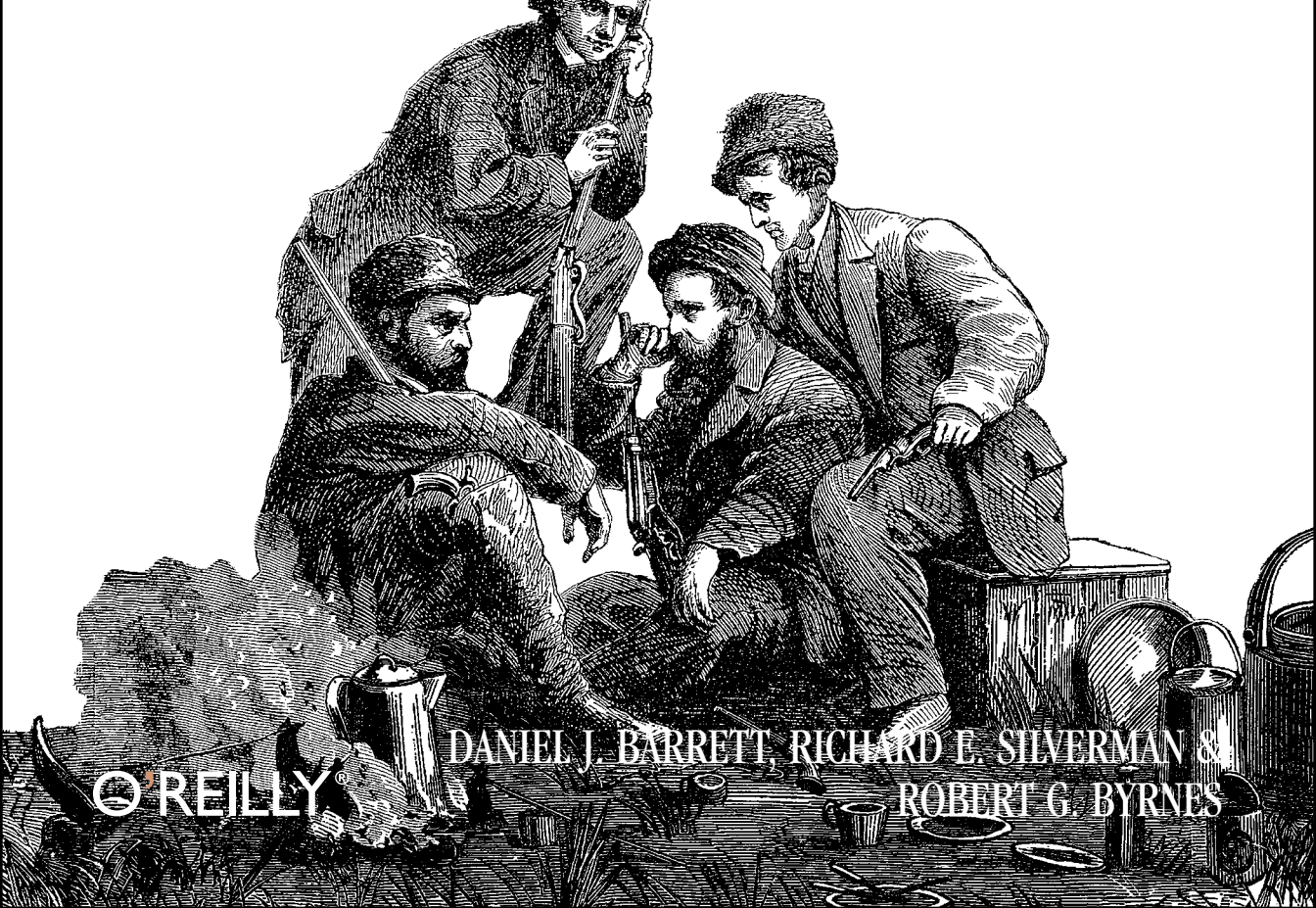
SECURITY, TOOLS & TECHNIQUES

LINUX SECURITY COOKBOOK



DANIEL J. BARRETT, RICHARD E. SILVERMAN &
ROBERT G. BYRNES

O'REILLY

LINUX
SECURITY
COOKBOOK

Other Linux resources from O'Reilly

Related titles

- Linux in a Nutshell
- Linux Network Administrator's Guide
- Running Linux
- Linux Device Drivers
- Understanding the Linux Kernel
- Building Secure Servers with Linux
- LPI Linux Certification in a Nutshell
- Learning Red Hat Linux
- Linux Server Hacks
- Linux Security Cookbook
- Managing RAID on Linux
- Linux Web Server CD Bookshelf
- Building Embedded Linux Systems

Linux Books Resource Center

linux.oreilly.com is a complete catalog of O'Reilly's books on Linux and Unix and related technologies, including sample chapters and code examples.

ONLamp.com is the premier site for the open source web platform: Linux, Apache, MySQL and either Perl, Python, or PHP.

Conferences

O'Reilly & Associates brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

LINUX SECURITY COOKBOOK

Daniel J. Barrett,
Richard E. Silverman, and
Robert G. Byrnes

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

9.1 Testing Login Passwords (John the Ripper)

Problem

You want to check that all login passwords in your system password database are strong.

Solution

Use John the Ripper, a password-cracking utility from the Openwall Project (*http://www.openwall.com*). After the software is installed, run:

```
# cd /var/lib/john
# umask 077
# unshadow /etc/passwd /etc/shadow > mypasswords
# john mypasswords
```

Cracked passwords will be written into the file *john.pot*. Cracked username/password pairs can be shown after the fact (or during cracking) with the `-show` option:

```
# john -show mypasswords
```

You can instruct `john` to crack the passwords of only certain users or groups with the options `-users:u1,u2,...` or `-groups:g1,g2,...`, e.g.:

```
# john -users:smith,jones,akhmed mypasswords
```

Running `john` with no options will print usage information.

Discussion

SuSE distributes John the Ripper, but Red Hat does not. If you need it, download the software in source form for Unix from *http://www.openwall.com/john*, together with its signature, and check the signature before proceeding. [7.15]

Unpack the source:

```
$ tar xvzpf john-*.tar.gz
```

Prepare to compile:

```
$ cd `ls -d john-* | head -1`/src
$ make
```

This will print out a list of targets for various systems; choose the appropriate one for your host, e.g.:

```
linux-x86-any-elf       Linux, x86, ELF binaries
```

and run `make` to build your desired target, e.g.:

```
$ make linux-x86-any-elf
```

Install the software, as root:

```
# cd ../run
# mkdir -p /usr/local/sbin
# umask 077
# cp -d john un* /usr/local/sbin
# mkdir -p /var/lib/john
# cp *.* mailer /var/lib/john
```

Then use the recipe we've provided.

By default, Red Hat 8.0 uses MD5-hashed passwords stored in */etc/shadow*, rather than the traditional DES-based `crypt( )` hashes stored in */etc/passwd*; this is effected by the `md5` and `shadow` directives in */etc/pam.d/system-auth*:

```
password    sufficient    /lib/security/pam_unix.so nullok use_authtok md5 shadow
```

The `unshadow` command gathers the account and hash information together again for cracking. This information should not be publicly available for security reasons—that's why it is split up in the first place—so be careful with this re-integrated file. If your passwords change, you will have to re-run the `unshadow` command to build an up-to-date password file for cracking.

In general, cracking programs use dictionaries of common words when attempting to crack a password, trying not only the words themselves but also permutations, misspellings, alternate capitalizations, and so forth. The default dictionary (*/var/lib/john/password.lst*) is small, so obtain larger ones for effective cracking. Also, add words appropriate to your environment, such as the names of local projects, machines, companies, and people. Some available dictionaries are:

ftp://ftp.ox.ac.uk/pub/wordlists/
ftp://ftp.cerias.purdue.edu/pub/dict/wordlists

Concatenate your desired word lists into a single file, and point to it with the `wordlist` directive in */var/lib/john/john.ini*.

`john` operates on a file of account records, so you can gather the password data from many machines and process them in one spot. You must ensure, however, that they all use the same hashing algorithms compiled into the version you built on your cracking host. For security, it might be wise to gather your account databases, then perform the cracking on a box off the network, in a secure location.

There are other crackers available, notably Crack by Alec Muffet. [9.2] We feature John the Ripper here not because it's necessarily better, but because it's simpler to use on Red Hat 8.0, automatically detecting and supporting the default MD5 hashes.

See Also

See the *doc* directory of the John the Ripper distribution for full documentation and examples.

Learn about Alec Muffet's Crack utility at *http://www.users.dircon.co.uk/~crypto/download/c50-faq.HTML*.

The Red Hat Guide to Password Security is at *http://www.redhat.com/docs/manuals/linux/RHL-8.0-Manual/security-guide/s1-wstation-pass.html*.

9.2 Testing Login Passwords (CrackLib)

Problem

You want assurance that your login passwords are secure.

Solution

Write a little program that calls the `FascistCheck` function from CrackLib:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <crack.h>
#define DICTIONARY "/usr/lib/cracklib_dict"
int main(int argc, char *argv[]) {
     char *password;
     char *problem;
     int status = 0;
     printf("\nEnter an empty password or Ctrl-D to quit.\n");
     while ((password = getpass("\nPassword: ")) != NULL && *password ) {
          if ((problem = FascistCheck(password, DICTIONARY)) != NULL) {
               printf("Bad password: %s.\n", problem);
               status = 1;
          } else {
               printf("Good password!\n");
          }
     }
     exit(status);
}
```

Compile and link it thusly:

```
$ gcc cracktest.c -lcrack -o cracktest
```

Run it (the passwords you type will not appear on the screen):

```
$ ./cracktest
Enter an empty password or Ctrl-D to quit.
Password: xyz
Bad password: it's WAY too short.
Password: elephant
Bad password: it is based on a dictionary word.
Password: kLu%ziF7
Good password!
```

Discussion

CrackLib is an offshoot of Alec Muffet's password cracker, Crack. It is designed to be embedded in other programs, and hence is provided only as a library (and dictionary). The `FascistCheck` function subjects a password to a variety of tests, to ensure that it is not vulnerable to guessing.

See Also

Learn more about CrackLib at *http://www.crypticide.org/users/alecm*.

Perl for System Administration (O'Reilly), section 10.5, shows how to make a Perl module to use CrackLib.

PAM can use CrackLib to force users to choose good passwords. [4.2]

9.3 Finding Accounts with No Password

Problem

You want to detect local login accounts that can be accessed without a password.

Solution

```
# awk -F: '$2 == "" { print $1, "has no password!" }' /etc/shadow
```

Discussion

The worst kind of password is no password at all, so you want to make sure every account has one. Any good password-cracking program can be employed here—they often try to find completely unprotected accounts first—but you can also look for missing passwords directly.

Encrypted passwords are stored in the second field of each entry in the shadow password database, just after the username. Fields are separated by colons.

Note that the *shadow* password file is readable only by superusers.

See Also

shadow(5).

9.4 Finding Superuser Accounts

Problem

You want to list all accounts with superuser access.

Solution

```
$ awk -F: '$3 == 0 { print $1, "is a superuser!" }' /etc/passwd
```

Discussion

A superuser, by definition, has a numerical user ID of zero. Be sure your system has only one superuser account: root. Multiple superuser accounts are a very bad idea because they are harder to control and track. (See Chapter 5 for better ways to share root privileges.)

Numerical user IDs are stored in the third field of each entry in the *passwd* database. The username is stored in the first field. Fields are separated by colons.

See Also

passwd(5).

9.5 Checking for Suspicious Account Use

Problem

You want to discover unusual or dangerous usage of accounts on your system: dormant user accounts, recent logins to system accounts, etc.

Solution

To print information about the last login for each user:

```
$ lastlog [-u username]
```

To print the entire login history:

```
$ last [username]
```

To print failed login attempts:

```
$ lastb [username]
```

To enable recording of bad logins:

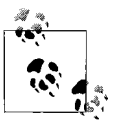
```
# touch /var/log/btmp
```

```
# chown --reference=/var/log/wtmp /var/log/btmp
# chmod --reference=/var/log/wtmp /var/log/btmp
```

Discussion

Attackers look for inactive accounts that are still enabled, in the hope that intrusions will escape detection for long periods of time. If Joe retired and left the organization last year, will anyone notice if his account becomes compromised? Certainly not Joe! To avoid problems like this, examine all accounts on your system for unexpected usage patterns.

Linux systems record each user's last login time in the database */var/log/lastlog*. The terminal (or X Window System display name) and remote system name, if any, are also noted. The `lastlog` command prints this information in a convenient, human-readable format.



/var/log/lastlog is a database, not a log file. It does not grow continuously, and therefore should not be rotated. The apparent size of the file (e.g., as displayed by `ls -l`) is often much larger than the actual size, because the file contains "holes" for ranges of unassigned user IDs.

Access is restricted to the superuser by recent versions of Red Hat (8.0 or later). If this seems too paranoid for your system, it is safe to make the file world-readable:

```
# chmod a+r /var/log/lastlog
```

In contrast, the *btmp* log file will grow slowly (unless you are under attack!), but it should be rotated like other log files. You can either add *btmp* to the *wtmp* entry in */etc/logrotate.conf*, or add a similar entry in a separate file in the */etc/logrotate.d* directory. [9.30]

A history of all logins and logouts (interspersed with system events like shutdowns, reboots, runlevel changes, etc.) is recorded in the log file */var/log/wtmp*. The `last` command scans this log file to produce a report of all login sessions, in reverse chronological order, sorted by login time.

Failed login attempts can also be recorded in the log file */var/log/btmp*, but this is not done by default. To enable recording of bad logins, create the *btmp* file manually, using the same owner, group, and permissions as for the *wtmp* file. The `lastb` command prints a history of bad logins.

The preceding methods do not scale well to multiple systems, so see our more general solution. [9.6]

See Also

lastlog(1), last(1), lastb(1).

9.6 Checking for Suspicious Account Use, Multiple Systems

Problem

You want to scan multiple computers for unusual or dangerous usage of accounts.

Solution

Merge the *lastlog* databases from several systems, using Perl:

```
use DB_File;
use Sys::Lastlog;
use Sys::Hostname;
my %omnilastlog;
tie(%omnilastlog, "DB_File", "/share/omnilastlog");
my $ll = Sys::Lastlog->new( );
while (my ($user, $uid) = (getpwent( ))[0, 2]) {
    if (my $llent = $ll->getlluid($uid)) {
        $omnilastlog{$user} = pack("Na*", $llent->ll_time( ),
                                          join("\0", $llent->ll_line( ),
                                                     $llent->ll_host( ),
                                                     hostname))
            if $llent->ll_time( ) >
                (exists($omnilastlog{$user}) ?
                    unpack("N", $omnilastlog{$user}) : -1);
    }
}
untie(%omnilastlog);
exit(0);
```

To read the merged *lastlog* database, *omnilastlog*, use another Perl script:

```
use DB_File;
my %omnilastlog;
tie(%omnilastlog, "DB_File", "/share/omnilastlog");
while (my ($user, $record) = each(%omnilastlog)) {
    my ($time, $rest) = unpack("Na*", $record);
    my ($line, $host_from, $host_to) = split("\0", $rest, -1);
    printf("%-8.8s %-16.16s -> %-16.16s %-8.8s %s\n",
        $user, $host_from, $host_to, $line,
        $time ? scalar(localtime($time)) : "**Never logged in**");
}
untie(%omnilastlog);
exit(0);
```

Discussion

Perusing the output from the `lastlog`, `last`, and `lastb` commands [9.5] might be sufficient to monitor activity on a single system with a small number of users, but the technique doesn't scale well in the following cases:

- If accounts are shared among many systems, you probably want to know a user's most recent login on *any* of your systems.
- Some system accounts intended for special purposes, such as bin or daemon, should *never* be used for routine logins.
- Disabled accounts should be monitored to make sure they have no login activity.

Legitimate usage patterns vary, and your goal should be to notice deviations from the norm. We need more flexibility than the preceding tools provide.

We can solve this dilemma through automation. The Perl modules `Sys::Lastlog` and `Sys::Utmp`, which are available from CPAN, can parse and display a system's last-login data. Despite its name, `Sys::Utmp` can process the *wtmp* and *btmp* files; they have the same format as */var/log/utmp*, the database containing a snapshot of currently logged-in users.

Our recipe merges *lastlog* databases from several systems into a single database, which we call *omnilastlog*, using Perl. The script steps through each entry in the password database on each system, looks up the corresponding entry in the *lastlog* database using the `Sys::Lastlog` module, and updates the entry in the merged *omnilastlog* database if the last login time is more recent than any other we have previously seen.

The merged *omnilastlog* database is tied to a hash for easy access. We use the Berkeley DB format because it is byte-order–independent and therefore portable: this would be important if your Linux systems run on different architectures. If all of your Linux systems are of the same type (e.g., Intel x86 systems), then any other Perl database module could be used in place of `DB_File`.

Our hash is indexed by usernames rather than numeric user IDs, in case the user IDs are not standardized among the systems (a bad practice that, alas, does happen). The record for each user contains the time, terminal (`ll_line`), and remote and local hostnames. The time is packed as an integer in network byte order (another nod to portability: for homogeneous systems, using the native "L" packing template instead of "N" would work as well). The last three values are glued together with null characters, which is safe because the strings never contain nulls.

Run the merge script on all of your systems, as often as desired, to update the merged *omnilastlog* database. Our recipe assumes a shared filesystem location, */share/omnilastlog*; if this is not convenient, copy the file to each system, update it, and then copy it back to a central repository. The merged database is compact, often smaller than the individual *lastlog* databases.

An even simpler Perl script reads and analyzes the merged *omnilastlog* database. Our recipe steps through and unpacks each record in the database, and then prints all of the information, like the `lastlog` command.

This script can serve as a template for checking account usage patterns, according to your own conventions. For example, you might notice dormant accounts by insisting that users with valid shells (as listed in the file */etc/shells*, with the exception of */sbin/nologin*) must have logged in somewhere during the last month. Conversely, you might require that system accounts (recognized by their low numeric user IDs) with invalid shells must never login, anywhere. Finally, you could maintain a database of the dates when accounts are disabled (e.g., as part of a standard procedure when people leave your organization), and demand that no logins occur for such accounts after the termination date for each.

Run a script frequently to verify your assumptions about legitimate account usage patterns. This way, you will be reminded promptly after Joe's retirement party that his account should be disabled, hopefully before crackers start guessing his password.

See Also

The `Sys::Lastlog` and `Sys::Utmp` Perl modules are found at *http://www.cpan.org*.

Perl for System Administration (section 9.2) from O'Reilly shows how to unpack the *utmp* records used for *wtmp* and *btmp* files. O'Reilly's *Perl Cookbook* also has sample programs for reading records from *lastlog* and *wtmp* files: see the `laston` and `tailwtmp` scripts in Chapter 8 of that book.

9.7 Testing Your Search Path

Problem

You want to avoid invoking the wrong program of a given name.

Solution

Ensure that your search path contains no relative directories:

```
$ perl -e 'print "PATH contains insecure relative directory \"$_\"\n"
              foreach grep ! m[^/], split /:/, $ENV{"PATH"}, -1;'
```

Discussion

Imagine you innocently type `ls` while your current working directory is */tmp*, and you discover to your chagrin that you have just run a malicious program, */tmp/ls*,

instead of the expected */bin/ls*. Worse, you might not notice at all, if the rogue program behaves like the real version while performing other nefarious activities silently.

This can happen if your search path contains a period ("."), meaning the current working directory. The possibility of unexpected behavior is higher if "." is early in your search path, but even the last position is not safe: consider the possibility of misspellings. A cracker could create a malicious */tmp/hwo*, a misspelling of the common `who` command, and hope you type "hwo" sometime while you're in */tmp*. As there is no earlier "hwo" in your search path, you'll unintentionally run the cracker's *./hwo* program. (Which no doubt prints, `` `basename $SHELL`: hwo: command not found `` to stderr while secretly demolishing your filesystem.) Play it safe and keep "." out of your search path.

An empty search path element—two adjacent colons, or a leading or trailing colon—also refers to the current working directory. These are sometimes created inadvertently by scripts that paste together the `PATH` environment variable with ":" separators, adding one too many, or adding an extra separator at the beginning or end.

In fact, any relative directories in your search path are dangerous, as they implicitly refer to the current working directory. Remove all of these relative directories: you can still run programs (securely!) by explicitly typing their relative directory, as in:

```
./myprogram
```

Our recipe uses a short Perl script to split the `PATH` environment variable, complaining about any directory that is not absolute (i.e., that does not start with a "/" character). The negative limit (-1) for split is important for noticing troublesome empty directories at the end of the search path.

See Also

environ(5).

9.8 Searching Filesystems Effectively

Problem

You want to locate files of interest to detect security risks.

Solution

Use `find` and `xargs`, but be knowledgeable of their important options and limitations.

Discussion

Are security risks lurking within your filesystems? If so, they can be hard to detect, especially if you must search through mountains of data. Fortunately, Linux provides the powerful tools `find` and `xargs` to help with the task. These tools have so many options, however, that their flexibility can make them seem daunting to use. We recommend the following good practices:

Know your filesystems

Linux supports a wide range of filesystem types. To see the ones configured in your kernel, read the file */proc/filesystems*. To see which filesystems are currently mounted (and their types), run:

```
$ mount
/dev/hda1 on / type ext2 (rw)
/dev/hda2 on /mnt/windows type vfat (rw)
remotesys:/export/spool/mail on /var/spool/mail type nfs
(rw,hard,intr,noac,addr=192.168.10.13)
//MyPC/C$ on /mnt/remote type smbfs (0)
none on /proc type proc (rw)
...
```

with no options or arguments. We see a traditional Linux ext2 filesystem (*/dev/hda1*), a Windows FAT32 filesystem (*/dev/hda2*), a remotely mounted NFS filesystem (*remotesys:/export/spool/mail*), a Samba filesystem (*//MyPC/C$*) mounted remotely, and the *proc* filesystem provided by the kernel. See mount(8) for more details.

Know which filesystems are local and which are remote

Searching network filesystems like NFS partitions can be quite slow. Furthermore, NFS typically maps your local root account to an unprivileged user on the mounted filesystem, so some files or directories might be inaccessible even to root. To avoid these problems when searching a filesystem, run `find` locally on the server that physically contains it.

Be aware that some filesystem types (e.g., for Microsoft Windows) use different models for owners, groups, and permissions, while other filesystems (notably some for CD-ROMs) do not support these file attributes at all. Consider scanning "foreign" filesystems on servers that recognize them natively, and just skip read-only filesystems like CD-ROMs (assuming you know and trust the source).

The standard Linux filesystem type is ext2. If your local filesystems are of this type only,* you can scan them all with a command like:

```
# find / ! -fstype ext2 -prune -o ... (other find options) ...
```

This can be readily extended to multiple local filesystem types (e.g., ext2 and ext3):

* And if they are not mounted on filesystems of other types, which would be an unusual configuration.

