14

Safe Hosts in a Hostile Environment

Probably the biggest cause of insecurity on the Internet is that the average host is not reasonably secure when it arrives from the manufacturer. The manufacturers know this, but they tend to focus on features and time-to-market instead of security. A secure computer usually has fewer services, and may be less convenient to use. Unless the product has security as its specific target, security tends to be overlooked. Most people tend to choose convenience over security. (Even reputable "security people" often take shortcuts and cheat a little.)

In this chapter, we supply a definition of "secure," and discuss the characteristics of various Internet hosts that we think meet this definition. Then we can configure a safe host, a *safe haven*, which can be used as a base to administer and manage other hosts.

A collection of such secure hosts can form a safe community using secure network transport. This community should be quite resistant to network attack from outside the community save one threat: denial-of-service attacks, which are discussed in Section 5.8.

14.1 What Do We Mean by "Secure"?

For the next few chapters, we use a restricted meaning for the word "secure" when applied to a host. There is no such thing as absolute security. Whether a host is penetrated depends on the time, money, and risk that an attacker is willing to spend, compared with the time, money, and diligence we are willing to commit to defending the host.

A major problem of Internet security these days is that attackers generally don't have to spend much time or money, and experience virtually no risk, to break into an average Internet server. For example, [Farmer, 1997] provides a survey of major Web servers and their likely network insecurities. Web servers, the most public of hosts, were *more* likely to be running insecure services than other hosts.

We can do better. It is not that difficult to make a specific host highly resistant to anonymous attack from the Internet. The trick is to have that host remain useful.

Non-networked attacks are possible, but are much riskier. The attacker may have to show up on the premises, or pay off our system administrator, or kidnap the CEO's dog. These risks may be worth it to an attacker if the prize is valuable enough, but they are beyond the scope of this book. Here we wish to insist that the attacker must be present to win.

In other words, for now we are saying that a host is "secure" if it cannot be successfully invaded through network access alone. The attacker will have to try something more risky and more traceable.

This is a fairly low standard to shoot for, and your installation may require much higher assurances. What we present here should be a good start. We will leave it to you to post Marine guards, pull the shutters, or take any other additional steps that you need.

14.2 Properties of Secure Hosts

A secure host has time-tested, robust, reliable network services, including the operating system. Its administrators are strongly authenticated and/or need physical access to the host. Other users add weakness, and should be avoided if possible. General access to a secure host should only be permitted only from a very small number of secure hosts in the same community, and their communication should be over private links or use strong encryption. Furthermore, any such access must be restricted to equally secure hosts.

This can be done, even on an open network. It takes careful engineering and a relentlessly paranoid approach.

A user may be authenticated by his or her physical presence in a building, leaving security to the guard at the door, cameras, and suspicious co-workers. He or she may be authenticated by the people who provide physical access to the machine. In some cases, biometrics may be used.

When traveling or calling in from home, a hardware token may be used (see Chapter 7.) It is not sufficient to trust the phone company's ANI ("caller ID") plus a password on a call from home over a phone line; even if you trust the phone switches and the law enforcement policies in your country, phone phreaks can play amazing games. Besides, this makes an employee's home physical security a component of the company's physical security. A spouse, child, or burglar could break this.

Hardware tokens are still the best remote authentication, and we encourage their use, even from home. You probably need keys to get into your home or car—why not to your computer account?

A secure host runs robust network software. It is difficult, and probably impossible, to determine if software is bug-free, but we can make some reasonable assumptions. The following guidelines can offer some indication of software's security:

- Is the program small and simple? Simple programs are more likely to be correct, and hence secure.

A Trusted Computing Base and Open Source Software

In the general computing field, software is seldom written for naked hardware. (It is true that the most common computers in the world are variants of the old Intel 8051, used in cars, thermostats, and so on.) The rest of us program on an operating system, which gives us an environment that helps us get the job done.

A *Trusted Computing Base (TCB)* is a programming environment that we place some trust in to help us remain secure. If our foundation is unsafe, it may not matter how secure the house is. The military envisioned various levels of trust in the famed Orange Book [Brand, 1985], going all the way up to a TCB that has every line of code mathematically proven to be correct.

This is impractical, and perhaps impossible. Even the U.S. Navy skipped this step in designing its “smart ships”: One battle cruiser sat dead in the water for half an hour because its TCB (Windows NT) could not handle an application’s division by zero. Where can we get a decent, inexpensive TCB for our secure hosts?

The surprising answer is that some of the best candidates for TCBs are free. While much of the free software on the Internet is overpriced, there is quality available. The GNU project and the Free Software Foundation have produced some very high-quality software, notably the *gcc* compiler. The GNU tools and other packages such as Perl have enabled other developers to produce more.

When BSDI faced the legal challenge to liberate the Berkeley UNIX source code, several versions of this time-tested kernel became available, including NetBSD, OpenBSD, and FreeBSD. Linus Torvalds wrote his own kernel and gave it away, spawning Debian Linux, Slackware, Red Hat Linux, and more. Each of these has its strengths and weaknesses, but in general they are quite good, and often run for months between reboots—a good sign.

Why can we tend to trust software often maintained by dozens or even thousands of developers? Because we can audit it at our leisure, take a look under the hood, and see how it works. We can find bugs and even recompile it. And thousands of other eyes do as well. While it is true that back doors can be inserted, we have a better chance of finding them. The world helps us audit the software.

But the public does miss errors in such software. Source code is comforting, but it isn’t a panacea.

