8

Using Some Tools and Services

Chapter 2 probably convinced you that we don't think much of the security of most standard network services. Very few fit our definition of "secure." We have three options:

- Live with the standard services we trust
- Build new ones that are more likely to be secure
- Find a way to tame those unsafe, but useful services

Note carefully our use of the word "service." By it, we include both the protocols and their common implementations. Sometimes the protocol itself is unsafe—reread Chapter 2, if necessary—but sometimes the problem is with the existing code base.

The first option limits us too much; there are very few standard or *Commercial Off-The-Shelf (COTS)* programs we trust. The second is a bit more appealing, but is not practical for everyone. If nothing else, writing secure code for a complex protocol is hard; even someone with the time and the will won't necessarily produce better code than the existing options provide.

In this chapter, we will tame some existing services, option 3. Most people hold their noses and use option 1, with a very broad or naïve definition of trust. Some opt for option 2, building it themselves. Great care must be taken, and few are qualified to do it right.

Note that we have not considered the option of running unsafe services behind a firewall. This does not make the host secure: it is still vulnerable to anyone with access to it.

8.1 *Inetd*—Network Services

Inetd is a general tool for launching network servers in response to incoming connections. It can launch a variety of services: UDP, TCP, RPC, and others. *Inetd* runs under account *root* because it usually listens to services in the privileged range and needs to run server programs under lesser accounts. A number of simple services can be processed by *inetd* itself.

This model is attractive from the standpoint of security and simplicity. Server programs often don't need explicit networking code—*inetd* connects the process to the network socket through standard input, standard output, and standard error. The process does not need to run as *root*, and we can further restrict the program through other programs such as TCP wrappers.

Typically, *inetd* launches a new instantiation of a server program for each incoming connection. This works for low-volume network services, but can become a problem under load, though modern computers can handle a remarkable number of connections per second using this model. Most *inetd* implementations—a number are available—allow limitations on network connection rates.

The standard *inetd* program has grown over the years. There is the rate-limiting code mentioned above, an internal TCP wrapper, IPsec security, and, of course, IPv6 support—some 3,000 lines of C code in all. Some of this complexity is not needed, and all of it is worrisome: We like to rely on *inetd* on some pretty important hosts. Historically, some versions of *inetd* have had a few bugs that can shut services down, but none we know of have had security problems.

8.2 *Ssh*—Terminal and File Access

Ssh is now a vital part of our security toolkit (see Sections 3.5.3 and 18.4.1). Though we are a little leery of it, it provides vital and probably robust end-to-end encryption for our most important problems. The reason our enthusiasm is not absolute is that *ssh* is so feature-rich that its inherent complexity is bound to introduce flaws in implementation and administration. Version 1 of the protocol was in widespread use when it was found to be insecure. Even version 2 has been found to be susceptible to statistical timing attacks [Song *et al.*, 2001]. To accommodate cryptosystem block sizes, *ssh* version 2 rounds up each packet to an eight-byte boundary. In interactive mode, every keystroke that a user types generates a new IP packet of distinctive size and timing, and packets containing password characters produce no echo packets. These properties help the attacker infer the size of passwords and statistical information that amounts to about one bit per packet.

We rely on *ssh* for interactive connections between hosts and for file transport. Besides *scp*, a number of important file transport programs—such as *rsync* and *rdist*— can use *ssh*. For these connections, it is important to configure the authentication correctly. Because they usually run in scripts when a user isn't present to supply a password, these need single-factor authentication: a key. For interactive authentication, we can use two-factor authentication.

The details of configuration are important. We refer to version 2 authentication methods and configurations in this section, as implemented in *OpenSSH*.

8.2.1 Single-Factor Authentication for *ssh*

Ssh has multiple configuration options. One form of authentication is HostbasedAuthentication or RhostsRSAAuthentication. This mimics the old BSD-style authentication used for *rlogin*/*rsh*, but in a much stronger way. Connection is granted if it comes from the proper IP address, has the appropriate host key, and the IP address appears in system- or user-supplied `hosts.equiv` or

Evaluating Server Software

Programming is hard to do, and safe programming is very hard to do. It's even harder to prove that a program is safe and secure. This is an open area for research.

But we can look for some indications of how the programmers approached their task. We can look for outright bugs or indications of trouble. If we find them, we lose confidence in the software. If we don't find them, or see signs of rigorous and systematic paranoia, we may gain some confidence, especially if the software has proved itself over time. What decreases our confidence in a piece of software?

- Lack of source code and a good compiler
- Dangerous programming languages. C certainly qualifies, though there have been security problems in type-safe languages.
- Long programs and numerous features. Less is more.
- Servers running as *root* that don't relinquish permissions as soon as they can
- Large configuration languages that are processed before privileges are reduced
- In C, the use of *gets*, *strcpy*, *strcat*, and *sprintf*, among others. All but the first can be used safely with very careful programming and numerous checks, but there are safer versions of each.
- Compilation warning messages
- The use of deprecated language features and libraries
- In C, excessive use of `#ifdefs` [Spencer and Collyer, 1992]. Programs should not be woven, unless they are literate [Knuth, 2001].
- A history of bugs

These are rough heuristics. Many attempts have been made to create formally secure languages and programs over the past 40 years. It would be very useful to continue these efforts with a special eye toward making safer network services.

Programming is hard.

