

6

Gateway Tools

Beyond those we have already discussed, several other software tools are useful for building an application-level gateway. The tools include *proxylib*, a portable version of the 10th Edition UNIX system *connection server* interface [Presotto and Ritchie, 1985], *proxy*, and others. Many of these tools are publicly available; see Appendix A for details.

In designing our own tools and libraries, we have, as much as possible, followed the oft-cited, seldom-heeded, “UNIX philosophy”. That is, our tools are simple and modular, and have few flags, frills, or options.

6.1 Proxylib

Building a firewall means writing lots of small programs that need to open network connections. That, in turn, means writing and rewriting the same set of routines to call `gethostbyname` and/or `inet_ntoa`, `socket`, `connect`, etc. Worse yet, you may need several versions of many of those programs to handle inside callers, outside callers, pass-through callers, etc. There has to be a better way.

Fortunately, there is. We developed a version of the 10th Edition connection server that will run on just about any UNIX system. Creating a file descriptor for a network connection is now a matter of one or two subroutine calls, with no mystic structures or system calls involved. The primary argument is a character string of the form

```
dest!service
```

where `dest` is a host name or numeric IP address and `service` is the port number or service name. *Proxylib* buys you more than that. There is an optional leading field specifying the dialer type. For example,

```
tcp!host!service
```

will connect via TCP. Other dialers might be `dk` (Datakit VCS), `dial` (a phone number), or `atm`. Our library includes a dialer named `proxy` that connects to the firewall proxy service somehow:

The method is determined at compile time or by the address(es) given in the environment variable \$PROXY. The benefits are considerable. For example, most of the proxy NFS we developed was debugged using an explicit request for TCP. Using it through the firewall required no changes whatsoever to the code.

The two principal entry points are

```
int
ipccopen(char *path, char *flags);
```

and

```
char *
ipccpath(char *dest, char *defdialer, char *defservice);
```

where `defdialer` and `defservice` are the default dialer and service, respectively. The `path` argument to `ipccopen` is of the form

```
dialer!destination!service
```

The `ipccpath` routine behaves as follows:

```
ipccpath("x!y!z", "defdial", "defservice") → "x!y!z"
ipccpath("x!y", "defdial", "defservice") → "x!y!defservice"
ipccpath("x", "defdial", "defservice") → "defdial!x!defservice"
```

The normal call in, say, *ptelnet* is

```
fd = ipccopen(ipccpath(dest, "proxymach", "telnet"), "");
```

That is, "proxymach" is the default dialer and "telnet" is the default service. Either or both can be overridden explicitly by the user. By convention, `proxymach` is the gateway administratively assigned to that user. The environment variable \$PROXY can contain a comma-separated list of gateways to use for *proxy* connections. The user could even call *ptelnet* with the `tcp` dialer and get normal, local *telnet* connections.

The `flags` argument to `ipccopen` is used for things like FTP, to create an incoming socket and pass back its address, and for our prototype proxy X11.

The converted version of *telnet* is *ptelnet*, *ftp* is *pftp*, etc. This leaves the local unmodified versions of these programs available and makes the existence of the modified versions obvious.

One criticism of our gateway approach is the need to modify internal programs. These have indeed raised questions of support and portability. By isolating the network-dependent code in the proxy library, portability has been much easier. Generally, the major AT&T Computer Center computers all have these modified programs available, and we make the complete package available on internal anonymous FTP servers. We know of no attempts to add the proxy protocol to PCs and Macintoshes, although it probably wouldn't be too hard. The protocol is simple to implement. Some of the services can be implemented at the gateway. For example, our users can *telnet* to our gateway and then give a command to *telnet* to an external destination. The Digital and TIS gateway packages contain similar arrangements for *ftp*.

Our proxy library does have one notable limitation: it provides no support for the TCP urgent pointer. This is normally used in *telnet* and especially *ftp* to abort processing. Historically, we could not implement this feature easily through Datakit and other networks. We could implement it through our current gateway, but its loss is a minor irritation and hasn't been worth the effort.

6.1.1 Socks

The *socks* package [Koblas and Koblas, 1992] is a publicly available TCP circuit gateway package. Like our *proxy* package, it can be installed easily into network applications. In fact, it has a one-for-one replacement for each standard networking call, i.e., `connect` becomes `Rconnect`, etc., which makes it easier to install.

Most releases of major software are quickly converted to use *socks*, including some for PCs and Macintoshes.

Socks works at the numeric IP address level rather than with host names as *proxylib* does. This means that the internal host needs access to external name service in some form. This is fairly easy to do given a single-host gateway. There is some danger to supplying external name server access to internal hosts and we are wary. Section 3.3.4 details our misgivings.

6.2 Syslog

Syslog is useful for managing the various logs. It has a variety of useful 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.

