new install.

---

4. Step 2: Updating

OK, this section should be comparatively short, simple and straightforward compared to the above, but no less important.

The very first thing after a new install you should check the errata notices at http://redhat.com/apps/errata/, and apply all relevant updates. Only a year old you say? That's a long time actually, and not current enough to be safe. Only a few months or few weeks? Check anyway. A day or two? Better safe than sorry. It is quite possible that security updates have been released during the pre−release phase of the development and release cycle. If you can't take this step, disable any publicly accessible services until you can.

Linux distributions are not static entities. They are updated with new, patched packages as the need arises. The updates are just as important as the original installation. Even more so, since they are fixes. Sometimes these updates are bug fixes, but quite often they are security fixes because some hole has been discovered. Such "holes" are *immediately* known to the cracker community, and they are quick to exploit them on a large scale. Once the hole is known, it is quite simple to get in through it, and there will be many out there looking for it. And Linux developers are also equally quick to provide fixes. Sometimes the same day as the hole has become known!

Keeping *all* installed packages current with your release is one of the most important steps you can take in maintaining a secure system. It can not be emphasized enough that all installed packages should be kept updated −− not just the ones you use. If this is burdensome, consider uninstalling any unused packages. Actually this is a good idea anyway.

But where to get this information in a timely fashion? There are a number of web sites that offer the latest security news. There are also a number of mailing lists dedicated to this topic. In fact, Red Hat has the "watch" list, just for this purpose at https://listman.redhat.com/mailman/listinfo/redhat−watch−list. This is a very low volume list by the way. This is an excellent way to stay abreast of issues effecting your release, and is *highly recommended*. http://linuxsecurity.com is a good site for Linux only issues. They also have weekly newsletters available: http://www.linuxsecurity.com/general/newsletter.html.

Red Hat also has the up2date utility for automatically keeping your system(s) up to date ;−). See the man page for details.

This is not a one time process −− it is ongoing. It is important to stay current. So watch those security notices. And subscribe to that security mailing list today! If you have cable modem, DSL, or other full time connection, there is no excuse not to do this religiously. All distributions make this easy enough!

One last note: any time a new package is installed, there is also a chance that a new or revised configuration has been installed as well. Which means that if this package is a server of some kind, it may be enabled as a result of the update. This is bad manners, but it can happen, so be sure to run netstat or comparable to verify your system is where you want it after any updates or system changes. In fact, do it periodically even if there are no such changes.

4.1. Summary and Conclusions for Step 2

It is very simple: make sure your Linux installation is current. Check the Red Hat errata for what updated packages may be available. There is nothing wrong with running an older release, just so the packages in it are updated according to what Red Hat has made available since the initial release. At least as long as Red Hat is still supporting the release and updates are still being provided. For instance, Red Hat has stopped providing updates for 5.0 and 5.1, but still does for 5.2.

---

5. Step 3: Firewalls and Setting Access Policies

So what is a "firewall"? It's a vague term that can mean anything that acts as a protective barrier between us and the outside world. This can be a dedicated system, or a specific application that provides this functionality. Or it can be a combination of components, including various combinations of hardware and software. Firewalls are built from "rules" that are used to define what is allowed to enter and exit a given system or network. Let's look at some of the possible components that are readily available for Linux, and how we might implement a reasonably safe firewalling strategy.

In Step 1 above, we have turned off all services we don't need. In our example, there were a few we still needed to have running. In this section, we will take the next step here and decide which we need to leave open to the world. And which we might be able to restrict in some way. If we can block them all, so much the better, but this is not always practical.

5.1. Strategy

What we want to do now is restrict connections and traffic so that we only allow the minimum necessary for whatever our particular situation is. In some cases we may want to block all incoming "new" connection attempts. Example: we want to run X, but don't want anyone from outside to access it, so we'll block it completely from outside connections. In other situations, we may want to limit, or restrict, incoming connections to trusted sources only. The more restrictive, the better. Example: we want to **ssh** into our system from outside, but we only ever do this from our workplace. So we'll limit **sshd** connections to our workplace address range. There are various ways to do this, and we'll look at the most common ones.

We also will not want to limit our firewall to any one application. There is nothing wrong with a "layered" defense−in−depth approach. Our front line protection will be a packet filter −− either ipchains or iptables (see below). Then we can use additional tools and mechanisms to reinforce our firewall.

We will include some brief examples. Our rule of thumb will be to deny everything as the default policy, then open up just what we need. We'll try to keep this as simple as possible since it can be an involved and complex topic, and just stick to some of the most basic concepts. See the Links section for further reading on this topic.

5.2. Packet Filters −− Ipchains and Iptables

"Packet filters" (like ipchains) have the ability to look at individual packets, and make decisions based on what they find. These can be used for many purposes. One common purpose is to implement a firewall.

Common packet filters on Linux are ipchains which is standard with 2.2 kernels, and iptables which is available with the more recent 2.4 kernels. iptables has more advanced packet filtering capabilities and is recommended for anyone running a 2.4 kernel. But either can be effective for our purposes. ipfwadm is a similar utility for 2.0 kernels (not discussed here).

If constructing your own ipchains or iptables firewall rules seems a bit daunting, there are various sites that can automate the process. See the Links section. Also the included examples may be used as a starting point. As of Red Hat 7.1, Red Hat is providing init scripts for ipchains and iptables, and gnome−lokkit for generating a very basic set of firewall rules (see below). This may be adequate, but it is still recommended to know the proper syntax and how the various mechanisms work as such tools rarely do more than a few very simple rules.

Various examples are given below. These are presented for illustrative purposes to demonstrate some of the concepts being discussed here. While they might also be useful as a starting point for your own script, please note that they are not meant to be all encompassing. You are strongly encouraged to understand how the scripts work, so you can create something even more tailored for your own situation.

5.2.1. ipchains

ipchains can be used with either 2.2 or 2.4 kernels. When ipchains is in place, it checks every packet that moves through the system. The packets move across different "chains", depending where they originate and where they are going. Think of "chains" as rule sets. In advanced configurations, we could define our own custom chains. The three default built−in chains are `input`, which is incoming traffic, `output`, which is outgoing traffic, and `forward`, which is traffic being forwarded from one interface to another (typically used for "masquerading"). Chains can be manipulated in various ways to control the flow of traffic in and out of our system. Rules can be added at our discretion to achieve the desired result.

At the end of every "chain" is a "target". The target is specified with the `−j` option to the command. The target is what decides the fate of the packet and essentially terminates that particular chain. The most common targets are mostly self−explanatory: `ACCEPT`, `DENY`, `REJECT`, and `MASQ`. `MASQ` is for "ipmasquerading". `DENY` and `REJECT` essentially do the same thing, though in different ways. Is one better than the other? That is the subject of much debate, and depends on other factors that are beyond the scope of this document. For our purposes, either should suffice.

ipchains has a very flexible configuration. Port (or port ranges), interfaces, destination address, source address can be specified, as well as various other options. The man page explains these details well enough that we won't get into specifics here.

Traffic entering our system from the Internet, enters via the `input` chain. This is the one that we need as tight as we can make it.

Below is a brief example script for a hypothetical system. We'll let the comments explain what this script does. Anything starting with a "#" is a comment. ipchains rules are generally incorporated into shell scripts, using shell variables to help implement the firewalling logic.

```
#!/bin/sh
#
# ipchains.sh
#
# An example of a simple ipchains configuration.
#
# This script allows ALL outbound traffic, and denies
# ALL inbound connection attempts.
#
###################################################################
```

```
# Begin variable declarations and user configuration options ######
#
IPCHAINS=/sbin/ipchains
# This is the WAN interface, that is our link to the outside world.
# For pppd and pppoe users.
# WAN_IFACE="ppp0"
WAN_IFACE="eth0"

## end user configuration options #################################
###################################################################

# The high ports used mostly for connections we initiate and return
# traffic.
LOCAL_PORTS=`cat /proc/sys/net/ipv4/ip_local_port_range |cut -f1`:\
`cat /proc/sys/net/ipv4/ip_local_port_range |cut -f2`

# Any and all addresses from anywhere.
ANYWHERE="0/0"

# Let's start clean and flush all chains to an empty state.
ipchains -F

# Set the default policies of the built-in chains. If no match for any
# of the rules below, these will be the defaults that ipchains uses.
ipchains -P forward DENY
ipchains -P output ACCEPT
ipchains -P input DENY

# Accept localhost/loopback traffic.
ipchains -A input -i lo -j ACCEPT

# Get our dynamic IP now from the Inet interface. WAN_IP will be our
# IP address we are protecting from the outside world. Put this
# here, so default policy gets set, even if interface is not up
# yet.
WAN_IP=`ifconfig $WAN_IFACE |grep inet |cut -d : -f 2 |cut -d \  -f 1`

# Bail out with error message if no IP available! Default policy is
# already set, so all is not lost here.
[ -z "$WAN_IP" ] && echo "$WAN_IFACE not configured, aborting." && exit 1

# Accept non-SYN TCP, and UDP connections to LOCAL_PORTS. These are
# the high, unprivileged ports (1024 to 4999 by default). This will
# allow return connection traffic for connections that we initiate
# to outside sources. TCP connections are opened with 'SYN' packets.
ipchains -A input -p tcp -s $ANYWHERE -d $WAN_IP $LOCAL_PORTS ! -y -j ACCEPT

# We can't be so selective with UDP since that protocol does not
# know about SYNs.
ipchains -A input -p udp -s $ANYWHERE -d $WAN_IP $LOCAL_PORTS -j ACCEPT

## ICMP (ping)
#
# ICMP rules, allow the bare essential types of ICMP only. Ping
# request is blocked, ie we won't respond to someone else's pings,
# but can still ping out.
ipchains -A input  -p icmp  --icmp-type echo-reply \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
ipchains -A input  -p icmp  --icmp-type destination-unreachable \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
ipchains -A input  -p icmp  --icmp-type time-exceeded \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
```

```
#################################################################
# Set the catchall, default rule to DENY, and log it all. All other
# traffic not allowed by the rules above, winds up here, where it is
# blocked and logged. This is the default policy for this chain
# anyway, so we are just adding the logging ability here with '-l'.
# Outgoing traffic is allowed as the default policy for the 'output'
# chain. There are no restrictions on that.

ipchains -A input -l -j DENY

echo "Ipchains firewall is up `date`."

##-- eof ipchains.sh
```

To use the above script would require that it is executable (i.e. `chmod +x ipchains.sh`), and run by root to build the chains, and hence the firewall.

To summarize what this example did was to start by setting some shell variables in the top section, to be used later in the script. Then we set the default rules (ipchains calls these "policies") of denying all inbound and forwarded traffic, and of allowing all our own outbound traffic. We had to open some holes in the high, unprivileged ports so that we could have return traffic from connections that bigcat initiates to outside addresses. If we connect to someone's web server, we want that HTML data to be able to get back to us, for instance. The same applies to other network traffic. We then allowed a few specific types of the ICMP protocol (most are still blocked). We are also logging any inbound traffic that violates any of our rules so we know who is doing what. Notice that we are only using IP address here, not hostnames of any kind. This is so that our firewall works, even in situation where there may be DNS failures. Also, to prevent any kind of DNS spoofing.

See the ipchains man page for a full explanation of syntax. The important ones we used here are:

> `-A input`: Adds a rule to the "input" chain. The default chains are input, output, and forward.
>
> `-p udp`: This rule only applies to the "UDP" "protocol". The `-p` option can be used with tcp, udp or icmp protocols.
>
> `-i $WAN_IFACE`: This rule applies to the specified interface only, and applies to whatever chain is referenced (input, output, or forward).
>
> `-s <IP address>` [port]: This rule only applies to the source address as specified. It can optionally have a port (e.g. 22) immediately afterward, or port range, e.g. 1023:4999.
>
> `-d <IP address>` [port]: This rule only applies to the destination address as specified. Also, it may include port or port range.
>
> `-l` : Any packet that hits a rule with this option is logged (lower case "L").

`−j ACCEPT`: Jumps to the "ACCEPT" "target". This effectively terminates this chain and decides the ultimate fate for this particular packet, which in this example is to "ACCEPT" it. The same is true for other `−j` targets like `DENY`.

By and large, the order in which command line options are specified is not significant. The chain name (e.g. `input`) must come first though.

Remember in Step 1 when we ran **netstat**, we had both X and print servers running among other things. We don't want these exposed to the Internet, even in a limited way. These are still happily running on bigcat, but are now safe and sound behind our ipchains based firewall. You probably have other services that fall in this category as well.

The above example is a simplistic all or none approach. We allow all our own outbound traffic (not necessarily a good idea), and block all inbound connection attempts. It would more than likely require a bit of fine tuning to make it work for you. For a more advanced set of rules, see the Appendix. And you might want to read http://linuxdoc.org/HOWTO/IPCHAINS−HOWTO.html.

Whenever you have made changes to your firewall, you should verify its integrity. One step to make sure your rules seem to be doing what you intended, is to see how ipchains has interpreted your script. You can do this by opening your xterm very wide, and issuing the following command:

```
# ipchains −L −n −v | less
```

The output is grouped according to chain. You should also find a way to scan yourself (see the Verifying section below). And then keep an eye on your logs to make sure you are blocking what is intended.

5.2.2. iptables

iptables is the next generation packet filter for Linux, and requires a 2.4 kernel. It can do everything ipchains can, but has a number of noteworthy enhancements. The syntax is similar to ipchains in many respects. See the man page for details.

The most noteworthy enhancement is "connection tracking", also known as "stateful inspection". This gives iptables more knowledge of the state of each packet. Not only does it know if the packet is a TCP or UDP packet, or whether it has the SYN or ACK flags set, but also if it is part of an existing connection, or related somehow to an existing connection. The implications for firewalling should be obvious.

The bottom line is that it is easier to get a tight firewall with iptables, than with ipchains. So this is the recommended way to go.

Here is the same script as above, revised for iptables:

```
#!/bin/sh
#
# iptables.sh
#
# An example of a simple iptables configuration.
#
```

```
# This script allows ALL outbound traffic, and denies
# ALL inbound connection attempts.
#
###################################################################
# Begin variable declarations and user configuration options ######
#

# Local Interfaces
# This is the WAN interface that is our link to the outside world.
# For pppd and pppoe users.
# WAN_IFACE="ppp0"
WAN_IFACE="eth0"
#

## end user configuration options #################################
###################################################################

# Any and all addresses from anywhere.
ANYWHERE="0/0"

# This module may need to be loaded:
modprobe ip_conntrack_ftp

# Start building chains and rules #################################
#
# Let's start clean and flush all chains to an empty state.
iptables -F

# Set the default policies of the built-in chains. If no match for any
# of the rules below, these will be the defaults that IPTABLES uses.
iptables -P FORWARD DROP
iptables -P OUTPUT ACCEPT
iptables -P INPUT DROP

# Accept localhost/loopback traffic.
iptables -A INPUT -i lo -j ACCEPT

## ICMP (ping)
#
# ICMP rules, allow the bare essential types of ICMP only. Ping
# request is blocked, ie we won't respond to someone else's pings,
# but can still ping out.
iptables -A INPUT  -p icmp  --icmp-type echo-reply \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
iptables -A INPUT  -p icmp  --icmp-type destination-unreachable \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
iptables -A INPUT  -p icmp  --icmp-type time-exceeded \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT

###################################################################
# Set the catchall, default rule to DENY, and log it all. All other
# traffic not allowed by the rules above, winds up here, where it is
# blocked and logged. This is the default policy for this chain
# anyway, so we are just adding the logging ability here with '-j
# LOG'. Outgoing traffic is allowed as the default policy for the
# 'output' chain. There are no restrictions on that.

iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -m state --state NEW -i ! $WAN_IFACE -j ACCEPT
iptables -A INPUT -j LOG -m limit --limit 30/minute --log-prefix "Dropping: "

echo "Iptables firewall is up `date`."
```

```
##-- eof iptables.sh
```

The same script logic is used here, and thus this does pretty much the same exact thing as the ipchains script in the previous section. There are some subtle differences as to syntax. Note the case difference in the chain names for one (e.g. INPUT vs input). Logging is handled differently too. It has its own "target" now (`−j LOG`), and is much more flexible.

There are some very fundamental differences as well, that might not be so obvious. Remember this section from the ipchains script:

```
# Accept non-SYN TCP, and UDP connections to LOCAL_PORTS. These are the high,
# unprivileged ports (1024 to 4999 by default). This will allow return
# connection traffic for connections that we initiate to outside sources.
# TCP connections are opened with 'SYN' packets. We have already opened
# those services that need to accept SYNs for, so other SYNs are excluded here
# for everything else.
ipchains -A input -p tcp -s $ANYWHERE -d $WAN_IP $LOCAL_PORTS ! -y -j ACCEPT

# We can't be so selective with UDP since that protocol does not know
# about SYNs.
ipchains -A input -p udp -s $ANYWHERE -d $WAN_IP $LOCAL_PORTS -j ACCEPT
```

We jumped through hoops here with ipchains so that we could restrict unwanted, incoming connections as much as possible. A bit of a kludge, actually.

That section is missing from the iptables version. It is not needed as connection tracking handles this quite nicely, and then some. This is due to the "statefulness" of iptables. It knows more about each packet than ipchains. For instance, it knows whether the packet is part of a "new" connection, or an "established" connection, or a "related" connection. This is the so−called "stateful inspection" of connection tracking.

There are many, many features of iptables that are not touched on here. For more reading on the Netfilter project and iptables, see http://netfilter.samba.org. And for a more advanced set of rules, see the Appendix.