`.rhosts`/`.shosts` files. We don't advise that you let your users make security policy, so the `sshd_config` file might have the following:

```
HostbasedAuthentication yes
IgnoreRhosts yes
IgnoreUserKnownHosts yes
PasswordAuthentication no
RhostsAuthentication no     # protocol 1 only
RhostsRSAAuthentication yes # protocol 1 only
```

As written, this authentication trusts any user on the client. DenyUsers and AllowUsers can be used to modify this trust a bit. This authentication depends on a constant IP address for the client, which probably won't do for a traveling laptop. This IP dependence probably *adds* a little security, as the host key, if stolen, can't be used from another host without IP spoofing. Of course, if the attacker can steal your host private key, you've probably already lost control of the host itself.

We can remove this IP dependence using DSA or RSA authentication. This is based on the presence of a private key in a user's key ring. It cannot be combined with the IP-based authentication—*ssh* tries one, then the other.

For DSA authentication with UNIX clients, we generate a key on the client:

```
ssh-keygen -t dsa
```

which puts a public/private key pair in `.ssh/id_dsa.pub` and `.ssh/id_dsa`, respectively. (Use `-t rsa` for RSA keys.) *ssh-keygen* asks for a password to lock this key entry; it must be empty for single factor authentication. Append `id_dsa.pub` to `.ssh/authorized_keys2` on the server, and add

```
DSAAuthentication yes
```

to both the client and server *ssh* configuration files.

The server now trusts the client using single-factor authentication. This trust is often asymmetric: The client may be at a higher trust level than the server. Automated scripts can now run *ssh*, *scp*, and other programs that use them, like *rsync*, without human intervention. Access to the server can be limited by restricting the programs it will run. This could be used to allow users to provision parts of a Web server or FTP archive on a DMZ without having access to the whole server.

Either of these authentication methods is better than nothing, even between relatively insecure clients and servers. These tools are a good first step toward tightening the security of these hosts and their communications, and routine encryption of low-priority traffic can make it harder for an eavesdropper to identify the high-value data streams and hosts. It is worthwhile even if only password authentication is used, as it masks some (but not all) of the information about the password.

8.2.2 Two-Factor Authentication

The single-factor authentication described above is fine if the client is highly unlikely to be compromised. *Ssh* does support various two-factor authentication schemes, though there are a bewildering array of options.

The second factor is a passphrase that must be entered. We must ask where the information needed to process that phrase is stored. If an attacker can find a way to mount a dictionary attack on the phrase, the security of the system is diminished considerably, because people pick lousy passwords.

For example, the DSA key mentioned in the previous section can be protected by a passphrase if we want two-factor authentication. The passphrase unlocks the key, which is then used to connect to the server. If the key resides on a laptop that is stolen, a passphrase may be the only obstacle protecting the server, at least until the theft is noticed.

Can the attacker run a dictionary attack on the passphrase? To do so, the attacker's program needs to determine if each guess is correct. Does the format of the key file enable the program to determine if it made the right guess? The *ssh* designers could go either way. They could make any guess produce a bit string that might be correct, with no way to verify the correctness other than actually connecting to the server and trying. This means the server would retain control over its incoming authentication queries. Replies could be limited to a few tries, attempts logged, and the access shut out. These are nice security properties, but they are confusing to the user. An authorized user who mistyped the passphrase would be denied access, and it would be harder to figure out why. User support has considerable costs.

The *ssh* designers picked the second option: A passphrase can be checked for validity immediately, without connecting to the server. This simplifies support issues. Moreover, the original public DSA key is probably still on the client host, without protection, so attackers could verify the key themselves, though with considerably more computing costs.

The passphrase improves the security of DSA authentication, but we have seen that it would be better to have the password processed off-machine. *Ssh* offers options for this. It supports Kerberos, which stores the password elsewhere, but it is not clear that this can be combined with a required host or DSA key—we have not tried it. Password authentication plus DSA authentication would do the trick, but *ssh* doesn't support the combination. The password checking would be performed by the server, which could check for dictionary attacks. Similarly OTP authentication is supported, but only as a single authentication method. The OTP printout is only a single factor, something you have. If it is implemented in a palmtop computer, for example, it can be true two-factor authentication.

Ssh does support some authentication tokens, and it is easy to modify the server to support others. These can provide genuine two-factor authentication on their own.

8.2.3 Authentication Shortcomings

Even with all these options, *ssh* doesn't allow us to implement some of the policies we think are best.

Oddly, *ssh* does not support known host plus password authentication. If the calling computer has an unknown host key, we might wish to enforce two-factor authentication by using an

authentication device (see Section 7.3). These permit a challenge/response authentication that gives us a two-factor authentication, and *ssh* can support this, but not based on whether the calling host is known or not. Of course, an unknown host may be untrusted for good reason.

Some versions of *ssh* support *Pluggable Authentication Modules (PAMs)*, which could probably be configured to implement the policies we desire. Alas, PAM is not always supported by *ssh*, and the `UsePrivilegeSeparation` option makes this implementation more difficult.

The real problem is that these different authentication methods are not orthogonal. This leads to complexity both in the code and in trying to administer such a system. We'd be happier if the administrator could configure authentication "chains," conditional on the source IP address:

```
10.0.0.0/8: RSA | RhostsRSAAuthentication Password

*: RSA | RhostsRSAAuthentication Kerberos
```

Note that this address–based authentication is very different from the IP address-based authentication we decry for the *r-* commands in Section 3.5.2. Those commands rely solely on the IP address for authentication. Here, the IP address is used for *identification*, but authentication is based on the possession of a strong cryptographic key.

