Computer Security

(EDA263, DIT641)

OFFPRINTS

2017/2018

Contents:

Revision: 150125-1

Like other general-purpose operating systems, Linux's wide range of features presents a broad attack surface. Even so, by leveraging native Linux security controls, carefully configuring Linux applications, and deploying certain add-on security packages, you can create highly secure Linux systems.

23.1 INTRODUCTION

Since Linus Torvalds created Linux in 1991, more or less on a whim, Linux has evolved into one of the world's most popular and versatile operating systems. Linux is free, open-sourced, and available in a wide variety of "distributions" targeted at almost every usage scenario imaginable. These distributions range from conservative, commercially supported versions such as Red Hat Enterprise Linux; to cutting-edge, completely free versions such as Ubuntu; to stripped-down but hyperstable "embedded" versions (designed for use in appliances and consumer products) such as uClinux.

The study and practice of Linux security therefore has wide-ranging uses and ramifications. New exploits against popular Linux applications affect many thousands of users around the world. New Linux security tools and techniques have just as profound of an impact, albeit a much more constructive one.

In this chapter we'll examine the Discretionary Access Control–based security model and architecture common to all Linux distributions and to most other UNIX-derived and UNIX-like operating systems (and also, to a surprising degree, to Microsoft Windows). We'll discuss the strengths and weaknesses of this ubiquitous model; typical vulnerabilities and exploits in Linux; best practices for mitigating those threats; and improvements to the Linux security model that are only slowly gaining popularity but that hold the promise to correct decades-old shortcomings in this platform.

23.2 LINUX'S SECURITY MODEL

Linux's traditional security model can be summed up quite succinctly: people or processes with "root" privileges can do anything; other accounts can do much less.

From the attacker's perspective, the challenge in cracking a Linux system therefore boils down to gaining root privileges. Once that happens, attackers can erase or edit logs; hide their processes, files, and directories; and basically redefine the reality of the system as experienced by its administrators and users. Thus, as it's most commonly practiced, Linux security (and UNIX security in general) is a game of "root takes all."

How can such a powerful operating system get by with such a limited security model? In fairness, many Linux system administrators fail to take full advantage of the security features available to them (features we're about explore in depth). People can and do run robust, secure Linux systems by making careful use of native Linux security controls, plus selected add-on tools such as sudo or Tripwire. However, the crux of the problem of Linux security in general is that like the

UNIX operating systems on which it was based, Linux's security model relies on **Discretionary Access Controls** (DAC).

In the Linux DAC system, there are users, each of which belongs to one or more groups; and there are also **objects**: files and directories. Users read, write, and execute these objects, based on the objects' **permissions**, of which each object has three sets: one each defining the permissions for the object's user-owner, group-owner, and "other" (everyone else). These permissions are enforced by the Linux kernel, the "brain" of the operating system.

Because a process/program is actually just a file that gets copied into executable memory when run, permissions come into play twice with processes. Prior to being executed, a program's file permissions restrict who can execute, access, or change it. When running, a process normally "runs as" (with the identity of) the user and group of the person or process that executed it.

Because processes "act as" users, if a running process attempts to read, write, or execute some other object, the kernel will first evaluate that object's permissions against the process's user and group identity, just as though the process was an actual human user. This basic transaction, wherein a **subject** (user or process) attempts some **action** (read, write, execute) against some **object** (file, directory, special file), is illustrated in Figure 23.1.

Whoever owns an object can set or change its permissions. Herein lies the Linux DAC model's real weakness: The system **superuser** account, called "root," has the ability to both take ownership and change the permissions of all objects in the system. And as it happens, it's not uncommon for both processes and administrator-users to routinely run with root privileges, in ways that provide attackers with opportunities to hijack those privileges.

Those are the basic concepts behind the Linux DAC model. The same concepts in a different arrangement will come into play later when we examine Mandatory Access Controls such as SELinux. Now let's take a closer look at how the Linux DAC implementation actually works.

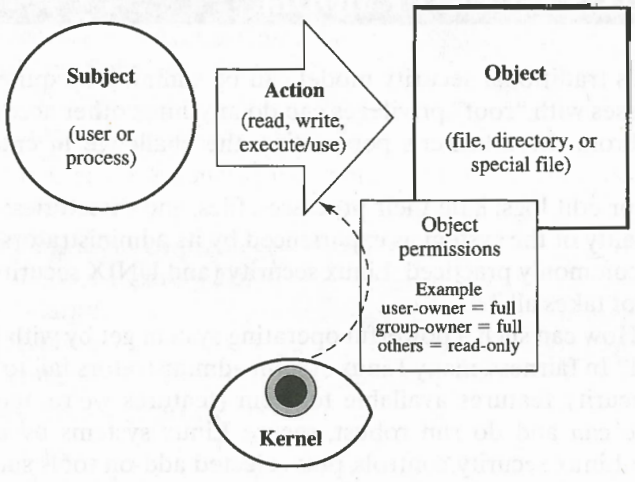


Figure 23.1 Linux Security Transactions

23.3 THE LINUX DAC IN DEPTH: FILE-SYSTEM SECURITY[1]

So far, we haven't said anything about memory, device drivers, named pipes, and other system resources. Isn't there more to system security than users, files, and directories? Yes and no: In a sense, Linux treats *everything* as a file.