```
# find / ! \( -fstype ext2 -o -fstype ext3 \) -prune -o ...
```

The `find -prune` option causes directories to be skipped, so we prune any filesystems that do *not* match our desired types (ext2 or ext3). The following `-o` ("or") operator causes the filesystems that survive the pruning to be scanned.

The `find -xdev` option prevents crossing filesystem boundaries, and can be useful for avoiding uninteresting filesystems that might be mounted. Our recipes use this option as a reminder to be conscious of filesystem types.

Carefully examine permissions

The `find -perm` option can conveniently select a subset of the permissions, optionally ignoring the rest. In the most common case, we are interested in testing for *any* of the permissions in the subset: use a "+" prefix with the permission argument to specify this. Occasionally, we want to test *all* of the permissions: use a "-" prefix instead.* If no prefix is used, then the entire set of permissions is tested; this is rarely useful.

Handle filenames safely

If you scan enough filesystems, you will eventually encounter filenames with embedded spaces or unusual characters like newlines, quotation marks, etc. The null character, however, *never* appears in filenames, and is therefore the only safe separator to use for lists of filenames that are passed between programs.

The `find -print0` option produces null-terminated filenames; `xargs` and `perl` both support a `-0` (zero) option to read them. Useful filters like `sort` and `grep` also understand a `-z` option to use null separators when they read and write data, and `grep` has a separate `-Z` option that produces null-terminated filenames (with the `-l` or `-L` options). Use these options whenever possible to avoid misinterpreting filenames, which can be disastrous when modifying filesystems as root!

Avoid long command lines

The Linux kernel imposes a 128 KB limit on the combined size of command-line arguments and the environment. This limit can be exceeded by using shell command substitution, e.g.:

```
$ mycommand `find ...`
```

Use the `xargs` program instead to collect filename arguments and run commands repeatedly, without exceeding this limit:

```
$ find ... -print0 | xargs -0 -r mycommand
```

* Of course, if the subset contains only a single permission, then there is no difference between "any" and "all," so either prefix can be used.

The `xargs -r` option avoids running the command if the output of `find` is empty, i.e., no filenames were found. This is usually desirable, to prevent errors like:

```
$ find ... -print0 | xargs -0 rm
rm: too few arguments
```

It can occasionally be useful to connect multiple `xargs` invocations in a pipeline, e.g.:

```
$ find ... -print0 | xargs -0 -r grep -lZ pattern | xargs -0 -r mycommand
```

The first `xargs` collects filenames from `find` and passes them to `grep`, as command-line arguments. `grep` then searches the file contents (which `find` cannot do) for the pattern, and writes another list of filenames to stdout. This list is then used by the second `xargs` to collect command-line arguments for `mycommand`.

If you want `grep` to select filenames (instead of contents), insert it directly into the pipe:

```
$ find ... -print0 | grep -z pattern | xargs -0 -r mycommand
```

In most cases, however, `find -regex` *`pattern`* is a more direct way to select filenames using a regular expression.

Note how `grep -Z` refers to writing filenames, while `grep -z` refers to reading and writing data.

`xargs` is typically much faster than `find -exec`, which runs the command separately for each file and therefore incurs greater start-up costs. However, if you need to run a command that can process only one file at a time, use either `find -exec` or `xargs -n 1`:

```
$ find ... -exec mycommand '{}' \;
$ find ... -print0 | xargs -0 -r -n 1 mycommand
```

These two forms have a subtle difference, however: a command run by `find -exec` uses the standard input inherited from `find`, while a command run by `xargs` uses the pipe as its standard input (which is not typically useful).

See Also

find(1), xargs(1), mount(8).

9.9 Finding setuid (or setgid) Programs

Problem

You want to check for potentially insecure setuid (or setgid) programs.

Solution

To list all setuid or setgid files (programs and scripts):

```
$ find /dir -xdev -type f -perm +ug=s -print
```

To list only setuid or setgid scripts:

```
$ find /dir -xdev -type f -perm +ug=s -print0 | \
perl -0ne 'chomp;
           open(FILE, $_);
           read(FILE, $magic, 2);
           print $_, "\n" if $magic eq "#!";
           close(FILE)'
```

To remove setuid or setgid bits from a file:

```
$ chmod u-s file        Remove the setuid bit
$ chmod g-s file        Remove the setgid bit
```

To find and interactively fix setuid and setgid programs:

```
$ find /dir -xdev -type f \
    \(  -perm +u=s -printf "setuid: %p\n" -ok chmod -v u-s {} \; , \
        -perm +g=s -printf "setgid: %p\n" -ok chmod -v g-s {} \;   \)
```

To ignore the setuid or setgid attributes for executables in a filesystem, mount it with the `nosuid` option. To prohibit executables entirely, use the `noexec` mount option. These options can appear on the command line:

```
# mount -o nosuid ...
# mount -o noexec ...
```

or in */etc/fstab*:

```
/dev/hdd3   /home   ext2    rw,nosuid   1 2
/dev/hdd7   /data   ext2    rw,noexec   1 3
```

Be aware of the important options and limitations of `find`, so you don't inadvertently overlook important files. [9.8]

Discussion

If your system has been compromised, it is quite likely that an intruder has installed backdoors. A common ploy is to hide a setuid root program in one of your filesystems.

The setuid permission bit changes the effective user ID to the owner of the file (even root) when a program is executed; the setgid bit performs the same function for the group. These two attributes are independent: either or both may be set.

Programs (and especially scripts) that use setuid or setgid bits must be written very carefully to avoid security holes. Whether you are searching for backdoors or auditing your own programs, be aware of any activity that involves these bits.

Many setuid and setgid programs are legitimately included in standard Linux distributions, so do not panic if you detect them while searching directories like */usr*. You can maintain a list of known setuid and setgid programs, and then compare the list

with results from more recent filesystem scans. Tripwire (Chapter 1) is an even better tool for keeping track of such changes.

Our recipe uses `find` to detect the setuid and setgid bits. By restricting attention to regular files (with `-type f`), we avoid false matches for directories, which use the setgid bit for an unrelated purpose. In addition, our short Perl program identifies scripts, which contain "#!" in the first two bytes (the magic number).

The `chmod` command removes setuid or setgid bits (or both) for individual files. We can also combine detection with interactive repair using `find`: our recipe tests each bit separately, prints a message if it is found, asks (using `-ok`) if a `chmod` command should be run to remove the bit, and finally confirms each repair with `chmod -v`. Commands run by `find -ok` (or `-exec`) must be terminated with a "\;" argument, and the "{}" argument is replaced by the filename for each invocation. The separate "," (comma) argument causes `find` to perform the tests and actions for the setuid and setgid bits independently.

Finally, `mount` options can offer some protection against misuse of setuid or setgid programs. The `nosuid` option prevents recognition of either bit, which might be appropriate for network filesystems mounted from a less trusted server, or for local filesystems like */home* or */tmp*.* The even more restrictive `noexec` option prevents execution of any programs on the filesystem, which might be useful for filesystems that should contain only data files.

See Also

find(1), xargs(1), chmod(1), perlsec(1).

9.10 Securing Device Special Files

Problem

You want to check for potentially insecure device special files.

Solution

To list all device special files (block or character):

```
$ find /dir -xdev \( -type b -o -type c \) -ls
```

To list any regular files in */dev* (except the `MAKEDEV` program):

```
$ find /dev -type f ! -name MAKEDEV -print
```

* Note that Perl's `suidperl` program does not honor the `nosuid` option for filesystems that contain setuid Perl scripts.

To prohibit device special files on a filesystem, use `mount -o nodev` or add the `nodev` option to entries in */etc/fstab*.

Be aware of the important options and limitations of `find`, so you don't inadvertently overlook important files. [9.8]

Discussion

Device special files are objects that allow direct access to devices (either real or virtual) via the filesystem. For the security of your system, you must carefully control this access by maintaining appropriate permissions on these special files. An intruder who hides extra copies of important device special files can use them as backdoors to read—or even modify—kernel memory, disk drives, and other critical devices.

Conventionally, device special files are installed only in the */dev* directory, but they can live anywhere in the filesystem, so don't limit your searches to */dev*. Our recipe looks for the two flavors of device special files: block and character (using `-type b` and `-type c`, respectively). We use the more verbose `-ls` (instead of `-print`) to list the major and minor device numbers for any that are found: these can be compared to the output from `ls -l /dev` to determine the actual device (the filename is irrelevant).

It is also worthwhile to monitor the */dev* directory, to ensure that no regular files have been hidden there, either as replacements for device special files, or as rogue (perhaps setuid) programs. An exception is made for the */dev/MAKEDEV* program, which creates new entries in */dev*.

The mount option `nodev` prevents recognition of device special files. It is a good idea to use this for any filesystem that does not contain */dev*, especially network filesystems mounted from less trusted servers.

See Also

find(1).

9.11 Finding Writable Files

Problem

You want to locate world-writable files and directories on your machine.

Solution

To find world-writable files:

```
$ find /dir -xdev -perm +o=w ! \( -type d -perm +o=t \) ! -type l -print
```

To disable world write access to a file:

```
$ chmod o-w file
```

To find and interactively fix world-writable files:

```
$ find /dir -xdev -perm +o=w ! \( -type d -perm +o=t \) ! -type l -ok chmod -v o-w {} \;
```

To prevent newly created files from being world-writable:

```
$ umask 002
```

Be aware of the important options and limitations of `find`, so you don't inadvertently overlook important files. [9.8]

Discussion

Think your system is free of world-writable files? Check anyway: you might be surprised. For example, files extracted from Windows Zip archives are notorious for having insecure or screwed-up permissions.

Our recipe skips directories that have the sticky bit set (e.g., */tmp*). Such directories are often world-writable, but this is safe because of restrictions on removing and renaming files. [7.2]

We also skip symbolic links, since their permission bits are ignored (and are usually all set). Only the permissions of the targets of symbolic links are relevant for access control.

The `chmod` command can disable world-write access. Combine it with `find -ok` and you can interactively detect and repair world-writable files.

You can avoid creating world-writable files by setting a bit in your `umask`. You also can set other bits for further restrictions. [7.1] Note that programs like `unzip` are free to override the umask, however, so you still need to check.

See Also

find(1), chmod(1). See your shell documentation for information on `umask`: bash(1), tcsh(1), etc.

9.12 Looking for Rootkits

Problem

You want to check for evidence that a rootkit—a program to create or exploit security holes—has been run on your system.

Solution

Use `chkrootkit`. Download the tarfile from *http://www.chkrootkit.org*, verify its checksum:

```
$ md5sum chkrootkit.tar.gz
```

unpack it:

```
$ tar xvzpf chkrootkit.tar.gz
```

build it:

```
$ cd chkrootkit-*
$ make sense
```

and run it as root:

```
# ./chkrootkit
```

More securely, run it using known, good binaries you have previously copied to a secure medium, such as CD-ROM, e.g.:

```
# ./chkrootkit -p /mnt/cdrom
```

Discussion

`chkrootkit` tests for the presence of certain rootkits, worms, and trojans on your system. If you suspect you've been hacked, this is a good first step toward confirmation and diagnosis.

`chkrootkit` invokes a handful of standard Linux commands. At press time they are `awk`, `cut`, `egrep`, `find`, `head`, `id`, `ls`, `netstat`, `ps`, `strings`, `sed`, and `uname`. If these programs have been compromised on your system, `chkrootkit`'s output cannot be trusted. So ideally, you should keep around a CD-ROM or write-protected floppy disk with these programs, and run `chkrootkit` with the `-p` option to use these known good binaries.

Be sure to use the latest version of `chkrootkit`, which will be aware of the most recently discovered threats.

See Also

The *README* file included with `chkrootkit` explains the tests conducted, and lists the full usage information.

9.13 Testing for Open Ports

Problem

You want a listing of open network ports on your system.

Solution

Probe your ports from a remote system.

To test a specific TCP port (e.g., SSH):

```
$ telnet target.example.com ssh
$ nc -v -z target.example.com ssh
```

To scan most of the interesting TCP ports:

```
# nmap -v target.example.com
```

To test a specific UDP port (e.g., 1024):

```
$ nc -v -z -u target.example.com 1024
```

To scan most of the interesting UDP ports (slowly!):

```
# nmap -v -sU target.example.com
```

To do host discovery (only) for a range of addresses, without port scanning:

```
# nmap -v -sP 10.12.104.200-222
```

To do operating system fingerprinting:

```
# nmap -v -O target.example.com
```

For a handy (but less flexible) GUI, run `nmapfe` instead of `nmap`.

Discussion

When attackers observe your systems from the outside, what do they see? Obviously, you want to present an image of an impenetrable fortress, not a vulnerable target. You've designed your defenses accordingly: a carefully constructed firewall, secure network services, etc. But how can you really be sure?

You don't need to wait passively to see what will happen next. Instead, actively test your own armor with the same tools the attackers will use.

Your vulnerability to attack is influenced by several interacting factors:

The vantage point of the attacker
: Firewalls sometimes make decisions based on the source IP address (or the source port).

All intervening firewalls
: You have your own, of course, but your ISP might impose additional restrictions on incoming or even outgoing traffic from your site.

The network configuration of your systems
: Which servers listen for incoming connections and are willing to accept them?

Start by testing the last two subsystems in isolation. Verify your firewall operation by simulating the traversal of packets through `ipchains`. [2.21] Examine the network state on your machines with `netstat`. [9.14]

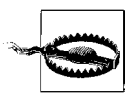
Next, the acid test is to probe from the outside. Use your own accounts on distant systems, if you have them (and if you have permission to do this kind of testing, of course). Alternatively, set up a temporary test system immediately outside your firewall, which might require cooperation from your ISP.

The `nmap` command is a powerful and widely used tool for network security testing. It gathers information about target systems in three distinct phases, in order:

Host discovery
: Initial probes to determine which machines are responding within an address range

Port scanning
: More exhaustive tests to find open ports that are not protected by firewalls, and are accepting connections

Operating system fingerprinting
: An analysis of network behavioral idiosyncrasies can reveal a surprising amount of detailed information about the targets



Use `nmap` to test only systems that you maintain. Many system administrators consider port scanning to be hostile and antisocial. If you intend to use `nmap`'s stealth features, obtain permission from third parties that you employ as decoys or proxies.

Inform your colleagues about your test plans, so they will not be alarmed by unexpected messages in system logs. Use the `logger` command [9.31] to record the beginning and end of your tests.

Use caution when probing mission-critical, production systems. You *should* test these important systems, but `nmap` deliberately violates network protocols, and this behavior can occasionally confuse or even crash target applications and kernels.

To probe a single target, specify the hostname or address:

```
# nmap -v target.example.com
# nmap -v 10.12.104.200
```

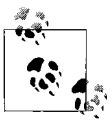
We highly recommend the `-v` option, which provides a more informative report. Repeat the option (`-v -v`...) for even more details.

You can also scan a range of addresses, e.g., those protected by your firewall. For a class C network, which uses the first three bytes (24 bits) for the network part of each address, the following commands are all equivalent:

```
# nmap -v target.example.com/24
# nmap -v 10.12.104.0/24
# nmap -v 10.12.104.0-255
# nmap -v "10.12.104.*"
```

Lists of addresses (or address ranges) can be scanned as well:

```
# nmap -v 10.12.104.10,33,200-222,250
```



nmapfe is a graphical front end that runs nmap with appropriate command-line options and displays the results. nmapfe is designed to be easy to use, though it does not provide the full flexibility of all the nmap options.

By default, nmap uses both TCP and ICMP pings for host discovery. If these are blocked by an intervening firewall, the nmap -P options provide alternate ping strategies. Try these options when evaluating your firewall's policies for TCP or ICMP. The goal of host discovery is to avoid wasting time performing port scans for unused addresses (or machines that are down). If you know that your targets are up, you can disable host discovery with the -P0 (that's a zero) option.

The simplest way to test an individual TCP port is to try to connect with telnet. The port might be open:

```
$ telnet target.example.com ssh
Trying 10.12.104.200...
Connected to target.example.com.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.1p1
```

or closed (i.e., passed by the firewall, but having no server accepting connections on the target):

```
$ telnet target.example.com 33333
Trying 10.12.104.200...
telnet: connect to address 10.12.104.200: Connection refused
```

or blocked (filtered) by a firewall:

```
$ telnet target.example.com 137
Trying 10.12.104.200...
telnet: connect to address 10.12.104.200: Connection timed out
```

Although telnet's primary purpose is to implement the Telnet protocol, it is also a simple, generic TCP client that connects to arbitrary ports.

The nc command is an even better way to probe ports:

```
$ nc -z -vv target.example.com ssh 33333 137
target.example.com [10.12.104.200] 22 (ssh) open
```

```
target.example.com [10.12.104.200] 33333 (?) : Connection refused
target.example.com [10.12.104.200] 137 (netbios-ns) : Connection timed out
```

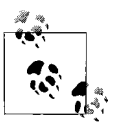
The `-z` option requests a probe, without transferring any data. The repeated `-v` options control the level of detail, as for `nmap`.

Port scans are a *tour de force* for `nmap`:

```
# nmap -v target.example.com
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
No tcp,udp, or ICMP scantype specified, assuming SYN Stealth scan.
Use -sP if you really don't want to portscan (and just want to see what hosts are
up).
Host target.example.com (10.12.104.200) appears to be up ... good.
Initiating SYN Stealth Scan against target.example.com (10.12.104.200)
Adding open port 53/tcp
Adding open port 22/tcp
The SYN Stealth Scan took 21 seconds to scan 1601 ports.
Interesting ports on target.example.com (10.12.104.200):
(The 1595 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
53/tcp     open        domain
137/tcp    filtered    netbios-ns
138/tcp    filtered    netbios-dgm
139/tcp    filtered    netbios-ssn
1080/tcp   filtered    socks
Nmap run completed -- 1 IP address (1 host up) scanned in 24 seconds
```

In all of these cases, be aware that intervening firewalls can be configured to return TCP RST packets for blocked ports, which makes them appear closed rather than filtered. *Caveat prober*.

`nmap` can perform more sophisticated (and efficient) TCP probes than ordinary connection attempts, such as the SYN or "half-open" probes in the previous example, which don't bother to do the full initial TCP handshake for each connection. Different probe strategies can be selected with the `-s` options: these might be interesting if you are reviewing your firewall's TCP policies, or you want to see how your firewall logs different kinds of probes.



Run `nmap` as root if possible. Some of its more advanced tests intentionally violate IP protocols, and require raw sockets that only the superuser is allowed to access.

If `nmap` can't be run as root, it will still work, but it may run more slowly, and the results may be less informative.

UDP ports are harder to probe than TCP ports, because packet delivery is not guaranteed, so blocked ports can't be reliably distinguished from lost packets. Closed ports can be detected by ICMP responses, but scanning is often very slow because many systems limit the rate of ICMP messages. Nevertheless, your firewall's UDP

policies are important, so testing is worthwhile. The `nc -u` and `nmap -sU` options perform UDP probes, typically by sending a zero-byte UDP packet and noting any responses.

By default, `nmap` scans all ports up to 1024, plus well-known ports in its extensive collection of services (used in place of the more limited */etc/services*). Use the `-F` option to quickly scan only the well-known ports, or the `-p` option to select different, specific, numeric ranges of ports. If you want to exhaustively scan *all* ports, use `-p 0-65535`.

If you are interested only in host discovery, disable port scanning entirely with the `nmap -sP` option. This might be useful to determine which occasionally-connected laptops are up and running on an internal network.

Finally, the `nmap -O` option enables operating system fingerprinting and related tests that reveal information about the target:

```
# nmap -v -O target.example.com
...
For OSScan assuming that port 22 is open and port 1 is closed and neither are
firewalled
...
Remote operating system guess: Linux Kernel 2.4.0 - 2.5.20
Uptime 3.167 days (since Mon Feb 21 12:22:21 2003)
TCP Sequence Prediction: Class=random positive increments
                         Difficulty=4917321 (Good luck!)
IPID Sequence Generation: All zeros

Nmap run completed -- 1 IP address (1 host up) scanned in 31 seconds
```

Fingerprinting requires an open and a closed port, which are chosen automatically (so a port scan is required). `nmap` then determines the operating system of the target by noticing details of its IP protocol implementation: Linux is readily recognized (even the version!). It guesses the uptime using the TCP timestamp option. The TCP and IPID Sequence tests measure vulnerability to forged connections and other advanced attacks, and Linux performs well here.

It is sobering to see how many details `nmap` can learn about a system, particularly by attackers with no authorized access. Expect that attacks on your Linux systems will focus on known Linux-specific vulnerabilities, especially if you are using an outdated kernel. To protect yourself, keep up to date with security patches.

`nmap` can test for other vulnerabilities of specific network services. If you run an open FTP server, try `nmap -b` to see if it can be exploited as a proxy. Similarly, if you allow access to an IDENT server, use `nmap -I` to determine if attackers can learn the username (especially root!) that owns other open ports. The `-sR` option displays information about open RPC services, even without direct access to your portmapper.

If your firewall makes decisions based on source addresses, run `nmap` on different remote machines to test variations in behavior. Similarly, if the source port is consulted by your firewall policies, use the `nmap -g` option to pick specific source ports.

The `nmap -o` options save results to log files in a variety of formats. The XML format (`-oX`) is ideal for parsing by scripts: try the `XML::Simple` Perl module for an especially easy way to read the structured data. Alternately, the `-oG` option produces results in a simplified format that is designed for searches using grep. The `-oN` option uses the same human-readable format that is printed to stdout, and `-oA` writes all three formats to separate files.

`nmap` supports several stealth options that attempt to disguise the source of attacks by using third-parties as proxies or decoys, or to escape detection by fragmenting packets, altering timing parameters, etc. These can occasionally be useful for testing your logging and intrusion detection mechanisms, like Snort. [9.20]

See Also

nmap(1), nmapfe(1), nc(1), telnet(1). The `nmap` home page is *http://www.insecure.org/nmap*.The `XML::Simple` Perl module is found on CPAN, *http://www.cpan.org*.

9.14 Examining Local Network Activities

Problem

You want to examine network use occurring on your local machine.

Solution

To print a summary of network use:

```
$ netstat --inet                 Connected sockets
$ netstat --inet --listening     Server sockets
$ netstat --inet --all           Both
# netstat --inet ... -p          Identify processes
```

To print dynamically assigned ports for RPC services:

```
$ rpcinfo -p [host]
```

To list network connections for all processes:

```
# lsof -i[TCP|UDP][@host][:port]
```

To list all open files for specific processes:

```
# lsof -p pid
# lsof -c command
# lsof -u username
```

The /proc Filesystem

Programs like `ps`, `netstat`, and `lsof` obtain information from the Linux kernel via the */proc* filesystem. Although */proc* looks like an ordinary file hierarchy (e.g., you can run */bin/ls* for a directory listing), it actually contains simulated files. These files are like windows into the kernel, presenting its data structures in an easy-to-read manner for programs and users, generally in text format. For example, the file */proc/mounts* contains the list of currently mounted filesystems:

```
$ cat /proc/mounts
/dev/root / ext2 rw 0 0
/proc /proc proc rw 0 0
/dev/hda9 /var ext2 rw 0 0
...
```

but if you examine the file listing:

```
$ ls -l /proc/mounts
-r--r--r--    1 root     root            0 Feb 23 17:07 /proc/mounts
```

you'll see several curious things. The file has zero size, yet it "contains" the mounted filesystem data, because it's a simulated file. Also its "last modified" timestamp is the current time. The permission bits are accurate: this file is world-readable but not writable.[a] The kernel enforces these access restrictions just as for ordinary files.

You can read */proc* files directly, but it's usually more convenient to use programs like `ps`, `netstat`, and `lsof` because:

- They combine data from a wide range of */proc* files into an informative report.
- They have options to control the output format or select specific information.
- Their output format is usually more portable than the format of the corresponding */proc* files, which are Linux-specific and can change between kernel versions (although considerable effort is expended to provide backward compatibility). For instance, the output of `lsof -F` is in a standardized format, and therefore easily parsed by other programs.

Nevertheless, */proc* files are sometimes ideal for scripts or interactive use. The most important files for networking are */proc/net/tcp* and */proc/net/udp*, both consulted by `netstat`. Kernel parameters related to networking can be found in the */proc/sys/net* directory.

Information for individual processes is located in */proc/<pid>* directories, where *<pid>* is the process ID. For example, the file */proc/12345/cmdline* contains the original command line that invoked the (currently running) process 12345. Programs like `ps` summarize the data in these files. Each process directory contains a */proc/<pid>/fd* subdirectory with links for open files: this is used by the `lsof` command.

For more details about the format of files in the */proc* filesystem, see the proc(5) manpage, and documentation in the Linux kernel source distribution, specifically:

/usr/src/linux/Documentation/filesystems/proc.txt*

[a] Imagine the havoc one could wreak by writing arbitrary text into a kernel data structure.

To list all open files (and network connections) for all processes:

```
# lsof
```

To trace network system calls, use `strace`. [9.15]

Discussion

Suppose you see a process with an unfamiliar name running on your system. Should you be concerned? What is it doing? Could it be surreptitiously transmitting data to some other machine on a distant continent?

To answer these kinds of questions, you need tools for observing network use and for correlating activities with specific processes. Use these tools frequently so you will be familiar with normal network usage, and equipped to focus on suspicious behavior when you encounter it.

The `netstat` command prints a summary of the state of networking on your machine, and is a good way to start investigations. The `--inet` option prints active connections:

```
$ netstat --inet
Active Internet connections (w/o servers)
Proto   Recv-Q Send-Q Local Address           Foreign Address         State
tcp     0      240    myhost.example.com:ssh  client.example.com:3672 ESTABLISHED
tcp     0      0      myhost.example.com:4099 server.example.com:ssh  TIME_WAIT
```

This example shows inbound and outbound `ssh` connections; the latter is shutting down (as indicated by `TIME_WAIT`). If you see an unusually large number of connections in the `SYN_RECV` state, your system is probably being probed by a port scanner like `nmap`. [9.13]

Add the `--listening` option to instead see server sockets that are ready to accept new connections (or use `--all` to see both kinds of sockets):

```
$ netstat --inet --listening
Active Internet connections (only servers)
Proto Recv-Q   Send-Q   Local Address   Foreign Address     State
tcp        0        0   *:ssh           *:*                 LISTEN
tcp        0        0   *:http          *:*                 LISTEN
tcp        0        0   *:814           *:*                 LISTEN
udp        0        0   *:ntp           *:*
udp        0        0   *:811           *:*
```

This example shows the `ssh` daemon, a web server (http), a network time server (which uses udp), and two numerical mystery ports, which might be considered suspicious. On a typical system, you would expect to see many more server sockets, and you should try to understand the purpose of each. Consider disabling services that you don't need, as a security precaution.

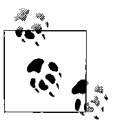
Port numbers for RPC services are assigned dynamically by the portmapper. The `rpcinfo` command shows these assignments:

```
$ rpcinfo -p | egrep -w "port|81[14]"
    program vers proto   port
     100007    2   udp    811  ypbind
     100007    1   udp    811  ypbind
     100007    2   tcp    814  ypbind
     100007    1   tcp    814  ypbind
```

This relieves our concerns about the mystery ports found by `netstat`.

You can even query the portmapper on a different machine, by specifying the hostname on the command line. This is one reason why your firewall should block access to your portmapper, and why you should run it only if you need RPC services.

The `netstat -p` option adds a process ID and command name for each socket, and the `-e` option adds a username.



Only the superuser can examine detailed information for processes owned by others. If you need to observe a wide variety of processes, run these commands as root.

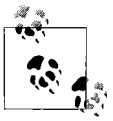
The `lsof` command lists open files for individual processes, including network connections. With no options, `lsof` reports on all open files for all processes, and you can hunt for information of interest using `grep` or your favorite text editor. This technique can be useful when you don't know precisely what you are looking for, because all of the information is available, which provides context. The voluminous output, however, can make specific information hard to notice.

`lsof` provides many options to select files or processes for more refined searches. By default, `lsof` prints information that matches *any* of the selections. Use the `-a` option to require matching *all* of them instead.

The `-i` option selects network connections: `lsof -i` is more detailed than but similar to `netstat --inet --all -p`. The `-i` option can be followed by an argument of the form [TCP|UDP][*@host*][:*port*] to select specific network connections—any or all of the components can be omitted. For example, to view all `ssh` connections (which use TCP), to or from any machine:

```
# lsof -iTCP:ssh
COMMAND PID  USER     FD TYPE DEVICE SIZE NODE NAME
sshd     678 root     3u IPv4   1279       TCP  *:ssh (LISTEN)
sshd    7122 root     4u IPv4 211494       TCP  myhost:ssh->client:367  (ESTABLISHED)
sshd    7125 katie    4u IPv4 211494       TCP  myhost:ssh->client:3672 (ESTABLISHED)
ssh     8145 marianne 3u IPv4 254706       TCP  myhost:3933->server:ssh (ESTABLISHED)
```

Note that a single network connection (or indeed, any open file) can be shared by several processes, as shown in this example. This detail is not revealed by `netstat -p`.



Both `netstat` and `lsof` convert IP addresses to hostnames, and port numbers to service names (e.g., `ssh`), if possible. You can inhibit these conversions and force printing of numeric values, e.g., if you are have many network connections and some nameservers are responding slowly. Use the `netstat --numeric-hosts` or `--numeric-ports` options, or the `lsof -n`, `-P`, or `-l` options (for host addresses, port numbers, and user IDs, respectively) to obtain numeric values, as needed.

To examine processes that use RPC services, the `+M` option is handy for displaying portmapper registrations:

```
# lsof +M -iTCP:814 -iUDP:811
COMMAND  PID  USER  FD  TYPE  DEVICE  SIZE  NODE  NAME
ypbind   633  root  6u  IPv4    1202         UDP  *:811[ypbind]
ypbind   633  root  7u  IPv4    1207         TCP  *:814[ypbind] (LISTEN)
ypbind   635  root  6u  IPv4    1202         UDP  *:811[ypbind]
ypbind   635  root  7u  IPv4    1207         TCP  *:814[ypbind] (LISTEN)
ypbind   636  root  6u  IPv4    1202         UDP  *:811[ypbind]
ypbind   636  root  7u  IPv4    1207         TCP  *:814[ypbind] (LISTEN)
ypbind   637  root  6u  IPv4    1202         UDP  *:811[ypbind]
ypbind   637  root  7u  IPv4    1207         TCP  *:814[ypbind] (LISTEN)
```

This corresponds to `rpcinfo -p` output from our earlier example. The RPC program names are enclosed in square brackets, after the port numbers.

You can also select processes by ID (`-p`), command name (`-c`), or username (`-u`):

```
# lsof -a -c myprog -u tony
COMMAND  PID  USER  FD TYPE  DEVICE    SIZE  NODE   NAME
myprog  8387  tony cwd  DIR   0,15    4096  42329  /var/tmp
myprog  8387  tony rtd  DIR    8,1    4096      2       /
myprog  8387  tony txt  REG    8,2   13798  31551  /usr/local/bin/myprog
myprog  8387  tony mem  REG    8,1   87341  21296  /lib/ld-2.2.93.so
myprog  8387  tony mem  REG    8,1   90444  21313  /lib/libnsl-2.2.93.so
myprog  8387  tony mem  REG    8,1   11314  21309  /lib/libdl-2.2.93.so
myprog  8387  tony mem  REG    8,1  170910  81925  /lib/i686/libm-2.2.93.so
myprog  8387  tony mem  REG    8,1   10421  21347  /lib/libutil-2.2.93.so
myprog  8387  tony mem  REG    8,1   42657  21329  /lib/libnss_files-2.2.93.so
myprog  8387  tony mem  REG    8,1   15807  21326  /lib/libnss_dns-2.2.93.so
myprog  8387  tony mem  REG    8,1   69434  21341  /lib/libresolv-2.2.93.so
myprog  8387  tony mem  REG    8,1 1395734  81923  /lib/i686/libc-2.2.93.so
myprog  8387  tony  0u  CHR  136,3              2  /dev/pts/3
myprog  8387  tony  1u  CHR  136,3              2  /dev/pts/3
myprog  8387  tony  2u  CHR  136,3              2  /dev/pts/3
myprog  8387  tony  3r  REG    8,5        0  98315  /var/tmp/foo
myprog  8387  tony  4w  REG    8,5        0  98319  /var/tmp/bar
myprog  8387  tony  5u  IPv4 274331            TCP  myhost:2944->www:http (ESTABLISHED)
```

Note that the arrow does not indicate the direction of data transfer for network connections: the order displayed is always *`local->remote`*.

The letters following the file descriptor (FD) numbers show that `myprog` has opened the file *foo* for reading (r), the file *bar* for writing (w), and the network connection bidirectionally (u).

The complete set of information printed by `lsof` can be useful when investigating suspicious processes. For example, we can see that `myprog`'s current working directory (cwd) is */var/tmp*, and the pathname for the program (txt) is */usr/local/bin/myprog*. Be aware that rogue programs may try to disguise their identity: if you find `sshd` using the executable */tmp/sshd* instead of */usr/sbin/sshd*, that is cause for alarm. Similarly, it would be troubling to discover a program called "ls" with network connections to unfamiliar ports![*]

See Also

netstat(8), rpcinfo(8), lsof(8).

9.15 Tracing Processes

Problem

You want to know what an unfamiliar process is doing.

Solution

To attach to a running process and trace system calls:

```
# strace -p pid
```

To trace network system calls:

```
# strace -e trace=network,read,write ...
```

Discussion

The `strace` command lets you observe a given process in detail, printing its system calls as they occur. It expands all arguments, return values, and errors (if any) for the system calls, showing all information passed between the process and the kernel. (It can also trace signals.) This provides a very complete picture of what the process is doing.

Use the `strace -p` option to attach to and trace a process, identified by its process ID, say, 12345:

```
# strace -p 12345
```

[*] Even `ls` can legitimately use the network, however, if your system uses NIS for user or group ID lookups. You need to know what to expect in each case.

To detach and stop tracing, just kill `strace`. Other than a small performance penalty, `strace` has no effect on the traced process.

Tracing all system calls for a process can produce overwhelming output, so you can select sets of interesting system calls to print. For monitoring network activity, the `-e trace=network` option is appropriate. Network sockets often use the generic `read` and `write` system calls as well, so trace those too:

```
$ strace -e trace=network,read,write finger katie@server.example.com
…
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 4
connect(4, {sin_family=AF_INET,
            sin_port=htons(79),
            sin_addr=inet_addr("10.12.104.222")}, 16) = 0
write(4, "katie", 5)                    = 5
write(4, "\r\n", 2)                     = 2
read(4, "Login: katie          \t\t\tName: K"..., 4096) = 244
read(4, "", 4096)                       = 0
…
```

The trace shows the creation of a TCP socket, followed by a connection to port 79 for the finger service at the IP address for the server. The program then follows the finger protocol by writing the username and reading the response.

By default, `strace` prints only 32 characters of string arguments, which can lead to the truncated output shown. For a more complete trace, use the `-s` option to specify a larger maximum data size. Similarly, `strace` abbreviates some large structure arguments, such as the environment for new processes: supply the `-v` option to print this information in full.

You can trace most network activity effectively by following file descriptors: in the previous example, the value is 4 (returned by the socket-creation call, and used as the first argument for the subsequent system calls). Then match these values to the file descriptors displayed in the FD column by `lsof`. [9.14]

When you identify an interesting file descriptor, you can print the transferred data in both hexadecimal and ASCII using the options `-e [read|write]=fd`:

```
$ strace -e trace=read -e read=4 finger katie@server.example.com
…
read(4, "Login: katie          \t\t\tName: K"..., 4096) = 244
 | 00000  4c 6f 67 69 6e 3a 20 6b  61 74 69 65 20 20 20 20  Login: k atie     |
 | 00010  20 20 20 20 20 20 09 09  09 4e 61 6d 65 3a 20 4b        .. .Name: K |
…
```

`strace` watches data transfers much like network packet sniffers do, but it also can observe input/output involving local files and other system activities.

If you trace programs for long periods, ask `strace` to annotate its output with timestamps. The `-t` option records absolute times (repeat the option for more detail), the `-r` option records relative times between system calls, and `-T` records time spent in

the kernel within system calls. Finally, add the `strace -f` option to follow child processes.*

Each line of the trace has the process ID added for children. Alternatively, you can untangle the system calls by directing the trace for each child process to a separate file, using the options:

```
$ strace -f -ff -o filename ...
```

See Also

strace(1), and the manpages for the system calls appearing in `strace` output.

9.16 Observing Network Traffic

Problem

You want to watch network traffic flowing by (or through) your machine.

Solution

Use a packet sniffer such as `tcpdump`.†

To sniff packets and save them in a file:

```
# tcpdump -w filename [-c count] [-i interface] [-s snap-length] [expression]
```

To read and display the saved network trace data:

```
$ tcpdump -r filename [expression]
```

To select packets related to particular TCP services to or from a host:

```
# tcpdump tcp port service [or service] and host server.example.com
```

For a convenient and powerful GUI, use Ethereal. [9.17]

To enable an unconfigured interface, for a "stealth" packet sniffer:

```
# ifconfig interface-name 0.0.0.0 up
```

To print information about all of your network interfaces with loaded drivers: [3.1]

```
$ ifconfig -a
```

* To follow child processes created by `vfork`, include the `-F` option as well, but this requires support from the kernel that is not widely available at press time. Also, `strace` does not currently work well with multi-threaded processes: be sure you have the latest version, and a kernel Version 2.4 or later, before attempting thread tracing.

† In spite of its name, `tcpdump` is not restricted to TCP. It can capture entire packets, including the link-level (Ethernet) headers, IP, UDP, etc.

Discussion

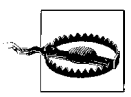
Is your system under attack? Your firewall is logging unusual activities, you see lots of half-open connections, and the performance of your web server is degrading. How can you learn what is happening so you can take defensive action? Use a *packet sniffer* to watch traffic on the network!

In normal operation, network interfaces are programmed to receive only the following:

- *Unicast packets*, addressed to a specific machine
- *Multicast packets*, targeted to systems that choose to subscribe to services like streaming video or sound
- *Broadcast packets*, for when an appropriate destination is not known, or for important information that is probably of interest to all machines on the network

The term "unicast" is not an oxymoron: all packets on networks like Ethernet are in fact sent (conceptually) to all systems on the network. Each system simply ignores unicast packets addressed to other machines, or uninteresting multicast packets.

A packet sniffer puts a network interface into *promiscuous mode*, causing it to receive all packets on the network, like a wiretap. Almost all network adapters support this mode nowadays. Linux restricts the use of promiscuous mode to the superuser, so always run packet-sniffing programs as root. Whenever you switch an interface to promiscuous mode, the kernel logs the change, so we advise running the `logger` command [9.27] to announce your packet-sniffing activities.



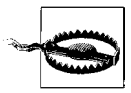
If promiscuous mode doesn't seem to be working, and your kernel is sending complaints to the system logger (usually in */var/log/messages*) that say:

```
modprobe: can't locate module net-pf-17
```

then your kernel was built without support for the packet socket protocol, which is required for network sniffers.

Rebuild your kernel with the option `CONFIG_PACKET=y` (or `CONFIG_PACKET=m` to build a kernel module). Red Hat and SuSE distribute kernels with support for the packet socket protocol enabled, so network sniffers should work.

Network switches complicate this picture. Unlike less intelligent hubs, switches watch network traffic, attempt to learn which systems are connected to each network segment, and then send unicast packets only to ports known to be connected to the destination systems, which defeats packet sniffing. However, many network switches support packet sniffing with a configuration option to send all traffic to designated ports. If you are running a network sniffer on a switched network, consult the documentation for your switch.



The primary purpose of network switches is to improve performance, not to enhance security. Packet sniffing is more difficult on a switched network, but not impossible: `dsniff` [9.19] is distributed with a collection of tools to demonstrate such attacks. Do not be complacent about the need for secure protocols, just because your systems are connected to switches instead of hubs.

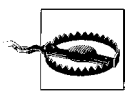
Similarly, routers and gateways pass traffic to different networks based on the destination address for each packet. If you want to watch traffic between machines on different networks, attach your packet sniffer somewhere along the route between the source and destination.

Packet sniffers tap into the network stack at a low level, and are therefore immune to restrictions imposed by firewalls. To verify the correct operation of your firewall, use a packet sniffer to watch the firewall accept or reject traffic.

Your network interface need not even be configured in order to watch traffic (it does need to be up, however). Use the `ifconfig` command to enable an unconfigured interface by setting the IP address to zero:

```
# ifconfig eth2 0.0.0.0 up
```

Unconfigured interfaces are useful for dedicated packet-sniffing machines, because they are hard to detect or attack. Such systems are often used on untrusted networks exposed to the outside (e.g., right next to your web servers). Use care when these "stealth" packet sniffers are also connected (by normally configured network interfaces) to trusted, internal networks: for example, disable IP forwarding. [2.3]



Promiscuous mode can degrade network performance. Avoid running a packet sniffer for long periods on important, production machines: use a separate, dedicated machine instead.

Almost all Linux packet-sniffing programs use `libpcap`, a packet capture library distributed with `tcpdump`. As a fortunate consequence, network trace files share a common format, so you can use one tool to capture and save packets, and others to display and analyze the traffic. The `file` command recognizes and displays information about `libpcap`-format network trace files:

```
$ file trace.pcap
trace.pcap: tcpdump capture file (little-endian) - version 2.4 (Ethernet, capture
length 96)
```

Kernels of Version 2.2 or higher can send warnings to the system logger like:

```
tcpdump uses obsolete (PF_INET,SOCK_PACKET)
```

These are harmless, and can be safely ignored. To avoid the warnings, upgrade to a more recent version of `libpcap`.

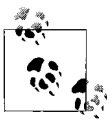
To sniff packets and save them in a file, use the `tcpdump -w` option:

```
# tcpdump -w trace.pcap [-c count] [-i interface] [-s snap-length] [expression]
```

Just kill `tcpdump` when you are done, or use the `-c` option to request a maximum number of packets to record.

If your system is connected to multiple networks, use the `-i` option to listen on a specific interface (e.g., eth2). The `ifconfig` command prints information about all of your network interfaces with loaded drivers: [3.1]

```
$ ifconfig -a
```



The special interface name "any" denotes *all* of the interfaces by any program that uses `libpcap`, but these interfaces are not put into promiscuous mode automatically. Before using `tcpdump -i any`, use `ifconfig` to enable promiscuous mode for specific interfaces of interest:

```
# ifconfig interface promisc
```

Remember to disable promiscuous mode when you are done sniffing:

```
# ifconfig interface -promisc
```

Support for the "any" interface is available in kernel Versions 2.2 or later.

Normally, `tcpdump` saves only the first 68 bytes of each packet. This snapshot length is good for analysis of low-level protocols (e.g., TCP or UDP), but for higher-level ones (like HTTP) use the `-s` option to request a larger snapshot. To capture entire packets and track all transmitted data, specify a snapshot length of zero. Larger snapshots consume dramatically more disk space, and can impact network performance or even cause packet loss under heavy load.

By default, `tcpdump` records all packets seen on the network. Use a *capture filter expression* to select specific packets: the criteria can be based on any data in the protocol headers, using a simple syntax described in the tcpdump(8) manpage. For example, to record FTP transfers to or from a server:

```
# tcpdump -w trace.pcap tcp port ftp or ftp-data and host server.example.com
```

By restricting the kinds of packets you capture, you can reduce the performance implications and storage requirements of larger snapshots.

To read and display the saved network trace data, use the `tcpdump -r` option:

```
$ tcpdump -r trace.pcap [expression]
```

Root access is not required to analyze the collected data, since it is stored in ordinary files. You may want to protect those trace files, however, if they contain sensitive data.

Use a *display filter expression* to print information only about selected packets; display filters use the same syntax as capture filters.

The capture and display operations can be combined, without saving data to a file, if neither the `-w` nor `-r` options are used, but we recommend saving to a file, because:

- Protocol analysis often requires displaying the data multiple times, in different formats, and perhaps using different tools.
- You might want to analyze data captured at some earlier time.
- It is hard to predict selection criteria in advance. Use more inclusive filter expressions at capture time, then more discriminating ones at display time, when you understand more clearly which data is interesting.
- Display operations can be inefficient. Memory is consumed to track TCP sequence numbers, for example. Your packet sniffer should be lean and mean if you plan to run it for long periods.
- Display operations sometimes interfere with capture operations. Converting IP addresses to hostnames often involves DNS lookups, which can be confusing if you are watching traffic to and from your nameservers! Similarly, if you tunnel `tcpdump` output through an SSH connection, that generates additional SSH traffic.

Saving formatted output from `tcpdump` is an even worse idea. It consumes large amounts of space, is difficult for other programs to parse, and discards much of the information saved in the `libpcap`-format trace file. Use `tcpdump -w` to save network traces.

`tcpdump` prints information about packets in a terse, protocol-dependent format meticulously described in the manpage. Suppose a machine 10.6.6.6 is performing a port scan of another machine, 10.9.9.9, by running `nmap -r`. [9.13] If you use `tcpdump` to observe this port scan activity, you'll see something like this:

```
# tcpdump -nn
...
23:08:14.980358 10.6.6.6.6180 > 10.9.9.9.20: S 5498218:5498218(0) win 4096 [tos 0x80]
23:08:14.980436 10.9.9.9.20 > 10.6.6.6.6180: R 0:0(0) ack 5498219 win 0 (DF) [tos 0x80]
23:08:14.980795 10.6.6.6.6180 > 10.9.9.9.21: S 5498218:5498218(0) win 4096 [tos 0x80]
23:08:14.980893 10.9.9.9.21 > 10.6.6.6.6180: R 0:0(0) ack 5498219 win 0 (DF) [tos 0x80]
23:08:14.983496 10.6.6.6.6180 > 10.9.9.9.22: S 5498218:5498218(0) win 4096
23:08:14.984488 10.9.9.9.22 > 10.6.6.6.6180: S 3458349:3458349(0) ack 5498219 win 5840
<mss 1460> (DF)
23:08:14.983907 10.6.6.6.6180 > 10.9.9.9.23: S 5498218:5498218(0) win 4096 [tos 0x80]
23:08:14.984577 10.9.9.9.23 > 10.6.6.6.6180: R 0:0(0) ack 5498219 win 0 (DF) [tos 0x80]
23:08:15.060218 10.6.6.6.6180 > 10.9.9.99.22: R 5498219:5498219(0) win 0 (DF)
23:08:15.067712 10.6.6.6.6180 > 10.9.9.99.24: S 5498218:5498218(0) win 4096
23:08:15.067797 10.9.9.9.24 > 10.6.6.6.6180: R 0:0(0) ack 5498219 win 0 (DF)
23:08:15.068201 10.6.6.6.6180 > 10.9.9.9.25: S 5498218:5498218(0) win 4096 [tos 0x80]
23:08:15.068282 10.9.9.9.25 > 10.6.6.6.6180: R 0:0(0) ack 5498219 win 0 (DF) [tos 0x80]
...
```

The `nmap -r` process scans the ports sequentially. For each closed port, we see an incoming TCP SYN packet, and a TCP RST reply from the target. An open SSH port (22) instead elicits a TCP SYN+ACK reply, indicating that a server is listening: the

scanner responds a short time later with a TCP RST packet (sent out of order) to tear down the half-open SSH connection. Protocol analysis is especially enlightening when a victim is confronted by sneakier probes and denial of service attacks that don't adhere to the usual network protocol rules.

The previous example used `-nn` to print everything numerically. The `-v` option requests additional details; repeat it (`-v -v ...`) for increased verbosity. Timestamps are recorded by the kernel (and saved in `libpcap`-format trace files), and you can select a variety of formats by specifying the `-t` option one or more times. Use the `-e` option to print link-level (Ethernet) header information.

See Also

ifconfig(8), tcpdump(8), nmap(8). The `tcpdump` home page is *http://www.tcpdump.org*, and the `nmap` home page is *http://www.insecure.org/nmap*.

A good reference on Internet protocols is found at *http://www.protocols.com*. Also, the book *Internet Core Protocols: The Definitive Guide* (O'Reilly) covers similar material.

9.17 Observing Network Traffic (GUI)

Problem

You want to watch network traffic via a graphical interface.

Solution

Use Ethereral and `tethereal`.

Discussion

Prolonged perusing of `tcpdump` output [9.16] can lead to eyestrain. Fortunately, alternatives are available, and Ethereal is one of the best.

Ethereal is a GUI network sniffer that supports a number of enhancements beyond the capabilities of `tcpdump`. When Ethereal starts, it presents three windows:

Packet List
: A summary line for each packet, in a format similar to `tcpdump`.

Tree View
: An expandable protocol tree for the packet selected in the previous window. An observer can drill down to reveal individual fields at each protocol level. Ethereal understands and can display an astounding number of protocols in detail.

Data View
: Hexadecimal and ASCII dumps of all bytes captured in the selected packet. Bytes are highlighted according to selections in the protocol tree.

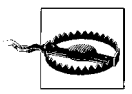
Ethereal uses the same syntax as `tcpdump` for capture filter expressions. However, it uses a different, more powerful syntax for display filter expressions. Our previous `tcpdump` example, to select packets related to FTP transfers to or from a server: [9.16]

```
tcp port ftp or ftp-data and host server.example.com
```

would be rewritten using Ethereal's display filter syntax as:

```
ftp or ftp-data and ip.addr == server.example.com
```

The display filter syntax is described in detail in the ethereal(1) manpage.



If you receive confusing and uninformative syntax error messages, make sure you are not using *display* filter syntax for *capture* filters, or vice-versa.

Ethereal provides a GUI to construct and update display filter expressions, and can use those expressions to find packets in a trace, or to colorize the display.

Ethereal also provides a tool to follow a TCP stream, reassembling (and reordering) packets to construct an ASCII or hexadecimal dump of an entire TCP session. You can use this to view many protocols that are transmitted as clear text.

Menus are provided as alternatives for command-line options (which are very similar to those of `tcpdump`). Ethereal does its own packet capture (using `libpcap`), or reads and writes network trace files in a variety of formats. On Red Hat systems, the program is installed with a wrapper that asks for the root password (required for packet sniffing), and allows running as an ordinary user (if only display features are used).

The easiest way to start using Ethereal is:

1. Launch the program.
2. Use the Capture Filters item in the Edit menu to select the traffic of interest, or just skip this step to capture all traffic.
3. Use the Start item in the Capture menu. Fill out the Capture Preferences dialog box, which allows specification of the interface for listening, the snapshot (or "capture length"), and whether you want to update the display in real time, as the packet capture happens. Click OK to begin sniffing packets.
4. Watch the dialog box (and the updated display, if you selected the real time update option) to see the packet capture in progress. Click the Stop button when you are done.
5. The display is now updated, if it was not already. Try selecting packets in the Packet List window, drill down to expand the Tree View, and select parts of the

protocol tree to highlight the corresponding sections of the Data View. This is a *great* way to learn about internal details of network protocols!

6. Select a TCP packet, and use the Follow TCP Stream item in the Tools menu to see an entire session displayed in a separate window.

Ethereal is amazingly flexible, and this is just a small sample of its functionality. To learn more, browse the menus and see the Ethereal User's Guide for detailed explanations and screen shots.

`tethereal` is a text version of Ethereal, and is similar in function to `tcpdump`, except it uses Ethereal's enhanced display filter syntax. The `-V` option prints the protocol tree for each packet, instead of a one-line summary.

Use the `tethereal -b` option to run in "ring buffer" mode (Ethereal also supports this option, but the mode is designed for long-term operation, when the GUI is not as useful). In this mode, `tethereal` maintains a specified number of network trace files, switching to the next file when a maximum size (determined by the `-a` option) is reached, and discarding the oldest files, similar to `logrotate`. [9.30] For example, to keep a ring buffer with 10 files of 16 megabytes each:

```
# tethereal -w ring-buffer -b 10 -a filesize:16384
```

See Also

ethereal(1), tethereal(1). The Ethereal home page is *http://www.ethereal.com*.

9.18 Searching for Strings in Network Traffic

Problem

You want to watch network traffic, searching for strings in the transmitted data.

Solution

Use `ngrep`.

To search for packets containing data that matches a regular expression and protocols that match a filter expression:

```
# ngrep [grep-options] regular-expression [filter-expression]
```

To search instead for a sequence of binary data:

```
# ngrep -X hexadecimal-digits [filter-expression]
```

To sniff packets and save them in a file:

```
# ngrep -O filename [-n count] [-d interface] [-s snap-length] \
  regular-expression [filter-expression]
```

To read and display the saved network trace data:

```
$ ngrep -I filename regular-expression [filter-expression]
```

Discussion

ngrep is supplied with SuSE but not Red Hat; however, it is easy to obtain and install if you need it. Download it from *http://ngrep.sourceforge.net* and unpack it:

```
$ tar xvpzf ngrep-*.tar.gz
```

compile it:

```
$ cd ngrep
$ ./configure --prefix=/usr/local
$ make
```

and install it into */usr/local* as root:*

```
# mkdir -p /usr/local/bin /usr/local/man/man8
# make install
```

Sometimes we are interested in observing the data delivered by network packets, known as the *payload*. Tools like tcpdump [9.16] and especially Ethereal [9.17] can display the payload, but they are primarily designed for protocol analysis, so their ability to select packets based on arbitrary data is limited.†

The ngrep command searches network traffic for data that matches extended regular expressions, in the same way that the egrep command (or grep -E) searches files. In fact, ngrep supports many of the same command-line options as egrep, such as -i (case-insensitive), -w (whole words), or -v (nonmatching). In addition, ngrep can select packets using the same filter expressions as tcpdump. To use ngrep as an ordinary packet sniffer, use the regular expression ".", which matches any nonempty payload.

ngrep is handy for detecting the use of insecure protocols. For example, we can observe FTP transfers to or from a server, searching for FTP request command strings to reveal usernames, passwords, and filenames that are transmitted as clear text:

```
$ ngrep -t -x 'USER|PASS|RETR|STOR' tcp port ftp and host server.example.com
interface: eth0 (10.44.44.0/255.255.255.0)
filter: ip and ( tcp port ftp )
```

* We explicitly install in */usr/local*, because otherwise the configure script would install into */usr*, based on the location of gcc. We recommend */usr/local* to avoid clashes with vendor-supplied software in */usr*; this recommendation is codified in the Filesystem Hierarchy Standard (FHS), *http://www.pathname.com/fhs*. The configure script used for ngrep is unusual—such scripts typically install into */usr/local* by default, and therefore do not need an explicit --prefix option. We also create the installation directories if they don't already exist, to overcome deficiencies in the make install command.

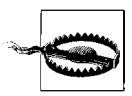
† The concept of a packet's payload is subjective. Each lower-level protocol regards the higher-level protocols as its payload. The highest-level protocol delivers the user data; for example, the files transferred by FTP.

```
match: USER|PASS|RETR|STOR
#############
T 2003/02/27 23:31:20.303636 10.33.33.33:1057 -> 10.88.88.88:21 [AP]
  55 53 45 52 20 6b 61 74    69 65 0d 0a               USER katie..
#####
T 2003/02/27 23:31:25.315858 10.33.33.33:1057 -> 10.88.88.88:21 [AP]
  50 41 53 53 20 44 75 6d    62 6f 21 0d 0a            PASS Dumbo!..
#############
T 2003/02/27 23:32:15.637343 10.33.33.33:1057 -> 10.88.88.88:21 [AP]
  52 45 54 52 20 70 6f 6f    68 62 65 61 72 0d 0a      RETR poohbear..
########
T 2003/02/27 23:32:19.742193 10.33.33.33:1057 -> 10.88.88.88:21 [AP]
  53 54 4f 52 20 68 6f 6e    65 79 70 6f 74 0d 0a      STOR honeypot..
###############exit
58 received, 0 dropped
```

The `-t` option adds timestamps; use `-T` instead for relative times between packets. The `-x` option prints hexadecimal values in addition to the ASCII strings.

`ngrep` prints a hash character (#) for each packet that matches the filter expression: only those packets that match the regular expression are printed in detail. Use the `-q` option to suppress the hashes.

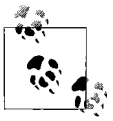
To search for binary data, use the `-X` option with a sequence of hexadecimal digits (of any length) instead of a regular expression. This can detect some kinds of buffer overflow attacks, characterized by known signatures of fixed binary data.



`ngrep` matches data only within individual packets. If strings are split between packets due to fragmentation, they will not be found. Try to match shorter strings to reduce (but not entirely eliminate) the probability of these misses. Shorter strings can also lead to false matches, however—a bit of experimentation is sometimes required. `dsniff` does not have this limitation. [9.19]

Like other packet sniffers, `ngrep` can write and read `libpcap`-format network trace files, using the `-O` and `-I` options. [9.16] This is especially convenient when running `ngrep` repeatedly to refine your search, using data captured previously, perhaps by another program. Usually `ngrep` captures packets until killed, or it will exit after recording a maximum number of packets requested by the `-n` option. The `-d` option selects a specific interface, if your machine has several. By default, `ngrep` captures entire packets (in contrast to `tcpdump` and `ethereal`), since `ngrep` is interested in the payloads. If your data of interest is at the beginning of the packets, use the `-s` option to reduce the snapshot and gain efficiency.

When `ngrep` finds an interesting packet, the adjacent packets might be of interest too, as context. The `ngrep -A` option prints a specified number of extra (not necessarily matching) packets for trailing context. This is similar in spirit to the `grep -A` option, but `ngrep` does not support a corresponding `-B` option for leading context.



A recommended practice: Save a generous amount of network trace data with `tcpdump`, then run `ngrep` to locate interesting data. Finally, browse the complete trace using Ethereal, relying on the timestamps to identify the packets matched by `ngrep`.

See Also

ngrep(8), egrep(1), grep(1), tcpdump(8). The home page for `ngrep` is *http://ngrep.sourceforge.net*, and the `tcpdump` home page is *http://www.tcpdump.org*.

Learn more about extended regular expressions in the O'Reilly book *Mastering Regular Expressions*.

9.19 Detecting Insecure Network Protocols

Problem

You want to determine if insecure protocols are being used on the network.

Solution

Use `dsniff`.

To monitor the network for insecure protocols:

```
# dsniff -m [-i interface] [-s snap-length] [filter-expression]
```

To save results in a database, instead of printing them:

```
# dsniff -w gotcha.db [other options...]
```

To read and print the results from the database:

```
$ dsniff -r gotcha.db
```

To capture mail messages from SMTP or POP traffic:

```
# mailsnarf [-i interface] [-v] [regular-expression [filter-expression]]
```

To capture file contents from NFS traffic:

```
# filesnarf [-i interface] [-v] [regular-expression [filter-expression]]
```

To capture URLs from HTTP traffic:

```
# urlsnarf  [-i interface] [-v] [regular-expression [filter-expression]]
```

`ngrep` is also useful for detecting insecure network protocols. [9.18]

Discussion

`dsniff` is not supplied with Red Hat or SuSE, but installation is straightforward. A few extra steps are required for two prerequisite libraries, `libnet` and `libnids`, not

distributed by Red Hat. SuSE provides these libraries, so you can skip ahead to the installation of `dsniff` itself on such systems.

If you need the libraries, first download `libnet`, a toolkit for network packet manipulation, from *http://www.packetfactory.net/projects/libnet*, and unpack it:

```
$ tar xvzpf libnet-1.0.*.tar.gz
```

Then compile it:*

```
$ cd Libnet-1.0.*
$ ./configure --prefix=/usr/local
$ make
```

and install it as root:

```
# make install
```

We explicitly configure to install in */usr/local* (instead of */usr*), to match the default location for our later configuration steps. Next, download `libnids`, which is used for TCP stream reassembly, from *http://www.packetfactory.net/projects/libnids*, and unpack it:

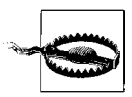
```
$ tar xvzpf libnids-*.tar.gz
```

Then compile it:

```
$ cd `ls -d libnids-* | head -1`
$ ./configure
$ make
```

and install it as root:

```
# make install
```



`dsniff` also requires the Berkeley database library, which is provided by both Red Hat and SuSE. Unfortunately, some systems such as Red Hat 7.0 are missing */usr/include/db_185.h* (either a plain file or a symbolic link) that `dsniff` needs. This is easy to fix:

```
# cd /usr/include
# test -L db.h -a ! -e db_185.h \
    && ln -sv `readlink db.h | sed -e 's,/db,&_185,'` .
```

Your link should look like this:

```
$ ls -l db_185.h
lrwxrwxrwx    1 root root 12 Feb 14 14:56 db_185.h -> db4/db_185.h
```

It's OK if the link points to a different version (e.g., db3 instead of db4).

* At press time, `dsniff` 2.3 (the latest stable version) cannot be built with the most recent version of `libnet`. Be sure to use the older `libnet` 1.0.2a with `dsniff` 2.3.

Finally, download dsniff from *http://naughty.monkey.org/~dugsong/dsniff*, and unpack it:

```
$ tar xvzpf dsniff-*.tar.gz
```

Then compile it:

```
$ cd `ls -d dsniff-* | head -1`
$ ./configure
$ make
```

and install it as root:

```
# make install
```

Whew! With all of the software in place, we can start using dsniff to audit the use of insecure network protocols:

```
# dsniff -m
dsniff: listening on eth0
-----------------
03/01/03 20:11:07 tcp client.example.com.2056 -> server.example.com.21 (ftp)
USER katie
PASS Dumbo!
-----------------
03/01/03 20:11:23 tcp client.example.com.1112 -> server.example.com.23 (telnet)
marianne
aspirin?
ls -l
logout
-----------------
03/01/03 20:14:56 tcp client.example.com.1023 -> server.example.com.514 (rlogin)
[1022:tony]
rm junque
-----------------
03/01/03 20:16:33 tcp server.example.com.1225 -> client.example.com.6000 (x11)
MIT-MAGIC-COOKIE-1 c166a754fdf243c0f93e9fecb54abbd8
-----------------
03/01/03 20:08:20 udp client.example.com.688 -> server.example.com.777 (mountd)
/home [07 04 00 00 01 00 00 00 0c 00 00 00 02 00 00 00 3b 11 a1 36 00 00 00 00 00 00
00 00 00 00 00 00 ]
```

dsniff understands a wide range of protocols, and recognizes sensitive data that is transmitted without encryption. Our example shows passwords captured from FTP and Telnet sessions, with telnet commands and other input. (See why typing the root password over a Telnet connection is a very bad idea?) The rlogin session used no password, because the source host was trusted, but the command was captured. Finally, we see authorization information used by an X server, and filehandle information returned for an NFS mount operation.

dsniff uses libnids to reassemble TCP streams, because individual characters for interactively-typed passwords are often transmitted in separate packets. This reassembly relies on observation of the initial three-way handshake that starts all TCP sessions, so dsniff does not trace sessions already in progress when it was invoked.

The `dsniff -m` option enables automatic pattern-matching of protocols used on non-standard ports (e.g., HTTP on a port other than 80). Use the `-i` option to listen on a specific interface, if your system is connected to multiple networks. Append a filter-expression to restrict the network traffic that is monitored, using the same syntax as `tcpdump`. [9.16] `dsniff` uses `libpcap` to examine the first kilobyte of each packet: use the `-s` option to adjust the size of the snapshot if necessary.

`dsniff` can save the results in a database file specified by the `-w` option; the `-r` option reads and prints the results. If you use a database, be sure to protect this sensitive data from unwanted viewers. Unfortunately, `dsniff` cannot read or write `libpcap`-format network trace files—it performs live network-monitoring only.

A variety of more specialized sniffing tools are also provided with `dsniff`. The `mailsnarf` command captures mail messages from SMTP or POP traffic, and writes them in the standard mailbox format:

```
# mailsnarf
mailsnarf: listening on eth0
From engh@example.com Sat Mar  1 21:00:02 2003
Received: (from engh@example.com)
        by mail.example.com (8.11.6/8.11.6) id h1DJAPe10352
        for liberace@example.com; Sat, 1 Mar 2003 21:00:02 -0500
Date: Sat, 1 Mar 2003 21:00:02 -0500
From: Engelbert Humperdinck <engh@example.com>
Message-Id: <200303020200.AED1D74A1@example.com>
To: liberace@example.com
Subject: Elvis lives!

I ran into Elvis on the subway yesterday.
He said he was on his way to Graceland.
```

Suppose you want to encourage users who are sending email as clear text to encrypt their messages with GnuPG (see Chapter 8). You could theoretically inspect every email message, but of course this would be a gross violation of their privacy. You just want to detect whether encryption was used in each message, and to identify the correspondents if it was not. One approach is:

```
# mailsnarf -v "-----BEGIN PGP MESSAGE-----" | \
  perl -ne 'print if /^From / .. /^$/;' | \
  tee insecure-mail-headers
```

Our regular expression identifies encrypted messages, and the `mailsnarf -v` option (similar to `grep -v`) captures only those messages that were *not* encrypted. A short Perl script then discards the message bodies and records only the mail headers. The `tee` command prints the headers to the standard output so we can watch, and also writes them to a file, which can be used later to send mass mailings to the offenders. This strategy never saves your users' sensitive email data in a file.

dsniff comes with similar programs for other protocols, but they are useful mostly as convincing demonstrations of the importance of secure protocols. We hope you are already convinced by now!

The filesnarf command captures files from NFS traffic, and saves them in the current directory:

```
# filesnarf
filesnarf: listening on eth0
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: known_hosts (1303@0)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: love-letter.doc (8192@0)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: love-letter.doc (4096@8192)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: .Xauthority (204@0)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: myprog (8192@0)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: myprog (8192@8192)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: myprog (8192@16384)
filesnarf: 10.220.80.1.2049 > 10.220.80.4.800: myprog (8192@40960)
```

The last values on each line are the number of bytes transferred, and the file offsets. Of course, you can capture only those parts of the file transmitted on the network, so the saved files can have "holes" (which read as null bytes) where the missing data would be. No directory information is recorded. You can select specific filenames using a regular expression (and optionally with the -v option, to invert the sense of the match, as for mailsnarf or grep).

The urlsnarf command captures URLs from HTTP traffic, and records them in the Common Log Format (CLF). This format is used by most web servers, such as Apache, and is parsed by many web log analysis programs.

```
# urlsnarf
urlsnarf: listening on eth1 [tcp port 80 or port 8080 or port 3128]
client.example.com - - [ 1/Mar/2003:21:06:36 -0500] "GET http://naughty.monkey.org/
cgi-bin/counter?ft=0|dd=E|trgb=ffffff|df=dugsong-dsniff.dat HTTP/1.1" - - "http://
naughty.monkey.org/~dugsong/dsniff/" "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:0.9.
9) Gecko/20020513"
client.example.com - - [ 1/Mar/2003:21:06:46 -0500] "GET http://naughty.monkey.org/
~dugsong/dsniff/faq.html HTTP/1.1" - - "http://naughty.monkey.org/~dugsong/dsniff/"
"Mozilla/5.0 (X11; U; Linux i686; en-US; rv:0.9.9) Gecko/20020513"
```

By default, urlsnarf watches three ports that commonly carry HTTP traffic: 80, 3128, and 8080. To monitor a different port, use a capture filter expression:

```
# urlsnarf tcp port 8888
urlsnarf: listening on eth1 [tcp port 8888]
...
```

To monitor all TCP ports, use a more general expression:

```
# urlsnarf -i eth1 tcp
urlsnarf: listening on eth1 [tcp]
...
```

A regular expression can be supplied to select URLs of interest, optionally with -v as for mailsnarf or filesnarf.

A few other programs are provided with `dsniff` as a proof of concept for attacks on switched networks, man-in-the-middle attacks, and slowing or killing TCP connections. Some of these programs can be quite disruptive, especially if used incorrectly, so we don't recommend trying them unless you have an experimental network to conduct penetration testing.

See Also

dsniff(8), mailsnarf(8), filesnarf(8), urlsnarf(8). The `dsniff` home page is *http://naughty.monkey.org/~dugsong/dsniff*.

9.20 Getting Started with Snort

Problem

You want to set up Snort, a network-intrusion detection system.

Solution

Snort is included with SuSE but not Red Hat. If you need it (or you want to upgrade), download the source distribution from *http://www.snort.org* and unpack it:

```
$ tar xvpzf snort-*.tar.gz
```

Then compile it:

```
$ cd `ls -d snort-* | head -1`
$ ./configure
$ make
```

and install the binary and manpage as root:

```
# make install
```

Next, create a logging directory. It should not be publicly readable, since it will contain potentially sensitive data:

```
# mkdir -p -m go-rwx /var/log/snort
```

Finally, install the configuration files and rules database:

```
# mkdir -p /usr/local/share/rules
# cp etc/* rules/*.rules  /usr/local/share/rules
```

Discussion

Snort is a *network intrusion detection system* (NIDS), sort of an early-warning radar system for break-ins. It sniffs packets from the network and analyzes them according to a collection of well-known signatures characteristic of suspicious or hostile activi-

ties. This may remind you of an anti-virus tool, which looks for patterns in files to identify viruses.

By examining the protocol information and payload of each packet (or a sequence of packets) and applying its pattern-matching rules, Snort can identify the telltale fingerprints of attempted buffer overflows, denial of service attacks, port scans, and many other kinds of probes. When Snort detects a disturbing event, it can log network trace information for further investigation, and issue alerts so you can respond rapidly.

See Also

snort(8). The Snort home page is *http://www.snort.org*.