8.2.4 Server Authentication

When using *ssh*, it's important that the client authenticate the server, too. There are existing tools, such as *sshmitm* and *ettercap*, that let an attacker hijack an *ssh* session. Users are warned about this—they're told that the server's public key is unknown or doesn't match—but most people ignore these warnings. This is an especially serious matter if passwords are being used. You may wish to consider using

```
IgnoreUserKnownHosts yes
```

if your user population can't be trusted to do the right thing.

8.3 *Syslog*

Syslog, written by Eric Allman, is useful for managing the various logs. It has a variety of features: the writes are atomic (i.e., they won't intermix output with other logging activities), particular logs can be recorded in several places simultaneously, logging can go off-machine, and it is a well-known tool with a standard format. The *syslogd* daemon listens for log entries on a local pipe and, optionally, from a UDP port.

The program has been a source of worry: it runs as *root*, and is used on vital hosts. There has been a serious advisory on it (see CERT Advisory CA-95:13) of the usual stack-smashing kind; see Section 5.3. Many versions let you turn off the network listener (check your local documentation; the magic letter differs from system to system); you should do this on important hosts. If your version doesn't let you turn off UDP access to it, download, compile, and install a version that does.

Syslog's UDP packets can get lost on the wire and in the kernel. There's a move afoot to document the *syslog* protocol as a standard, and add reliable delivery to it; see RFC 3195 [New and Rose, 2001].

Besides being safer, it eliminates a potential denial-of-service attack. A vandal who sends 100 KB/sec of phony log messages would fill up a 200 MB disk partition in about half an hour. That would be a lovely prelude to an attack. Make sure that your filters do not let that happen.

It is often a good idea to keep your files in an off-machine *logging drop safe*. Hackers generally go after the log files before they do anything else, even before they plant their back doors and Trojan horses. You're much more likely to detect any successful intrusions if the log files are on the protected inside machine.

8.4 Network Administration Tools

This topic is vast, and so are the number of tools available for network administration. The following sections describe a couple of standbys worth mentioning.

8.4.1 Network Monitoring

It is a difficult job to police and understand Internet traffic. There can be billions of packets involving millions of players. The packet rates can challenge the latest hardware running highly efficient software. Fortunately, most of the traffic is stereotypical: We can understand much of what's going on and ignore it, focusing on the unusual packets. Chapter 15 examines this problem in some detail.

We can monitor a network from a host that is actually under attack, or even compromised, but it is not a good idea—it is better to pick another host with access to the packet flow. It is even better if this host does not interact with the network, as sniffing computers usually run in promiscuous mode. Dave Wagner suggested some techniques developed by students in his class for detecting hosts in promiscuous mode (they often respond to packets that they shouldn't see) [Wu and Wong, 1998], and there are tools available, such as L0pht's AntiSniff.

8.4.2 Using Tcpdump

By far, the best alternative is external monitoring à la *The Cuckoo's Egg* [Stoll, 1989, 1988]. For network monitoring, we recommend the *tcpdump* program. Though its primary purpose is protocol analysis—and, indeed, it provides lovely translations of most important network protocols—it can also record every packet going across the wire. Equally important, it can refrain from recording them; *tcpdump* includes a rich language to specify what packets should be recorded.

The raw output from *tcpdump* isn't too useful for intrusion monitoring—several simultaneous conversations may be intermixed in the output file. You can find a number of publicly available tools to process *tcpdump* data—Stephen Northcutt's *Shadow* IDS is a good example.

Some monitoring tools have contained security holes—special packets can crash or even subvert the monitoring host! All of these monitoring programs share another common danger: The very kernel driver that allows them to monitor the Net can be abused by

Those With Evil Intentions to do their own monitoring—and their monitoring is usually geared toward password collection or connection hijacking. You may want to consider omitting such device drivers from any machine that does not absolutely need it. But do so thoroughly; many modern systems include the capability to load new drivers at runtime. If you can, delete that capability as well. (If you can't delete that capability, consider using a different operating system for such tasks.)

Conversely, if you have any unprotected machines on your DMZ net—for example, experimental machines—you must protect yourself from eavesdropping attacks launched from those systems. Any passwords typed by your users on outgoing calls (or any passwords you type when administering the gateway machine) are exposed on the path from the inside router to the regional net's router; these could easily be picked up by a compromised host on that net. The easiest way to stop this is to install a *filtering bridge* or a "smart" hub to isolate the experimental machines. Figure 8.1 shows how a DMZ net could be modified to accomplish this.

Note well: Such bridges, hubs, and switches are generally *not* designed as security devices, and should not be relied upon. There are many well-known ways to subvert the filtering, such as sending to or from sufficiently many MAC addresses that you overflow the filtering tables, or engaging in ARP-spoofing. If you're serious, you need a dedicated network tap, such as those made by NetOptics or Finisar. If you don't want to go that far, use a separate router port.

Another popular monitoring program is *ethereal*, which features a GUI interface that reminds us of some commercial network monitoring devices.