Documents, pictures, and even executable programs are very easy to conceptualize as files on your hard disk. But although we *think* of a directory as a container of files, in UNIX a directory is actually itself a file containing a list of other files.

Similarly, the CD-ROM drive attached to your system seems tangible enough, but to the Linux kernel, it too is a file: the "special" device-file /dev/cdrom. To send data from or write data to the CD-ROM drive, the Linux kernel actually reads to and writes from this special file. (Actually, on most systems, "/dev/cdrom" is a symbolic link to /dev/hdb or some other special file. And a symbolic link is in turn nothing more than a file that contains a pointer to another file.)

Other special files, such as named pipes, act as input/output (I/O) "conduits," allowing one process or program to pass data to another. One common example of a named pipe on Linux systems is /dev/urandom: When a program reads this file, /dev/urandom returns random characters from the kernel's random number generator.

These examples illustrate how in Linux/UNIX, *nearly everything* is represented by a file. Once you understand this, it's much easier to understand why file-system security is such a big deal (and how it works).

Users, Groups, and Permissions

There are two things on a UNIX system that aren't represented by files: user accounts and group accounts, which for short we can call **users** and **groups**. (Various files contain information about a system's users and groups, but none of those files actually represents them.)

A user account represents someone or something capable of using files. As we saw in the previous section, a user account can be associated both with actual human beings and with processes. The standard Linux user account "lp," for example, is used by the Line Printer Daemon (lpd): The lpd program runs as the user lp.

A group account is simply a list of user accounts. Each user account is defined with a **main group** membership, but may in fact belong to as many groups as you want or need it to. For example, the user "maestro" may have a main group membership in "conductors" and also belong to the group "pianists."

A user's main group membership is specified in the user account's entry in /etc/password; you can add that user to additional groups by editing /etc/group and adding the username to the end of the entry for each group the user needs to belong to, or via the **usermod** command [see the usermod(8) manpage for more information].

---

[1]This section is adapted from [BAUE04], with permission of the *Linux Journal*.

Listing 23-1 shows "maestro"'s entry in the file /etc/password, and Listing 23-2 shows part of the corresponding /etc/group file:

```
maestro:x:200:100:Maestro Edward Hizzersands:/home/
maestro:/bin/bash
```

Listing 23-1: An /etc/password Entry for the User "maestro"

```
conductors:x:100:
pianists:x:102:maestro,volodya
```

Listing 23-2: Two /etc/group Entries

In Listing 23-1, we see that the first field contains the name of the user account, "maestro;" the second field ("x") is a placeholder for maestro's password (which is actually stored in /etc/shadow); the third field shows maestro's numeric userid (or "uid," in this case "200"); and the fourth field shows the numeric groupid (or "gid," in this case "100") of maestro's main group membership. The remaining fields specify a comment, maestro's home directory, and maestro's default login shell.

In Listing 23-2, from /etc/group, each line simply contains a groupname, a group-password (usually unused — "x" is a placeholder), numeric group-id (gid), and a comma-delimited list of users with "secondary" memberships in the group. Thus we see that the group "conductors" has a gid of "100", which corresponds to the gid specified as maestro's main group in Listing 23-1; and also that the group "pianists" includes the user "maestro" (plus another named "volodya") as a secondary member.

The simplest way to modify /etc/password and /etc/group in order to create, modify, and delete user accounts is via the commands **useradd**, **usermod**, and **userdel**, respectively. All three of these commands can be used to set and modify groupmemberships, and all three commands are well documented in their respective manpages. (To see a quick usage summary, you can also type the command followed by "—help," for example, "useradd —help".)

So we've got user accounts, which are associated with different group accounts. Just what is all this good for?

Simple File Permissions

Each file on a UNIX system (which, as we've seen, means "practically every single thing on a UNIX system") has two owners: a user and a group, each with its own set of permissions that specify what the user or group may do with the file (read it, write to it or delete it, and execute it). A third set of permissions pertains to **other**, that is, user accounts that don't own the file or belong to the group that owns it.