- Is it widely tested and used? The leading edge is the bleeding edge; let someone else blaze the trail for you, if you can.
- Is source code available? This is not a guarantee—Kerberos version 4 was available in source form for years before an important security bug was found.[1] But it helps.
- Is the author finicky about details? Does the software remain in beta-test for a long time, and with minor tweaks? A careful programmer has better habits, and it shows in the product. Bugs are rare. (Wietse Venema fits this description as well as any one we know.) Of course, software can sit in beta too long. We simply lack the technology to know when the software is absolutely, positively ready.
- A client is more likely to be secure from a directed attack than a server. A server must be available all the time, and deal with any comer. Clients usually run while the user is watching, though of course it is nearly impossible to understand what a complex system is doing. Clients are more likely targets of opportunity, when a Web browser or mail reader encounters some evil software.
- Does the software have a continuing history of security problems? If so, chances are good it will have more, especially if the same developer or developers are working on it. Repeated patches to security-critical code are a bad sign.
- Was security designed into the program from the beginning? Retrofits usually don't fit very well. You want every line of text coded with the thought of attacks in mind. Often, the fundamental design is the most security-critical aspect, and that's difficult to change late in the game.
- How does the author deal with the possibility of buffer overflows? (Dave Presotto, the author of the *upas* mailer [Presotto, 1985], wrote his own string library to avoid such problems—and he did this in 1984, years *before* the Morris Worm called attention to the problem. He wasn't worried about attacks; he just didn't like to write buggy code.)
- Does it run with unnecessary privileges? (On many systems, *xterm*, the standard terminal emulator for X11, runs with root privileges. As the late Fred Grampp once remarked in a similar context, "you don't give privileges to a whale.") Unnecessary privileges often denote a lazy programmer who didn't want to take the time to do things the right way.

A secure host trusts only other secure hosts, and only as far as it needs to. Don't give full access to a remote host, even a trusted one, if lesser access will do. This is the concept of *least privilege*, and it tends to limit vulnerability and damage if attacks do succeed. Carefully question people who say they need full access, and try to find a better solution.

Secure hosts must communicate over secure channels. A channel may be a private serial line or Ethernet. It may be some form of cryptography over a public network. This does not necessarily mean a fully encrypted link, though it can. Sometimes it is good enough to use authenticated and

1. See CERT Advisory CA-96.03

signed messages, with the text in the clear. To our knowledge, use of this last form of cryptography is acceptable to even the most repressive governments, because they can read the messages. They have no acknowledged need to forge messages and interfere with our web of trust.

14.2.1 Secure Clients

Most network interactions on the Internet use the client/server model. A client calls another host for some service. This asymmetry extends to the kinds of computers that are typically used as clients and servers.

Windows and Macintoshes

The most common client is a PC running a recent flavor of Microsoft's Windows. Windows 3.1 was not distributed with TCP/IP network software; you had to buy it separately. Each supplier had its own particular configurations, network servers, and defaults. Most machines were used as clients only, but sometimes ran dangerous server software by default. A port scan of one security specialist's PC discovered an anonymous FTP server on the host—he had no idea it was running, and had to figure out how to shut it off. Such a PC is not a secure host.

Starting with Windows 95, the TCP/IP stack was built into the operating system. These client machines did not have default TCP/IP services turned on, which made the basic host reasonably secure from overt network attack. If file- or print-sharing were turned on, though, various suspect services were started on TCP ports 137–139. This is still true.

A wide variety of things can be done to improve the security of a Windows host. Some services can be turned off or configured for tighter security. There is personal firewall software, which can block external access to services and add a layer of protection. Applications that process content created elsewhere usually have options to turn off dangerous features like macros and execution of remotely supplied programs.

Of course, virus scanners are a vital part of a network-connected component. E-mail from friends may contain viruses, or even be sent by viruses and worms. The great flexibility and vast array of features available on a Windows box offer countless opportunities to corrupt the host, and very few defensive layers are available to contain these threats.

With the introduction of .NET, Microsoft has enabled great flexibility for establishing security policy on Windows machines. The basic idea behind the .NET Framework is that programs are packaged as *assemblies* containing code and metadata. The metadata includes information such as a strong name, based on a public key whose private component was used to sign the code portion. These assemblies are cryptographically sealed containers; the strong names, which consist of a public key and a signature, are used as credentials. In the execution environment, an administrator sets a policy; the policy examines the credentials to determine whether or not to execute the code in the assembly, and if so, which resources the methods can access. Assemblies that are developed using .NET tools are called *managed code* and may be allowed more access to the executing host than other code, depending on whether or not they carry the right credentials. The system examines the execution stack to see if particular method calls are allowed. This is necessary because it is possible for managed code to call into unmanaged code. Thus, the runtime execution environment must examine the call stack to make sure that all of the calls leading up to a particular method call are managed code, and that they all have enough privilege to execute.

The .NET Framework provides powerful tools to control software. At the same time, it introduces all kinds of risks. Code that is signed the right way can execute as trusted local code, regardless of its origin. For example, two business partners in remote areas can put executables on the Web that will run on each others' hosts. This puts quite a value on the private signing keys of those organizations. The trade-off between security and complexity is a recurring theme in this book; .NET takes complexity to new heights. The book that Microsoft put out to explain the security framework [LaMacchia *et al.*, 2002] is 793 pages long. It is filled with warnings to administrators about commands and settings that they should use with extreme caution. Is this safe? In our opinion, .NET provides more rope than any previous environment in such widespread use.