8.4.3 Ping, Traceroute, and Dig

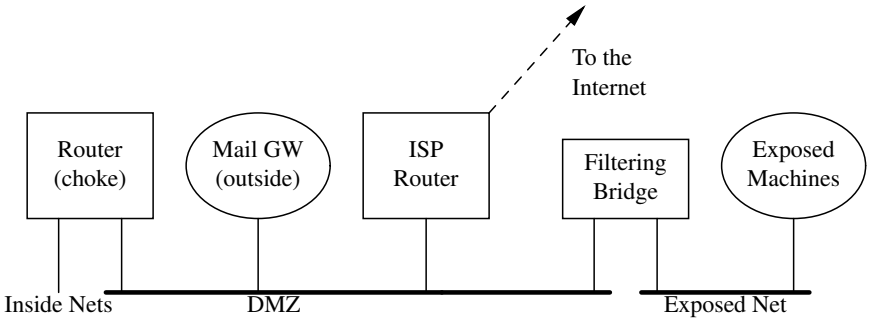
Although not principally security tools, the *ping* and *traceroute* programs have been useful in tracing packets back to their source. *Ping* primarily establishes connectivity. It indicates whether or not hosts are reachable, and it will often tell you what the problem is if you cannot get through.

Traceroute is more verbose; it shows each hop along the path to a destination. It sends out packets with increasing *time-to-live (TTL)* fields. This field is decremented each time it arrives at a new router. When it hits zero, most routers return a packet death notice (an ICMP Time Exceeded) and the packet is dropped. This lets *traceroute*, or similar programs, deduce the outgoing paths of the packets. There are limitations to this information: The routing may change during the scan and packets may travel down different paths, imputing connections that aren't there. More important, the return paths can be quite different: A large percentage of Internet connections are asymmetric [Paxson, 1997].

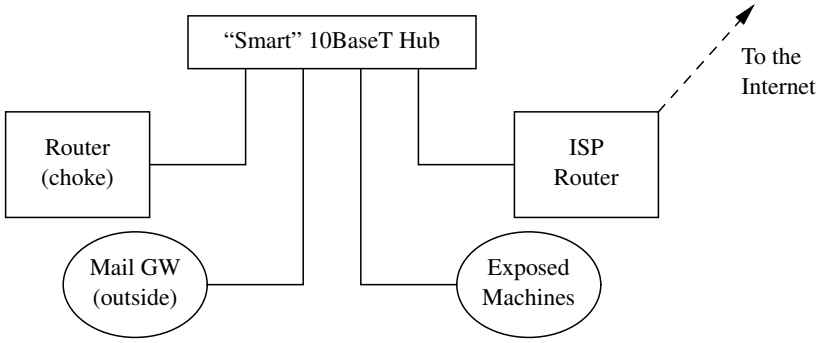
Both *ping* and *traceroute* can use a number of different packets to probe a network. ICMP echo packets are the typical default, and usually work well. Some firewalls block UDP packets (always a good idea) but allow various ICMP messages through. Probes to TCP port 80 (http) often travel where others are not allowed—which makes the program *tcptraceroute* useful.

'Tis a thin line between good and evil. These tools can be used for hacking, and hacking tools can be used for network administration (see Section 6.8).

We rely on *dig* to perform DNS queries. We use it to find `SOA` records, to dump subtrees when trying to resolve an address, and so on. You may already have the *nslookup* program on your machine, which performs similar functions. We prefer *dig* because it is more suitable for use in pipelines.



Isolation via a filtering bridge



Isolation via a "smart" 10BaseT hub

Figure 8.1: Preventing exposed machines from eavesdropping on the DMZ net. A router, instead of the filtering bridge, could be used to guard against address-spoofing. It would also do a better job protecting against layer-2 attacks.

The name server can supply more complete information—many name servers are configured to dump their entire database to anyone who asks for it. You can limit the damage by blocking TCP access to the name server port, but that won't stop a clever attacker. Either way provides a list of important hosts, and the numeric IP addresses provide network information. *Dig* can supply the following data:

```
dig axfr zone @target.com +pfset=0x2020
```

Specifying `+pfset=0x2020` suppresses most of the extraneous information *dig* generates, making it more suitable for use in pipelines.

8.5 Chroot—Caging Suspect Software

UNIX provides a privileged system call named *chroot* that confines a process to a subtree of the file system. This process cannot open or create a file outside this subtree, though it can inherit file handles that point to files outside the restricted area.

Chroot is a powerful tool for limiting the damage that buggy or hostile programs can do to a UNIX system. It is another very important layer in our defenses. If a service is compromised, we don't lose the entire machine. It is not perfect—user *root* may, with difficulty, be able to break out of a *chroot*-limited process—but it is pretty good.

Chroot is one of a class of software tools that create a jail, or *sandbox*, for software execution. This can limit damage to files should that program misbehave. Sandboxes in general provide an important layer for defense-in-depth against buggy software. They are another battleground in the war between convenience and security: The original sandboxes containing Java programs have often been extended to near impotence by demands for greater access to a client's host.

Chroot does not confine all activities of a process, only its access to the file system. It is a limited but quite useful tool for creating sandboxes. A program can still cause problems, most of them in the denial-of-service category:

- **File System Full:** The disk can be filled, perhaps with logging information. Many UNIX systems support disk quota checks that can confine this. Sometimes it is best to *chroot* to a separate partition.
- **Core Dumps:** These can fall under the file-system-full category. The *chroot* command assures that the core dump will go into the confining directory, not somewhere else.
- **CPU Hog:** We can use *nice* to control this, if necessary.
- **Memory Full:** The process can grab as much memory as it wants. This can also cause thrashing to the swap device. There are usually controls available to limit memory usage.
- **Open Network Connections:** *Chroot* doesn't stop a program from opening connections to other hosts. Someone might trust connections from our address, a foolish reliance on address-based authentication. It might scan reachable hosts for holes, and act as a conduit back to a human attacker. Or, the program might try to embarrass us (see Chapter 17).