Listing 23-3 shows a "long file-listing" for the file/home/maestro/baton_ dealers.txt:

```
-rw-rw-r-- 1 maestro conductors 35414 Mar 25 01:38
baton_dealers.txt
```

Listing 23-3: File-Listing Showing Permissions

Permissions are listed in the order "user permissions, group permissions, other permissions." Thus we see that for the file shown in Listing 23-3, its user-owner ("maestro") may read and write/delete the file ("rw-"); its group-owner ("conductors") may also read and write/delete the file ("rw-"); but that other users (who are neither "maestro" nor members of "conductors") may only read the file.

There's a third permission besides "read" and "write": "execute," denoted by "x" (when set). If maestro writes a shell script named "punish_bassoonists.sh", and if he sets its permissions to "-rwxrw-r--", then maestro will be able to execute his script by entering the name of the script at the command line. If, however, he forgets to do so, he won't be able to run the script, even though he owns it. Permissions are usually set via the "chmod" command (short for "change mode").

Directory Permissions

Directory permissions work slightly differently from permissions on regular files. "Read" and "write" are similar; for directories these permissions translate to "list the directory's contents" and "create or delete files within the directory", respectively. "Execute" is less intuitive; for directories, "execute" translates to "use anything within or change working directory to this directory".

That is, if a user or group has execute permissions on a given directory, the user or group can list that directory's contents, read that directory's files (assuming those individual files' own permissions include this), and change its own working directory to that directory, as with the command "cd". If a user or group does not have execute permissions on a given directory, its will be unable to list or read anything in it, regardless of the permissions set on the things inside.

(Note that if you lack execute permissions on a directory but do have read permissions on an the directory, and you try to list its contents with ls, you will receive an error message that, in fact, lists the directory's contents. But this doesn't work if you have neither read nor execute permissions on the directory.)

Suppose our example system has a user named "biff" who belongs to the group "drummers". And suppose further that his home directory contains a directory called "extreme_casseroles" that he wishes to share with his fellow percussionists. Listing 23-4 shows how biff might set that directory's permissions:

```
bash-$ chmod g+rx extreme_casseroles
bash-$ ls -l extreme_casseroles
drwxr-x--- 8 biff drummers 288 Mar 25 01:38
extreme_casseroles
```

Listing 23-4: A Group-Readable Directory

Per Listing 23-4, only biff has the ability to create, change, or delete files inside extreme_casseroles. Other members of the group "drummers" may list its contents and cd to it. Everyone else on the system, however (except root, who is always all powerful), is blocked from listing, reading, cd-ing, or doing anything else with the directory.

The Sticky Bit

Suppose that our drummer friend Biff wants to allow his fellow drummers not only to read his recipes, but also to add their own. As we saw last time, all he needs to do is set the "group-write" bit for this directory, like this:

```
chmod g+w ./extreme_casseroles
```

There's only one problem with this: "write" permissions include both the ability to create new files in this directory, but also to delete them. What's to stop one of his drummer pals from deleting other people's recipes? The "sticky bit."

In older UNIX operating systems, the sticky bit was used to write a file (program) to memory so it would load more quickly when invoked. On Linux, however, it serves a different function: When you set the sticky bit on a directory, it limits users' ability to delete things in that directory. That is, to delete a given file in the directory you must either own that file or own the directory, even if you belong to the group that owns the directory and group-write permissions are set on it.

To set the sticky bit, issue the command

```
chmod +t directory_name
```

In our example, this would be "chmod +t extreme_casseroles". If we set the sticky bit on extreme_casseroles and then do a long listing of the directory itself, using "ls -ld extreme_casseroles", we'll see

```
drwxrwx--T 8 biff drummers 288 Mar 25 01:38
  extreme_casseroles
```

Note the "T" at the end of the permissions string. We'd normally expect to see either "x" or "-" there, depending on whether the directory is "other-writable". "T" denotes that the directory is not "other-executable" but has the sticky bit set. A lowercase "t" would denote that the directory is other-executable and has the sticky bit set.

To illustrate what effect this has, suppose a listing of the contents of extreme_casseroles/ looks like this (Listing 23-5):

```
drwxrwxr-T 3 biff drummers 192 2004-08-10 23:39.
drwxr-xr-x 3 biff drummers 4008 2004-08-10 23:39..
-rw-rw-r-- 1 biff drummers 18 2004-07-08 07:40
  chocolate_turkey_casserole.txt
-rw-rw-r-- 1 biff drummers 12 2004-08-08 15:10
  pineapple_mushroom_suprise.txt
drwxr-xr-x 2 biff drummers 80 2004-08-10 23:28 src
```

Listing 23-5: Contents of extreme_casseroles/

Suppose further that the user "crash" tries to delete the recipe file "pineapple_mushroom_surprise.txt", which crash finds offensive. crash expects this to

work, because he belongs to the group "drummers" and the group-write bit is set on this file.