A Macintosh's configuration can vary based on the operating system version and third-party software. OS/X.2 (Jaguar) ships with most services off by default. A glaring exception is the *Rendezvous* service, which implements the mDNS protocol. The purpose of *Rendezvous* is to automatically discover computers, printers, and other devices on an IP network, without requiring user configuration. We suggest you turn this off, unless you really need it. A few other services are on by default, including print server configuration (the IPP protocol), a *syslog* daemon, and a couple of open ports to support *NetInfo*.

Client software can threaten the security of the client; Web browsers leap to mind. These are huge programs with histories of security problems. To minimize these threats to the clients, turn off Java, JavaScript, browser plug-ins, and ActiveX, if you can. Of course, many useful network sites stop working when you do so. A computer that runs foreign programs with faulty or no containment is not secure; the host may be secure if these are disabled.

Single-User, UNIX-Like Systems

Many people have their own workstations or laptops running one of the UNIX-style operating systems, such Linux or FreeBSD. They don't share these machines with anyone. If properly secured and maintained, these are the most trustable clients available. They share files with no one, and allow no logins except through the console. All or most services are turned off (see Section 14.4). But these machines may still run browsers and other elephants.

Sometimes, even local use of local hardware on a workstation, like a video camera, can open the host up to possible attack. SGI hosts accessed their local cameras through a network connection, as user *root*. (Why didn't they use UNIX sockets or shared memory instead of network sockets?) In more recent versions of Irix, they even accessed the DNS resolver through an NFS-style query, opening a number of serious holes in what used to be a securable workstation.

Multi-User Hosts

In our experience, it is hard to make multi-user, general purpose hosts secure. The crowd tends to desire services like NFS, and dislikes strong authentication, preferring the ease of passwords.

We will allow such community machines limited access to secure hosts through carefully configured services. See, for example, our anonymous FTP service in Section 8.7.

14.2.2 Secure Servers

Servers run on many different platforms. At this writing, the fastest and cheapest tend to be UNIX-based, though your religion may vary. We suggest that you select servers that run the operating system you know best. You are less likely to make rookie mistakes, and can concentrate on securing the services.

A safe server runs safe services. This book is mostly about safe and unsafe services. If you can't decide whether you can trust a service, use the list of suggestions in Section 14.2.

A secure server generally has very few users, probably only the administrators. We find that users are a tremendous burden on a system. They complicate and compromise security arrangements. We suggest that you avoid them. It is reasonable to give each administrator a separate account, and monitor the use of the *su* command to help audit changes.

Section 14.4 describes the procedure to secure a UNIX-like client or server.

14.2.3 Secure Routers and Other Network Elements

Like all hosts, routers and similar *network elements* should run only the services they absolutely need. This is especially important given the vital role they play in gluing our networks together. Network elements include routers, switches, hubs, firewalls, cable modems, wireless base stations, dial-in boxes ("NAS"), back-end authentication servers, and so on.

There are several concerns for these devices: administrative access, network services (as usual), and default passwords come to mind. Many network devices are configured once, at installation, and then forgotten. This configuration can be done at the console, a terminal connected to a serial port. Remote access is often not needed unless you are running a large network with geographically diverse equipment. All network services should be shut off. (In some cases, you can shut off SNMP; if you can't, use SNMPv3, with its cryptographic authentication.)

Watch your trust model. We've seen a case where gear that was going to be on customer premises had a wired-in password on all units. If a single Bad Guy reverse-engineered it or wiretapped the management traffic, *all* such units would be vulnerable.

Some network elements do require frequent reconfiguration. These need secure access and strong authentication to remain trustable. At least, change the default administrative password; an astonishing number of important network elements still have the manufacturer's default passwords installed.

14.3 Hardware Configuration

Don't skimp on the hardware supplied for each server machine. A generous hardware configuration will reduce the need to upgrade a system, and reduce the corresponding interruption. In these days of cheap PCs, the hardware costs are nearly zero compared to the cost of competent system administration.

Configure plenty of memory, and make sure that it is easy to get more. It is cheap, improves performance, and provides some resistance to denial-of-service attacks.

Install plenty of disk space: big disks are cheap. FTP, Web pages, spool files, and log files all can take a lot of space, and are likely to grow faster than you think. It is also nice to have spare disk partitions for backup. Large disk partitions are much harder to overflow with network traffic.

14.4 Field-Stripping a Host

UNIX system administration is a nightmare.

—DENNIS M. RITCHIE

A typical UNIX-style system comes with many available network services. If all these services are turned off, and only a very few carefully selected services are installed, such a machine can be highly resistant to invasion from the network. These services may still be susceptible to denial-of-service attacks, and the system's TCP/IP implementation itself might be crashed by carefully crafted packets, but the data and programs on the host are very likely to remain uncorruptible by known or theoretical network hacking methods.

It isn't hard to strip most services from a host; most appear in `/etc/inetd.conf`. The remaining ones come from programs that are started at system boot time.

It has surprised us how often administrators of important hosts have failed to turn off unnecessary services. Even if you think we are too severe in our judgment of the safety of particular services, clearly it is a good idea to turn off those that you don't use.

A number of UNIX-like operating systems are available. The details for field-stripping these vary, but the goal is the same: Remove the network doors into the computer. Some possible options include the following:

Linux There are several versions of Linux. Many allow you to install minimal versions of the system, in which case field-stripping is not required. These can be quite spartan, which is good. Linux system administration details are quite different from the older, commercial UNIX systems.

FreeBSD This BSD variant was designed for server speed. Some of the authors tend to use this one, but it is a close call between it and the other two BSD systems.

OS/X This is Apple's UNIX-based operating system, based on FreeBSD. It provides a platform for running Macintosh programs with nice GUIs, as well as the standard UNIX with X Windows. It is rapidly gaining in popularity.