> A *root* program running in such an environment can also operate a sniffer, but if the attacking program has *root* privileges, it can break out in any event.

Life can be difficult in a *chroot* environment. We have to install enough files and directories to support the needs of the program and all the libraries it uses. This can include at least some of the following:

file	use
`/etc/resolv.conf`	network name resolution
`/etc/passwd`	user name/UID lookups
`/etc/group`	group name/GID lookups
`/usr/lib/libc.so.1`	general shared library routines
`/usr/lib/libm.so`	
`/lib/rld`	shared library information (sometimes)
`/dev/tty`	for seeing *rld* error messages

Statically loaded programs are fairly easy to provide, but shared libraries add complications. Each shared library must be provided, usually in `/lib` or `/usr/lib`.

It can be hard to figure out why a program isn't executing properly in a jail. Are the error messages reported inside or outside the jail? It depends on when they happen. It can take some fussing to get these to work.

The UNIX *chroot* system call is available via the *chroot* command. The command it executes must reside in the jail, which means we have to be careful that the confined process does not have write permission to that binary. The standard version of the *chroot* command lacks a mechanism for changing user and group IDs, *i.e.*, for reducing privileges. This means that the jailed program is running as *root* (because *chroot* requires *root* privileges) and must change accounts itself. It is a bad idea to allow the jailed program *root* access: All known and likely security holes that allow escape from *chroot* require *root* privileges.

Chrootuid is a common program that changes the account and group in addition to calling *chroot*. This simple extension makes things much safer. Alas, we still have to include the binary in the jail.

We can use this program to try to convince some other system administrator to run a service we like on their host. The *jail* source is small and easy to audit. If the administrator is willing to run this small program (as *root*), he or she can install our service with some assurance of safety.

Many other sandboxing technologies are available under various operating systems. Some involve special libraries to check system calls, i.e., [LeFebvre, 1992]. Janus [Goldberg *et al.*, 1996] examines system calls for dangerous behavior; it has been ported to Linux. There is an entire field of study on *domain and type enforcement (DTE)* that specifies and controls the privileges a program has [Grimm and Bershad, 2001; Badger *et al.*, 1996]. A number of secure Linux projects are trying to make a more trustable trusted computing base, and provide finer access controls than the all-encompassing permissions that *root* has on a UNIX host. Of course, the finer-grained the controls, the more difficult it is for the administrator to understand just what privileges are being granted. There are no easy answers here.

The Trouble with Shared Libraries

Shared libraries have become very common. Instead of including copies of all the library routines in each executable file, they are loaded into virtual memory, and a single common copy is available to all. Multiple executions of a single binary file have shared text space on most systems since the dawn of time. But more RAM led to tremendous software bloat, especially in the X Window System, which resulted in a need to share code among multiple programs.

Shared libraries can greatly reduce the size and load time of binaries. For example, *echo* on a NetBSD system is 404 bytes long. But *echo* calls the *stdio* library, which is quite large. Linked statically, the program requires 36K bytes, plus 11K of data; linked dynamically, it needs just 2 K of program and 240 bytes of data. These are substantial savings, and probably reduce load time as well.

Shared libraries also offer a single point of control, a feature we like when using a firewall. Patches are installed and compiled only once. Some security research projects have used shared libraries to implement their ideas. It's easier than hacking the kernel.

So what are our security objections to using shared libraries in security-critical programs? They provide a new way to attack the security of a host. The shared libraries are part of the critical code, though they are not part of the physical binary. They are one more thing to secure, in a system that is already hard to tighten up. Indeed, hackers have installed trap doors into shared library routines. One mod adds a special password to the password-processing routine, opening holes in every *root* program that asks for a password.

It is no longer sufficient to checksum the *login* binary: now the routines in the shared libraries have to be verified as well, and that's a somewhat more complicated job. Flaws in the memory management software become more critical. A way to overwrite the address space of an unprivileged program might turn into a way to attack a privileged program, if the attacker can overwrite the shared segment. That shouldn't be possible, of course, but the unprivileged program shouldn't have had any holes either.

There have been problems with *setuid* programs and shared libraries as well.[a] In some systems, users can control the search path used to find various library routines. Imagine the mischief if a user-written library can be fed to a privileged program.

Chroot environments become more difficult to install. Suddenly, programs have this additional necessary baggage, complicating the security concerns.

We are not persuaded that the single point of update is a compelling reason either. You should know which are your security-sensitive routines, and recompile them. The back door update muddles the situation. For programs not critical to security, go ahead and use shared libraries.

a. CERT Advisory CA-1992-11; CERT Vulnerability Note VU#846832

8.6 Jailing the Apache Web Server

At this writing, the Apache Web server (see WWW.APACHE.ORG) is the most popular one on the Net. It is free, efficient, and comes with source code. It has a number of security features: It tries to relinquish *root* privileges when they aren't needed, user scripts can be run under given user names, and these can even be confined using jail-like programs such as *suexec* and *CGIWrap*.