However, remember, biff just set the parent directory's sticky bit. crash's attempted deletion will fail, as we see in Listing 23-6 (user input in boldface):

```
crash@localhost:/extreme_casseroles> rm pineapple_
  mushroom_suprise.txt
rm: cannot remove 'pineapple_mushroom_suprise.txt':
  Operation not permitted
```

Listing 23-6: Attempting Deletion with Sticky Bit Set

The sticky bit only applies to the directory's first level downward. In Listing 23-5 you may have noticed that besides the two nasty recipes, extreme_casseroles/ also contains another directory, "src". The contents of src will not be affected by extreme_casserole's sticky bit (though the directory src itself will be). If biff wants to protect src's contents from group deletion, he'll need to set src's own sticky bit.

Setuid and Setgid

Now we come to two of the most dangerous permissions bits in the UNIX world: setuid and segid. If set on an executable binary file, the setuid bit causes that program to "run as" its owner, no matter who executes it. Similarly, the setgid bit, when set on an executable, causes that program to run as a member of the group that owns it, again regardless of who executes it.

By *run as* we mean "to run with the same privileges as." For example, suppose biff writes and compiles a C program, "killpineapple", that behaves the same as the command "rm /extreme_casseroles/pineapple_mushroom_surprise.txt". Suppose further that biff sets the setuid bit on killpineapple, with the command "chmod +s ./killpineapple", and also makes it group executable. A long-listing of killpineapple might look like this:

```
-rwsr-xr-- 1 biff drummers 22 2004-08-11 23:01 killpineapple
```

If crash runs this program he will finally succeed in his quest to delete the Pineapple Mushroom Surprise recipe: killpineapple will run as though biff had executed it. When killpineapple attempts to delete pineapple_mushroom_suprise.txt, it will succeed because the file has user-write permissions and killpineapple is acting as its user-owner, biff.

Note that setuid and setgid are *very dangerous* if set on any file owned by root or any other privileged account or group. We illustrate setuid and setgid in this discussion so you understand what they do, not because you should actually *use* them for anything important. The command "sudo" is a much better tool for delegating root's authority.

If you want a program to run setuid, that program must be group executable or other executable, for obvious reasons. Note also that the Linux kernel ignores the setuid and setgid bits on shell scripts; these bits only work on binary (compiled) executables.

setgid works the same way, but with group permissions: If you set the setgid bit on an executable file via the command "chmod g+s filename", and if the file is also "other-executable" (-r-xr-sr-x), then when that program is executed it will run with the group-ID of the file rather than of the user who executed it.

In the preceding example, if we change killpineapple's "other" permissions to "r-x" (chmod o+x killpineapple) and make it setgid (chmod g+s killpineapple), then no matter who executes killpineapple, killpineapple will exercise the permissions of the "drummers" group, because drummers is the group-owner of killpineapple.

Setgid and Directories

Setuid has no effect on directories, but setgid does, and it's a little nonintuitive. Normally, when you create a file, it's automatically owned by your user ID and your (primary) group ID. For example, if biff creates a file, the file will have a user-owner of "biff" and a group-owner of "drummers" (assuming that "drummers" is biff's primary group, as listed in /etc/passwd).

Setting a directory's setgid bit, however, causes any file created in that directory to inherit the directory's group-owner. This is useful if users on your system tend to belong to secondary groups and routinely create files that need to be shared with other members of those groups.

For example, if the user "animal" is listed in /etc/group as being a secondary member of "drummers" but is listed in /etc/passwd has having a primary group of "muppets", then animal will have no trouble creating files in the extreme_casseroles/ directory, whose permissions are set to drwxrwx--T. However, by default animal's files will belong to the group muppets, not to drummers, so unless animal manually reassigns his files' group-ownership (chgrp drummers newfile) or resets their other-permissions (chmod o+rw newfile), then other members of drummers won't be able to read or write animal's recipes.

If, on the other hand, biff (or root) sets the setgid bit on extreme_casseroles/ (chmod g+s extreme_casseroles), then when animal creates a new file therein, the file will have a group-owner of "drummers", just like extreme_casseroles/ itself. Note that all other permissions still apply: If the directory in question isn't group-writable, then the setgid bit will have no effect (because group members won't be able to create files inside it).

Numeric Modes

So far we've been using mnemonics to represent permissions: "r" for read, "w" for write, and so on. But internally, Linux uses numbers to represent permissions; only user-space programs display permissions as letters. The chmod command recognizes both mnemonic permission modifiers ("u+rwx,go-w") and **numeric modes**.

A numeric mode consists of four digits: As you read left to right, these represent special permissions, user permissions, group permissions, and other permissions (where, you'll recall, "other" is short for "other users not covered by user permissions or group permissions"). For example, 0700 translates to "no special permissions set, all user permissions set, no group permissions set, no other permissions set."