5.2.3. Red Hat Firewall Configuration Tools

Red Hat has not included firewall configuration tools until 7.1, when the GUI utility **gnome−lokkit** started being bundled. **gnome−lokkit** does a minimalist set of rules for ipchains only. Explicit support for iptables configuration is not an option, despite the fact that the default kernel is 2.4.

gnome−lokkit is an option on non−upgrade installs, and can also be run as a stand−alone app any time after installation. It will ask a few simple questions, and dump the resulting rule−set into `/etc/sysconfig/ipchains`.

As mentioned, this is a fairly minimalist set of rules, and possibly a sufficient starting point. An example `/etc/sysconfig/ipchains` created by **gnome−lokkit**:

```
# Firewall configuration written by lokkit
# Manual customization of this file is not recommended.
# Note: ifup−post will punch the current nameservers through the
#       firewall; such entries will *not* be listed here.
:input ACCEPT
:forward ACCEPT
:output ACCEPT
−A input −s 0/0 −d 0/0 80 −p tcp −y −j ACCEPT
−A input −s 0/0 −d 0/0 25 −p tcp −y −j ACCEPT
−A input −s 0/0 −d 0/0 22 −p tcp −y −j ACCEPT
−A input −s 0/0 −d 0/0 23 −p tcp −y −j ACCEPT
−A input −s 0/0 −d 0/0 −i lo −j ACCEPT
−A input −s 0/0 −d 0/0 −i eth1 −j ACCEPT
−A input −s 127.0.0.1 53 −d 0/0 −p udp −j ACCEPT
−A input −s 0/0 −d 0/0 −p tcp −y −j REJECT
−A input −s 0/0 −d 0/0 −p udp −j REJECT
```

This is in a format that can be read by the ipchains command **ipchains−restore**. Consequently, a new or modified set or rules can be generated with the **ipchains−save**, and redirecting the output to this file. **ipchains−restore** is indeed how the ipchains init script processes this file. So for this to work, the ipchains service must be activated:

```
# chkconfig ipchains on
```

Conversely, if you want to roll your own iptables rules instead, you should make sure the ipchains init service is disabled. There is also an iptables init script, that works much the same as the ipchains version. There is just no support from **gnome−lokkit** at this time.

5.3. Tcpwrappers (libwrap)

Tcpwrappers provides much the same desired results as ipchains and iptables above, though works quite differently. Tcpwrappers actually intercepts the connection attempt, then examines its configurations files, and decides whether to accept or reject the request. Tcpwrappers controls access at the application level, rather than the socket level like iptables and ipchains. This can be quite effective, and is a standard component on most Linux systems.

Tcpwrappers consists of the configuration files `/etc/hosts.allow` and `/etc/hosts.deny`. The functionality is provided by the libwrap library.

Tcpwrappers first looks to see if access is permitted in `/etc/hosts.allow`, and if so, access is granted. If not in `/etc/hosts.allow`, the file `/etc/hosts.deny` is then checked to see if access is *not* allowed. If so, access is denied. Else, *access is granted*. For this reason, `/etc/hosts.deny` should contain only one uncommented line, and that is: `ALL: ALL`. Access should then be permitted through entries in `/etc/hosts.allow`, where specific services are listed, along with the specific host addresses allowed to access these services. While hostnames can be used here, use of hostnames opens the limited possibility for name spoofing.

Tcpwrappers is commonly used to protect services that are started via inetd (or xinetd). But also any program that has been compiled with libwrap support, can take advantage of it. Just don't assume that all programs have built in libwrap support −− they do not. In fact, most probably don't. So we will only use it in our examples here to protect services start via inetd. And then rely on our packet filtering firewall, or other mechanism, to protect non−(x)inetd services.

Below is a small snippet from a typical `inetd.conf` file:

```
# Pop and imap mail services et al
#
#pop-2   stream  tcp     nowait  root    /usr/sbin/tcpd ipop2d
#pop-3   stream  tcp     nowait  root    /usr/sbin/tcpd ipop3d
#imap    stream  tcp     nowait  root    /usr/sbin/tcpd imapd
#
```

The second to last column is the tcpwrappers daemon −− **/usr/sbin/tcpd**. Immediately after is the daemon it is protecting. In this case, POP and IMAP mail servers. Your distro probably has already done this part for you. For the few applications that have built−in support for tcpwrappers via the libwrap library, specifying the daemon as above is not necessary.

We will use the same principles here: default policy is to deny everything, then open holes to allow the minimal amount of traffic necessary.

So now with your text editor, **su** to root and open `/etc/hosts.deny`. If it does not exist, then create it. It is just a plain text file. We want the following line:

```
ALL: ALL
```

If it is there already, fine. If not, add it in and then save and close file. Easy enough. "ALL" is one of the keywords that tcpwrappers understands. The format is `$SERVICE_NAME : $WHO`, so we are denying all connections to all services here. At least all services that are using tcpwrappers. Remember, this will primarily be inetd services. See `man 5 hosts_access` for details on the syntax of these files. Note the "5" there!

Now let's open up just the services we need, as restrictively as we can, with a brief example:

```
ALL: 127.0.0.1
sshd,ipop3d: 192.168.1.
sshd: .myworkplace.com, hostess.mymomshouse.com
```

The first line allows all "localhost" connections. You will need this. The second allows connections to the sshd and ipop3d services from IP addresses that start with `192.168.1.`, in this case the private address range for our hypothetical home LAN. Note the trailing ".". It's important. The third line allows connections to only our sshd daemon from any host associated with `.myworkplace.com`. Note the leading "." in this example. And then also, the single host `hostess.mymomshouse.com`. In summary, localhost and all our LAN connections have access to any and all tcpwrappered services on bigcat. But only our workplace addresses, and our mother can use sshd on bigcat from outside connections. Everybody else is denied by the default policy in `/etc/hosts.deny`.

The types of wild cards above (`.myworkplace.com` and `192.168.1.`) are not supported by ipchains and iptables, or most other Linux applications for that matter. Also, tcpwrappers can use hostnames in place of IP addresses which is quite handy in some situations. This does not work with ipchains and iptables.

You can test your tcpwrappers configuration with the included **tcpdchk** utility (see the man page). Note that at this time this does not work with xinetd, and may not even be included in this case.

There is nothing wrong with using both tcpwrappers and a packet filtering firewall like ipchains. In fact, it is recommended to use a "layered" approach. This helps guard against accidental misconfigurations. In this case, each connection will be tested by the packet filter rules first, then tcpwrappers.

Remember to make backup copies before editing system configuration files, restart the daemon afterward, and then check the logs for error messages.

5.3.1. xinetd

As mentioned, xinetd is an enhanced inetd , and replaces inetd as of Red Hat 7.0. It has much of the same functionality, with some notable enhancements. One is that tcpwrappers support be is compiled in, eliminating the need for explicit references to **tcpd**. Which means `/etc/hosts.allow` and `/etc/hosts.deny` are automatically in effect.

Some of xinetd's other enhancements: specify IP address to listen on, which is a very effective method of access control; limit the rate of incoming connections and the total number of simultaneous connections; limit services to specific times of day. See the xinetd and xinetd.conf man pages for more details.

The syntax is quite different though. An example from `/etc/xinetd.d/tftp`:

```
service tftp
{
        socket_type     = dgram
        bind            = 192.168.1.1
        instances       = 2
        protocol        = udp
        wait            = yes
        user            = nobody
        only_from       = 192.168.1.0
        server          = /usr/sbin/in.tftpd
        server_args     = /tftpboot
        disable         = no
}
```

Notice the `bind` statement. We are only listening on, or "binding" to, the private, LAN interface here. No outside connections can be made since the outside port is not even opened. We are also only accepting connections from `192.168.1.0`, our LAN. For xinetd's purposes, this denotes any IP address beginning with "192.168.1". Note that the syntax is different from inetd. The `server` statement in this case is the **tftp** daemon, **in.tftpd**. Again, this assumes that libwrap/tcpwrappers support is compiled into xinetd. The `user` running the daemon will be "nobody". Yes, there is a user account called "nobody", and it is wise to run such daemons as non−root users whenever possible. Lastly, the `disable` statement is xinetd's way of turning services on or off. In this case, it is "on". This is on here only as an example. Do NOT run tftp as a public service as it is unsafe.

5.4. PortSentry

Portsentry works quite differently than the other tools discussed so far. Portsentry does what its name implies −− it guards ports. Portsentry is configured with the `/etc/portsentry/portsentry.conf` file.

Unlike the other applications discussed above, it does this by actually becoming the listening server on those ports. Kind of like baiting a trap. Running `netstat -taup` as root while portsentry is running, will show portsentry as the `LISTENER` on whatever ports portsentry is configured for. If portsentry senses a connection attempt, it blocks it completely. And then goes a step further and blocks the route to that host to stop all further traffic. Alternately, ipchains or iptables can be used to block the host completely. So it makes an excellent tool to stop port scanning of a range of ports.

But portsentry has limited flexibility as to whether it allows a given connection. It is pretty much all or nothing. You can define specific IP addresses that it will ignore in `/etc/portsentry/portsentry.ignore`. But you cannot allow selective access to individual ports. This is because only one server can bind to a particular port at the same time, and in this case that is portsentry itself. So it has limited usefulness as a stand−alone firewall. As part of an overall firewall strategy, yes, it can be quite useful. For most of us, it should not be our first line of defense, and we should only use it in conjunction with other tools.

Suggestion on when portsentry might be useful:

- As a second layer of defense, behind either ipchains or iptables. Packet filtering will catch the packets first, so that anything that gets to portsentry would indicate a misconfiguration. Do not use in conjunction with inetd services −− it won't work. They will butt heads.
- As a way to catch full range ports scans. Open a pinhole or two in the packet filter, and let portsentry catch these and re−act accordingly.
- If you are *very sure* you have no exposed public servers at all, and you just want to know who is up to what. But do not assume anything about what portsentry is protecting. By default it does not watch all ports, and may even leave some very commonly probed ports open. So make sure you configure it accordingly. And make sure you have tested and verified your set up first, and that nothing is exposed.

All in all, the packet filters make for a better firewall.

5.5. Proxies

The dictionary defines "proxy" as "the authority or power to act on behalf of another". This pretty well describes software proxies as well. It is an intermediary in the connection path. As an example, if we were using a web proxy like "squid" (http://www.squid−cache.org/), every time we browse to a web site, we would actually be connecting to our locally running squid server. Squid in turn, would relay our request to the ultimate, real destination. And then squid would relay the web pages back to us. It is a go−between. Like "firewalls", a "proxy" can refer to either a specific application, or a dedicated server which runs a proxy application.

Proxies can perform various duties, not all of which have much to do with security. But the fact that they are an intermediary, makes them a good place to enforce access control policies, limit direct connections through a firewall, and control how the network behind the proxy looks to the Internet. So this makes them strong candidates to be part of an overall firewall strategy. And, in fact, are sometimes used instead of packet filtering firewalls. Proxy based firewalls probably make more sense where many users are behind the same firewall. And it probably is not high on the list of components necessary for home based systems.

Configuring and administering proxies can be complex, and is beyond the scope of this document. The Firewall and Proxy Server HOWTO, http://linuxdoc.org/HOWTO/Firewall−HOWTO.html, has examples of setting up proxy firewalls. Squid usage is discussed at http://squid−docs.sourceforge.net/latest/html/book1.htm

---

5.6. Individual Applications

Some servers may have their own access control features. You should check this for each server application you run. We'll only look at a few of the common ones in this section. Man pages, and other application specific documentation, is your friend here. This should be done whether you have confidence in your firewall or not. Again, layers of protection is always best.

- BIND − a very common package that provides name server functionality. The daemon itself is "named". This only requires full exposure to the Internet if you are providing DNS look ups for one or more domains to the rest of the world. If you are not sure what this means, *you do not need, or want*, it exposed. For the overwhelming majority of us this is the case. It is a very common crack target.

  But it may be installed, and can be useful in a caching only mode. This does not require full exposure to the Internet. Limit the interfaces on which it "listens" by editing `/etc/named.conf` (random example shown):

  ```
  options {
    directory "/var/named";
    listen-on { 127.0.0.1; 192.168.1.1; };
    version "N/A";
  };
  ```

  The "listen−on" statement is what limits where named listens for DNS queries. In this example, only on localhost and bigcat's LAN interface. There is no port open for the rest of the world. It just is not there. Restart **named** after making changes.

- X11 can be told not to allow TCP connections by using the `-nolisten tcp` command line option. If using **startx**, you can make this automatic by placing `alias startx="startx -- -nolisten tcp"` in your `~/.bashrc`, or the system−wide file, `/etc/bashrc`, with your text editor. If using xdm (or variants such as gdm, kdm, etc), this option would be specified in /etc/X11/xdm/Xservers (or comparable) as `:0 local /usr/bin/X11/X -nolisten tcp`. gdm actually uses `/etc/X11/gdm/gdm.conf`.

  If using xdm (or comparable) to start X automatically at boot, `/etc/inittab` can be modified as: `xdm -udpPort 0`, to further restrict connections. This is typically near the bottom of

`/etc/inittab`.

- Recent versions of sendmail can be told to listen only on specified addresses:

```
# SMTP daemon options
O DaemonPortOptions=Port=smtp,Addr=127.0.0.1, Name=MTA
```

  The above excerpt is from `/etc/sendmail.cf` which can be carefully added with your text editor. The `sendmail.mc` directive is:

```
dnl This changes sendmail to only listen on the loopback device 127.0.0.1
dnl and not on any other network devices.
DAEMON_OPTIONS(`Port=smtp,Addr=127.0.0.1, Name=MTA')
```

  In case you would prefer to build a new `sendmail.cf`, rather than edit the existing one. Other mail server daemons likely have similar configuration options. Check your local documentation. As of Red Hat 7.1, sendmail has compiled in support for tcpwrappers as well.

- SAMBA connections can be restricted in `smb.conf`:

```
bind interfaces = true
interfaces = 192.168.1. 127.
hosts allow = 192.168.1. 127.
```

  This will only open, and allow, connections from localhost (127.0.0.1), and the local LAN address range. Adjust the LAN address as needed.

- The CUPS print daemon can be told where to listen for connections. Add to `/etc/cups/cupsd.conf`:

```
Listen 192.168.1.1:631
```

  This will only open a port at the specified address and port number.

- xinetd can force daemons to listen only on a specified address with its "bind" configuration directive. For instance, an internal LAN interface address. See `man xinetd.conf` for this and other syntax. There are various other control mechanisms as well.

As always, anytime you make system changes, backup the configuration file first, restart the appropriate daemon afterward, and then check the appropriate logs for error messages.

5.7. Verifying

The final step after getting your firewall in place, is to verify that it is doing what you intended. You would be wise to do this anytime you make even minor changes to your system configuration.

So how to do this? There are several things you can do.

For our packet filters like ipchains and iptables, we can list all our rules, chains, and associated activity with `iptables −nvL | less` (substitute **ipchains** if appropriate). Open your xterm as wide as possible to avoid wrapping long lines.

This should give you an idea if your chains are doing what you think they should. You may want to perform some of the on−line tasks you normally do first: open a few web pages, send and retrieve mail, etc. This will, of course, not give you any information on tcpwrappers or portsentry. **tcpdchk** can be used to verify tcpwrappers configuration (except with xinetd).

And then, scan yourself. nmap is the scanning tool of choice and is included with recent Red Hat releases, or from http://www.insecure.org/nmap/nmap_download.html. nmap is very flexible, and essentially is a "port prober". In other words, it looks for open ports, among other things. See the nmap man page for details.

If you do run nmap against yourself (e.g. `nmap localhost`), this should tell you what ports are open −− and *visible locally* only! Which hopefully by now, is quite different from what can be seen from the outside. So, scan yourself, and then find a trusted friend, or site (see the Links section), to scan you from the outside. Make sure you are not violating your ISPs Terms of Service by port scanning. It may not be allowed, even if the intentions are honorable. Scanning from outside is the best way to know how the rest of the world sees you. This should tell you how well that firewall is working. See the nmap section in the Appendix for some examples on nmap usage.

One caveat on this: some ISPs may filter some ports, and you will not know for sure how well your firewall is working. Conversely, they make it look like certain ports are open by using web, or other, proxies. The scanner may see the web proxy at port 80 and mis−report it as an open port on your system.

Another option is to find a website that offers *full range* testing. http://www.hackerwhacker.com is one such site. Make sure that any such site is not just scanning a relatively few well known ports.

Repeat this procedure with every firewall change, every system upgrade or new install, and when any key components of your system changes.

You may also want to enable logging all the denied traffic. At least temporarily. Once the firewall is verified to be doing what you think it should, and if the logs are hopelessly overwhelming, you may want to disable logging.

If relying on portsentry at all, please read the documentation. Depending on your configuration it will either drop the route to the scanner, or implement a ipchains/iptables rule doing the same thing. Also, since it "listens" on the specified ports, all those ports will show as "open". A false alarm in this case.

5.8. Logging

Linux does a lot of logging. Usually to more than one file. It is not always obvious what to make of all these entries −− good, bad or indifferent? Firewall logs tend to generate a fair amount of each. Of course, you are wanting to stop only the "bad", but you will undoubtedly catch some harmless traffic as well. The 'net has a lot of background noise.

In many cases, knowing the intentions of an incoming packet are almost impossible. Attempted intrusion? Misbehaved protocol? Mis−typed IP address? Conclusions can be drawn based on factors such as destination port, source port, protocol, and many other variables. But there is no substitute for experience in interpreting firewall logs. It is a black art in many cases.