Why does Apache need to run as *root*? It runs on port 80, which is a privileged port. It may run a CGI script as a particular user, or in a *chroot* environment, both requiring *root* permissions.

In any case, the Apache Web server is fairly complex. When it is run under its own recognizance, we are trusting the Apache code and our own configuration skills. The Apache manual is clear that misconfiguration can cause security problems.

The trusted computing base for Apache is problematic. It uses shared libraries when available, as well as *dynamic shared objects (DSOs)* to load various capabilities at runtime. These optimizations are usually made in the name of efficiency, though in this case they can slow down the server. In these days of cheap memory and disk space, we should be moving toward simpler programs.

If we really want high assurance that a bug in the Apache server software won't compromise our host, we can confine the program in a box of our own devising. In the following example, we have *inetd* serve port 80, and call the *jail* program to confine the server to directory `/usr/apache`. We get much more control, but lose the optimizations Apache provides by serving the port itself. (For a high-volume Web server, this can be a critical issue.) A typical line in `/etc/inetd.conf` might be

```
http stream tcp nowait root /usr/local/etc/jail
   jail -u 99 -g 60001 -l /tmp/jail.log /usr/apache /bin/httpd -d /
```

(Note that this recipe specifies *root*. It has to for the *chroot* in Apache to work.)

Life is much simpler and safer in the jail if we generate a static binary, with fixed modules. For Apache 1.3.26, the following *configure* call sufficed on a FreeBSD system:

```
CFLAGS="-static" CFLAGS_SHLIB="-static" LD_SHLIB="-static"
     ./configure --disable-shared=all
```

The binary *src/httpd* can be copied into the jail.

It can be a fight to generate a static binary for a program. The documentation usually doesn't contain instructions, so one has to wade through configuration files and often source code. Apache 2.0 uses *libtool*, and it appears to be impossible to generate what we want without modifying the release software.

The Apache configuration files are pretty simple. For this arrangement, you will need to include the following in `httpd.conf`:

```
ServerType inetd
HostnameLookups off
ServerRoot /
DocumentRoot "/pages"
UserDir Disabled
```

along with the various other normal configuration options.

As usual with *chroot* environments, we have to include various system files to keep the server happy. The contents of the jail can become ridiculous (as was the case for Irix 6.2), but here we have:

```
drwxr-xr-x   2 root  wheel   512 Jun 21 10:44 bin
drwxr-xr-x   3 root  wheel   512 Nov 25  2001 conf
drwxr-xr-x   2 root  wheel   512 Nov 25  2001 etc
drwxr-xr-x   3 root  wheel  2048 Nov 25  2001 icons
drwxr-xr-x   2 root  wheel  2048 Jun  1 00:02 logs
drwxr-xr-x  14 root  wheel   512 Jan  2 20:39 pages
```

Directory	Files	Reason
`bin`	*httpd*	server executable
`conf`	`httpd.conf`	server configuration
	`mime.types`	server needs them
`etc`	`group`	GID/name mappings
	`pwd.db`	UID/name mappings
`icons`	(various)	images for the server
`logs`	(various)	all the logging data
`pages`	(various)	the Web pages

Of course, the server runs as account *daemon*, and has write permission only on the specific log files in the `log` directory. An exploited server can overwrite the logs (append-only files would be better) and fill up the log file system. It can fill up the file system and swap space, taking the machine down. But it can't deface the Web pages, as there is a separate instantiation of the server for each request, and it doesn't have write access to the binary. (What we'd really like is a *chroot* that takes effect just after the program load is completed, so the binary wouldn't have to exist in the jail at all.) It would be able to read all of our pages, and even our SSL keys if we ran that too. (See Section 8.12 for a way around that last problem.)

One file we don't need is `/bin/sh`. Marcus Ranum suggests that this is a fine opportunity for a burglar alarm. Put in its place an executable that copies its arguments and inputs to a safe place and generates a high-priority alarm if it is ever invoked. This extra defensive layer can make sudden heros when a day-zero exploit is discovered.

Many Web servers could be run this way. If the host is resistant to attack, and the Web server is configured this way, it is almost impossible for a net citizen to corrupt a Web page. This arrangement could have saved a number of organizations great embarrassment, at the expense of some performance.

Clearly, this solution works only for read-only Web offerings, with limited loads. Active content implies added capabilities and dangers.

8.6.1 CGI Wrappers

CGI scripts are programs that run to generate Web responses. These programs are often simple shell or Perl scripts, but they can also be part of a complex database access arrangement. They have often been used to break into Web servers.

Program flaws are the usual reason: they don't check their input or parameters. Input string length may be unchecked, exposing the program to *stack-smashing*. Special characters may be given uncritically to Perl for execution, allowing the sender to execute arbitrary Perl commands. (The Perl *Taint* feature helps to avoid this.) Even some sample scripts shipped with browsers have had security holes (see CERT Advisory CA-96.06 and CERT Advisory CA-97.24).

CGI scripts are often the wildcard on an otherwise secure host. The paranoid system administrator can arrange to secure a host, exclude users, provide restricted file access, and run safe or contained servers. But other users often have to supply CGI scripts. If they make a programming error, do we risk the entire machine? Careful inspection and review of CGI scripts may help, but it is hard to spot all the bugs in a program.