We chose to use the local log classes and assign our own names:

```
#define LOG_INETD      LOG_LOCAL0
#define LOG_FTPD       LOG_LOCAL1
#define LOG_TELNETD    LOG_LOCAL2
#define LOG_SMTPD      LOG_LOCAL3
#define LOG_PROXY      LOG_LOCAL4
#define LOG_SMTP        LOG_LOCAL5
#define LOG_SMTPSCHED  LOG_LOCAL6
```

Most are self-explanatory. `LOG_INETD` receives the TCP wrapper information. `LOG_PROXY` handles proxy and relay connection reports, and `LOG_SMTPSCHED` records our mail queue scanner's activities.

It is important to remember what *syslog* does *not* do. It does not guarantee that your logs will be complete, useful, readable, or available in any form amenable to automated analysis. It is worth considerable effort to ensure that your calls to *syslog* do meet these criteria. In our experience, the easiest way to do that is to build a security log subroutine library.

When designing such a library, one wants to pick a file format that will let you answer a question like this: three months ago—and 1.8 GB ago—what were the malign activities from FOO.BAR.EDU, sorted by type? In other words, you want concise summary records, with a single line per incident, and different fields delimited by some fixed character. Add extra fields to some messages if necessary to achieve consistency. Do not worry if this format is not human-friendly: you want it suitable for standard UNIX tools like *awk* or *perl*.

If you are comfortable with a commercial database package, you may wish to use it to process your log data. But that is probably overkill. We have felt no such need, despite the voluminous logs our gateways create.

If your *syslogd* supports it, keep the logs on a different machine. Hackers generally go after the log files before they do anything else, even before they plant their backdoors and Trojan horses. You're much more likely to detect any successful intrusions if the log files are on the protected inside machine.

Many *syslog* daemons listen for messages on a UDP port, which leaves them open to denial-of-service attacks. 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.


6.3 Watching the Network: Tcpcdump and Friends

Sometimes, it is necessary for the Good Guys to monitor traffic on a network. Naturally, this occurs most often during an actual intrusion, when you need to see exactly what is being done to your system. Monitoring from the system being attacked is often possible, various opinions on the likelihood notwithstanding [Maryland Hacker, 1993], but it is a bad idea. It's just too easy for the intruder to notice or disable such logging.

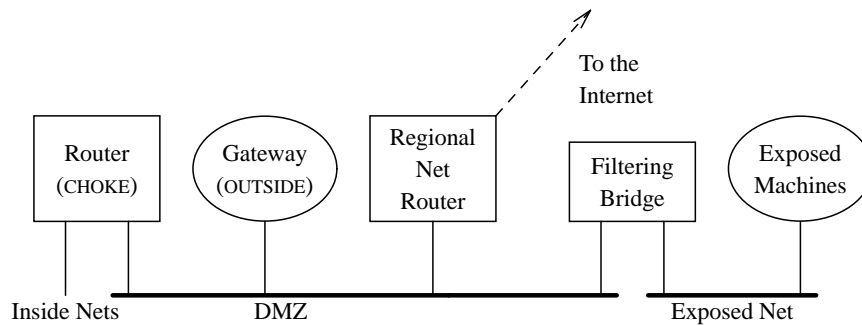
6.3.1 Using Tcpcdump

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 of the 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.

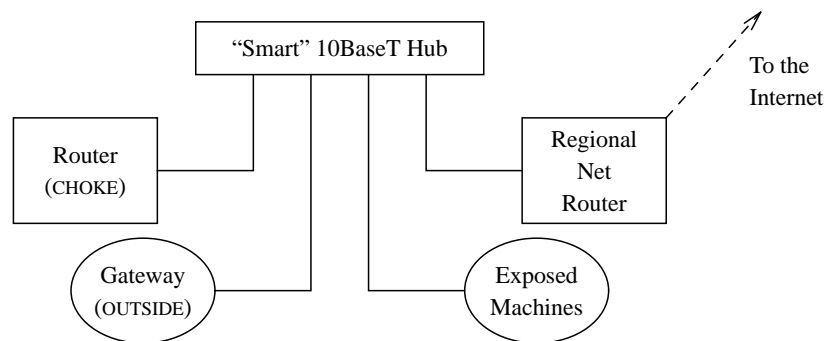
The raw output from *tcpdump* isn't too useful for intrusion monitoring. There is (by design) no ASCII output mode, and there may be several simultaneous conversations intermixed in the output file. But it is not hard to separate the streams and print them, although we do not know of any publicly available tools to do so.

 Many operating systems include similar tools. For example, SunOS has its *etherfind* program, which even uses a similar syntax to *tcpdump*. But all of these programs share one common danger: the very kernel driver which 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. 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 ability to load new drivers at run-time. If you can, delete that ability as well.

A number of packages are available for those of the MS-DOS faith. These, and commercial LAN monitors, may not be as useful as UNIX-based tools for this sort of application. Continuous contingency monitoring of a gateway LAN requires much more storage, but considerably less monitoring of back-to-back packets than do ordinary network problems. The ability to transfer the data to a machine where you can sort, analyze, and archive it is important as well.



Isolation via a filtering bridge



Isolation via a "smart" 10BaseT hub

Figure 6.1: Preventing exposed machines from eavesdropping on the DMZ net. A router instead of the filtering bridge could be used to guard against address-spoofing.