So do we really need to log? And how much should we be trying to log? Logging is good in that it tells us that the firewall is functional. Even if we don't understand much of it, we know it is doing "something". And if we have to, we can dig into those logs and find whatever data might be called for.

On the other hand, logging can be bad if it is so excessive, it is difficult to find pertinent data, or worse, fills up a partition. Or if we over re−act and take every last entry as an all out assault. Some perspective is a great benefit, but something that new users lack almost by definition. Again, once your firewall is verified, and you are perplexed or overwhelmed, home desktop users may want to disable as much logging as possible. Anyone with greater responsibilities should log, and then find ways to extract the pertinent data from the logs by filtering out extraneous information.

Not sure where to look for log data? The two logs to keep an eye on are `/var/log/messages` and `/var/log/secure`. There may be other application specific logs, depending on what you have installed, or using. FTP, for instance, logs to `/var/log/xfer` on Red Hat.

Portsentry and tcpwrappers do a certain amount of logging that is not adjustable. xinetd has logging enhancements that can be turned on. Both ipchains and iptables, on the other hand, are very flexible as to what is logged.

For ipchains the `−l` option can be added to any rule. iptables uses the `−j LOG` target, and requires its own, separate rule instead. iptables goes a few steps further and allows customized log entries, and rate limiting. See the man page. Presumably, we are more interested in logging blocked traffic, so we'd confine logging to only our `DENY` and `REJECT` rules.

So whether you log, and how much you log, and what you do with the logs, is an individual decision, and probably will require some trial and error so that it is manageable. A few auditing and analytical tools can be quite helpful:

Some tools that will monitor your logs for you and notify you when necessary. These likely will require some configuration, and trial and error, to make the most out of them:

- A nice log entry analyzer for ipchains and iptables from Manfred Bartz: http://www.logi.cc/linux/NetfilterLogAnalyzer.php3. What does all that stuff mean anyway?
- LogSentry (formerly logcheck) is available from http://www.psionic.org/products/logsentry.html, the same group that is responsible for portsentry. LogSentry is an all purpose log monitoring tool with a flexible configuration, that handles multiple logs.
- http://freshmeat.net/projects/firelogd/, the Firewall Log Daemon from Ian Jones, is designed to watch, and send alerts on iptables or ipchains logs data.

* http://freshmeat.net/projects/fwlogwatch/ by Boris Wesslowski, is a similar idea, but supports more log formats.

5.9. Where to Start

Let's take a quick look at where to run our firewall scripts from.

Portsentry can be run as an init process, like other system services. It is not so important when this is done. Tcpwrappers will be automatically be invoked by inetd or xinetd, so not to worry there either.

But the packet filtering scripts will have to be started somewhere. And many scripts will have logic that uses the local IP address. This will mean that the script must be started after the interface has come up and been assigned an IP address. Ideally, this should be immediately after the interface is up. So this depends on how you connect to the Internet. Also, for protocols like PPP or DHCP that may be dynamic, and get different IP's on each re−connect, it is best to have the scripts run by the appropriate daemon.

Red Hat uses `/etc/ppp/ip-up.local` for any user defined, local PPP configuration. If this file does not exist, create it, and make it executable (`chmod +x`). Then with your text editor, add a reference to your firewall script.

For DHCP, it depends on which client. **dhcpcd** will execute **/etc/dhcpcd/dhcpcd−<interface>.exe** (e.g. dhcpcd−eth0.exe) whenever a lease is obtained or renewed. So this is where to put a reference to your firewall script. For **pump** (the default on Red Hat), the main configuration file is `/etc/pump.conf`. **Pump** will run whatever script is defined by the "script" statement any time there is a new or renewed lease:

```
script /usr/local/bin/ipchains.sh
```

If you have a static IP address (i.e. it never changes), the placement is not so important and should be *before* the interface comes up!

5.10. Summary and Conclusions for Step 3

In this section we looked at various components that might be used to construct a "firewall". And learned that a firewall is as much a strategy and combination of components, as it is any one particular application or component. We looked at a few of the most commonly available applications that can be found on most, if not all, Linux systems. This is not a definitive list.

This is a lot of information to digest at all at one time and expect anyone to understand it all. Hopefully this can used as a starting point, and used for future reference as well. The packet filter firewall examples can be used as starting points as well. Just use your text editor, cut and paste into a file with an appropriate name, and then run `chmod +x` against it to make it executable. Some minor editing of the variables may be necessary. Also look at the Links section for sites and utilities that can be used to generate a custom script. This may be a little less daunting.

Now we are done with Steps 1, 2 and 3. Hopefully by now you have already instituted some basic measures to protect your system(s) from the various and sundry threats that lurk on networks. If you haven't implemented any of the above steps yet, now is a good time to take a break, go back to the top, and have at it. The most important steps are the ones above.

A few quick conclusions...

"What is best iptables, ipchains, tcpwrappers, or portsentry?" The quick answer is that iptables can do more than any of the others. So if you are using a 2.4 kernel, use iptables. Then, ipchains if using a 2.2 kernel. The long answer is "it just depends on what you are doing and what the objective is". Sorry. The other tools all have some merit in any given situation, and all can be effective in the right situation.

"Do I really need all these packages?" No, but please combine more than one approach, and please follow all the above recommendations. iptables by itself is good, but in conjunction with some of the other approaches, we are even stronger. Do not rely on any single mechanism to provide a security blanket. "Layers" of protection is always best. As is sound administrative practices. The best iptables script in the world is but one piece of the puzzle, and should not be used to hide other system weaknesses.

"If I have a small home LAN, do I need to have a firewall on each computer?" No, not necessary as long as the LAN gateway has a properly configured firewall. Unwanted traffic should be stopped at that point. And as long as this is working as intended, there should be no unwanted traffic on the LAN. But, by the same token, doing this certainly does no harm. And on larger LANs that might be mixed platform, or with untrusted users, it would be advisable.

---

6. Intrusion Detection

This section will deal with how to get early warning, how to be alerted after the fact, and how to clean up from intrusion attempts.

6.1. Intrusion Detection Systems (IDS)

Intrusion Detection Systems (IDS for short) are designed to catch what might have gotten past the firewall. They can either be designed to catch an active break−in attempt in progress, or to detect a successful break−in after the fact. In the latter case, it is too late to prevent any damage, but at least we have early awareness of a problem. There are two basic types of IDS: those protecting networks, and those protecting individual hosts.

For host based IDS, this is done with utilities that monitor the filesystem for changes. System files that have changed in some way, but should not change −− unless we did it −− are a dead give away that something is amiss. Anyone who gets in, and gets root, will presumably make changes to the system somewhere. This is usually the very first thing done. Either so he can get back in through a backdoor, or to launch an attack against someone else. In which case, he has to change or add files to the system.

This is where tools like tripwire (http://www.tripwire.org) play a role. Tripwire is included beginning with Red Hat 7.0. Such tools monitor various aspects of the filesystem, and compare them against a stored database. And can be configured to send an alert if *any* changes are detected. Such tools should only be installed on a known "clean" system.

For home desktops and home LANs, this is probably not an absolutely necessary component of an overall security strategy. But it does give peace of mind, and certainly does have its place. So as to priorities, make sure the Steps 1, 2 and 3 above are implemented and verified to be sound, before delving into this.

We can get somewhat the same results with `rpm −Va`, which will verify all packages, but without all the same functionality. For instance, it will not notice new files added to most directories. For this to be helpful, it needs to be done after a clean install, and then each time any packages are upgraded or added. Example:

```
# rpm −Va > /root/system.checked
```

Then we have a stored system snapshot that we can refer back to.

Another idea is to run **chkrootkit** (http://www.chkrootkit.org/) as a weekly cron job. This will detect common "rootkits".

6.2. Have I Been Hacked?

Maybe you are reading this because you've noticed something "odd" about your system, and are suspicious that someone was gotten in? This can be a clue.

The first thing an intruder typically does is install a "rootkit". There are many prepackaged rootkits available on the Internet. The rootkit is essentially a script, or set of scripts, that makes quick work of modifying the system so the intruder is in control, and he is well hidden. He does this by installing modified binaries of common system utilities and tampering with log files. Or by using special kernel modules that achieve similar results. So common commands like **ls** may be modified so as to not show where he has his files stored. Clever!

A well designed rootkit can be quite effective. Nothing on the system can really be trusted to provide accurate feedback. Nothing! But sometimes the modifications are not as smooth as intended and give hints that something is not right. Some things that *might* be warning signs:

- **Login** acts weird. Maybe no one can login. Or only root can login. Any **login** weirdness at all should be suspicious. Similarly, any weirdness with adding or changing passwords.
- System utilities are slower, or awkward, or show strange and unexpected results. Common utilities that might be modified are: **ls**, **find**, **who**, **w**, **last**, **netstat**, **login**, **ps**, **top**. This is not a definitive list!
- Files or directories named "..." or ".. " (dot dot space). A sure bet in this case. Files with haxor looking names like "r00t−something".
- Unexplained bandwidth usage.
- Logs that are missing completely, or missing large sections. Or a sudden change in **syslog** behavior.
- Mysterious open ports, or processes.
- Files that cannot be deleted or moved. Some rootkits use **chattr** to make files "immutable", or not changable. This kind of change will not show up with **ls**, so the files look normal at first glance. See the man pages for **chattr** and **lsattr** on how to reverse this. Then see the next section below on restoring your system as the jig is up at this point.
- Indications of a "sniffer", such as log messages of an interface entering "promiscuous" mode.
- Modifications to `/etc/inetd.conf`, or `/etc/passwd`. Especially, any additions. Try using **cat** or **tail** to view these files. Additions will most likely be appended to the end. Remember though such changes may not be "visible" to any system tools.

Sometimes the intruder is not so smart and forgets about root's `.bash_history`, or cleaning up log entries, or even leaves strange, leftover files in `/tmp`. So these should always be checked. Just don't necessarily expect them to be accurate.

Packet sniffers, like tcpdump (http://www.tcpdump.org), might be useful in finding any uninvited traffic. Interpreting sniffer output is probably beyond the average new user. snort (http://www.snort.org), and ethereal (http://www.ethereal.com), are also good. Ethereal has a GUI.

As mentioned, a compromised system will undoubtedly have altered system binaries, and the output of system utilities is not to be trusted. Nothing on the system can be relied upon to be telling you the truth. Re−installing individual packages may or may not help since it could be system libraries or kernel modules that are doing the dirty work.

We can use `rpm −Va |less` to attempt to verify the integrity all packages. But again there is no assurance that rpm itself has not been tampered with, or the system components that RPM relies on.

If you have **pstree** on your system, try this instead of the standard **ps**. Sometimes the script kiddies forget about this one. No guarantees though that this is accurate either.

You can also try querying the `/proc` filesystem, which contains everything the kernel knows about processes that are running:

```
# cat /proc/*/stat | awk '{print $1,$2}'
```

This will provide a list of all processes and PIDs numbers (assuming a malicious kernel module is not hiding this).

Another approach is to visit http://www.chkrootkit.org, download their rootkit checker, and see what it says.

Some interesting discussions on issues surrounding forensics can be found at http://www.fish.com/security/. There is also a collection of tools available, aptly called "The Coroner's Toolkit" (TCT).

Read below for steps on recovering from an intrusion.

6.3. Reclaiming a Compromised System

So now you've confirmed a break−in, and know that someone else has root access, and quite likely one or more hidden backdoors to your system. You've lost control. How to clean up and regain control?

There is no sure fire way of doing this short of a complete re−install. There is no way to find with assurance all the modified files and backdoors that may have been left. Trying to patch up a compromised system risks a false sense of security and may actually aggravate an already bad situation.

The steps to take, in this order:

- Pull the plug and disconnect the machine. You may be unwittingly participating in criminal activity, and doing to others what has been done to you.
- Depending on the needs of the situation and time available to restore the system, it is advantageous to learn as much as you can about how the attacker got in, and what was done in order to plug the hole and avoid a recurrence. This could conceivably be time consuming, and is not always feasible. And it may require more expertise than the typical user possesses.
- Backup important data. Do *not* include any system files in the backup, and system configuration files like `inetd.conf`. Limit the backup to personal data files only! You don't want to backup, then restore something that might open a backdoor or other hole.
- Re−install from scratch, and reformat the drive during the installation (**mke2fs**) to make sure no remnants are hiding.
- Restore from backups. After a clean install is the best time to install an IDS (Intrusion Detection System) such as tripwire (http://www.tripewire.org).
- Apply all updates from ftp://updates.redhat.com.
- Re−examine your system for unnecessary services. Re−examine your firewall and access policies, and tighten all holes. *Use new passwords*, as these were stolen in all likelihood.
- Re−connect system ;−)

At this time, any rootkit cleanup tools that may be available on−line are not recommended. They probably do work just fine most of the time. But again, how to be absolutely sure that all is well and all vestiges of the intrusion are gone?

7. General Tips

This section will quickly address some general concepts for maintaining a more secure and reliable system or network. Let's emphasize "maintaining" here since computer systems change daily, as does the environment around them. As mentioned before, there isn't any one thing that makes a system secure. There are too many variables. Security is an approach and an attitude more than it is a reliance on any particular product, application or specific policy.

* Do not allow remote root logins. This may be controlled by a configuration file such as `/etc/securetty`. Remove any lines that begin "pts". This is one big security hole.
* In fact, don't log in as root at all. Period. Log in on your user account and **su** to root when needed. Whether the login is remote or local. Or use **sudo**, which can run individual commands with root privileges. (Red hat includes a **sudo** package.) This takes some getting used to, but it is the "right" way to do things. And the safest. And will become more a more natural way of doing this as time goes on.

  I know someone is saying right now "but that is so much trouble, I am root, and it is my system". True, but root is a specialized account that was not ever meant to be used as a regular user account. Root has access to everything, even hardware devices. The system "trusts" root. It believes that you know what you are doing. If you make a mistook, it assumes that you meant that, and will do it's best to do what you told it to do...even if that destroys the system!

  As an example, let's say you start X as root, open Netscape, and visit a web site. The web page has badly behaved java script. And conceivably now that badly written java script might have access to much more of your system than if you had done it the "right" way.

* Take passwords seriously. Don't give them out to anyone. Don't use the same one for everything. Don't use root's password for anything else −− except root's password! Never sign up or register on line, using any of your system passwords. Passwords should be a combination of mixed case letters, numbers and/or punctuation and a reasonable length (eight characters or longer). Don't use so−called "dictionary" words that are easy to guess like "cat" or "dog". Don't incorporate personal information like names or dates or hostnames. Don't write down system passwords −− memorize them.

  Use the more secure "shadow" passwords. This has been the default on Red Hat for some time now. If the file `/etc/shadow` exists, then it is enabled already. The commands **pwconv** and **grpconv**, can be used to convert password and group files to shadow format if available.

* Avoid using programs that require clear text logins over untrusted networks like the Internet. **Telnet** is a prime example. **ssh** is much better. If there is any support for SSL (Secure Socket Layers), use it. For instance, does your ISP offer POP or IMAP mail via SSL? Recent Red Hat releases do include openssl, and many Linux applications can use SSL where support is available.
* Set resource limits. There are various ways to do this. The need for this probably increases with the number of users accessing a given system. Not only does setting limits on such things as disk space prevent intentional mischief, it can also help with unintentionally misbehaved applications or processes. **quota** (`man quota`) can be used to set disk space limits. Bash includes the **ulimit** command (`man ulimit` or `man bash`), that can limit various functions on a per user basis.

  Also, not discussed here at any length, but PAM (Pluggable Authentication Modules) has a very sophisticated approach to controlling various system functions and resources. See `man pam` to get started. PAM is configured via either `/etc/pam.conf` or `/etc/pam.d/*`. Also files in

`/etc/security/*`, including `/etc/security/limits.conf`, where again various sane limits can be imposed. An in depth look at PAM is beyond the scope of this document. The User−Authentication HOWTO (http://linuxdoc.org/HOWTO/User−Authentication−HOWTO/index.html) has more on this.

* Make sure someone with a clue is getting root's mail. This can be done with an "alias". Typically, the mail server will have a file such as `/etc/aliases` where this can defined. This can conceivably be an account on another machine if need be:

```
# Person who should get root's mail.  This alias
# must exist.
# CHANGE THIS LINE to an account of a HUMAN
root:           hal@bigcat
```

  Remember to run **newaliases** (or equivalent) afterward.