NetBSD Designed to run on a wide variety of hardware, this is an excellent choice for embedded systems. Note that running something that isn't a SPARC or a Pentium will give you practical immunity to most garden-variety attack-smashing attacks.

OpenBSD The maintainers focus on security issues. Their diligence has helped them avoid some of the vulnerabilities found in other systems. A good choice. Many of the application-level fixes have been ported to Linux and the other BSDs.

Solaris An old UNIX workhorse.

A computer should be configured before connecting it to a network, as it will be running unsafe network services by default. We perform this configuration, and indeed most of its system administration, through its console. Following are the things we do to prepare a UNIX-like host for a hostile environment:

1. Comment out all the lines in `/etc/inetd.conf`. By default, we want none of these services turned on. If a specific one is needed, turn it on. We comment these out, rather than deleting them, because we might want to temporarily turn one on during setup. Figure 14.1 shows a fairly typical `inetd.conf` file before editing.

2. If no services are needed in `/etc/inetd.conf`, disable the call to *inetd*. This program has grown too much over the decades—don't run it if you don't need it.

3. Reboot the machine and run *ps* to make sure that *inetd* is gone. Then run

   ```
   netstat -a
   ```

   (*Netstat* is the best auditing tool in the business.) There will still be network services showing, doubtless served by daemons run in the start-up script.

4. Disable the daemons that run these network services. They will probably include *sendmail* (SMTP), *rpcbind*, *rstatd*, and so on.

5. Reboot and repeat until no unwanted network services are running. At this point, our *netstat* might look like the following:

   ```
   Active Internet connections (including servers)
   Proto Recv-Q Send-Q  Local Address   Foreign Address (state)
   udp        0      0  0.0.0.0.syslog  0.0.0.0.*
   ```

   syslog is a useful program for collecting logs. Most versions can be run without a network listener (switches "-s -s" on FreeBSD.) Many systems want to print documents but don't have a local printer, or need to send but not receive mail. They can be configured to do so without running any network services.

6. When the *netstat* shows what we want, we run a final *ps* to see what processes are running after a fresh reboot. Here's a list from an old SGI Irix system:

   ```
    UID   PID  PPID  C    STIME TTY      TIME CMD
   root     0     0  0 09:55:29 ?        0:01 sched
   root     1     0  0 09:55:29 ?        0:00 /etc/init
   root     2     0  0 09:55:29 ?        0:00 vhand
   root     3     0  0 09:55:29 ?        0:00 bdflush
   root     4     0  0 09:55:29 ?        0:00 munldd
   root     5     0  0 09:55:29 ?        0:00 vfs_sync
   root   342     1  0 09:55:50 tport    0:00 -csh
   ```

```
root      7     0  0 09:55:29 ?       0:00 shaked
root      8     0  0 09:55:29 ?       0:00 xfsd
root      9     0  0 09:55:29 ?       0:00 xfsd
root     10     0  0 09:55:29 ?       0:00 xfsd
root     11     0  0 09:55:29 ?       0:00 xfsd
root     12     0  0 09:55:29 ?       0:00 pdflush
root    343     1  0 09:55:50 ttyd1   0:00 /sbin/getty ttyd1 co_9600
root    130     1  0 09:55:41 ?       0:00 /usr/etc/inetd
root     65     1  0 09:55:35 ?       0:00 /usr/etc/syslogd
root    344     1  0 09:55:50 ttyd2   0:00 /sbin/getty -N ttyd2 co_9600
root    243     1  0 09:55:45 ?       0:00 /sbin/cron
root    364   353  6 10:01:03 tport   0:00 ps -ef
root    353   342  0 09:56:12 tport   0:00 sh
```

Unless you are very familiar with the operating system, there will probably be daemons you don't understand. Most of these are familiar, and we think we (dimly) understand their function. *Shaked* was a new one to us. Its process ID suggests that it is involved with the file system. The man pages say nothing. The string "shake" does not appear in the startup directory.

7. It is also work checking the `/etc/passwd` and `/etc/group` files. Try to figure out the functions of accounts you don't understand. Make sure there are passwords on each account that has a login shell. Accept no default passwords.

8. Check for world-writable files in `/etc`. We once saw a production host heading out the door with world-write permissions on `/etc/group`. There should be no world-writable file in the main executable directories either. Newer systems seem to get this right.

9. Perhaps install IP filtering on the closed ports to ensure that nothing is getting through.

This approach is piecemeal, and not nearly as complete as running something like *COPS*. But a little wandering can turn up some interesting things, and we may not have a compiler on this host, which *COPS* requires.

The kernel may need some reconfiguration. If you aren't using IPv6 yet, it might be a good idea to turn it off in the kernel.

Other changes we might want to make to a secure host include the following:

- Set `/etc/motd` to warn all users that they might be monitored and prosecuted. On a restricted host, warn *all* users that they are not allowed on the machine. The notice about monitoring is considered necessary, or at least helpful, by some legal authorities in some jurisdictions.

- Configure extra disk partitions, and be generous with the space. Remember that the outside world has the ability to fill the logs, spool directory, and FTP directories. Each of these should be in a separate large disk partition.