Another solution is to jail the scripts with *chroot*. The Apache server comes with a program called *suexec*, which is similar to the *jail* discussed in Section 8.6. This carefully checks its execution environment, and runs the given CGI script if it believes it is called from the Web server. Another program, *CGIWrap*, does the same thing. Note, though, that such scripts still need read access to many resources, perhaps including your user database.

8.6.2 Security of This Web Server

Many organizations have suffered public humiliation when their Web servers have been cracked. Can this happen here?

We are on pretty firm ground if the Web server offers read-only Web pages, without CGI scripts. The server runs as a nonprivileged user. That user has write permission only on the log files: The binaries and Web contents are read-only for this account. Assuming that the *jail* program can't be cracked, our Web page contents are safe, even if there is a security hole in the Web server. Such a hole could allow the attacker to damage or alter the log files, a minor annoyance, not a public event. They could also fill our disk partition, probably bringing down the service.

The rest of the host has to be secure from attack, as do the provisioning link and master computer. With very simple host configurations, this can be done with reasonably high assurance of security.

As usual, we can always be overwhelmed with a denial-of-service attack. The real challenge is in securing high-end Web servers.

8.7 Aftpd—A Simple Anonymous FTP Daemon

Anonymous FTP is an old file distribution method, but it still works and is compatible with Web browsers. It is relatively easy to set up an anonymous FTP service. For the concerned gatekeeper, the challenge is selecting the right version of *ftpd* to install. In general, the default *ftpd* that comes with most systems has too much privilege. Versions of *ftpd* range from inadequate to dangerously baroque. An example of the latter is *wu-ftpd*, which has many convenient features, but also a long history of security problems.

We use a heavily modified version of a standard *ftpd* program developed with help from Marcus Ranum and Norman Wilson. Many cuts and few pastes were used. The server allows anonymous FTP logins only, and relinquishes privileges immediately after it confines itself with *chroot*.

By default, it offers only read access to the directory tree; write access is a compilation option. We don't run this anymore, but if we did, it would certainly be jailed.

The actual setup of an anonymous FTP service is described well in the vendor manual pages. Several caveats are worth repeating, though: Be absolutely certain that the root of the FTP area is not writable by anonymous users; be sure that such users cannot change the access permissions; don't let the *ftp* account own anything in the tree; don't let users create directories (they could store stolen files there); and do *not* put a copy of the real `/etc/passwd` file into the FTP area (even if the manual tells you to). If you get the first three wrong, an intruder can deposit a `.rhosts` file there, and use it to *rlogin* as user *ftp*, and the problems caused by the last error should be obvious by now.

8.8 Mail Transfer Agents

8.8.1 Postfix

We think that knowledge of a programmer's security attitudes is one of the best predictors of a program's security. Wietse Venema is one of the fussiest programmers we know. A year after his mailer, *postfix*, was running almost perfectly, it still wasn't out of alpha release. This is quite a contrast to the typical rush to get software to market. Granted, the financial concerns are different: Wietse had the support of IBM Research; a start-up company may depend on early release for their financial survival.

But Wietse's meticulous care shows in his software. This doesn't mean it is bug-free, or even free of security holes, but he designed security in from the start. *Postfix* was designed to be a safe and secure replacement for *sendmail*. It handles large volumes of mail well, and does a reasonable job handling spam.

It can be configured to send mail, receive mail, or replace *sendmail* entirely. The send-only configuration is a good choice for secure servers that need to report things to an administrator, but don't need to receive mail themselves.

The compilation is easy on any of the supported operating systems. Its lack of compilation warnings is another good sign of clean coding. None of its components run *setuid*; most of them don't even run as *root*. The installation has a lot of options, particularly for spam filtering, but mail environments differ too much for one size to fit all. We do suggest that the *smptd* daemon be run in *chroot* jail, just in case.

Because *postfix* runs as a *sendmail* replacement, there is the usual danger that a system upgrade will overwrite *postfix*'s `/usr/lib/sendmail` with some newer version of *sendmail*.

8.9 POP3 and IMAP

The POP3 and IMAP services require read and write access to users' mailboxes. They can be run in *chroot* jail under an account that has full access to the mailboxes, but not to anything else. The protocols require read access to passwords, so the keys have to be stored in the jail, or loaded before jailing the software.

Numerous implementations of POP3 are available. The protocol is easy to implement, and many of these can be jailed with the *chroot* command. One can even use *sslwrap* to implement an encrypted server. It would be nice to have an *inetd*-based server that jails itself after reading in the mail passwords.

IMAP4 has a lot more features than POP3. This makes it more convenient, but fundamentally more dangerous to implement, as the server needs more file system access. In the default configuration, user mailboxes are in their home directories so jailing IMAP4 configuration is less beneficial. This is another case where a protocol, POP3, seems to be better than its successors, at least from a security point of view.

8.10 Samba: An SMB Implementation

Samba is a set of programs that implement the SMB protocol (see Section 3.4.3) and others on a UNIX system. A UNIX system can offer printer, file system, and naming services to a collection of PCs. For example, it can be a convenient way to let PC users edit pages on a Web server.

It is clear that a great deal of care has gone into the Samba system. Unfortunately, it is a large and complex system, and the protocols themselves, especially the authentication protocols, are weak. Like the Apache Web server, it has a huge configuration file, and mistakes in configuration can expose the UNIX host to unintended access.