* Be careful where you get software. Use trusted sources. How well do you trust complete strangers? Check Red Hat's ftp site (or mirrors) first if looking for a specific package. It will probably be best suited for your system any way. Or, the original package's project site is good as well. Installing from raw source (either tarball or src.rpm) at least gives you the ability to examine the code. Even if you don't understand it ;−) While this does not seem to be a wide spread problem with Linux software sites, it is very trivial for someone to add a very few lines of code, turning that harmless looking binary into a "Trojan horse" that opens a backdoor to your system. Then the jig is up.
* So someone has scanned you, probed you, or otherwise seems to want into your system? Don't retaliate. There is a good chance that the source IP address is a compromised system, and the owner is a victim already. Also, you may be violating someone's Terms of Service, and have trouble with your own ISP. The best you can do is to send your logs to the abuse department of the source IP's ISP, or owner. This is often something like "abuse@someisp.com". Just don't expect to hear much back. Generally speaking, such activity is not legally criminal, unless an actual break−in has taken place. Furthermore, even if criminal, it will never be prosecuted unless significant damage (read: big dollars) can be shown.
* Red Hat users can install the "Bastille Hardening System", http://www.bastille−linux.org/. This is a multi−purpose system for "hardening" Red Hat and Mandrake system security. It has a GUI interface which can be used to construct firewall scripts from scratch and configure PAM among many other things. Debian support is new.
* So you have a full−time Internet connection via cable−modem or DSL. But do you always use it, or always need it? There's an old saying that "the only truly secure system, is a disconnected system". Well, that's certainly one option. So take that interface down, or stop the controlling daemon (**dhcpcd**, **pppoed**, etc). Or possibly even set up cron jobs to bring your connection up and down according to your normal schedule and usage.
* What about cable and DSL routers that are often promoted as "firewalls"? The lower priced units are mostly equating NAT (Network Address Translation), together with the ability to open holes for ports through it, as a firewall. While NAT itself does provide a fair degree of security for the systems behind the NAT gateway, this does not constitute anything but a very rudimentary firewall. And if holes are opened, there is still exposure. Also, you are relying on the router's firmware and implementation not to be flawed. It is wise to have some kind of additional protection behind such routers.
* What about wireless network cards and hubs? Insecure, despite what the manufacturers may claim. Treat these connections just as you would an Internet connection. Use secure protocols like ssh only!

Even if it is just one LAN box to another.

- If you find you need to run a particular service, and it is for just you, or maybe a relatively small number of people, use a non−standard port. Most server daemons support this. For instance, **sshd** runs on port 22 by default. All worms and script kiddies will expect it there, and look for it there. So, run it on another port! See the **sshd** man page.
- Lastly, know your system! Let's face it, if you are new to Linux, you can't already know something you have never used. Understood. But in the process of learning, learn how to do things the right way, not the easiest way. There is several decades of history behind "the right way" of doing things. This has stood the test of time. What may seem unnecessary or burdensome now, will make sense in due time.

  Be familiar with whatever services you are running, and the implications these services might have to the overall health of your system if something does go wrong. Read what you can, and ask questions. Don't run something as a service "just because I can", or because the installer put it there. You can't start out being an experienced System Administrator clearly. But you can work to learn enough about your own system, that you are in control. This is one thing that separates *nix from MS systems: we can never be in complete control with MS, but we can with *nix. Conversely, if something bad happens, we often have no one else to blame.

8. Appendix

8.1. Servers, Ports, and Packets

Let's take a quick, non−technical look at some networking concepts, and how they can potentially impact our own security. We don't need to know much about networking, but a general idea of how things work is certainly going to help us with firewalls and other related issues.

As you may have noticed Linux is a very network oriented Operating System. Much is done by connecting to "servers" of one type or another −− X servers, font servers, print servers, etc.

Servers provide "services", which provide various capabilities, both to the local system and potentially other remote systems. The same server generally provides both functionalities. Some servers work quietly behind the scenes, and others are more interactive by nature. We may only be aware of a print server when we need to print something, but it is there running, listening, and waiting for connection requests whether we ever use it or not (assuming of course we have it enabled). A typical Linux installation will have many, many types of servers available to it. Default installations often will turn some of these "on".

And even if we are not connected to a real network all the time, we are still "networked" so to speak. Take our friendly local X server for instance. We may tend to think of this as just providing a GUI interface, which is only true to a point. It does this by "serving" to client applications, and thus is truly a server. But X Windows is also capable of serving remote clients over a network −− even large networks like the Internet. Though we probably don't really want to be doing this ;−)

And yes, if you are not running a firewall or have not taken other precautions, and are connected to the Internet, it is quite possible that someone −− anyone −− could connect to your X server. X11 "listens" on TCP "port" 6000 by default. This principle applies to most other servers as well −− they can be easily connected to, unless something is done to restrict or prevent connections.

In TCP/IP (Transmission Control Protocol/Internet Protocol) networks like we are talking about with Linux and the Internet, every connected computer has a unique "IP Address". Think of this like a phone number. You have a phone number, and in order to call someone else, you have to know that phone number, and then dial it. The phone numbers have to be unique for the system to work. IP address are generally expressed as "dotted quad" notation, e.g. 152.19.254.81.

On this type of network, servers are said to "listen". This means that they have a "port" opened, and are awaiting incoming connections to that port. Connections may be local, as is typically the case with our X server, or remote −− meaning from another computer "somewhere". So servers "listen" on a specific "port" for incoming connections. Most servers have a default port, such as port 80 for web servers. Or 6000 for X11. See `/etc/services` for a list of common ports and their associated service.

The "port" is actually just an address in the kernel's networking stack, and is a method that TCP, and other protocols, use to organize connections and the exchange of data between computers. There are total of 65,536 TCP and UDP ports available, though usually only a relatively few of these are used at any one time. These are classified as "privileged", those ports below 1024, and "unprivileged", 1024 and above. Most servers use the privileged ports.

Only one server may listen on, or "bind" to, a port at a time. Though that server may well be able to open multiple connections via that one port. Computers talk to each other via these "port" connections. One computer will open a connection to a "port" on another computer, and thus be able to exchange data via the connection that has been established between their respective ports.

Getting back to the phone analogy, and stretching it a bit, think of calling a large organization with a complex phone system. The organization has many "departments": sales, shipping, billing, receiving, customer service, R&D, etc. Each department has it's own "extension" number. So the shipping department might be extension 21, the sales might be department 80 and so on. The main phone number is the IP Address, and the department's extension is the port in this analogy. The "department's" number is always the same when we call. And generally they can handle many simultaneous incoming calls.

The data itself is contained in "packets", which are small chunks of data, generally 1500 bytes or less each. Packets are used to control and organize the connection, as well as carry data. There are different types of packets. Some are specifically used for controlling the connection, and then some packets carry our data as their payload. If there is a lot of data, it will be broken up into multiple packets which is almost always how it works. The packets will be transmitted one at a time, and then "re−assembled" at the other end. One web page for instance, will take many packets to transmit −− maybe hundreds or even thousands. This all happens very quickly and transparently.

We can see a typical connection between two computers in this one line excerpt from **netstat** output:

```
tcp    30    0 169.254.179.139:1359    18.29.1.67:21      CLOSE_WAIT
```

The interesting part is the IP addresses and ports in the fourth and fifth columns. The port is the number just to the right of the colon. The left side of the colon is the IP address of each computer. The fourth column is the local address, or our end of the connection. In the example, 169.254.179.139 is the IP address assigned by my ISP. It is connected to port 21 (FTP) on 18.29.1.67, which is rpmfind.net. This is just after an FTP download from rpmfind.net. Note that while I am connected to their FTP server on their port 21, the port on my end that is used by my FTP client is 1359. This is a randomly assigned "unprivileged" port, used for my end of the two−way "conversation". The data moves in both directions: me:port#1359 <−> them:port#21. The FTP protocol is actually a little more complicated than this, but we won't delve into the finer points here. The `CLOSE_WAIT` is the TCP state of the connection at this particular point in time. Eventually the connection will close completely on both ends, and netstat will not show anything for this.

The "unprivileged" port that is used for my end of the connection, is temporary and is not associated with a locally running server. It will be closed by the kernel when the connection is terminated. This is quite different than the ports that are kept open by "listening" servers, which are permanent and remain "open" even after a remote connection is terminated.

So to summarize using the above example, we have client (me) connecting to a server (rpmfind.net), and the connection is defined and controlled by the respective ports on either end. The data is transmitted and controlled by packets. The server is using a "privileged" port (i.e. a port below number 1024) which stays open listening for connections. The "unprivileged" port used on my end by my client application is temporary, is only opened for the duration of the connection, and only responds to the server's port at the other end of the connection. This type of port is not vulnerable to attacks or break−ins generally speaking. The server's port is vulnerable since it remains open. The administrator of the FTP server will need to take appropriate precautions that his server is secure. Other Internet connections, such as to web servers or mail servers, work similar to the above example, though the server ports are different. SMTP mail servers use port 25, and web servers typically use port 80. See the Ports section for other commonly used ports and services.

One more point on ports: ports are only accessible if there is something listening on that port. No one can force a port open if there is no service or daemon listening there, ready to handle incoming connection requests. A closed port is a totally safe port.

And a final point on the distinction between clients and servers. The example above did not have a **telnet** or **ftp** server in the `LISTENER` section in the **netstat** example above. In other words, no such servers were running locally. You do not need to run a **telnet** or **ftp** server daemon in order to connect to *somebody else's* **telnet** or **ftp** server. These are only for providing these services to others that would be making connections to you. Which you don't really want to be doing in most cases. This in no way effects the ability to use **telnet** and **ftp** client software.

8.2. Common Ports

A quick run down of some commonly seen and used ports, with the commonly associated service name, and risk factor. All have *some* risk. It is just that some have historically had more exploits than others. That is how they are evaluated below, and not necessarily to be interpreted as whether any given service is safe or not.

1−19, assorted protocols, many of which are antiquated, and probably none of which are needed on a modern system. If you don't know what any of these are, then you definitely don't need them. The echo service (port 7) should not be confused with the common **ping** program. Leave all these off.

20 − FTP−DATA. "Active" FTP connections use two ports: 21 is the control port, and 20 is where the data comes through. Passive FTP does not use port 20 at all. Low risk, but see below.

21 − FTP server port, aka File Transfer Protocol. A well entrenched protocol for transferring files between systems. Very high risk, and maybe the number one crack target.

22 − SSH (Secure Shell), or sometimes PCAnywhere. Low to moderate risk (yes there are exploits even against so called "secure" services).

23 − Telnet server. For LAN use only. Use ssh instead in non−secure environments. Moderate risk.

25 − SMTP, Simple Mail Transfer Protocol, or mail server port, used for sending outgoing mail, and transferring mail from one place to another. Moderate risk. This has had a bad history of exploits, but has improved lately.

37 − Time service. This is the built−in inetd time service. Low risk. For LAN use only.

53 − DNS, or Domain Name Server port. Name servers listen on this port, and answer queries for resolving host names to IP addresses. High Risk.

67 (UDP) − BOOTP, or DHCP, server port. Low risk. If using DHCP on your LAN, this does not need to be exposed to the Internet.

68 (UDP) − BOOTP, or DHCP, client port. Low risk.

69 – tftp, or Trivial File Transfer Protocol. Extremely insecure. LAN only, if really, really needed.

79 – Finger, used to provide information about the system, and logged in users. Low risk as a crack target, but gives out way too much information and should not be run.

80 – WWW or HTTP standard web server port. The most commonly used service on the Internet. Low risk.

98 – Linuxconf web access administrative port. LAN only, if really needed at all.

110 – POP3, aka Post Office Protocol, mail server port. POP mail is mail that the user retrieves from a remote system. Low risk.

111 – sunrpc (Sun Remote Procedure Call), or portmapper port. Used by NFS (Network File System), NIS (Network Information Service), and various related services. Sounds dangerous and is high risk. LAN use only. A favorite crack target.

113 – identd, or auth, server port. Used, and sometimes required, by some older style services (like SMTP and IRC) to validate the connection. Probably not needed in most cases. Low risk, but could give an attacker too much information about your system.

119 –– nntp or news server port. Low risk.

123 – Network Time Protocol for synchronizing with time servers where a high degree of accuracy is required. Low risk, but probably not required for most users. rdate makes an easier and more secure way of updating the system clock. And then inetd's built in time service for synchronizing LAN systems is another option.

137–139 – NetBios (SMB) services. Mostly a Windows thing. Low risk on Linux, but LAN use only. 137 is a very commonly seen port attempt. A rather obnoxious protocol from Redmond that generates a lot of "noise", much of which is harmless.

143 – IMAP, Interim Mail Access Protocol. Another mail retrieval protocol. Low to moderate risk.

161 – SNMP, Simple Network Management Protocol. More commonly used in routers and switches to monitor statistics and vital signs. Not needed for most of us, and low risk.

177 – XDMCP, the X Display Management Control Protocol for remote connections to X servers. Low risk, but LAN only is recommended.

443 – HTTPS, a secure HTTP (WWW) protocol in fairly wide use. Low risk.

465 – SMTP over SSL, secure mail server protocol. Low risk.

512 (TCP) – exec is how it shows in netstat. Actually the proper name is **rexec,** for Remote Execution. Sounds dangerous, and is. High risk, LAN only if at all.

512 (UDP) – biff, a mail notification protocol. Low risk, LAN only.

513 − login, actually **rlogin**, aka Remote Login. No relation to the standard **/bin/login** that we use every time we log in. Sounds dangerous, and is. High risk, and LAN only if really needed.

514 (TCP) − shell is the nickname, and how netstat shows it. Actually, **rsh** is the application for "Remote Shell". Like all the "r" commands, this is a throw back to kindler, gentler times. Very insecure, so high risk, and LAN only usage, if at all.

514 (UDP) − syslog daemon port, only used for remote logging purposes. The average user does not need this. Probably low risk, but definitely LAN only if really required.

515 − lp or print server port. High risk, and LAN only of course. Someone on the other side of the world does not want to use your printer for it's intended purpose!

587 − MSA, or "submission", the Mail Submission Agent protocol. A new mail handling protocol supported by most MTA's (mail servers). Low risk.

631 − the CUPS (print daemon) web management port. LAN only, low risk.

635 − mountd, part of NFS. LAN use only.

901 − SWAT, Samba Web Administration Tool port. LAN only.

993 − IMAP over SSL, secure IMAP mail service. Very low risk.

995 − POP over SSL, secure POP mail service. Very low risk.

1024 − This is the first "unprivileged" port, which is dynamically assigned by the kernel to whatever application requests it. This can be almost anything. Ditto for ports just above this.

1080 − Socks Proxy server. A favorite crack target.

1243 − SubSeven Trojan. Windows only problem.

1433 − MS SQL server port. A sometimes target. N/A on Linux.

2049 − nfsd, Network File Service Daemon port. High risk, and LAN usage only is recommended.

3128 − Squid proxy server port. Low risk, but for most should be LAN only.

5432 − Postgres SQL server port. LAN only, relatively low risk.

5631 (TCP), 5632 (UDP) − PCAnywhere ports. Windows only. PCAnywhere can be quite "noisy", and broadcast wide address ranges.

6000 − X11 TCP port for remote connections. Low to moderate risk, but again, this should be LAN only. Actually, this can include ports 6000−6009 since X can support multiple displays and each display would have its own port. ssh's X11Forwarding will start using ports at 6010.

6346 – gnutella.

6667 – ircd, Internet Relay Chat Daemon.

6699 – napster.

7100–7101 – Some font servers use these ports. Low risk, but LAN only.

8000 and 8080 – common web cache and proxy server ports. LAN only.

10000 – webmin, a web based system administration utility. Low risk at this point.

27374 – SubSeven, a commonly probed for Windows only Trojan. Also, 1243.

31337 – Back Orifice, another commonly probed for Windows only Trojan.

More services and corresponding port numbers can be found in `/etc/services`. Also, the "official" list is http://www.iana.org/assignments/port–numbers.

A great analysis of what probes to these and other ports might mean from Robert Graham: http://www.linuxsecurity.com/resource_files/firewalls/firewall–seen.html. A very good reference.

Another point here, these are the *standard* port designations. There is no law that says any service has to run on a specific port. Usually they do, but certainly they don't always have to.

Just a reminder that when you see these types of ports in your firewall logs, it is not anything to go off the deep end about. Not if you have followed Steps 1–3 above, and verified your firewall works. You are fairly safe. Much of this traffic may be "stray bullets" too –– Internet background noise, misconfigured clients or routers, noisy Windows stuff, etc.

8.3. Netstat Tutorial

8.3.1. Overview

netstat is a very useful utility for viewing the current state of your network status –– what servers are listening for incoming connections, what interfaces they listen on, who is connected to us, who we are connect to, and so on. Take a look at the man page for some of the many command line options. We'll just use a relative few options here.

As an example, let's check all currently listening servers and active connections for both TCP and UDP on our hypothetical host, bigcat. bigcat is a home desktop installation, with a DSL Internet connection in this example. bigcat has two ethernet cards: one for the external connection to the ISP, and one for a small LAN with an address of 192.168.1.1.

```
$ netstat -tua
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:printer               *:*                     LISTEN
tcp        0      0 bigcat:8000             *:*                     LISTEN
tcp        0      0 *:time                  *:*                     LISTEN
tcp        0      0 *:x11                   *:*                     LISTEN
tcp        0      0 *:http                  *:*                     LISTEN
tcp        0      0 bigcat:domain           *:*                     LISTEN
tcp        0      0 bigcat:domain           *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 *:631                   *:*                     LISTEN
tcp        0      0 *:smtp                  *:*                     LISTEN
tcp        0      1 dsl-78-199-139.s:1174   64.152.100.93:nntp      SYN_SENT
tcp        0      1 dsl-78-199-139.s:1175   64.152.100.93:nntp      SYN_SENT
tcp        0      1 dsl-78-199-139.s:1173   64.152.100.93:nntp      SYN_SENT
tcp        0      0 dsl-78-199-139.s:1172   207.153.203.114:http    ESTABLISHED
tcp        1      0 dsl-78-199-139.s:1199   www.xodiax.com:http     CLOSE_WAIT
tcp        0      0 dsl-78-199-139.sd:http  63.236.92.144:34197     TIME_WAIT
tcp      400      0 bigcat:1152             bigcat:8000             CLOSE_WAIT
tcp     6648      0 bigcat:1162             bigcat:8000             CLOSE_WAIT
tcp      553      0 bigcat:1164             bigcat:8000             CLOSE_WAIT
udp        0      0 *:32768                 *:*
udp        0      0 bigcat:domain           *:*
udp        0      0 bigcat:domain           *:*
udp        0      0 *:631                   *:*
```

This output probably looks very different from what you get on your own system. Notice the distinction between "Local Address" and "Foreign Address", and how each includes a corresponding port number (or service name if available) after the colon. "Local Address" is our end of the connection. The first group with `LISTEN` in the far right hand column are services that are running on this system. These are servers that are running in the background on bigcat, and "listen" for incoming connections. So they have a port opened, and this is where they "listen". These connections might come from the local system (i.e. bigcat itself), or remote systems. This is very important information to have! The others just below this are connections that have been established from this system to other systems. The respective connections are in varying states as indicated by the key words in the last column. Those with no key word in the last column at the end are servers responding to UDP connections. UDP is a different protocol from TCP altogether, but is used for some types of low priority network traffic.

