

7

Traps, Lures, and Honey Pots

We have said that a secure gateway should run as few servers as possible. Since there is generally no mechanism for logging connection requests for unused services, most system administrators cannot tell that someone is probing them.

If logging routines are attached to these ports, there is some assurance that an attack will be detected. Sometimes entire machines or even networks may be set up as *honey pots* to detect browsers. These are especially comforting when installed “near” a valuable target, behind the security walls. This is equivalent to placing a surveillance camera inside a locked vault.

Note carefully the distinction between these security logs and those added to standard network services. The latter monitor the behavior of normal operations, much as accountants monitor a business’s cash flow. Security logs are your sentries, your early warning system—and in some cases your packet of money with a dye bomb attached.

7.1 What to Log

Our research gateway is bristling with logging programs. We do this not just to protect the machine, but to try to judge the rate and sophistication of the probing technology used on the net, and to learn of new attack strategies. Some of ours are quite specialized; others are generic packet suckers. All of them log the incoming data, attempt to trace back the call, and—when feasible—try to distinguish between legitimate users and attackers.

The *finger* server is a good example. Attempts to *finger* a particular user are usually benign attempts to learn an electronic mail address. But that would not work even without our monitor program, since most users do not have logins on the gateway machine. Instead, we print a message explaining how to send mail by name. *Generic finger* attempts are often used to gather login names, personal information, and account usage information for hacking attempts. Therefore, completely bogus output is returned, showing that *guest* and *berferd*—a dummy user name—are logged in. Counterintelligence moves, which include “reverse *finger*”, are not done in this case, for fear of

```

From: adm@research.att.com
To: trappers

Attempted rsh to inet[24640]
Call from host Some.Random.COM (176.75.92.87)
remuser: bin
locuser: bin
command: domainname

(/usr/ucb/finger @176.75.92.87; /usr/ucb/finger bin@176.75.92.87) 2>&1
[176.75.92.87]
Login      Name                TTY Idle   When      Where
rel        R. Locke              co   4d Sat 11:26
afu        Albert Urban          p0  10: Fri 13:51  seed.random.com
rlh        Richard L Hart       p2  3:18 Sat 20:27  fatsol.random.c
rel        R. Locke              p4   3d Mon 09:05  taxi.random.com
[176.75.92.87]
Login name: bin
Directory: /bin
Never logged in.
No unread mail
No Plan.

```

Figure 7.1: An attack via *rsh*.

triggering a *finger* war if the other end is running similar software. All attempts are logged for later analysis.

The so-called “*r* commands” also merit a special server, because of the extra information they provide. For *rlogin* and *rsh*, the protocol includes both the originating user’s login name and the login name desired on our gateway. Thus, we can do a precisely aimed reverse *finger*, and we can assess the level of the threat. A login attempt by some user *foo*, and a request for the same login on RESEARCH.ATT.COM, is probably a harmless error. On the other hand, an attempt by *bin* to execute the *domainname* command as *bin* (see Figure 7.1) represents enemy action. (It also suggests that the attacking machine has been compromised. Note, too, that all of the people shown as logged in are idle.) Attempts to *rlogin* as *guest* from a legitimate account usually fall in the doorknob-twisting category.

For most other services, we rely on a simple packet sucker. This is a program invoked by *inetd* that sits on the socket, reading and logging anything that comes along. While that is happening, counterintelligence moves are initiated. The TCP packet sucker exits when the connection is closed; the UDP version relies on a timeout, but will also exit if a packet arrives from some other source. The information gained from such a simple technique can be quite interesting; see Figure 7.2. It shows an attempt to grab our password file via TFTP.

```
From: adm@research.att.com
To: trappers
Subject: udpsuck tftp(69)

UDP packet from host cs.visigoth.edu (125.76.83.163): port 1406, 23 bytes
  0:  00012f65 74632f70 61737377 64006e65  ../etc/passwd.ne
 16: 74617363 696900          tascii.
/usr/ucb/finger @125.76.83.163 2>&1
[125.76.83.163]
No one logged on

4 more packets received
```

Figure 7.2: Spoof of an attack detected by the UDP packet sucker.