Each permission has a numeric value, and the permissions in each digit-place are additive: The digit represents the sum of all permission-bits you wish to set. If,

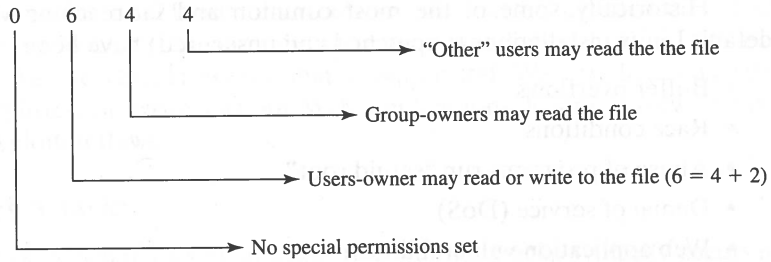


Figure 23.2 Permissions on mycoolfile

for example, user permissions are set to "7", this represents 4 (the value for "read") plus 2 (the value for "write") plus 1 (the value for "execute").

As just mentioned, the basic numeric values are 4 for read, 2 for write, and 1 for execute. (You can remember these by mentally repeating the phrase, "read-write-execute, 4-2-1.") Why no "3," you might wonder? Because (a) these values represent bits in a binary stream and are therefore all powers of 2; and (b) this way, no two combination of permissions have the same sum.

Special permissions are as follows: 4 stands for setuid, 2 stands for setgid, and 1 stands for sticky bit. For example, the numeric mode 3000 translates to "setgid set, stickybit set, no other permissions set" (which is, actually, a useless set of permissions).

Here's one more example of a numeric mode. If I issue the command "chmod 0644 mycoolfile," I'll be setting the permissions of "mycoolfile" as shown in Figure 23.2.

For a more complete discussion of numeric modes, see the Linux "info" page for "coreutils," node "Numeric Modes" (that is, enter the command "info coreutils numeric").

Kernel Space versus User Space

It is a simplification to say that users, groups, files, and directories are all that matter in the Linux DAC: Memory is important, too. Therefore, we should at least briefly discuss kernel space and user space.

Kernel space refers to memory used by the Linux kernel and its loadable modules (e.g., device drivers). **User space** refers to memory used by all other processes. Because the kernel enforces the Linux DAC and, in real terms, dictates system reality, it's extremely important to isolate kernel space from user space. For this reason, kernel space is never swapped to hard disk.

It's also the reason that only root may load and unload kernel modules. As we're about to see, one of the worst things that can happen on a compromised Linux system is for an attacker to gain the ability to load kernel modules.

23.4 LINUX VULNERABILITIES

In this section we'll discuss the most common weaknesses in Linux systems.

First, a bit of terminology. A **vulnerability** is a specific weakness or security-related bug in an application or operating system. A **threat** is the combination of a vulnerability, an attacker, and a means for the attacker to exploit the vulnerability (called an **attack vector**).

Historically, some of the most common and far-reaching vulnerabilities in default Linux installations (unpatched and unsecured) have been

- Buffer overflows
- Race conditions
- Abuse of programs run "setuid root"
- Denial of service (DoS)
- Web application vulnerabilities
- Rootkit attacks

While you've already had exposure to most of these concepts earlier in this book, let's take a closer look at how several of them apply to Linux.

Abuse of Programs Run "setuid root"

As we discussed in the previous section, any program whose "setuid" permission bit is set will run with the privileges of the user that owns it, rather than those of the process or user executing it. A **setuid root** program is a root-owned program with its setuid bit set; that is, a program that runs as root *no matter who executes it.*

If a setuid root program can be exploited or abused in some way (for example, via a buffer overflow vulnerability or race condition), then otherwise unprivileged users may be able to use that program to wield unauthorized root privileges, possibly including opening a **root shell** (a command-line session running with root privileges).

Running setuid root is necessary for programs that need to be run by unprivileged users yet must provide such users with access to privileged functions (for example, changing their password, which requires changes to protected system files). But such a program must be programmed very carefully, with impeccable user-input validation, strict memory management, and so on. That is, the program must be *designed* to be run setuid (or setgid) root. Even then, a root-owned program should only have its setuid bit set if absolutely necessary.

Due to a history of abuse against setuid root programs, major Linux distributions no longer ship with unnecessary setuid-root programs. But system attackers still scan for them.

Web Application Vulnerabilities

This is a very broad category of vulnerabilities, many of which also fall into other categories in this list. It warrants its own category because of the ubiquity of the World Wide Web: There are few attack surfaces as big and visible as an Internet-facing Web site.

While Web applications written in scripting languages such as PHP, Perl, and Java may not be as prone to classic buffer overflows (thanks to the additional layers of abstraction presented by those languages' interpreters), they're nonetheless prone to similar abuses of poor input handling, including cross-site scripting, SQL code injection, and a plethora of other vulnerabilities described in depth by the Open Web Application Security Project on the Project's Web site (http://www.owasp.org).