Now, the same thing with the "−n" flag to suppress converting to "names" so we can actually see the port numbers:

```
$ netstat -taun
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address      State
tcp        0      0 0.0.0.0:515             0.0.0.0:*            LISTEN
tcp        0      0 127.0.0.1:8000          0.0.0.0:*            LISTEN
tcp        0      0 0.0.0.0:37              0.0.0.0:*            LISTEN
tcp        0      0 0.0.0.0:6000            0.0.0.0:*            LISTEN
tcp        0      0 0.0.0.0:80              0.0.0.0:*            LISTEN
tcp        0      0 192.168.1.1:53          0.0.0.0:*            LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*            LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*            LISTEN
tcp        0      0 0.0.0.0:631             0.0.0.0:*            LISTEN
tcp        0      0 0.0.0.0:25              0.0.0.0:*            LISTEN
tcp        0      1 169.254.179.139:1174    64.152.100.93:119    SYN_SENT
```

```
tcp        0      1 169.254.179.139:1175    64.152.100.93:119    SYN_SENT
tcp        0      1 169.254.179.139:1173    64.152.100.93:119    SYN_SENT
tcp        0      0 169.254.179.139:1172    207.153.203.114:80   ESTABLISHED
tcp        1      0 169.254.179.139:1199    216.26.129.136:80    CLOSE_WAIT
tcp        0      0 169.254.179.139:80      63.236.92.144:34197  TIME_WAIT
tcp      400      0 127.0.0.1:1152          127.0.0.1:8000       CLOSE_WAIT
tcp     6648      0 127.0.0.1:1162          127.0.0.1:8000       CLOSE_WAIT
tcp      553      0 127.0.0.1:1164          127.0.0.1:8000       CLOSE_WAIT
udp        0      0 0.0.0.0:32768           0.0.0.0:*
udp        0      0 192.168.1.1:53          0.0.0.0:*
udp        0      0 127.0.0.1:53            0.0.0.0:*
udp        0      0 0.0.0.0:631             0.0.0.0:*
```

Let's look at the first few lines of this in detail. On line one,

```
 tcp        0      0 0.0.0.0:515            0.0.0.0:*          LISTEN
```

"Local Address" is `0.0.0.0`, meaning "all" interfaces that are available. The local port is 515, or the standard print server port, usually owned by the lpd daemon. You can find a listing of common service names and corresponding ports in the file `/etc/services`.

The fact that it is listening on all interfaces is significant. In this case, that would be lo (localhost), eth0, and eth1. Printer connections could conceivably be made over any of these interfaces. Should a user on this system bring up a PPP connection, then the print daemon would be listening on that interface (ppp0) as well. The "Foreign Address" is also `0.0.0.0`, meaning from "anywhere".

It is also worth noting here, that even though this server is telling the kernel to listen on all interfaces, the **netstat** output does not reflect whether there may be a firewall in place that may be filtering incoming connections. We just can't tell that at this point. Obviously, for certain servers, this is very desirable. Nobody outside your own LAN has any reason whatsoever to connect to your print server port for instance.

Line two is a little different:

```
 tcp        0      0 127.0.0.1:8000         0.0.0.0:*          LISTEN
```

Notice the "Local Address" this time is localhost's address of `127.0.0.1`. This is very significant as only connections local to this machine will be accepted. So only bigcat can connect to bigcat's TCP port 8000. The security implications should be obvious. Not all servers have configuration options that allow this kind of restriction, but it is a very useful feature for those that do. Port 8000 in this example, is owned by the web proxy Junkbuster.

With the next three entries, we are back to listening on all available interfaces:

```
 tcp        0      0 0.0.0.0:37             0.0.0.0:*           LISTEN
 tcp        0      0 0.0.0.0:6000           0.0.0.0:*           LISTEN
 tcp        0      0 0.0.0.0:80             0.0.0.0:*           LISTEN
```

Looking at `/etc/services`, we can tell that port 37 is a "time" service, which is a time server. 6000 is X11, and 80 is the standard port for HTTP servers like Apache. There is nothing really unusual here as these are all readily available services on Linux.

The first two above are definitely not the kind of services you'd want just anyone to connect to. These should be firewalled so that all outside connections are refused. Again, we can't tell from this output whether any firewall is in place, much less how effectively implemented it may be.

The web server on port 80 is not a huge security risk by itself. HTTP is a protocol that is often open to all comers. For instance, if we wanted to host our own home page, Apache can certainly do this for us. It is also possible to firewall this off, so that it is for use only to our LAN clients as part of an Intranet. Obviously too, if you do not have a good justification for running a web server, then it should be disabled completely.

The next two lines are interesting:

```
tcp        0      0 192.168.1.1:53          0.0.0.0:*           LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*           LISTEN
```

Again notice the "Local Address" is not `0.0.0.0`. This is good! The port this time is 53, or the DNS port used by nameserver daemons like **named.** But we see the nameserver daemon is only listening on the lo interface (localhost), and the interface that connects bigcat to the LAN. So the kernel only allows connections from localhost, and the LAN. There will be no port 53 available to outside connections at all. This is a good example of how individual applications can sometimes be securely configured. In this case, we are probably looking at a caching DNS server since a real nameserver that is responsible for handling DNS queries would have to have port 53 open to the world. This is a security risk and requires special handling.

The last three `LISTENER` entries:

```
tcp        0      0 0.0.0.0:22              0.0.0.0:*           LISTEN
tcp        0      0 0.0.0.0:631             0.0.0.0:*           LISTEN
tcp        0      0 0.0.0.0:25              0.0.0.0:*           LISTEN
```

These are back to listening on all available interfaces. Port 22 is sshd, the Secure Shell server daemon. This is a good sign! Notice that the service for port 631 does not have a service name if we look at the output in the first example. This might be a clue that something unusual is going on here. (See the next section for the answer to this riddle.) And lastly, port 25, the standard port for the SMTP mail daemon. Most Linux installations probably will have an SMTP daemon running, so this is not necessarily unusual. But is it necessary?

The next grouping is established connections. For our purposes the state of the connection as indicated by the last column is not so important. This is well explained in the man page.

```
tcp        0      1 169.254.179.139:1174    64.152.100.93:119    SYN_SENT
tcp        0      1 169.254.179.139:1175    64.152.100.93:119    SYN_SENT
tcp        0      1 169.254.179.139:1173    64.152.100.93:119    SYN_SENT
tcp        0      0 169.254.179.139:1172    207.153.203.114:80   ESTABLISHED
tcp        1      0 169.254.179.139:1199    216.26.129.136:80    CLOSE_WAIT
tcp        0      0 169.254.179.139:80      63.236.92.144:34197  TIME_WAIT
tcp      400      0 127.0.0.1:1152          127.0.0.1:8000       CLOSE_WAIT
```

```
tcp     6648      0 127.0.0.1:1162          127.0.0.1:8000       CLOSE_WAIT
tcp      553      0 127.0.0.1:1164          127.0.0.1:8000       CLOSE_WAIT
```

There are nine total connections here. The first three is our external interface connecting to a remote host on their port 119, the standard NNTP (News) port. There are three connections here to the same news server. Apparently the application is multi−threaded, as it is trying to open multiple connections to the news server. The next two entries are connections to a remote web server as indicated by the port 80 after the colon in the fifth column. Probably a pretty common looking entry for most of us. But the one just after is reversed and has the port 80 in the fourth column, so this is someone that has connected to bigcat's web server via its external, Internet−side interface. The last three entries are all connections from localhost to localhost. So we are connecting to ourselves here. Remembering from above that port 8000 is bigcat's web proxy, this is a web browser that is connected to the locally running proxy. The proxy then will open an external connection of its own, which probably is what is going on with lines four and five.

Since we gave **netstat** both the `-t` and `-u` options, we are getting both the TCP and UDP listening servers. The last few lines are the UDP ones:

```
udp        0      0 0.0.0.0:32768          0.0.0.0:*
udp        0      0 192.168.1.1:53         0.0.0.0:*
udp        0      0 127.0.0.1:53           0.0.0.0:*
udp        0      0 0.0.0.0:631            0.0.0.0:*
```

The last three entries have ports that are familiar from the above discussion. These are servers that are listening for both TCP and UDP connections. Same servers in this case, just using two different protocols. The first one on local port 32768 is new, and does not have a service name available to it in `/etc/services`. So at first glance this should be suspicious and pique our curiosity. See the next section for the explanation.

Can we draw any conclusions from this hypothetical situation? For the most part, these look to be pretty normal looking network services and connections for Linux. There does not seem to be an unduly high number of servers running here, but that by itself does not mean much since we don't know if all these servers are really required or not. We know that **netstat** can not tell us if any of these are effectively firewalled, so there is no way to say how secure all this might be. We also don't really know if all the listening services are really required by the owner here. That is something that varies widely from installation to installation. Does bigcat even have a printer attached for instance? Presumably it does, or this is a completely unnecessary risk.

8.3.2. Port and Process Owners

We've learned a lot about what is going on with bigcat's networking from the above section. But suppose we see something we don't recognize and want to know what started that particular service? Or we want to stop a particular server and it is not obvious from the above output?

The `-p` option should give us the process's PID and the program name that started the process in the last column. Let's look at the TCP servers again (with first three columns cropped for spacing). We'll have to run this as root to get all the available information:

```
# netstat −tap
Active Internet connections (servers and established)
  Local Address           Foreign Address      State       PID/Program name
  *:printer               *:*                  LISTEN       988/inetd
  bigcat:8000             *:*                  LISTEN       1064/junkbuster
  *:time                  *:*                  LISTEN       988/inetd
  *:x11                   *:*                  LISTEN       1462/X
  *:http                  *:*                  LISTEN       1078/httpd
  bigcat:domain           *:*                  LISTEN       956/named
  bigcat:domain           *:*                  LISTEN       956/named
  *:ssh                   *:*                  LISTEN       972/sshd
  *:631                   *:*                  LISTEN       1315/cupsd
  *:smtp                  *:*                  LISTEN       1051/master
```

Some of these we already know about. But we see now that the printer daemon on port 515 is being started via **inetd** with a PID of "988". **inetd** is a special situation. **inetd** is often called the "super server", since it's main role is to spawn sub−services. xinetd replaces inetd as of Red Hat 7.0. If we look at the first line, **inetd** is listening on port 515 for printer services. If a connection comes for this port, **inetd** intercepts it, and then will spawn the appropriate daemon, i.e. the print daemon in this case. The configuration of how **inetd** handles this is typically done in `/etc/inetd.conf`. This should tell us that if we want to stop an **inetd** controlled server on a permanent basis, then we will have to dig into the **inetd** (or perhaps **xinetd**) configuration. Also the time service above is started via **inetd** as well. This should also tell us that these two services can be further protected by tcpwrappers (discussed in Step 3 above). This is one benefit of using **inetd** to control certain system services.

We weren't sure about the service on port 631 above since it did not have a standard service name, which means it is something maybe unusual or off the beaten path. Now we see it is owned by **cupsd** (not included with Red Hat by the way), which is one of several print daemons available under Linux. This happens to be the web interface for controlling the printer service. Something **cupsd** does that is indeed a little different than other print servers.

The last entry above is the SMTP mail server on bigcat. Often, this is **sendmail**. But not in this case. The command is "master", which may not ring any bells. Armed with the program name we could go searching the filesystem with tools like the **locate** or **find** commands. After we found it, we could then probably discern what package it belonged to. But with the PID available now, we can look at **ps** output, and see if that helps us any:

```
 $ /bin/ps ax |grep 1051 |grep −v grep
  1051 ?        S        0:24 /usr/libexec/postfix/master
```

We took a shortcut here by combining **ps** with **grep**. It looks like that this file belongs to **postfix**, which is indeed a mail server package comparable to **sendmail** (and is included with Powertools, not the base distribution).

Running **ps** with the `−−forest` flag (`−f` for short) can be helpful in determining what processes are parent or child process or another process. An edited example:

```
 $ /bin/ps −axf
  956 ?        S      0:00 named −u named
  957 ?        S      0:00  \_ named −u named
```

```
  958 ?        S      0:46      \_ named -u named
  959 ?        S      0:47      \_ named -u named
  960 ?        S      0:00      \_ named -u named
  961 ?        S      0:11      \_ named -u named
 1051 ?        S      0:30 /usr/libexec/postfix/master
 1703 ?        S      0:00  \_ tlsmgr -l -t fifo -u -c
 1704 ?        S      0:00  \_ qmgr -l -t fifo -u -c
 1955 ?        S      0:00  \_ pickup -l -t fifo -c
 1863 ?        S      0:00  \_ trivial-rewrite -n rewrite -t unix -u -c
 2043 ?        S      0:00  \_ cleanup -t unix -u -c
 2049 ?        S      0:00  \_ local -t unix
 2062 ?        S      0:00  \_ smtpd -n smtp -t inet -u -c
```

A couple of things to note here. We have two by now familiar daemons here: **named** and **postfix (smtpd)**. Both are spawning sub−processes. In the case of **named**, what we are seeing is threads, various sub−processes that it always spawns. **Postfix** is also spawning sub−processes, but not as "threads". Each sub−process has its own specific task. It is worth noting that child processes are dependent on the parent process. So killing the parent PID, will in turn kill all child processes.

If all this has not shed any light, we might also try **locate**:

```
$ locate /master
/etc/postfix/master.cf
/var/spool/postfix/pid/master.pid
/usr/libexec/postfix/master
/usr/share/vim/syntax/master.vim
/usr/share/vim/vim60z/syntax/master.vim
/usr/share/doc/postfix-20010202/html/master.8.html
/usr/share/doc/postfix-20010202/master.cf
/usr/share/man/man8/master.8.gz
```

find is perhaps the most flexible file finding utility, but doesn't use a database the way **locate** does, so is much slower:

```
$ find / -name master
/usr/libexec/postfix/master
```

If **lsof** is installed, it is another command that is useful for finding who owns processes or ports:

```
# lsof -i :631
COMMAND   PID  USER    FD   TYPE DEVICE SIZE NODE NAME
cupsd   1315  root    0u   IPv4   3734       TCP *:631 (LISTEN)
```

This is again telling us that the **cupsd** print daemon is the owner of port 631. Just a different way of getting at it. Yet one more way to get at this is with **fuser**, which should be installed:

```
# fuser -v -n tcp 631
```

```
                     USER        PID  ACCESS  COMMAND
631/tcp              root       1315  f....   cupsd
```

See the man pages for **fuser** and **lsof** command syntax.

Another place to look for where a service is started, is in the `init.d` directory, where the actual init scripts live. Something like `ls -l /etc/rc.d/init.d/`, should give us a list of these. Often the script name itself gives a hint as to which service(s) it starts, though it may not necessarily exactly match the "Program Name" as provided by **netstat**. Or we can use **grep** to search inside files and match a search pattern. Need to find where **rpc.statd** is being started, and we don't see a script by this name?

```
# grep rpc.statd /etc/init.d/*
/etc/init.d/nfslock: [ -x /sbin/rpc.statd ] || exit 0
/etc/init.d/nfslock:    daemon rpc.statd
/etc/init.d/nfslock:    killproc rpc.statd
/etc/init.d/nfslock:    status rpc.statd
/etc/init.d/nfslock:    /sbin/pidof rpc.statd >/dev/null 2>&1; STATD="$?"
```

We didn't really need all that information, but at least we see now exactly which script is starting it. Remember too that not all services are started this way. Some may be started via inetd, or xinetd.

The `/proc` filesystem also keeps everything we want to know about processes that are running. We can query this to find out more information about each process. Do you need to know the full path of the command that started a process?

```
# ls -l /proc/1315/exe
lrwxrwxrwx  1 root  root   0 July 4 19:41 /proc/1315/exe -> /usr/sbin/cupsd
```

Finally, we had a loose end or two in the UDP listening services. Remember we had a strange looking port number 32768, that also had no service name associated with it:

```
# netstat -aup
Active Internet connections (servers and established)
 Local Address           Foreign Address         State       PID/Program name
  *:32768                 *:*                                  956/named
  bigcat:domain           *:*                                  956/named
  bigcat:domain           *:*                                  956/named
  *:631                   *:*                                  1315/cupsd
```

Now by including the "PID/Program name" option with the `-p` flag, we see this also belongs to **named**, the nameserver daemon. Recent versions of BIND use an unprivileged port for some types of traffic. In this case, this is BIND 9.x. So no real alarms here either. The unprivileged port here is the one **named** uses to to talk to other nameservers for name and address lookups, and should not be firewalled.

So we found no big surprises in this hypothetical situation.