Experience with the packet sucker showed us that there were a significant number of requests for the *portmapper* service. The usual protocol is for the client to contact the server's *portmapper* to learn what port that service is currently using. The *portmapper* supplies that information, and the client proceeds to contact the server directly. This meant that we were seeing only the identifier of the service being requested and not the actual call to it. We decided to simulate the *portmapper* itself.

Our version, called the *portmopper*, does not keep track of any real registrations. When someone requests a service, a new socket is created, and its (random) port number is used in the reply. Naturally we attach a packet sucker to this new port so we can capture the RPC call.

Figure 7.3 shows excerpts from a typical session. We print and decode all the goo in the packet because we do not know if someone might try RPC-level subversion. The first useful datum is delimited by lines of asterisks; it shows a request for the mount daemon, using TCP. Our reply (not shown) assigned a random port number to this session. Finally, the input on that port shows that procedure 2 is being called, with no parameters. There is currently no code to interpret the procedure numbers, but a quick glance at `/usr/include/rpcsvc/mount.h` shows that it's an RPC dump request, i.e., a request for a list of all machines mounting any of our file systems. Our counterintelligence attempt failed: The machine in question is not running a *finger* daemon.

An alternative approach would have been to use the standard *portmapper*, and to have packet suckers registered for each interesting service. We rejected this approach for several reasons. First and foremost, we have no reason to trust the security of the *portmapper* code or the associated RPC library. We are not saying that they have known security holes. We are saying that we do not know if they do.¹ And we are morally certain that legions of would-be hackers are studying the code at this very moment, looking for holes. To be sure, we do not know that our code is bug-free. But our code is smaller and simpler, and hence less likely to be buggy. (It is also relatively unknown, a nontrivial advantage.)

¹Well, we do know that some older versions have holes. But we still don't know about the current versions.

```

From: adm@research.att.com
To: trappers
Subject: UDP portmopper from Vandel.COM (176.143.143.175)

Request:
  0:  2974eaca 00000000 00000002 000186a0  )t.....
 16:  00000002 00000003 00000000 00000000  .....
 32:  00000000 00000000 000186a5 00000001  .....
 48:  00000006 00000000  .....
xid: 2974eaca msgtype: 0 (call)
rpcvers: 2 prog: 100000 (portmapper) vers: 2 proc: 3 (getport)
Authenticator: credentials
Authtype: 0 (none) length: 0
Authenticator: verifier
Authtype: 0 (none) length: 0

***
reqprog: 100005 (mountd) vers: 1 proto: 6 port: 0
***
...
/usr/ucb/finger @176.143.143.175 2>&1
[176.143.143.175]
connect: Connection refused

Server input:
  0:  2976c57d 00000000 00000002 000186a5  )v.).....
 16:  00000001 00000002 00000000 00000000  .....
 32:  00000000 00000000  .....
xid: 2976c57d msgtype: 0 (call)
rpcvers: 2 prog: 100005 (mountd) vers: 1 proc: 2
Authenticator: credentials
Authtype: 0 (none) length: 0
Authenticator: verifier
Authtype: 0 (none) length: 0
Parameters:

```

Figure 7.3: Output from the *portmopper*.

A second reason for eschewing the *portmapper* is that we do not know what the “interesting” services are. Our approach does not require that we know in advance. We can detect requests for anything.

A final reason is that by its nature, the RPC library provides a high-level abstraction to the actual packets. This is useful for programmers, but bad for us. If someone is playing games with, say, the authenticators, we want to know about it.

7.1.1 Address Space Probes

Gateways are well-known machines, and hence attract hackers. A clever hacker might investigate further, looking for other likely machines to try. There seem to be two possibilities: blind probing of the address space or examination of our DNS data. We monitor for such attempts.

The obvious way to do such monitoring is to put a network controller into promiscuous mode and watch the packets fly. We do just that, though with a few modifications. Our gateway does not support promiscuous mode, so we used a spare workstation that did. And we had to seed the ARP table with enough entries that we could see the source of the offending packets (see Section 8.6).

The results of this trap have been rather curious. We have noticed a large number of FTP connection requests to 192.20.225.1, an old gateway machine that was powered off years ago. Furthermore, the large majority of these connection attempts have come from non-U.S. sites. Clearly some organizations are still using static host files that are at least five years old.