- Use static routes. Do not run *routed* on the external host: Whose information would you trust, anyway?

```
ftp     stream  tcp nowait  root    /usr/etc/ftpd    ftpd -l
telnet  stream  tcp nowait  root    /usr/etc/telnetd telnetd
shell   stream  tcp nowait  root    /usr/etc/rshd    rshd
login   stream  tcp nowait  root    /usr/etc/rlogind rlogind
exec    stream  tcp nowait  root    /usr/etc/rexecd  rexecd
finger  stream  tcp nowait  guest   /usr/etc/fingerd fingerd
http    stream  tcp nowait  nobody  ?/var/www/server/httpd httpd
wn-http stream tcp nowait nobody ?/var/www/server/wn-httpd ...
bootp   dgram   udp wait    root    /usr/etc/bootp   bootp
tftp dgram udp wait guest /usr/etc/tftpd tftpd -s /usr/local/boot ...
ntalk   dgram   udp wait    root    /usr/etc/talkd   talkd
tcpmux  stream  tcp nowait  root    internal
echo    stream  tcp nowait  root    internal
discard stream  tcp nowait  root    internal
chargen stream  tcp nowait  root    internal
daytime stream  tcp nowait  root    internal
time    stream  tcp nowait  root    internal
echo    dgram   udp wait    root    internal
discard dgram   udp wait    root    internal
chargen dgram   udp wait    root    internal
daytime dgram   udp wait    root    internal
time    dgram   udp wait    root    internal
sgi-dgl stream  tcp nowait  root/rcv /usr/etc/dgld   dgld -IM -tDGLTSOCKET
#uucp   stream  tcp nowait  root    /usr/lib/uucp/uucpd     uucpd
# RPC-based services: These use rpcbind instead of /etc/services.
mountd/1    stream  rpc/tcp wait/lc root /usr/etc/rpc.mountd     mountd
mountd/1    dgram   rpc/udp wait/lc root /usr/etc/rpc.mountd     mountd
sgi_mountd/1 stream rpc/tcp wait/lc root /usr/etc/rpc.mountd     mountd
sgi_mountd/1 dgram  rpc/udp wait/lc root /usr/etc/rpc.mountd     mountd
rstatd/1-3  dgram   rpc/udp wait    root /usr/etc/rpc.rstatd     rstatd
walld/1     dgram   rpc/udp wait    root /usr/etc/rpc.rwalld     rwalld
rusersd/1   dgram   rpc/udp wait    root /usr/etc/rpc.rusersd    rusersd
rquotad/1   dgram   rpc/udp wait    root /usr/etc/rpc.rquotad    rquotad
sprayd/1    dgram   rpc/udp wait    root /usr/etc/rpc.sprayd     sprayd
bootparam/1 dgram   rpc/udp wait    root /usr/etc/rpc.bootparamd bootparam
#ypupdated and rexd are somewhat insecure, and not really necessary
#ypupdated/1 stream  rpc/tcp wait   root /usr/etc/rpc.ypupdated  ypupdated
#rexd/1     stream  rpc/tcp wait    root /usr/etc/rpc.rexd       rexd
sgi_videod/1 stream rpc/tcp wait    root ?/usr/etc/videod         videod
sgi_fam/1   stream  rpc/tcp wait    root ?/usr/etc/fam            fam
#sgi_toolkitbus/1 stream rpc/tcp wait root/rcv /usr/etc/rpc.toolkitbus ...
sgi_snoopd/1 stream rpc/tcp wait    root ?/usr/etc/rpc.snoopd     snoopd
sgi_pcsd/1  dgram   rpc/udp wait    root ?/usr/etc/cvpcsd        pcsd
sgi_pod/1   stream  rpc/tcp wait    root ?/usr/etc/podd           podd
sgi_xfsmd/1 stream  rpc/tcp wait    root ?/usr/etc/xfsmd     xfsmd
ttdbserverd/1 stream rpc/tcp wait root ?/usr/etc/rpc.ttdbserverd rpc.ttdbserverd
# TCPMUX based services
tcpmux/sgi_scanner stream tcp nowait root   ?/usr/lib/scan/net/scannerd scannerd
tcpmux/sgi_printer stream tcp nowait root   ?/usr/lib/print/printerd printerd
```

Figure 14.1: The default `/etc/inetd.conf` file for Irix 6.2. Do any of these programs running as *root* have security problems? (Some lines were cut short and comments edited to fit the page.)

- Take a full dump of the host, and save the tapes or CD-ROMs forever. Make sure they are readable. Do this before plugging in the cable that allows external access for the first time. These are “day-zero backups,” and they are your last resort if someone breaks into your machines.

14.5 Loading New Software

Where do you get new software from? Whether it’s *OpenSSH*, a Web browser, $\LaTeX$, or desktop synchronization software, most people download programs from the net. In fact, there are very convenient programs, such as *dselect* and *fink* on Linux and OS/X, respectively, that can keep track of which packages you have on your machine, and provide a convenient way to download, install, and configure new programs in a few simple steps. Linux programs are distributed in convenient *Red Hat Package Manager (RPM)* archives. Often, these contain binaries. Programs for Windows and the Macintosh are distributed as similar self-extracting packages.

The *ports collection* for FreeBSD contains almost 4,000 programs—packages that people download from the net. The packages often come with checksums, but of course these only guarantee that the download matches the checksum; they say nothing about whether or not the code is malicious. An attacker can modify checksums that came from the same site as the download—checksums stored elsewhere require more work. Sometimes, for security reasons, the managers of the ports collection make changes to standard packages. An example of this is a package called *Xbreaky*, which had a `setuid` bit set. The FreeBSD and NetBSD ports patched the installation files to turn off that bit. That was fortunate, because it turned out to have a security hole. Interestingly, OpenBSD, which is supposed to be the most secure, did not catch this.