If all else fails, and you can't find a process owner for an open port, suspect that it may be an RPC (Remote Procedure Call) service of some kind. These use randomly assigned ports without any seeming logic or consistency, and are typically controlled by the **portmap** daemon. In some cases, these may not reveal the process owner to **netstat** or **lsof**. Try stopping **portmap**, and then see if the mystery service goes away. Or you can use **rpcinfo −p localhost** to see what RPC services may be running (**portmap** must be running for this to work).

If you suspect you have been broken into, *do not trust* **netstat** or **ps** output. There is a good chance that they, and other system components, has been tampered with in such a way that the output is not reliable.

8.4. Attacks and Threats

In this section, we will take a quick look at some of the common threats and techniques that are out there, and attempt to put them into some perspective.

The corporate world, government agencies and high profile Internet sites have to be concerned with a much more diverse and challenging set of threats than the typical home desktop user. There are many reasons someone may want to break in to someone else's computer. It may be just for kicks, or any number of malicious reasons. They may just want a base from which to attack someone else. This is a very common motivation.

The most common "attack" for most of us is from already compromised systems. The Internet is littered with computers that have been broken into, and are now doing their master's bidding blindly, in zombie−like fashion. They are programmed to scan massively large address ranges, probing each individual IP address as they go. Looking for one or more open ports, and then probing for known weaknesses if they get the chance. Very impersonal. Very methodical. And very effective. We are all in the path of such robotic scans. All because those responsible for these systems fail to do what you are doing now − taking steps to protect their system(s), and avoid being r00ted.

These scans do not look at login banners that may be presented on connection. It will do little good to change your `/etc/issue.net` to pretend that you are running some obscure operating system. If they find something listening, they will try all of the exploits appropriate to that port, without regard to any indications your system may give. If it works, they are in −− if not, they will move on.

8.4.1. Port Scans and Probes

First, let's define "scan" and "probe" since these terms come up quite a bit. A "probe" implies testing if a given port is open or closed, and possibly what might be listening on that port. A "scan" implies either "probing" multiple ports on one or more systems. Or individual ports on multiple systems. So you might "scan" all ports on your own system for instance. Or a cracker might "scan" the 216.78.*.* address range to see who has port 111 open.

Black hats can use scan and probe information to know what services are running on a given system, and then they might know what exploits to try. They may even be able to tell what Operating System is running, and even kernel version, and thus get even more information. "Worms", on the other hand, are automated and scan blindly, generally just looking for open ports, and then a susceptible victim. They are not trying to "learn" anything, the way a cracker might.

The distinction between "scan" and "probe"is often blurred. Both can used in good ways, or in bad ways, depending on who is doing it, and why. You might ask a friend to scan you, for instance, to see how well your firewall is working. This is a legitimate use of scanning tools such as nmap. But what if someone you don't know does this? What is their intent? If it's your ISP, they may be trying to enforce their Terms of Service Agreement. Or maybe, it is someone just playing, and seeing who is "out there". But more than likely it is someone or something with not such good intentions.

Full range port scans (meaning probing of many ports on the same machine) seem to be a not so common threat for home based networks. But certainly, scanning individual ports across numerous systems is a very, very common occurrence.

8.4.2. Rootkits

A "rootkit" is the script kiddie's stock in trade. When a successful intrusion takes place, the first thing that is often done, is to download and install such "rootkits". The rootkit is a set of scripts designed to take control of the system, and then hide the intrusion. Rootkits are readily available on the web for various Operating Systems.

A rootkit will typically replace critical system files such as **ls**, **ps**, **netstat**, **login** and others. Passwords may be added, hidden daemons started, logs tampered with, and surely one of more backdoors are opened. The hidden backdoors allow easy access any time the attacker wants back in. And often the vulnerability itself may even be fixed so that the new "owner" has the system all to himself. The entire process is scripted so it happens very quickly. The rightful owners of these compromised systems generally have no idea what is going on, and are victims themselves. A well designed rootkit can be very difficult to detect.

8.4.3. Worms and Zombies

A "worm" is a self replicating exploit. It infects a system, then attempts to spread itself typically via the same vulnerability. Various "worms" are weaving their way through the entire Internet address space constantly, spreading themselves as they go.

But somewhere behind the zombie, there is a controller. Someone launched the worm, and they will be informed after a successful intrusion. It is then up to them how the system will be used.

Many of these are Linux systems, looking for other Linux systems to "infect" via a number of exploits. But most Operating Systems share in this threat. Once a vulnerable system is found, the actual entry and take over is quick, and may be difficult to detect after the fact. The first thing an intruder (whether human or "worm") will do is attempt to cover their tracks. A "rootkit" is downloaded and installed. This trend has been exacerbated by the growing popularity of cable modems and DSL. The number of full time Internet connections is growing rapidly, and this makes fertile ground for such exploits since often these aren't as well secured as larger sites.

While this may sound ominous, a few simple precautions can effectively deter this type of attack. With so many easy victims out there, why waste much effort breaking into *your* system? There is no incentive to really try very hard. Just scan, look, try, move on if unsuccessful. There is always more IPs to be scanned. If your firewall is effectively bouncing this kind of thing, it is no threat to you at all. Take comfort in that, and don't over re−act.

It is worth noting, that these worms cannot "force" their way in. They need an open and accessible port, *and* a known vulnerability. If you remember the "Iptables Weekly Log Summary" in the opening section above, many of those may have all been the result of this type of scan. If you've followed the steps in this HOWTO, you should be reasonably safe here. This one is easy enough to deflect.

8.4.4. Script Kiddies

A "script kiddie" is a "cracker" wanna be who doesn't know enough to come up with his/her own exploits, but instead relies on "scripts" and exploits that have been developed by others. Like "worms", they are looking for easy victims, and may similarly scan large address ranges looking for specific ports with known vulnerabilities. Often, the actual scanning is done from already comprised systems so that it is difficult to trace it back to them.

The script kiddie has a bag of ready made tricks at his disposal, including an arsenal of "rootkits" for various Operating Systems. Finding susceptible victims is not so hard, given enough time and address space to probe. The motives are a mixed bag as well. Simple mischief, defacement of web sites, stolen credit card numbers, and the latest craze, "Denial of Service" attacks (see below). They collect zombies like trophies and use them to carry out whatever their objective is.

Again, the key here is that they are following a "script", and looking for easy prey. Like the worm threat above, a functional firewall and a few very basic precautions, should be sufficient to deflect any threat here. By now, you should be relatively safe from this nuisance.

8.4.5. Spoofed IPs

How easy is it to spoof an IP address? With the right tools, very easy. How much of a threat is this? Not much, for most of us, and is over−hyped as a threat.

Because of the way TCP/IP works, each packet must carry both the source and destination IP addresses. Any return traffic is based on this information. So a spoofed IP can never return any useful information to an attacker who is sending out spoofed packets. The traffic would go back to wherever that spoofed IP address was pointed. The attacker gets nothing back at all.

This does have potential for "DoS" attacks (see below) where learning something about the targeted system is not important. And may be used for some general mischief making as well.

8.4.6. Targeted Attacks

The worm and wide ranging address type scans, are impersonal. They are just looking for any vulnerable system. It makes no difference whether it is a top secret government facility, or your mother's Window's box. But there are "black hats" that will spend a great deal of effort to get into a system or network. We'll call these "targeted" attacks since there has been a deliberate decision made to break in to a specific system or network.

In this case, the attacker will look the system over for weaknesses. And possibly make many different kinds of attempts, until he finds a crack to wiggle through. Or gives up. This is more difficult to defend against. The attacker is armed and dangerous, so to speak, and is stalking his prey.

Again, this scenario is very unlikely for a typical home system. There just generally isn't any incentive to take the time and effort when there are bigger fish to fry. For those who may be targets, the best defense here includes many of things we've discussed. Vigilance is probably more important than ever. Good logging practices and an IDS (Intrusion Detection System) should be in place. And subscribing to one or more security related mailing lists like BUGTRAQ. And of course, reading those alerts daily, and taking the appropriate actions, etc.

8.4.7. Denial of Service (DoS)

"DoS" is another type of "attack" in which the intention is to disrupt or overwhelm the targeted system or network in such a way that it cannot function normally. DoS can take many forms. On the Internet, this often means overwhelming the victim's bandwidth or TCP/IP stack, by sending floods of packets and thus effectively disabling the connection. We are talking about many, many packets per second. Thousands in some cases. Or perhaps, the objective is to crash a server.

This is much more likely to be targeted at organizations or high profile sites, than home users. And can be quite challenging to stop depending on the technique. And it generally requires the co−operation of networks between the source(s) and the target, so that the floods are stopped, or minimized, before they reach the targeted destination. Once they hit the destination, there is no good way to completely ignore them.

"DDoS", Distributed Denial of Service, is where multiple sources are used to maximize the impact. Again, not likely to be directly targeted at home users. These are "slaves" that are "owned" by a cracker, or script kiddie, that are woken up and are targeted at the victim. There may be many computers involved in the attack.

If you are home user, and with a dynamic IP address, you might find disconnecting, then re−connecting to get a new IP, an effective way out if you are the target. Maybe.

8.4.8. Brute Force

"Brute force" attacks are where the attacker makes repetitive attempts at the same perceived weakness(es). Like a battering ram. A classic example would be where someone tries to access a telnet server simply by continually throwing passwords at it, hoping that one will eventually work. Or maybe crash the server. This doesn't require much imagination, and is not a commonly used tactic against home systems.

By the way, this is one good argument against allowing remote root logins. The root account exists on all systems. It is probably the only one that this is true of. You'd like to make a potential attacker guess both the login name *and* password. But if root is allowed remote logins, then the attacker only needs to guess the password!

8.4.9. Viruses

And now something *not* to worry about. Viruses seem to be primarily a Microsoft problem. For various reasons, viruses are not a significant threat to Linux users. This is not to say that it will always be this way, but the current virus explosion that plagues Microsoft systems, can not spread to Linux (or Unix) based systems. In fact, the various methods and practices that enable this phenomena, are not exploitable on Linux. So Anti−Virus software is not recommended as part of our arsenal. At least for the time being with Linux only networks.

8.5. Links

Some references for further reading are listed below. Not listed is your distribution's site, security page or ftp download site. You will have to find these on your own. Then you should bookmark them!

- Redhat sites of interest:

  The Redhat watch/security mailing list: https://listman.redhat.com/mailman/listinfo/redhat−watch−list

  Red Hat errata and security notices: http://redhat.com/errata/

  The Red Hat update FTP site: ftp://updates.redhat.com/
- Other relevant documents available from the Linux Documentation Project:

  Security HOWTO: http://linuxdoc.org/HOWTO/Security−HOWTO.html

  Firewall HOWTO: http://linuxdoc.org/HOWTO/Firewall−HOWTO.html

  Ipchains HOWTO: http://linuxdoc.org/HOWTO/IPCHAINS−HOWTO.html

  User Authentication: http://linuxdoc.org/HOWTO/User−Authentication−HOWTO/index.html, includes a nice discussion on PAM.

  VPN (Virtual Private Network): http://linuxdoc.org/HOWTO/VPN−HOWTO.html and http://linuxdoc.org/HOWTO/VPN−Masquerade−HOWTO.html

  The Remote X Apps Mini HOWTO, http://www.linuxdoc.org/HOWTO/mini/Remote−X−Apps.html, includes excellent discussions on the security implications of running X Windows.

  The Linux Network Administrators Guide: http://linuxdoc.org/LDP/nag2/index.html, includes a good overview of networking and TCP/IP, and firewalling.

  The Linux Administrator's Security Guide: http://www.seifried.org/lasg/, includes many obvious topics of interest, including firewalling, passwords and authentication, PAM, and more.

Securing Red Hat:
http://linuxdoc.org/LDP/solrhe/Securing−Optimizing−Linux−RH−Edition−v1.3/index.html

- Tools for creating custom ipchains and iptables firewall scripts:

  PMFirewall: http://www.pmfirewall.com/PMFirewall/

  Firestarter: http://firestarter.sourceforge.net

  Two related projects: http://seawall.sourceforge.net/ for ipchains, and http://shorewall.sourceforge.net/ for iptables.

- Netfilter and iptables documentation from the netfilter developers (available in many other languages as well):

  FAQ: http://netfilter.samba.org/documentation/FAQ/netfilter−faq.html
  Packet filtering: http://netfilter.samba.org/documentation/HOWTO/packet−filtering−HOWTO.html
  Networking: http://netfilter.samba.org/documentation/HOWTO/networking−concepts−HOWTO.html
  NAT/masquerading: http://netfilter.samba.org/documentation/HOWTO/NAT−HOWTO.html

- Port number assignments, and what that scanner may be scanning for:

  http://www.linuxsecurity.com/resource_files/firewalls/firewall−seen.html

  http://www.sans.org/newlook/resources/IDFAQ/oddports.htm

  http://www.iana.org/assignments/port−numbers, the official assignments.

- General security sites. These all have areas on documentation, alerts, newsletters, mailing lists, and other resources.

  Linux Security.com: http://www.linuxsecurity.com, loaded with good info, and Linux specific. Lots of good docs: http://www.linuxsecurity.com/docs/

  CERT, http://www.cert.org

  The SANS Institute: http://www.sans.org/

  The Coroner's Toolkit (TCT): http://www.fish.com/security/, discussions and tools for dealing with post break−in issues (and preventing them in the first place).

- Privacy:

  Junkbuster: http://www.junkbuster.com, a web proxy and cookie manager.

  PGP: http://www.gnupg.org/

* Other documentation and reference sites:

  Linux Security.com: http://www.linuxsecurity.com/docs/

  Linux Newbie: http://www.linuxnewbie.org/nhf/intel/security/index.html

  The comp.os.linux.security FAQ: http://www.linuxsecurity.com/docs/colsfaq.html

  The Internet Firewall FAQ: http://pubweb.nfr.net/~mjr/pubs/fwfaq/

  The Site Security Handbook RFC: http://www.ietf.org/rfc/rfc2196.txt
* Miscellaneous sites of interest:

  http://www.bastille−linux.org, for Mandrake and Red Hat only.

  SAINT: http://www.wwdsi.com/saint/, system security analysis.

  SSL: http://www.openssl.org/

  SSH: http://www.openssh.org/

  Scan yourself: http://www.hackerwhacker.com

  PAM: http://www.kernel.org/pub/linux/libs/pam/index.html

  Detecting Trojaned Linux Kernel Modules: http://members.prestige.net/tmiller12/papers/lkm.htm

  Rootkit checker: http://www.chkrootkit.org

  Port scanning tool nmap's home page: http://www.insecure.org

  Nessus, more than just a port scanner: http://www.nessus.org

  Tripwire, intrusion detection: http://www.tripwire.org

  Snort, sniffer and more: http://www.snort.org

  http://www.mynetwatchman.com and http://dshield.org are "Distributed Intrusion Detection Systems". They collect log data from subscribing "agents", and collate the data to find and report malicious activity. If you want to fight back, check these out.

8.6. Editing Text Files

By Bill Staehle

All the world is a file.

There are a great many types of files, but I'm going to stretch it here, and class them into two really broad families:

```
 Text files are just that.
 Binary files are not.
```

Binary files are meant to be read by machines, text files can be easily edited, and are generally read by people. But text files can be (and frequently are) read by machines. Examples of this would be configuration files, and scripts.

There are a number of different text editors available in *nix. A few are found on every system. That would be '/bin/ed' and '/bin/vi'. 'vi' is almost always a clone such as 'vim' due to license problems. The problem with 'vi' and 'ed' is that they are terribly user unfriendly. Another common editor that is not always installed by default is 'emacs'. It has a lot more features and capability, and is not easy to learn either.

As to 'user friendly' editors, 'mcedit' and 'pico' are good choices to start with. These are often much easier for those new to *nix.

The first things to learn are how to exit an editing session, how to save changes to the file, and then how to avoid breaking long lines that should not be broken (wrapped).

The 'vi' editor

'vi' is one of the most common text editors in the Unix world, and it's nearly always found on any *nix system. Actually, due to license problems, the '/bin/vi' on a Linux system is always a 'clone', such as 'elvis', 'nvi', or 'vim' (there are others). These clones can act exactly like the original 'vi', but usually have additional features that make it slightly less impossible to use.

So, if it's so terrible, why learn about it? Two reasons. First, as noted, it's almost guaranteed to be installed, and other (more user friendly) editors may not be installed by default. Second, many of the 'commands' work in other applications (such as the pager 'less' which is also used to view man pages). In 'less', accidentally pressing the 'v' key starts 'vi' in most installations.

'vi' has two modes. The first is 'command mode', and keystrokes are interpreted as commands. The other mode is 'insert' mode, where nearly all keystrokes are interpreted as text to be inserted.

==> Emergency exit from 'vi' 1. press the <esc> key up to three times, until the computer beeps, or the screen flashes. 2. press the keys :q! <Enter>

That is: colon, the letter Q, and then the exclamation point, followed by the Enter key.

'vi' commands are as follows. All of these are in 'command' mode:

```
a    Enter insertion mode after the cursor.
A    Enter insertion mode at the end of the current line.
i    Enter insertion mode before the cursor.
o    Enter insertion mode opening a new line BELOW current line.
O    Enter insertion mode opening a new line ABOVE current line.
h    move cursor left one character.
l    move cursor right one character.
j    move cursor down one line.
k    move cursor up one line.
/mumble  move cursor forward to next occurrence of 'mumble' in
         the text
?mumble  move cursor backward to next occurrence of 'mumble'
         in the text
n    repeat last search (? or / without 'mumble' to search for
     will do the same thing)
u    undo last change made

^B   Scroll back one window.
^F   Scroll forward one window.
^U   Scroll up one half window.
^D   Scroll down one half window.

:w   Write to file.
:wq  Write to file, and quit.
:q   quit.
:q!  Quit without saving.

<esc>   Leave insertion mode.
```

NOTE: The four 'arrow' keys almost always work in 'command' or 'insert' mode.