We have noticed a few attempts to connect to other machines. For the most part, these have been to DNS-listed addresses (we have a few stale and phony entries), rather than to random places on our network, and the one or two exceptions appear to be accidental. This log file is not examined in real time, so we have not been able to engage in our usual counterintelligence measures. Comparison of the source addresses and timestamps with our other log files tends to show other forms of snooping. Evil probes tend to come in bursts that ring many alarms.

One set of probes was especially alarming. Immediately following the arrest of two alleged non-U.S. hackers, someone else from that country launched a systematic probe of our network’s address space. Our known machine was ignored. We believe that this was an attempt at revenge, and that our well-instrumented gateway machine was ignored because the attackers knew it for what it was.

Other sites with sophisticated monitors have detected similar patterns [Safford *et al.*, 1993b]. Probes based on DNS data are more common, but address space probes occur as well.

Of late, we have seen concerted attempts to connect to random addresses of ours. The pattern does not suggest an attack; rather, it suggests hosts that are quite confused about our proper IP address. The problem appears to be corrupted DNS entries, which we have also experienced, rather than any security problem. This problem is discussed further in [Bellovin, 1993].


7.1.2 ICMP Monitoring

Over the last few years, there have been a number of reports of ICMP attacks of the type described in Section 2.1.5. To detect them, we wrote *icmpmon*, which we produced by deleting most of the code from *ping*.

The *icmpmon* program is fairly stupid. It does nothing but translate the packets and write them to `stdout`. We have not detected any attacks *per se*, though we've seen a fair amount of anomalous behavior [Bellovin, 1993]. Nevertheless, we regard *icmpmon* as a useful program to have around, precisely because of the incidents that have occurred.

7.1.3 Counterintelligence

When a probe occurs, we try to learn as much about the originating machine and user as we can. This usually isn't very much. Thus far, the only generally available mechanism to do that is the *finger* command. While far better than nothing, it has some weaknesses. A user can avoid appearing in a *finger* printout in a number of ways. He or she can overwrite `/etc/utmp` (it is world-writable on many systems) or use certain options of *xterm*. Indeed, we have seen attacks from machines that claim to have no one logged in, such as shown in Figure 7.2.

 The standard *finger* command is not a safe thing to use in all cases. Some hackers have been known to fight back by modifying either their machine's *finger* daemon or their own `.plan` or `.project` files to emit harmful control sequences, infinite output streams, etc. The *safe.finger* program, which is part of recent releases of the TCP wrapper, is recommended instead.

There is also the problem of pokes originating from security-conscious sites. Often these sites restrict or disable the *finger* daemon. Figure 7.3 shows an example. Security-conscious sites are probably the least-likely to be penetrated. But no one is immune; recall the story from Chapter 1 about the installation of a *guest* account on one of our supposedly secure gateways.

Some sites take their own security precautions. One (unsolicited) prober noticed our reverse *finger* attempt and congratulated us on it. Others who thought we were running a "cracker challenge contest" were able to detect our activities when specifically looking for them. The worst possibility would be an active response to our probe; it could easily trigger a looping *finger* contest. For these reasons we do not currently do reverse *finger*s in response to *finger* queries, but the problem could still arise. For example, an *rusers* query to us would trigger the *portmopper's* counterintelligence probes. These in turn could cause the remote site to query our *rusers* daemon. It may be necessary to add some locking to some daemons to avoid this sort of loop.

We have contemplated adding other arrows to our counterintelligence quiver, but there are few choices available. The *rusers* command is an obvious possibility, but it offers less information than *finger* does. Since it goes through the *portmapper*, it is harder to block or monitor. Many sites wisely block all outside calls to the *portmapper* because of concerns about the security of some RPC-based services. Another choice would be the Authentication Server *authd* [St. Johns, 1993] (see Section 7.3). Or we could try SNMP [Case *et al.*, 1990], but it is generally implemented on routers, not hosts, and regional networks often block SNMP packets from remote machines.

A totally different set of investigations is performed using DNS data. First of all, we attempt to learn the host name associated with the prober's numeric IP address, which should be a trivial matter. All addresses are supposed to be listed in the inverse mapping tree, but in practice many are not. This problem seems to be especially commonplace in sites new to the Internet. In such cases, we have to look for the SOA and NS records associated with the inverse domain. Using them, we attempt a zone transfer of the inverse domain and scan it for any host names at all.

That gives the zone name. We then transfer the forward-mapping zone and search for the target's address.