On the other hand, such machines are considerably less vulnerable to penetration than are multi-user systems. As we have noted, hackers *like* to find machines with promiscuous mode Ethernet drivers. Keeping such facilities off of an exposed net is a good idea.

Conversely, if you have any unprotected machines on your DMZ net—say, 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” 10BaseT hub to isolate the experimental machines. Figure 6.1 shows how our Plan C net could be modified to accomplish this.

6.3.2 Ping, Traceroute, and Dig

Although not principally security tools, the *ping* and *traceroute* programs have been useful to us in tracing packets back to their source. *Ping* primarily establishes connectivity. It says 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.

Often, *ping* will succeed, whereas *traceroute* will hit a barrier. The reason is the technology they use: *ping* uses ICMP Echo packets, which are often (but perhaps unwisely) permitted through firewalls, while *traceroute* uses UDP packets.

We rely on *dig* to perform DNS queries. We use it to find SOA records, to dump subtrees when trying to resolve an address, etc. 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.

6.4 Adding Logging to Standard Daemons

Traditionally, the standard network daemons don't log enough data for our purposes. For example, *login* loudly reports login failures only after a certain number of unsuccessful attempts, and it only reports the last user name tried. The hackers know this and can hide a certain number of password attempts. We wanted to log every try, successful or not. (This level of logging is bound to yield some passwords as users get out of sync with *login*, so this particular log should be read-protected from any user community [Grampp and Morris, 1984].)

The administrator must choose between real time notification of an event and after-the-fact scanning of the logs. We have found that real time mail notification of most events is annoying and tedious for our busy gateway. Probes there have become old hat. But probes of a honey pot machine should prompt a quick response.

In our early approach to network gateways we modified local daemons with impunity. We added logging and reduced privilege everywhere we could. In our latest attempts, we have tried to use stock software and publicly-available code whenever possible. But some modifications were necessary. These modifications are general enough that we recommend their inclusion in future versions of this software.

The source code for most of the daemons is available by FTP from public sources. We use the *syslog* to record the logs on the local machine. We made the following changes:

inetd Log the source of each incoming call and the desired service. If you use a logging TCP wrapper throughout, you can omit this modification.

ftpd Allow only *anonymous* logins, and log each FTP command and the full path and number of bytes of every file sent or received. Switch from user *root* to *ftp* very early in the program, and use the changes described earlier to avoid problems with port 20.

Logging from *ftpd* is problematic in many versions, because *syslog* uses the UNIX-domain socket `/dev/log` to communicate with the logging daemon. Once you've done a `chroot`, this no longer works. A number of solutions are possible.

The easiest is to change the `syslog` subroutine to use UDP to send the message, via the loopback interface. Alternatively, some versions of `syslogd` support the `-p` option to specify an alternative socket. In that case, create a `/dev` directory in the anonymous FTP area, and point `syslogd` at `/dev/log`. The latter is necessary so that `syslogd` will read from the proper socket. Create a symbolic link from the real `/dev` to point to the new socket, for the benefit of non-`chroot`'ed programs.

The final solution is to change `syslog` to use a UNIX-domain stream socket, rather than a datagram socket. This allows the open connection to persist across the `chroot` call. But it makes life much more difficult for things like `port20`.

telnetd Make sure that *telnet* passes the caller's address to *login*. Add an option to call a program other than *login*.

login Add the `$CALLER` environment variable, and log all attempts.

Generally, the source to *login* is not publicly available. The code provides a number of useful compile-time options not available to those without source. The binary-only community should also have a chance to disable them. At the very least, a number of standard subroutine calls should be provided, along with a linkable `.o` file for *login*. But we do not recommend doing this via a shared library; such facilities have been responsible for security problems on a number of different platforms.

rshd, rlogind We do not recommend that you run these daemons on a gateway machine; their authentication mechanism is simply too weak for such an exposed situation. Nevertheless, there is one good reason for them to exist: to avoid the necessity of storing hashed passwords on such a machine. If you do run them, be sure to add logging. The standard versions don't tell anyone about attempts (and especially about unsuccessful attempts) to use them. Also, disable the `.rhosts` file processing. System administrators should decide which hosts and accounts to trust, not users.

The FTP logging has taught us that people like to browse. This was not a surprise. Though the browsing occasionally indicated evil intentions, the sheer volume of logging all the FTP commands may not be worth it. The FTP daemon (or a log processor) should provide a one-line entry containing the file name, length, destination, and perhaps a process number for each file transferred. This is useful for security and also gives file distributors an idea of the public interest in their distribution files.

We have not put logging in *named*, the DNS server, and it should be there. This system provides the basis for a number of security attacks (see Section 2.3). Things that should be logged include zone transfers from sites other than authorized secondary servers, a too-high frequency of inverse queries, especially for nonexistent addresses, and attempts at cache contamination. Some of these things are difficult to do. Given the importance and fragility of *named*, we suggest letting an outboard daemon do the analysis.