The 'ed' editor.

The 'ed' editor is a line editor. Other than the fact that it is virtually guaranteed to be on any *nix computer, it has no socially redeeming features, although some applications may need it. A _lot_ of things have been offered to replace this 'thing' from 1975.

==> Emergency exit from 'ed'

1. type a period on a line by itself, and press <Enter> This gets you to the command mode or prints a line of text if you were in command mode. 2. type q and press <Enter>. If there were no changes to the file, this action quits ed. If you then see a '?' this means that the file had changed, and 'ed' is asking if you want to save the changes. Press q and <Enter> a second time to confirm that you want out.

The 'pico' editor.

'pico' is a part of the Pine mail/news package from the University of Washington (state, USA). It is a very friendly editor, with one minor failing. It silently inserts a line feed character and wraps the line when it exceeds (generally) 74 characters. While this is fine while creating mail, news articles, and text notes, it is often fatal when editing system files. The solution to this problem is simple. Call the program with the −w option, like this:

pico −w file_2_edit

Pico is so user friendly, no further instructions are needed. It _should_ be obvious (look at the bottom of the screen for commands). There is an extensive help function. Pico is available with nearly all distributions, although it _may_ not be installed by default.

==> Emergency exit from 'pico'

Press and hold the <Ctrl> key, and press the letter x. If no changes had been made to the file, this will quit pico. If changes had been made, it will ask if you want to save the changes. Pressing n will then exit.

The 'mcedit' editor.

'mcedit' is part of the Midnight Commander shell program, a full featured visual shell for Unix−like systems. It can be accessed directly from the command line (mcedit file_2_edit) or as part of 'mc' (use the arrow keys to highlight the file to be edited, then press the F4 key).

mcedit is probably the most intuitive editor available, and comes with extensive help. "commands" are accessed through the F* keys. Midnight Commander is available with nearly all distributions, although it _may_ not be installed by default.

==> Emergency exit from 'mcedit'

Press the F10 key. If no changes have been made to the file, this will quit mcedit. If changes had been made, it will ask if you want to Cancel this action. Pressing n will then exit.

8.7. nmap

Let's look at a few quick examples of what **nmap** scans look like. The intent here is to show how to use **nmap** to verify our firewalling, and system integrity. **nmap** has other uses that we don't need to get into. Do NOT use **nmap** on systems other than your own, unless you have permission from the owner, and you know it is not a violation of anyone's Terms of Service. This kind of thing *will* be taken as hostile by most people.

As mentioned previously, **nmap** is a sophisticated port scanning tool. It tries to see if a host is "there", and what ports might be open. Barring that, what states those ports might be in. **nmap** has a complex command line and can do many types of "scans". See the man page for all the nitty gritty.

A couple of words of warning first. If using portsentry, turn it off. It will drop the route to wherever the scan is coming from. You might want to turn off any logging also, or at least be aware that you might get copious logs if doing multiple scans.

A simple, default scan of "localhost":

```
# nmap localhost

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
Interesting ports on bigcat (127.0.0.1):
(The 1507 ports scanned but not shown below are in state: closed)

Port       State       Service
22/tcp     open        ssh
25/tcp     open        smtp
37/tcp     open        time
53/tcp     open        domain
80/tcp     open        http
3000/tcp   open        ppp

Nmap run completed -- 1 IP address (1 host up) scanned in 2 seconds
```

If you've read most of this document, you should be familiar with these services by now. These are some of the same ports we've seen in other examples. Some things to note on this scan: it only did 1500+ "interesting" ports −− not all ports. This can be configured differently if more is desirable (see man page). It only did TCP ports too. Again, configurable. It only picks up "listening" services, unlike **netstat** that shows all open ports −− listening or otherwise. Note the last "open" port here is 3000 is identified as "PPP". Wrong! That is just an educated guess by nmap based on what is contained in `/etc/services` for this port number. Actually in this case it is ntop (a network traffic monitor). Take the service names with a grain of salt. There is no way for **nmap** to really know what is on that port. Matching port numbers with service names can at times be risky. Many do have standard ports, but there is nothing to say they have to use the commonly associated port numbers.

Notice that in all our **netstat** examples, we had two classes of open ports: listening servers, and then established connections that we initiated to other remote hosts (e.g. a web server somewhere). **nmap** only sees the first group −− the listening servers! The other ports connecting us to remote servers are not visible, and thus not vulnerable. These ports are "private" to that single connection, and will be closed when the connection is terminated.

So we have open and closed ports here. Simple enough, and gives a pretty good idea what is running on bigcat −− but not necessarily what we look like to the outside world since this was done from localhost, and wouldn't reflect any firewalling or other access control mechanisms.

Let's do a little more intensive scan. Let's check all ports −− TCP and UDP.

```
# nmap -sT -sU -p 1-65535 localhost

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
Interesting ports on bigcat (127.0.0.1):
(The 131050 ports scanned but not shown below are in state: closed)

Port       State       Service
22/tcp     open        ssh
25/tcp     open        smtp
37/tcp     open        time
53/tcp     open        domain
53/udp     open        domain
80/tcp     open        http
3000/tcp   open        ppp
8000/tcp   open        unknown
```

```
32768/udp  open        unknown

Nmap run completed -- 1 IP address (1 host up) scanned in 385 seconds
```

This is more than just "interesting" ports −− it is everything. We picked up a couple of new ones in the process too. We've seen these before with **netstat**, so we know what they are. That is the **Junkbuster** web proxy on port 8000/tcp and **named** on 32768/udp. This scan takes much, much longer, but it is the only way to see all ports.

So now we have a pretty good idea of what is open on bigcat. Since we are scanning localhost from localhost, everything should be visible. We still don't know how the outside world sees us though. Now I'll **ssh** to another host on the same LAN, and try again.

```
# nmap bigcat

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
Interesting ports on bigcat (192.168.1.1):
(The 1520 ports scanned but not shown below are in state: closed)

Port       State       Service
22/tcp     open        ssh
3000/tcp   open        ppp

Nmap run completed -- 1 IP address (1 host up) scanned in 1 second
```

I confess to tampering with the iptables rules here to make a point. Only two visible ports on this scan. Everything else is "closed". So says nmap. Once again:

```
# nmap bigcat

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
Note: Host seems down. If it is really up, but blocking our ping probes, try −P0

Nmap run completed -- 1 IP address (0 hosts up) scanned in 30 seconds
```

Oops, I blocked ICMP (ping) while I was at it this time. One more time:

```
# nmap −P0 bigcat

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
All 1523 scanned ports on bigcat (192.168.1.1) are: filtered

Nmap run completed -- 1 IP address (1 host up) scanned in 1643 seconds
```

That's it. Notice how long that took. Notice ports are now "filtered" instead of "closed". How does **nmap** know that? Well for one, "closed" means bigcat sent a packet back saying "nothing running here", i.e. port is closed. In this last example, the iptables rules were changed to not allow ICMP (ping), and to "DROP" all incoming packets. In other words, no response at all. A subtle difference since **nmap** seems to still know there was a host there, even though no response was given. One lesson here, is if you want to slow a scanner down, "DROP" (or "DENY") the packets. This forces a TCP time out for the remote end on each port probe. Anyway, if your scans look like this, that is probably as well as can be expected, and your firewall is doing its job.

A brief note on UDP: **nmap** can not accurately determine the status of these ports if they are "filtered". You probably will get a false−positive "open" condition. This has to do with UDP being a connectionless protocol. If **nmap** gets no answer (e.g. due to a "DROP"), it assumes the packets reached the target, and thus the port will be reported as "open". This is "normal" for **nmap**.

We can play with firewall rules in a LAN set up to try to simulate how the outside world sees us, and if we are smart, and know what we are doing, and don't have a brain fart, we probably will have a pretty good picture. But it is still best to try to find a way to do it from outside if possible. Again, make sure you are not violating any ISP rules of conduct. Do you have a friend on the same ISP?

8.8. Sysctl Options

The "sysctl" options are kernel parameters that can be configured via the `/proc` filesystem. These can be dynamically adjusted at run−time. Typically these options are off if set to "0", and on if set to "1".

Some of these have security implications, and thus is why we are here ;−) We'll just list the ones we think are relevant. Feel free to cut and paste these into a firewall script, or other file that is run during boot (like `/etc/rc.local`). Red Hat provides the **sysctl** command for dynamically adjusting these values (see man page). Or they can permanently be set in `/etc/sysctl.conf` with your text editor of choice. **sysctl** is executed during init, and uses these values. You can read up on what these mean in `/usr/src/linux/Documentation/sysctl/README` and other files in the kernel Documentation directories.

The traditional method:

```
#!/bin/sh
#
# Configure kernel sysctl run−time options.
#
###################################################################

# Anti−spoofing blocks
for i in /proc/sys/net/ipv4/conf/*/rp_filter;
do
 echo 1 > $i
done

# Ensure source routing is OFF
for i in /proc/sys/net/ipv4/conf/*/accept_source_route;
 do
  echo 0 > $i
 done

# Ensure TCP SYN cookies protection is enabled
[ −e /proc/sys/net/ipv4/tcp_syncookies ] &&\
 echo 1 > /proc/sys/net/ipv4/tcp_syncookies

# Ensure ICMP redirects are disabled
```

```
for i in /proc/sys/net/ipv4/conf/*/accept_redirects;
 do
  echo 0 > $i
 done

# Ensure oddball addresses are logged
[ -e /proc/sys/net/ipv4/conf/all/log_martians ] &&\
 echo 1 > /proc/sys/net/ipv4/conf/all/log_martians

[ -e /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts ] &&\
 echo 1 > /proc/sys/net/ipv4/icmp_echo_ignore_broadcasts

[ -e /proc/sys/net/ipv4/icmp_ignore_bogus_error_responses ] &&\
 echo 1 > /proc/sys/net/ipv4/icmp_ignore_bogus_error_responses

## Optional from here on down, depending on your situation. ############

# Ensure ip-forwarding is enabled if
# we want to do forwarding or masquerading.
[ -e /proc/sys/net/ipv4/ip_forward ] &&\
 echo 1 > /proc/sys/net/ipv4/ip_forward

# On if your IP is dynamic (or you don't know).
[ -e /proc/sys/net/ipv4/ip_dynaddr ] &&\
 echo 1 > /proc/sys/net/ipv4/ip_dynaddr

# eof
```

The same effect by using `/etc/sysctl.conf` instead:

```
#
# Add to existing sysctl.conf
#

# Anti-spoofing blocks
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.all.rp_filter = 1

# Ensure source routing is OFF
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.conf.all.accept_source_route = 0

# Ensure TCP SYN cookies protection is enabled
net.ipv4.tcp_syncookies = 1

# Ensure ICMP redirects are disabled
net.ipv4.conf.default.accept_redirects = 0
net.ipv4.conf.all.accept_redirects = 0

# Ensure oddball addresses are logged
net.ipv4.conf.default.log_martians = 1
net.ipv4.conf.all.log_martians = 1


net.ipv4.icmp_echo_ignore_broadcasts = 1

net.ipv4.icmp_ignore_bogus_error_responses = 1

## Optional from here on down, depending on your situation. ############
```

```
# Ensure ip-forwarding is enabled if
# we want to do forwarding or masquerading.
net.ipv4.ip_forward = 1

# On if your IP is dynamic (or you don't know).
net.ipv4.ip_dynaddr = 1

# end of example
```

8.9. Secure Alternatives

This section will give a brief run down on secure alternatives to potentially insecure methods. This will be a hodge podge of clients and servers.

* telnet, rsh − ssh
* ftp, rcp − scp or sftp. Both are part of ssh packages. Also, files can easily be transfered via HTTP if Apache is already running anyway. Apache can be buttoned down even more by using SSL (HTTPS).
* sendmail − postfix, qmail. Not to imply that current versions of sendmail are insecure. Just that there is some bad history there, and just because it is so widely used that it makes an inviting crack target.

  As noted above, Linux installations often include a fully functional mail server. While this may have some advantages, it is not necessary in many cases for simply sending mail, or retrieving mail. This can all be done without a "mail server daemon" running locally.
* POP3 − SPOP3, POP3 over SSL. If you really need to run your own POP server, this is the way to do it. If retrieving your mail from your ISP's server, then you are at their mercy as to what they provide.
* IMAP − IMAPS, same as above.
* If you find you need a particular service, and it is for just you or a few friends, consider running it on a non−standard port. Most server daemons support this, and is not a problem as long as those who will be connecting, know about it. For instance, the standard port for **sshd** is 22. Any worm or scan will probe for this port number. So run it on a randomly chosen port. See the **sshd** man page.

8.10. Ipchains and Iptables Redux

This section offers a little more advanced look at some of things that ipchains and iptables can do. These are basically the same scripts as in Step 3 above, just with some more advanced configuration options added. These will provide "masquerading", "port forwarding", allow access to some user definable services, and a few other things. Read the comments for explanations.