On a few occasions this procedure has failed. We have been forced to resort to the use of *tracert*, manual *finger* attempts, and even a few *telnet* connections to various ports to see if any servers announce the host and domain name. Needless to say, none of this is automated: if a simple `gethostbyaddr` call fails, we perform any further investigations ourselves.

There is one DNS-related check that we do automate: we look for occurrence of the evil games that can be played with the inverse mapping tree of the DNS (Section 2.3). To detect these, our code performs the cross-check manually. If it fails, alarms, gongs, and tocsins are sounded.

7.1.4 Log-Based Monitoring Tools

A number of our monitors are based on periodic analyses of logs. For example, attempts to grab a (phony) password file via FTP are detected by a *grep* job run via *cron*. We thus cannot engage in real time counterintelligence activity in response to such pokes. Nevertheless, they remain very useful. Other things to monitor include attempts to exploit the DEBUG hole (see Chapter 10) or to confuse *sendmail* via oddly placed “pipe to” requests.


Our gateway machine has on occasion been used as a repository for (presumably stolen) PC software. Assorted individuals would store such programs under a directory named `..^T`, where “`^T`” represents the control-T character; others would retrieve it at their leisure. To ward off this activity, we clear out the public FTP area daily. That often works, though a better solution is to add the notion of “inside versus outside” to the daemon, and to prohibit transfers that did not cross the boundary.

Other sites report similar incidents, often involving erotic images. We leave to the readers' imagination what we could insert in place of these files.

Real-time analyzers can be added to some log files. It can be done very simply:

```
tail -f logfile | awk -f script
```

This is an especially useful technique for the FTP daemon's logs because attempts to add more sophisticated mechanisms to the daemon itself would run afoul of the `chroot` environment in which it should run.

 There is danger lurking here. Simple versions could easily fall victim to a sophisticated attacker who uses file names containing embedded shell commands. For this reason, among others, we run all of our traps with as few privileges as possible. In particular, where possible we do not run them as *root*.

A more sophisticated log monitor, *Swatch* [Hansen and Atkins, 1992], was developed at Stanford in response to the Berferd incident. It uses a pattern-match file to describe significant entries. A variety of actions can be associated with each pattern, including the ability to execute an arbitrary program. The TIS firewall toolkit includes an enhanced version of *syslogd* that has similar features built in. Unfortunately, the implementers of these packages have found, as have we, that it was necessary to modify various standard network daemons so that the right information was logged in the first place.

7.2 Dummy Accounts

Many sites offer accounts to the public named *guest*, *demo*, or (rarely) *visitor*. Their passwords, if present at all, are obvious. The *guest* accounts are offered in the spirit of generosity and convenience. Demo accounts provide a fine opportunity to test software, but they must be installed carefully. Do the programs have a shell escape? Can the user abort into a shell prompt? If so, what access to your machine does he have? Does the demo account have an `.rhosts` file? Can the demo user create one? Once they have a shell prompt, an invader can scrutinize your system security from the easy side: the inside. And they can launder connections to other sites.

Guest accounts give all that access for free. Worse, there is no accountability if the account is misused. If guest accounts are used, we recommend that each individual receive a login name of his or her own to promote accountability. These accounts should be reauthorized periodically.

Even novice hackers know all this, of course. Anyone can *telnet* to a machine and try to log in to account *guest*.

What will a dummy *guest* account prove? We get numerous *guest* probes daily. For a public, well-known machine like our gateway, it does provide an extensive list of IP laundering sites, as well as a wide variety of educational computers. We are showered with a daily barrage of guest login attempts. Many, perhaps most, are innocent. These probes may correlate to an actual attack, but they don't provide much information. They can give a statistical guide to the actively nosy sites out there.

On a protected network, presumed to be shielded from bored high school students and corporate spies, an attempted *guest* login may be much more serious. Certainly a honey pot machine should detect attempted *guest* logins.

A dummy guest account is a simple exercise for a novice system administrator, at least on UNIX systems. One of our favorite password entries and shell scripts is:

```
guest::9999:1:Joe Guest:/usr/guest:/usr/adm/guestlogin

#!/bin/sh

echo "Guest login attempt!" | mail goodguys
echo "Guest login attempt" >>/log/warn

sleep 20 # make 'em wait:  the network is slow
echo "/tmp full"
sleep 5
echo "/tmp full"
echo "/tmp full"
echo "/tmp full"
sleep 120 # make the hacker wait
exit
```

This script can waste their time and give them hope that the file system problem might be fixed sometime in the future. The script lacks one vital piece of information: the calling host address. We modified our *login* program to provide the calling IP address in the environment variable `$CALLER`. A reverse *finger* could be included in the script as well.