Digital signatures could help, in theory [Rubin, 1995]. Microsoft does this with ActiveX. However, they require that end hosts have the public key of the code signers, along with programs for checking signatures. A difficult question is who should sign the code. If the authors sign it, then the archive cannot make any changes to it, and the public key distribution problem is more difficult. If archive maintainers sign code, then they have to verify that it is not malicious. Their signature means that the code has not changed since they signed it, but that does not mean that the code writer was not malicious, nor that the code was not modified before the person actually signed it. In other words, digital signatures at most provide accountability, not security.

There are those who maintain that it is safer to distribute source code than binaries. We caution against taking this assumption too far. Perhaps it is true that because many people are likely to download the program, and *some* of them might actually look at the code, and *some* of them might actually be qualified to tell if there are security problems, that it is safer to compile your own source than to download binaries. However, there is nothing inherently safer about source code, and you can compile a Trojan horse on your machine just as easily as the attacker can on his or hers. Answer this: Have you read and understood the source code to, say, Apache, the popular open-source Web server? Hint: There are over 1,000 files, comprising more than 300,000 lines of code.

14.6 Administering a Secure Host

Secure hosts provide special problems for the system administrator. Stronger security usually makes system administration less convenient, as usual. At least the sysadmin doesn't have to meet the access demands of a large user community, because these hosts seldom have many direct users. Of course, many people may depend on the proper functioning of, for example, a KDC.

14.6.1 Access

System administrators need access to secure hosts, often from their homes and at late hours. Because a secure host is usually an important one, they rightly point out that a troubled system will be down until they can gain access to it.

Similarly, an ISP needs access to far-flung routers and other network elements. The most common monitoring method is SNMP, and that's a risky choice. (See Section 3.6 for a discussion of the protocol.) Even read-only SNMP access to a firewall's configuration information can leak information useful to the attacker. You should disable SNMP write access if you can; it's rarely a useful way to configure a network element. SNMPv3 is a much better choice, as it has strong security built in; if you can't run it, use packet filters to prevent outsiders from sending SNMP packets to your network elements. You need SNMP access; the Bad Guys don't.

Many routers accept *telnet* sessions, but the risks of that are obvious. You can often use *ssh*, a better choice.

Conversely, access through the *Public Switched Telephone Network (PSTN)* can expose the router to phone-based attacks, unless strong authentication is available. Besides, your network management software probably can't talk over a serial line.

By far, the safest way to access a secure host is through its physical console, at the machine itself. This reduces access security to the realm of physical security.

If physical access is not feasible, telephone access through a modem to the serial port, with strong authentication, is the next best choice. (You may need that anyway, for emergency access when your network is having a bad hair day.) The calling machine or terminal must be secure, of course. In this case, where does the host keep the keys needed for strong authentication? If it has to connect to an authentication server over a network, how do you access it if the network is down?

Ssh is probably a reasonable choice if the calling host itself is secure. Remember that there are hacking tools that can take over a user's keyboard on a multi-user host if the host has been compromised.

Other protocols, such as IPsec, newer versions of SNMP, or perhaps even encrypted PPTP may be an option. In any case, you should carefully consider the consequences if your access method is compromised.

14.6.2 Console Access

One can sit at the console itself, though these often reside in noisy computer rooms. System administration is better performed in a quieter, more relaxing atmosphere. Often, several computers share a console through some sort of serial or video switch. This enables us to stack the computers in a rack, with a single terminal or display head.

Here we rely on physical security to protect the host, which is reasonable. Sometimes our host will even lack a root password: If someone can touch the computer, we have already lost. This assumes that there are no other ways to log in to the host, which is true for most of the computers we leave in a dirty environment. It doesn't hurt to have the extra layer—the password—to further protect us. But that may not add very much. Use of an empty *root* password focuses the mind on security wonderfully.

When console access is through a remote serial line, it should be protected by some strong authentication. It is reasonable to require a one-time password for incoming phone access to a console.

We used to have a simple RS-232 hardware switch installed that selected between a local console terminal and the remote dial-up access. Console server software allows multiple administrators to connect to the same port simultaneously. One quickly develops a protocol to avoid stepping on one another's work. There are a number of fine commercial console servers; a nice free one is available from `http://www.conserver.com`.

It is important not to connect to these consoles from a compromised host. If someone taps that session, the outside machine is breached. You don't want your console session hijacked.

Physical access to the console is less convenient for system administration, but should be impossible for a typical hacker. And many secure hosts don't require frequent access after they have been set up. Again, though, you need to balance that against the requirements for availability.

14.6.3 Logging

Logging is essential when administering a host in a hostile environment. It tells us what is going on, and may be essential to forensics. When attackers break into a machine, the first thing they go for are the logs. Therefore, it is important to ensure that the logs are robust against attack. The best way to do that is to make the logs unmodifiable from the machine. For example, burning them onto CDs periodically guarantees that the attacker will not be able to erase or delete them.

Logging to a drop safe is a great idea—bytes check in but they don't check out. *Syslog* has a nice facility for doing this, by sending the log messages to another machine for safekeeping. One problem that is not avoided by write-only logs is that attackers can create so many logged events that they fill the disk and further logging is unsuccessful. You may be able to avoid this with a disk that is large enough. (Attackers may also try to talk directly to your log server. Be sure that your filter rules prevent this.)

What do you do with all those logs? If you are an expert, you can look at them yourself. You can write or acquire tools for parsing log files into more readable form. There are also commercial companies to whom you can send your logs, and they will help you determine if you are under attack. While this is not very useful for real-time attack detection, there is some value to knowing that someone was trying to break in, even if they were unsuccessful. Moreover, if the logs are append-only (so an invader cannot change them), they can be useful for *post mortem* analysis.

For log processing, it is very important to have time synchronized among your machines. Even a few seconds of skew can really mess things up. NTP is well-suited for this.