8.10.1. ipchains II

```
#!/bin/sh
#
# ipchains.sh
```

```
#
# An example of a simple ipchains configuration. This script
# can enable 'masquerading' and will open user definable ports.
#
###################################################################
# Begin variable declarations and user configuration options ######
#
# Set the location of ipchains (default).
IPCHAINS=/sbin/ipchains

# Local Interfaces
#
# This is the WAN interface, that is our link to the outside world.
# For pppd and pppoe users.
# WAN_IFACE="ppp0"
WAN_IFACE="eth0"
#
# Local Area Network (LAN) interface.
#LAN_IFACE="eth0"
LAN_IFACE="eth1"

# Our private LAN address(es), for masquerading.
LAN_NET="192.168.1.0/24"

# For static IP, set it here!
#WAN_IP="1.2.3.4"

# Set a list of public server port numbers here...not too many!
# These will be open to the world, so use caution. The example is
# sshd, and HTTP (www). Any services included here should be the
# latest version available from your vendor. Comment out to disable
# all PUBLIC services.
#PUBLIC_PORTS="22 80 443"
PUBLIC_PORTS="22"

# If we want to do port forwarding, this is the host
# that will be forwarded to.
#FORWARD_HOST="192.168.1.3"

# A list of ports that are to be forwarded.
#FORWARD_PORTS="25  80"

# If you get your public IP address via DHCP, set this.
DHCP_SERVER=66.21.184.66

# If you need identd for a mail server, set this.
MAIL_SERVER=

# A list of unwelcome hosts or nets. These will be denied access
# to everything, even our 'PUBLIC' services. Provide your own list.
#BLACKLIST="11.22.33.44 55.66.77.88"

# A list of "trusted" hosts and/or nets. These will have access to
# ALL protocols, and ALL open ports. Be selective here.
#TRUSTED="1.2.3.4/8  5.6.7.8"

## end user configuration options #################################
###################################################################

# The high ports used mostly for connections we initiate and return
# traffic.
LOCAL_PORTS=`cat /proc/sys/net/ipv4/ip_local_port_range |cut -f1`:\
```

```
`cat /proc/sys/net/ipv4/ip_local_port_range |cut -f2`

# Any and all addresses from anywhere.
ANYWHERE="0/0"

# Start building chains and rules #################################
#
# Let's start clean and flush all chains to an empty state.
$IPCHAINS -F

# Set the default policies of the built-in chains. If no match for any
# of the rules below, these will be the defaults that ipchains uses.
$IPCHAINS -P forward DENY
$IPCHAINS -P output ACCEPT
$IPCHAINS -P input DENY

# Accept localhost/loopback traffic.
$IPCHAINS -A input -i lo -j ACCEPT

# Get our dynamic IP now from the Inet interface. WAN_IP will be our
# IP address we are protecting from the outside world. Put this
# here, so default policy gets set, even if interface is not up
# yet.
[ -z "$WAN_IP" ] &&\
  WAN_IP=`ifconfig $WAN_IFACE |grep inet |cut -d : -f 2 |cut -d \  -f 1`

# Bail out with error message if no IP available! Default policy is
# already set, so all is not lost here.
[ -z "$WAN_IP" ] && echo "$WAN_IFACE not configured, aborting." && exit 1

WAN_MASK=`ifconfig $WAN_IFACE | grep Mask | cut -d : -f 4`
WAN_NET="$WAN_IP/$WAN_MASK"

## Reserved IPs:
#
# We should never see these private addresses coming in from outside
# to our external interface.
$IPCHAINS -A input -l -i $WAN_IFACE -s 10.0.0.0/8     -j DENY
$IPCHAINS -A input -l -i $WAN_IFACE -s 172.16.0.0/12  -j DENY
$IPCHAINS -A input -l -i $WAN_IFACE -s 192.168.0.0/16 -j DENY
$IPCHAINS -A input -l -i $WAN_IFACE -s 127.0.0.0/8    -j DENY
$IPCHAINS -A input -l -i $WAN_IFACE -s 169.254.0.0/16 -j DENY
$IPCHAINS -A input -l -i $WAN_IFACE -s 224.0.0.0/4    -j DENY
$IPCHAINS -A input -l -i $WAN_IFACE -s 240.0.0.0/5    -j DENY
# Bogus routing
$IPCHAINS -A input -l -s 255.255.255.255 -d $ANYWHERE -j DENY

## LAN access and masquerading
#
# Allow connections from our own LAN's private IP addresses via the LAN
# interface and set up forwarding for masqueraders if we have a LAN_NET
# defined above.
if [ -n "$LAN_NET" ]; then
 echo 1 > /proc/sys/net/ipv4/ip_forward
 $IPCHAINS -A input  -i $LAN_IFACE  -j ACCEPT
 $IPCHAINS -A forward -s $LAN_NET -d $LAN_NET -j ACCEPT
 $IPCHAINS -A forward  -s $LAN_NET -d ! $LAN_NET -j MASQ
fi

## Blacklist hosts/nets
#
# Get the blacklisted hosts/nets out of the way, before we start opening
```

```
# up any services. These will have no access to us at all, and will be
# logged.
for i in $BLACKLIST; do
 $IPCHAINS -A input -l -s $i -j DENY
done

## Trusted hosts/nets
#
# This is our trusted host list. These have access to everything.
for i in $TRUSTED; do
 $IPCHAINS -A input -s $i -j ACCEPT
done

# Port Forwarding
#
# Which ports get forwarded to which host. This is one to one
# port mapping (ie 80 -> 80) in this case.
# NOTE: ipmasqadm is a separate package from ipchains and needs
# to be installed also. Check first!
[ -n "$FORWARD_HOST" ] && ipmasqadm portfw -f &&\
 for i in $FORWARD_PORTS; do
   ipmasqadm portfw -a -P tcp -L $WAN_IP $i -R $FORWARD_HOST $i
 done

## Open, but Restricted Access ports/services
#
# Allow DHCP server (their port 67) to client (to our port 68) UDP traffic
# from outside source.
[ -n "$DHCP_SERVER" ] &&\
 $IPCHAINS -A input -p udp -s $DHCP_SERVER 67 -d $ANYWHERE 68 -j ACCEPT

# Allow 'identd' (to our TCP port 113) from mail server only.
[ -n "$MAIL_SERVER" ] &&\
 $IPCHAINS -A input -p tcp -s $MAIL_SERVER  -d $WAN_IP 113 -j ACCEPT

# Open up PUBLIC server ports here (available to the world):
for i in $PUBLIC_PORTS; do
 $IPCHAINS -A input -p tcp -s $ANYWHERE -d $WAN_IP $i -j ACCEPT
done

# So I can check my home POP3 mailbox from work. Also, so I can ssh
# in to home system. Only allow connections from my workplace's
# various IPs. Everything else is blocked.
$IPCHAINS -A input -p tcp -s 255.10.9.8/29 -d $WAN_IP 110 -j ACCEPT

# Uncomment to allow ftp data back (active ftp). Not required for 'passive'
# ftp connections.
#$IPCHAINS -A input -p tcp -s $ANYWHERE 20 -d $WAN_IP $LOCAL_PORTS -y -j ACCEPT

# Accept non-SYN TCP, and UDP connections to LOCAL_PORTS. These are
# the high, unprivileged ports (1024 to 4999 by default). This will
# allow return connection traffic for connections that we initiate
# to outside sources. TCP connections are opened with 'SYN' packets.
# We have already opened those services that need to accept SYNs
# for, so other SYNs are excluded here for everything else.
$IPCHAINS -A input -p tcp -s $ANYWHERE -d $WAN_IP $LOCAL_PORTS ! -y -j ACCEPT

# We can't be so selective with UDP since that protocol does not know
# about SYNs.
$IPCHAINS -A input -p udp -s $ANYWHERE -d $WAN_IP $LOCAL_PORTS -j ACCEPT

# Allow access to the masquerading ports conditionally. Masquerading
```

```
# uses it's own port range -- on 2.2 kernels ONLY! 2.4 kernels, do not
# use these ports, so comment out!
[ -n "$LAN_NET" ] &&\
 $IPCHAINS -A input -p tcp -s $ANYWHERE -d $WAN_IP 61000: ! -y -j ACCEPT &&\
 $IPCHAINS -A input -p udp -s $ANYWHERE -d $WAN_IP 61000: -j ACCEPT

## ICMP (ping)
#
# ICMP rules, allow the bare essential types of ICMP only. Ping
# request is blocked, ie we won't respond to someone else's pings,
# but can still ping out.
$IPCHAINS -A input  -p icmp  --icmp-type echo-reply \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
$IPCHAINS -A input  -p icmp  --icmp-type destination-unreachable \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT
$IPCHAINS -A input  -p icmp  --icmp-type time-exceeded \
   -s $ANYWHERE -i $WAN_IFACE -j ACCEPT

#######################################################################
# Set the catchall, default rule to DENY, and log it all. All other
# traffic not allowed by the rules above, winds up here, where it is
# blocked and logged. This is the default policy for this chain
# anyway, so we are just adding the logging ability here with '-l'.
# Outgoing traffic is allowed as the default policy for the 'output'
# chain. There are no restrictions on that.

$IPCHAINS -A input -l -j DENY

echo "Ipchains firewall is up `date`."

##-- eof ipchains.sh
```

8.10.2. iptables II

```
#!/bin/sh
#
# iptables.sh
#
# An example of a simple iptables configuration. This script
# can enable 'masquerading' and will open user definable ports.
#
###################################################################
# Begin variable declarations and user configuration options ######
#
# Set the location of iptables (default).
IPTABLES=/sbin/iptables

# Local Interfaces
# This is the WAN interface that is our link to the outside world.
# For pppd and pppoe users.
# WAN_IFACE="ppp0"
WAN_IFACE="eth0"
#
# Local Area Network (LAN) interface.
#LAN_IFACE="eth0"
LAN_IFACE="eth1"
```

```
# Our private LAN address(es), for masquerading.
LAN_NET="192.168.1.0/24"

# For static IP, set it here!
#WAN_IP="1.2.3.4"

# Set a list of public server port numbers here...not too many!
# These will be open to the world, so use caution. The example is
# sshd, and HTTP (www). Any services included here should be the
# latest version available from your vendor. Comment out to disable
# all Public services. Do not put any ports to be forwarded here,
# this only direct access.
#PUBLIC_PORTS="22 80 443"
PUBLIC_PORTS="22"

# If we want to do port forwarding, this is the host
# that will be forwarded to.
#FORWARD_HOST="192.168.1.3"

# A list of ports that are to be forwarded.
#FORWARD_PORTS="25  80"

# If you get your public IP address via DHCP, set this.
DHCP_SERVER=66.21.184.66

# If you need identd for a mail server, set this.
MAIL_SERVER=

# A list of unwelcome hosts or nets. These will be denied access
# to everything, even our 'Public' services. Provide your own list.
#BLACKLIST="11.22.33.44 55.66.77.88"

# A list of "trusted" hosts and/or nets. These will have access to
# ALL protocols, and ALL open ports. Be selective here.
#TRUSTED="1.2.3.4/8  5.6.7.8"

## end user configuration options #################################
###################################################################

# Any and all addresses from anywhere.
ANYWHERE="0/0"

# These modules may need to be loaded:
modprobe ip_conntrack_ftp
modprobe ip_nat_ftp

# Start building chains and rules #################################
#
# Let's start clean and flush all chains to an empty state.
$IPTABLES −F
$IPTABLES −X


# Set the default policies of the built−in chains. If no match for any
# of the rules below, these will be the defaults that IPTABLES uses.
$IPTABLES −P FORWARD DROP
$IPTABLES −P OUTPUT ACCEPT
$IPTABLES −P INPUT DROP

# Accept localhost/loopback traffic.
$IPTABLES −A INPUT −i lo −j ACCEPT
```

```
# Get our dynamic IP now from the Inet interface. WAN_IP will be the
# address we are protecting from outside addresses.
[ -z "$WAN_IP" ] &&\
  WAN_IP=`ifconfig $WAN_IFACE |grep inet |cut -d : -f 2 |cut -d \  -f 1`

# Bail out with error message if no IP available! Default policy is
# already set, so all is not lost here.
[ -z "$WAN_IP" ] && echo "$WAN_IFACE not configured, aborting." && exit 1

WAN_MASK=`ifconfig $WAN_IFACE |grep Mask |cut -d : -f 4`
WAN_NET="$WAN_IP/$WAN_MASK"

## Reserved IPs:
#
# We should never see these private addresses coming in from outside
# to our external interface.
$IPTABLES -A INPUT -i $WAN_IFACE -s 10.0.0.0/8      -j DROP
$IPTABLES -A INPUT -i $WAN_IFACE -s 172.16.0.0/12   -j DROP
$IPTABLES -A INPUT -i $WAN_IFACE -s 192.168.0.0/16  -j DROP
$IPTABLES -A INPUT -i $WAN_IFACE -s 127.0.0.0/8     -j DROP
$IPTABLES -A INPUT -i $WAN_IFACE -s 169.254.0.0/16  -j DROP
$IPTABLES -A INPUT -i $WAN_IFACE -s 224.0.0.0/4     -j DROP
$IPTABLES -A INPUT -i $WAN_IFACE -s 240.0.0.0/5     -j DROP
# Bogus routing
$IPTABLES -A INPUT -s 255.255.255.255 -d $ANYWHERE -j DROP

# Unclean
$IPTABLES -A INPUT -i $WAN_IFACE -m unclean -m limit \
        --limit 15/minute -j LOG --log-prefix "Unclean: "
$IPTABLES -A INPUT -i $WAN_IFACE -m unclean -j DROP

## LAN access and masquerading
#
# Allow connections from our own LAN's private IP addresses via the LAN
# interface and set up forwarding for masqueraders if we have a LAN_NET
# defined above.
if [ -n "$LAN_NET" ]; then
 echo 1 > /proc/sys/net/ipv4/ip_forward
 $IPTABLES -A INPUT -i $LAN_IFACE  -j ACCEPT
# $IPTABLES -A INPUT -i $LAN_IFACE -s $LAN_NET -d $LAN_NET  -j ACCEPT
 $IPTABLES -t nat -A POSTROUTING -s $LAN_NET -o $WAN_IFACE -j MASQUERADE
fi

## Blacklist
#
# Get the blacklisted hosts/nets out of the way, before we start opening
# up any services. These will have no access to us at all, and will
# be logged.
for i in $BLACKLIST; do
 $IPTABLES -A INPUT -s $i -m limit --limit 5/minute \
   -j LOG --log-prefix "Blacklisted: "
 $IPTABLES -A INPUT -s $i -j DROP
done

## Trusted hosts/nets
#
# This is our trusted host list. These have access to everything.
for i in $TRUSTED; do
 $IPTABLES -A INPUT -s $i -j ACCEPT
done

# Port Forwarding
```

```
#
# Which ports get forwarded to which host. This is one to one
# port mapping (ie 80 -> 80) in this case.
[ -n "$FORWARD_HOST" ] &&\
 for i in $FORWARD_PORTS; do
   $IPTABLES -A FORWARD -p tcp -s $ANYWHERE -d $FORWARD_HOST \
     --dport $i -j ACCEPT
   $IPTABLES -t nat -A PREROUTING -p tcp -d $WAN_IP --dport $i \
     -j DNAT --to $FORWARD_HOST:$i
 done

## Open, but Restricted Access ports
#
# Allow DHCP server (their port 67) to client (to our port 68) UDP
# traffic from outside source.
[ -n "$DHCP_SERVER" ] &&\
 $IPTABLES -A INPUT -p udp -s $DHCP_SERVER --sport 67 \
   -d $ANYWHERE --dport 68 -j ACCEPT

# Allow 'identd' (to our TCP port 113) from mail server only.
[ -n "$MAIL_SERVER" ] &&\
 $IPTABLES -A INPUT -p tcp -s $MAIL_SERVER  -d $WAN_IP --dport 113 -j ACCEPT

# Open up Public server ports here (available to the world):
for i in $PUBLIC_PORTS; do
 $IPTABLES -A INPUT -p tcp -s $ANYWHERE -d $WAN_IP --dport $i -j ACCEPT
done

# So I can check my home POP3 mailbox from work. Also, so I can ssh
# in to home system. Only allow connections from my workplace's
# various IPs. Everything else is blocked.
$IPTABLES -A INPUT -p tcp -s 255.10.9.8/29 -d $WAN_IP --dport 110 -j ACCEPT

## ICMP (ping)
#
# ICMP rules, allow the bare essential types of ICMP only. Ping
# request is blocked, ie we won't respond to someone else's pings,
# but can still ping out.
$IPTABLES -A INPUT  -p icmp  --icmp-type echo-reply \
   -s $ANYWHERE -d $WAN_IP -j ACCEPT
$IPTABLES -A INPUT  -p icmp  --icmp-type destination-unreachable \
   -s $ANYWHERE -d $WAN_IP -j ACCEPT
$IPTABLES -A INPUT  -p icmp  --icmp-type time-exceeded \
   -s $ANYWHERE -d $WAN_IP -j ACCEPT

# Identd Reject
#
# Special rule to reject (with rst) any identd/auth/port 113
# connections. This will speed up some services that ask for this,
# but don't require it. Be careful, some servers may require this
# one (IRC for instance).
#$IPTABLES -A INPUT -p tcp --dport 113 -j REJECT --reject-with tcp-reset

###################################################################
# Build a custom chain here, and set the default to DROP. All
# other traffic not allowed by the rules above, ultimately will
# wind up here, where it is blocked and logged, unless it passes
# our stateful rules for ESTABLISHED and RELATED connections. Let
# connection tracking do most of the worrying! We add the logging
# ability here with the '-j LOG' target. Outgoing traffic is
# allowed as that is the default policy for the 'output' chain.
# There are no restrictions placed on that in this script.
```

```
# New chain...
$IPTABLES -N DEFAULT
# Use the 'state' module to allow only certain connections based
# on their 'state'.
$IPTABLES -A DEFAULT -m state --state ESTABLISHED,RELATED -j ACCEPT
$IPTABLES -A DEFAULT -m state --state NEW -i ! $WAN_IFACE -j ACCEPT
# Enable logging for anything that gets this far.
$IPTABLES -A DEFAULT -j LOG -m limit --limit 30/minute --log-prefix "Dropping: "
# Now drop it, if it has gotten here.
$IPTABLES -A DEFAULT -j DROP

# This is the 'bottom line' so to speak. Everything winds up
# here, where we bounce it to our custom built 'DEFAULT' chain
# that we defined just above. This is for both the FORWARD and
# INPUT chains.

$IPTABLES -A FORWARD -j DEFAULT
$IPTABLES -A INPUT   -j DEFAULT

echo "Iptables firewall is up `date`."

##-- eof iptables.sh
```

8.10.3. Summary

A quick run down of the some highlights...

We added some host based access control rules: "blacklisted", and "trusted". We then showed several types of service and port based access rules. For instance, we allowed some very restrictive access to bigcat's POP3 server so we could connect only from our workplace. We allowed a very narrow rule for the ISP's DHCP server. This rule only allows one port on one outside IP address to connect to only one of our ports and only via the UDP protocol. This is a very specific rule! We are being specific since there is no reason to allow any other traffic to these ports or from these addresses. Remember our goal is the minimum amount of traffic necessary for our particular situation.

So we made those few exceptions mentioned above, and all other services running on bigcat should be effectively blocked completely from outside connections. These are still happily running on bigcat, but are now safe and sound behind our packet filtering firewall. You probably have other services that fall in this category as well.

We also have a small, home network in the above example. We did not take any steps to block that traffic. So the LAN has access to all services running on bigcat. And it is further "masqueraded", so that it has Internet access (different HOWTO), by manipulating the "forward" chain. And the LAN is still protected by our firewall since it sits behind the firewall. We also didn't impose any restrictive rules on the traffic leaving bigcat. In some situations, this might be a good idea.

Of course, this is just a hypothetical example. Your individual situation is surely different, and would require some changes and likely some additions to the rules above. For instance, if your ISP does not use DHCP (most do not), then that rule would make no sense. PPP works differently and such rules are not needed.

Please don't interpret that running any server as we did in this example is necessarily a "safe" thing to do. We shouldn't do it this way unless a) we really need to and b) we are running the current, safe version, and c) we are able to keep abreast of security related issues that might effect these services. Vigilance and caution are part of our responsibilities here too.

8.10.4. iptables mini−me

Just to demonstrate how succinctly iptables can be configured in a minimalist situation, the below is from the Netfilter team's *Rusty's Really Quick Guide To Packet Filtering*:

"Most people just have a single PPP connection to the Internet, and don't want anyone coming back into their network, or the firewall:"

```
## Insert connection−tracking modules (not needed if built into kernel).
insmod ip_conntrack
insmod ip_conntrack_ftp

## Create chain which blocks new connections, except if coming from inside.
iptables −N block
iptables −A block −m state −−state ESTABLISHED,RELATED −j ACCEPT
iptables −A block −m state −−state NEW −i ! ppp0 −j ACCEPT
iptables −A block −j DROP

## Jump to that chain from INPUT and FORWARD chains.
iptables −A INPUT −j block
iptables −A FORWARD −j block
```

This simple script will allow all outbound connections that we initiate, i.e. any NEW connections (since the default policy of ACCEPT is not changed). Then any connections that are "ESTABLISHED" and "RELATED" to these are also allowed. And, any connections that are not incoming from our WAN side interface, `ppp0`, are also allowed. This would be lo or possibly a LAN interface like eth1. So we can do whatever we want, but no unwanted, incoming connection attempts are allowed from the Internet. None.

This script also demonstrates the creation of a custom chain, defined here as "block", which is used both for the INPUT and FORWARD chains.