Nowadays, few Linux distributions ship with "enabled-by-default" Web applications (such as the default cgi scripts included with older versions of the Apache Web server). However, many users install Web applications with known vulnerabilities, or write custom Web applications having easily identified and easily exploited flaws.

Rootkit Attacks

This attack, which allows an attacker to cover her tracks, typically occurs *after* root compromise: If a successful attacker is able to install a rootkit before being detected, all is very nearly lost.

Rootkits began as collections of "hacked replacements" for common UNIX commands (ls, ps, etc.) that behaved like the legitimate commands they replaced, except for hiding an attacker's files, directories and processes. For example, if an attacker was able to replace a compromised Linux system's ls command with a rootkit version of ls, then anyone executing the ls command to view files and directories would see everything except the attacker's files and directories.

In the Linux world, since the advent of **loadable kernel modules** (LKMs), rootkits have more frequently taken the form of LKMs. This is particularly devious: An **LKM rootkit** does its business (covering the tracks of attackers) *in kernel space*, intercepting system calls pertaining to any user's attempts to view the intruder's resources.

In this way, files, directories, and processes owned by an attacker are hidden even to a compromised system's standard, un-tampered-with commands, including customized software. Besides operating at a lower, more global level, another advantage of the LKM rootkit over traditional rootkits is that system integrity-checking tools such as Tripwire won't generate alerts from system commands being replaced.

Luckily, even LKM rootkits do not always ensure complete invisibility for attackers. Many traditional and LKM rootkits can be detected with the script **chkrootkit**, available at **www. chkrootkit.org**. In general, however, if an attacker gets far enough to install an LKM rootkit, your system can be considered to be completely compromised; when and if you detect the breach (e.g., via a defaced Website, missing data, suspicious network traffic, etc.), the only way to restore your system with any confidence of completely shutting out the intruder will be to erase its hard disk (or replace it, if you have the means and inclination to analyze the old one), reinstall Linux, and apply all the latest software patches.

23.5 LINUX SYSTEM HARDENING

We've seen how Linux security is supposed to work, and how it most typically fails. The remainder of this chapter will focus on how to mitigate Linux security risks at the system and application levels. This section, obviously, deals with the first of these: OS-level security tools and techniques that protect the entire system. The final section in this chapter, on mandatory access controls, also describes system-level controls, but because this is both an advanced topic and an emerging technology (in the Linux world), we'll consider it separately from the more fundamental controls in this section.

provide. Many people manually create their own startup script for this purpose (an iptables "policy" is actually just a list of iptables commands), but a tool such as Shorewall or Firewall Builder may instead be used.

Antivirus Software

Historically, Linux hasn't been nearly so vulnerable to viruses as other operating systems (e.g., Windows). This may be due less to Linux's being inherently more secure than to its lesser popularity as a desktop platform: Virus writers wanting to maximize the return on their efforts prefer to target Windows because of its ubiquity.

To some extent, then, Linux users have tended not to worry about viruses. To the degree that they have, most Linux system administrators have tended to rely on keeping up to date with security patches for protection against malware, which is arguably a more proactive technique than relying on signature-based antivirus tools.

And indeed, prompt patching of security holes is an effective protection against worms, which have historically been a much bigger threat against Linux systems than viruses. A worm is simply an automated network attack that exploits one or more specific application vulnerabilities. If those vulnerabilities are patched, the worm won't infect the system.

Viruses, however, typically abuse the privileges of whatever user unwittingly executes them. Rather than actually exploiting a software vulnerability, the virus simply "runs as" the user. This may not have system-wide ramifications so long as that user isn't root, but even relatively unprivileged users can execute network client applications, create large files that could fill a disk volume, and perform any number of other problematic actions.

Unfortunately, there's no security patch to prevent users from double-clicking on e-mail attachments or loading hostile Web pages. Furthermore, as Linux's popularity continues to grow, especially as a general-purpose desktop platform (versus its currently-prevalent role as a back-end server platform), we can expect Linux viruses to become much more common. Sooner or later, therefore, antivirus software will become much more important on Linux systems than it is presently.

(Nowadays, it's far more common for antivirus software on Linux systems to be used to scan FTP archives, mail queues, etc., for viruses that target *other* systems than to be used to protect the system the antivirus software actually runs on.)

There are a variety of commercial and free antivirus software packages that run on (and protect) Linux, including products from McAfee, Symantec, and Sophos; and the free, open-source tool ClamAV.

User Management

As you'll recall from Sections 23.2 and 23.3, the guiding principles in Linux user account security are as follows:

- Be very careful when setting file and directory permissions;
- Use group memberships to differentiate between different roles on your system; and
- Be extremely careful in granting and using root privileges.

Let's discuss some of the nuts and bolts of user- and group-account management, and delegation of root privileges. First, some commands.