In the preferred and most efficient implementation, *samba* runs as a stand-alone daemon under account *root*. It switches to the user's account after authentication. Several authentication schemes are offered, including the traditional (and very weak) Lan Manager authentication.

A second option is to run the server from *inetd*. As usual, the start-up time is a bit longer, but we haven't noticed the difference in actual usage. In this case, *smbd* can run under any given user; for example, *nobody*. Then it has the lowest possible file permissions. This is a lot better than *root* access, but it still means that every file and directory to be shared must be checked for world-read and world-write access.

If we forgo the printer access, and just wish to share a piece of the file system, we can try to jail the whole package. For our experimental implementation we are supporting four Windows users on a home network. Each user is directed to a different TCP port on the same IP address using a program that implements the NetBIOS *retarget* command. This simple protocol answers "map network drive" queries on TCP port 139 to alternate IP addresses and TCP ports. Each of these alternate ports runs *smbd* in a jail specific to that user.

Each jail has a mostly unwritable `smbd` directory that contains `lib/etc/smbpasswd`, `lib/codepages`, `smb.conf`, a writable `locks` directory, and a log file. Besides these boilerplate files, the directory contains the files we wish to store and share. One share is used by the entire family to share files and store backups, which we can save by backing up the UNIX server. Our Windows machines do not need to run file sharing. We have not yet shared the printers in this manner.

This arrangement works well on a local home network. It might be robust against outside attack, but if it isn't, the server host is still safe. Because the SMB protocol is not particularly secure, we can't use this safely from traveling laptops. Hence, we can hide these ports on an

unannounced network of the home net, so they can't even be reached from the Internet except by compromising a local host first. This isn't impossible, but it does give the attackers another layer to penetrate.

With IPsec, we might be able to extend this service to off-site hosts.

8.11 Taming Named

The domain name service is vital for nearly all Internet operations. Clients use the service to locate hosts on the Internet using a resolver. DNS servers publish these addresses, and must be accessible to the general public.

The most widespread DNS server, *named*, does cause concern. It is large, and runs as *root* because it needs to access UDP port 53. This is a bad combination, and we have to run this server externally to service the world's queries about our namespace. There have been a number of successful attacks on this code (see, for example, CERT Advisory CA-1997-22, CERT Advisory CA-1998-05, CERT Advisory CA-1999-14, and CERT Advisory CA-2001-02). (See Figure 14.2 for more on the response to CERT Advisory CA-1998-05.) Note that these attacks are on the server code itself, rather than the more common DNS attacks involving the delivery of incorrect answers.

The *named* program can contain itself in a *chroot* environment, and that certainly makes it safer. Some versions can even give up *root* access after binding to UDP port 53. Because the privileges aren't relinquished until after the configuration file is processed, it may still be subject to attack from the configuration file, but that should be a hard file for an attacker to access. The following call is an example of this:

```
named -c /named.conf -u bind -g bind -t /usr/local/etc/named.d
```

This runs *named* in a jail with user and group *bind*. If *named* is conquered, the damage is limited to the DNS system. This is not trivial, but much easier to repair: we can still have confidence in the host itself. Of course, we have to compile *named* with static libraries, or else include all the shared libraries in the jail.

Adam Shostack has conspired to contain *named* in a *chroot* environment [Shostack, 1997]. It is more involved than our examples here because shared libraries and related problems are involved, but it's a very useful guide if your version of *named* can't isolate itself.

8.12 Adding SSL Support with Sslwrap

A crypto layer can add a lot of security to a message stream. SSL is widely implemented in clients, and is well suited to the task. The program *sslwrap* provides a neat, clean front end to TCP services. It is a simple program that is called by *inetd* to handle the SSL handshake with the client using a locally generated certificate. When the handshake is complete, it forwards the plaintext byte stream to the actual service, perhaps on a private IP address or over a local, physically secure network. Several similar programs are available, including *stunnel*.

This implementation does not limit who can connect to the service, but it does ensure that the byte stream is encrypted over the public networks. This encryption can protect passwords that the underlying protocol normally sends in the clear. A number of important protocols have SSL-secured alternates available on different TCP ports:

Service	Standard TCP Port	SSL TCP Port	SSL Name	Type of Service
POP3	110	995	POP3S	fetch mail
IMAP	143	993	IMAPS	fetch/manage mail
SMTP	25	465	SMTPS	deliver mail (smtps is deprecated)
telnet	21	992	telnets	terminal session
http	80	443	HTTPS	Web access
ftp	21	990	FTPS	file transfer control channel
ftp/data	20	989	FTPS-data	file transfer data channel

There are monolithic servers that support SSL for some of these, but the SSL routines are large and possible sources of security holes in the server. *Sslwrap* is easily jailed, isolating this risk nicely. (When the *slapper* SSL worm struck—see CERT Advisory CA-2002-27—a Web server we run was *not* at risk. Rather than running HTTPS on port 443, the machine ran *sslwrap*. Yes, that could have been penetrated, but there were no writable files in its tiny jail, and only the current instantiation of *sslwrap* was at risk, not the Web server itself. Of course, the private key could still be compromised, although *slapper* did not do that. *Apache* ran in a separate jail.)

RFC 2595 [Newman, 1999] has some complaints about the use of alternate ports for the TLS/SSL versions of these services. The current philosophy is to avoid creating any more such ports; [Hoffman, 2002] is an example of the current philosophy. While there are advantages to doing things that way, it does make it harder to use outboard wrappers.