14.6.4 Backup

Backups are always important, but safe hosts often have special backup needs. If there is the slightest chance that they may be hacked, it is invaluable to have a dump of the system made before the hackers touched the machine. This *day-zero backup* is a source of clean binaries, useful for checksum and comparison with newer, possibly modified files.

A day-zero backup should be taken before a host sees its first network packet, and additional full dumps made after patches or other major updates to the system. These backups should be stored well out of harm's way, and should be kept until the system is decommissioned.

They also should be checked. We know of one site some years ago that religiously backed up their data to the video track of a VCR—but the data was supposed to go to the audio track. Every backup was useless; too often, problems like this are not discovered until the backup is needed. (We still have painful memories of an all-night session rebuilding a system whose disk controller died 30 minutes *after* the backup tapes were found to be useless, and 30 minutes *before* the new emergency backup was to be taken.)

Backups can be made with *dump* or *tar*, compressed, and written on a large empty partition on the local disk. This file can be shipped to safer places via *scp*.

Backups can be written to a local tape drive. A newer option is backup to CD. These are a handy and relatively permanent form of storage. Of course, the (possibly compressed) data has to fit on the CD. DVDs hold more data, but they're expensive. Besides, the standards seem to be in a state of flux; you may not be able to read your old backups in a few years.

A computer should be backed up to some media off the machine, and perhaps off-site. The frequency of backup varies depending on how often important things change on the host. We have had some network servers that we back up once a year. The basic software does not change. It is easy to forget, though, and it is better to back up too often than not enough.

Most backups are needed because the system administrator made a mistake. A file may be accidentally edited or deleted. These boneheaded errors happen to all of us on occasion. A nightly backup to a separate partition on the same computer can save the administrator an embarrassing walk to the backup tapes. It is reasonable to us *dd* to back up the root partition to an empty partition. Make sure the backup partition is bootable.

Important binaries are often copied before they are updated, providing an easy recovery path:

```
mv inetd oinetd &&  mv ninetd inetd
```

Another point to consider is the physical security of the backup media. You probably want to keep off-site copies; however, if Bad Guys get their hands on a backup, they'll be able to read sensitive files, possibly including secret cryptographic keys.

14.6.5 Software Updates

The software in these trusted hosts needs to be updated. While it is true that we have left little exposed to the elements, sometimes important updates have to be installed. This is especially true for network services like *sshd*. We count on this service a lot, and sometimes a serious flaw is found.

How do we update a safe-haven host? We can update software from a trusted CD-ROM, or install new ROMs in network elements. This last approach offers high assurance that you are getting the code you expect, but it risks hardware problems. ROM updates are falling out of favor—ROMs have mostly been replaced by flash memory now, with software updates. (The thought of what a piece of malware can do to a flash-resident BIOS is scary.)

We can copy new software out to relevant hosts using encrypted links. Many use *rdist* or *rsync* over *ssh* links.

The client can attach to a network server to obtain updates. This is dangerous: How does the client know it is connecting to the correct server? Has the server been compromised, and now contain modified software? Did the software support team add back doors or other security holes to the software? If the vendor or the connection path is compromised, the local client will import Trojan software, and the client is lost.

This *client pull* approach is used across the industry: Netscape, Microsoft, Linux, the BSD systems, Mac OS/X, and others like the FreeBSD "ports" collection all obtain their software from remote servers. This software is compiled and installed with high system privileges. Certificates and checksums are available to mitigate these problems, but they are often ignored.

Though client pull has dangers, its simplicity is a strong plus, especially for client hosts owned by naïve computer users. We think the advantages far outweigh the risks for standard hosts, but they are quite dangerous for the safe-haven hosts we are relying upon.

When software updates are automated without user control, there are inherent risks. How do you know that the update, which is perhaps being distributed because of a security flaw, does not have a flaw itself? Programs such as *RealPlayer* for Windows often make users' lives miserable until they agree to upgrade to newer versions. You have to go through at least three different pop-up windows every time you run the program if an update is available. Software that insists on updating itself is a pain. Software that continuously updates itself without informing the user is dangerous and downright impolite. An extreme example of this is the TiVo video recorder: When the company updates the operating system, it automatically downloads a new system image to all users, along with a message indicating where to find the new user manual for the new features. Users are given no choice about upgrading.

When you are given a choice about updating software, there are several things to consider. There is really no way to understand all of the patches that a vendor issues, not just for the average user, but even for advanced programmers and administrators. If a machine is a production server, you need to test it in a lab. For home machines, perhaps you should test the update on a less important machine before putting it on the machine that you use to do your taxes. In the U.S., you don't want to do *anything* to that machine on April 14 if you have not filed your tax return yet.

Some software comes with license agreements that specify update policies. For example, Windows Media Player states that Microsoft has the right to remotely change the software on your machine if they believe that there is a *digital rights management (DRM)* violation. That is, if Microsoft suspects that there is a way to defeat the copy protection of content, they have the right to change the software on the customers' machines, without the customer's consent. In other words, when you install software on important machines, you should look at the fine print in the agreements to ensure that not only will you make the decision about when to upgrade, but that you will have the opportunity to make a decision at all.

Almost no one takes the time to read and try to understand the click-through license agreements.

How often should software on a minimal, high-security system be updated? There is a tension here. Updates take time, and mistakes can open unintended holes. If the system is running no network services, but is just routing packets, the original software might be good enough. This is certainly not true for most network services; flaws are eventually found, and the software needs to be updated. Most successful system cracks involve well-known problems for which patches exist.