You'll recall that in Section 23.3 we used the **chmod** command to set and change permissions for objects belonging to existing users and groups. To create, modify, and delete user accounts, use the **useradd**, **usermod**, and **userdel** commands, respectively. To create, modify, and delete group accounts, use the **groupadd**, **groupmod**, and **groupdel** commands, respectively. Alternatively, you can simply edit the file **/etc/passwd** directly to create, modify, or delete users, or edit **/etc/group** to create, modify, or delete groups.

Note that initial (primary) group memberships are set in each user's entry in /etc/passwd; supplementary (secondary) group memberships are set in /etc/group. (You can use the usermod command to change either primary or supplementary group memberships for any user.) To change your password, use the **passwd** command. If you're logged on as root, you can also use this command to change other users' passwords.

Password Aging **Password aging** (that is, maximum and minimum lifetime for user passwords) is set globally in the files **/etc/login.defs** and **/etc/default/useradd**, but these settings are only applied when new user accounts are created. To modify the password lifetime for an existing account, use the **change** command.

As for the actual minimum and maximum password ages, passwords should have some minimum age to prevent users from rapidly "cycling through" password changes in attempts to reuse old passwords; seven days is a reasonable minimum password lifetime. Maximum lifetime is trickier: If this is too long, the odds of passwords being exposed before being changed will increase, but if it's too short, users frustrated with having to change their passwords frequently may feel justified in selecting easily guessed but also easily remembered passwords, writing passwords down, and otherwise mistreating their passwords in the name of convenience. Sixty days is a reasonable balance for many organizations.

In any event, it's much better to disable or delete defunct user accounts promptly and to educate users on protecting their passwords than it is to rely too much on password aging.

"Root Delegation:" su and sudo As we've seen, the fundamental problem with Linux and UNIX security is that far too often, permissions and authority on a given system boil down to "root can to anything, users can't do much of anything." Provided you know the root password, you can use the su command to promote yourself to root from whatever user you logged in as. Thus, the su command is as much a part of this problem as it is part of the solution.

Sadly, it's much easier to do a quick su to become root for a while than it is to create a granular system of group memberships and permissions that allows administrators and sub-administrators to have exactly the permissions they need. You can use the su command with the "-c" flag, which allows you to specify a single command to run as root rather than an entire shell session (for example, "su -c rm somefile.txt"), but because this requires you to enter the root password, everyone who needs to run a particular root command via this method will need to be given the root password. But it's never good for more than a small number of people to know root's password.

Another approach to solving the "root takes all" problem is to use SELinux's Role-Based Access Controls (RBAC) (see Section 23.7), which enforce access controls that reduce root's effective authority. However, this is much more complicated than setting up effective groups and group permissions. (However, adding that degree of complexity may be perfectly appropriate, depending on what's at stake.)

A reasonable middle ground is to use the **sudo** command, which is a standard package on most Linux distributions. "sudo" is short for "superuser do", and it allows users to execute specified commands as root without actually needing to know the root password (unlike su). sudo is configured via the file **/etc/sudoers**, but you shouldn't edit this file directly; rather, you should use the command **visudo**, which opens a special vi (text editor) session.

As handy as it is, sudo is a very powerful tool, so use it wisely: Root privileges are never to be trifled with. It really is better to use user- and group permissions judiciously than to hand out root privileges even via sudo, and it's better still to use an RBAC-based system like SELinux if feasible.

Logging

Logging isn't a proactive control; even if you use an automated "log watcher" to parse logs in real time for security events, logs can only tell you about bad things that have already happened. But effective logging helps ensure that in the event of a system breach or failure, system administrators can more quickly and accurately identify what happened and thus most effectively focus their remediation and recovery efforts.

On Linux systems, system logs are handled either by the ubiquitous **Berkeley Syslog daemon** (syslogd) in conjunction with the **kernel log daemon** (klogd), or by the much-more-feature-rich **Syslog-NG**. System log daemons receive log data from a variety of sources (the kernel via /proc/kmsg, named pipes such as /dev/log, or the network), sort data by **facility** (category) and **severity**, and then write the log messages to log files (or to named pipes, the network, etc.). Figure 23.3 lists the facilities and severities, both in their mnemonic and numeric forms, of Linux logging facilities, plus syslogd's actions (log targets).

Syslog-NG, the creation of Hungarian developer Balazs Scheidler, is preferable to syslogd for two reasons. First, it can use a much wider variety of log-data sources and destinations. Second, its "rules engine" (usually configured in /etc/syslog-ng/syslog-ng.conf) is much more flexible than syslogd's simple configuration file (/etc/syslogd.conf), allowing you to create a much more sophisticated set of rules for evaluating and processing log data.