7.3 Tracing the Connection

It seems a simple task to trace an IP connection. Each packet arrives with the IP address of the sender. What could be easier?

Lots of things.

First of all, it is possible for the sending computer to put in any sending IP address it wishes. Most operating systems prevent unprivileged users from doing this, but PCs don't. This is a well-known and obvious problem, and it is the basis for some difficult and obscure attacks [Morris, 1985; Bellovin, 1989]. It is not clear how to detect this activity, or whether these attacks can actually be carried out over the Internet.

In some cases, a daemon known in different incarnations as *authd*, *pauthd*, *identd*, and *ident* can help [St. Johns, 1993]. They all implement a similar protocol, and are descended from RFC 931. If some machine has an open TCP connection to your machine, your machine can query the daemon and ask who owns that connection. The daemon will reply with some sort of operating system-specific string that can be used for maintaining audit trails.

RFC 1413 is quite explicit about how to use the information returned:

The Identification Protocol is not intended as an authorization or access control protocol. At best, it provides some additional auditing information with respect to TCP connections. At worst, it can provide misleading, incorrect, or maliciously incorrect information.

The use of the information returned by this protocol for other than auditing is strongly discouraged. Specifically, using Identification Protocol information to make access control decisions—either as the primary method (i.e., no other checks) or as an adjunct to other methods may result in a weakening of normal host security.

Others disagree, and point out that the reliability is no worse than the address-based authentication currently in vogue. Regardless, there are two major caveats. First, if the machine is not trustworthy, or has been subverted, the information its daemons return is not trustworthy. Second, under certain circumstances, if you try to contact one of these daemons you may actually be launching a denial-of-service attack against yourself!

Recall that most firewalls will not permit incoming calls except to specific ports. Some implementations, especially older ones, will return an ICMP `Destination Unreachable` message in response, and that, in turn, may tear down all of your connections to that machine. That is, if you attempt to validate an incoming call, you could cause that call to be terminated! The only fix is to upgrade your kernel; it needs to use the port number fields returned by ICMP to decide which connection should be terminated.

In general, the documentation you receive with your host will not tell you how your machine will behave under such circumstances. The only way to tell for sure is to try it. Some software packages that use the *ident* protocol include documentation that lists some candidate firewalls and hosts that you can use for your experiments.

This same phenomenon can affect reverse *finger* attempts, too. In general, this is a less serious problem when tracing a call to a phony service, since you do not much care if the connection is

terminated. But much more care is necessary if you are trying to log extra information before invoking an actual server.

These difficulties are not the major problem when trying to trace connections. Rather, the real problem is that hackers launder their connections through other machines. When an attack comes, the best automated counterintelligence can only trace two hops. This limit comes from the availability of the *finger* service on the attacking machine, which may give the address of the calling machine if the user is listed at all. Since *finger*s have varying formats in different operating systems, this is hard to automate.

A human can do better, but it takes quick work. CERT can provide contacts for many sites. Sometimes those sites already know they have a problem and can supply more information. A desperate administrator might be tempted to break into the most distant hop to follow the trail. We *do not* recommend this.

Often the hacker returns to try again, so it is possible to make arrangements for tracing future attacks. Be careful using email, though. Most hackers monitor the administrator's mail when they are attacking a computer.

If a terminal server or X.25 gateway is involved, the tracking gets much harder. Public X.25 data networks tend to be an infested soup of corruption for which the hackers have a separate set of laundering tricks. It requires a court order, often from many jurisdictions, to trace a telephone call. Several well-known hacking cases involve publicly accessible out-dial modems, which can provide the ultimate in laundering convenience. You don't think they use their own phones for this, do you?

Given enough time and persistence, cooperative authorities, and demonstrable damage, it may be possible to track down a Bad Guy. One must be endlessly on call and willing to sacrifice one's personal life. (Cliff Stoll's shower scene on Nova was intriguing television, but probably wasn't much fun at the time [Stoll, 1989, 1988].)