When a security flaw is found in a vital network service, it has to be fixed quickly. If the operating system hasn't been kept up-to-date, a sudden upgrade may require changes and installations that would have been better done at a quieter time. Conversely, a patch has a 20% chance of being wrong or needing further modifications—see the discussion of optimal timing in [Beattie *et al.*, 2002].

Network administrators have to keep up with software releases of their vital servers as well. For example, we watched and waited for security holes in *bind* to appear. It is an essential service, a persistent daemon, and tends to run as *root*. A hole would have a widespread affect on critical services, a ripe arena for the propagation of worms. Furthermore, DNS is a service that *must* be available to random Internet hosts. CERT Advisory CA-1998-05, "Multiple Vulnerabilities in BIND," was issued on 8 April 1998. How fast did people upgrade their critical software?

We started a scan of *bind* version numbers about two months after the CERT advisories. We checked the versions of *bind* on some 1,000 name servers for a year and a half to examine the propagation of safe software on critical services. The results are shown in Figure 14.2. Niels Provos and Peter Honeyman [2001] have run a similar analysis of dangerous *ssh* servers at the University of Michigan. It takes a while for people to catch up, even when the upgrade is vital.

Finally, the initial patches to a severe problem may be flawed themselves, requiring repeated updates. For example, CERT Advisory CA-2002-18 reported a serious problem with *OpenSSH*. Four levels of patches came out within three weeks of the original announcement, and it turned out that some of the patches also included a Trojan horse (see CERT Advisory CA-2002-24)! Deciding when it is right to install patches to software is a tough judgment call.

14.6.6 Watching the Roost

We should monitor our safe-haven hosts. Do they emit unusual packets? Have important files changed? Do the logs have unusual entries?

A number of programs watch systems and the networks around them. Programs such as *Tripwire* can check for modified files on a host.

Programs like *snort*, *clog*, and even *tcpdump* can watch network traffic fairly simply. They can discard expected traffic and report unusual activity. Chapter 15 covers this in more detail.

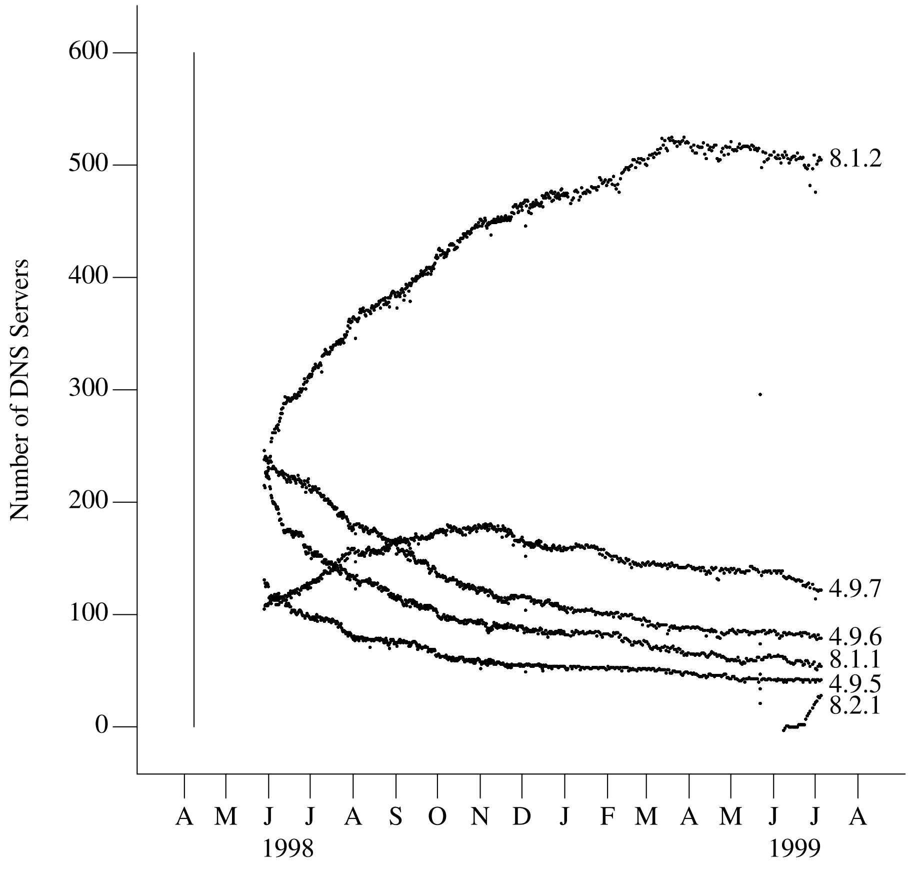

Figure 14.2: Versions of *bind* running on a number of hosts following the announcement of a major security hole. The security hole appeared in versions 4.9.5, 4.9.6, 4.9.7, and 8.1.1. Even though the scans started some two months after the bugs were announced, the adoption curves are clear.

14.7 Skinny-Dipping: Life Without a Firewall

If your safe client is sufficiently attack-resistant, and your network access needs are well-defined and well-constrained, it is feasible to connect safely to the Internet without a firewall. Connecting to the Internet without a firewall is like skinny-dipping: some unusual extra freedom, but with an added element of danger. It focuses the security-minded wonderfully.

Such hosts run few or no network servers: *ssh* may be it for incoming connections. If the system is used to read mail or browse the Web, these programs should be too stupid to run viruses, plug-ins, Java, JavaScript, or anything else imported from the outside world. In fact, these programs should be run jailed, which is difficult and inconvenient. Better kernel support for running untrusted clients is needed for nearly all current operating systems.

The lack of firewall does allow unusual testing and services.