Naturally, both syslogd and Syslog-NG install with default settings for what gets logged, and where. While these default settings are adequate in many cases, you should never take for granted that they are. At the very least, you should decide what combination of local and remote logging to perform. If logs remain local to the system that generates them, they may be tampered with by an attacker. If some or all log data are transmitted over the network to some central log server, audit trails can be more effectively preserved, but log data may also be exposed to network eavesdroppers.

(The risk of eavesdropping is still another reason to use Syslog-NG; whereas syslogd only supports remote logging via the connectionless UDP protocol, Syslog-NG also supports logging via TCP, which can be encrypted via a TLS "wrapper" such as Stunnel or Secure Shell.)

Facilities	Facility Codes†	Priorities (in increasing order)	Priority Codes†	Actions
auth	4	none	n/a	/some/file (log to specified file)
auth-priv	10	debug	7	-some/file (log to spec'd file)
cron	9	info	6	but don't sync afterwards)
daemon	3	notice	5	/some/pipe (log to specified
kern	0	warning	4	pipe)
lpr	6	err	3	dec/some/tty_or_console
mail	2	crit	2	(log to specified console)
mark	n/a	alert	1	@remote.hostname.or.IP
news	7	emerg	0	(log to specified remote host)
syslog	5	* {"any	n/a	username1, username2, etc
user	1	facility"}		(log to these users' screens)
uucp	8			* (log to all users' screens)
local {0-7}	16-23	Usage of ! and = as prefixes with priorities		
* {"any facility"}	n/a	*.notice (no prefix) = "any event with priority of notice or higher" *.!notice = "no event with priority of notice or higher" *.=notice = "only events with priority of notice" *.!=notice = "only events with priority of notice"		

†Numeric facility codes should not be used under Linux; they're here for reference only, as some other syslogd implementations (e.g., Cisco IOS) do use them.

Figure 23.3 Syslogd Reference

Local log files must be carefully managed. Logging messages from too many different log facilities to a single file may result in a logfile from which it is difficult to cull useful information; having too many different log files may make it difficult for administrators to remember where to look for a given audit trail. And in all cases, log files must not be allowed to fill disk volumes.

Most Linux distributions address this last problem via the **logrotate** command (typically run as a cron job), which decides how to rotate (archive or delete) system and application log files based both on global settings in the file /etc/logrotate.conf and on application-specific settings in the scripts contained in the directory /etc/logrotate.d/.

The Linux logging facility provides a local "system infrastructure" for both the kernel and applications, but it's usually also necessary to configure applications themselves to log appropriate levels of information. We revisit the subject of application-level logging in Section 23.6.

Other System Security Tools

Other tools worth mentioning that can greatly enhance Linux system security include the following:

- **Bastille:** A comprehensive system-hardening utility that educates as it secures
- **Tripwire:** A utility that maintains a database of characteristics of crucial system files and reports all changes made to them

- **Snort:** A powerful free Intrusion Detection System (IDS) that detects common network-based attacks
- **Nessus:** A modular security scanner that probes for common system and application vulnerabilities

23.6 APPLICATION SECURITY

Application security is a large topic; entire chapters in [BAUE05] are devoted to securing particular applications. However, many security features are implemented in similar ways across different applications. In this brief but important section, we'll examine some of these common features.

Running as an Unprivileged User/Group

Remember that in Linux and other UNIX-like operating systems, every process "runs as" some user. For network daemons in particular, it's extremely important that this user not be root; any process running as root is never more than a single buffer overflow or race condition away from being a means for attackers to achieve remote root compromise. Therefore, one of the most important security features a daemon can have is the ability to run as a nonprivileged user or group.

Running network processes as root isn't entirely avoidable; for example, only root can bind processes to "privileged ports" (TCP and UDP ports lower than 1024). However, it's still possible for a service's *parent* process to run as root in order to bind to a privileged port, but to then spawn a new child process that runs as an unprivileged user, each time an incoming connection is made.

Ideally, the unprivileged users and groups used by a given network daemon should be dedicated for that purpose, if for no other reason than for auditability (i.e., if entries start appearing in /var/log/messages indicating failed attempts by the user *ftpuser* to run the command /sbin/halt, it will be much easier to determine precisely what's going on if the *ftpuser* account isn't shared by five different network applications).

Running in a chroot Jail

If an FTP daemon serves files from a particular directory, say, /srv/ftp/public, there shouldn't be any reason for that daemon to have access to the rest of the file system. The **chroot** system call confines a process to some subset of /, that is, it maps a virtual "/" to some other directory (e.g., /srv/ftp/public). We call this directory to which we restrict the daemon a **chroot jail**. To the "chrooted" daemon, everything in the chroot jail appears to actually be in / (e.g., the "real" directory /srv/ftp/public/etc/myconfigfile appears as /etc/myconfigfile in the chroot jail). Things in directories outside the chroot jail (e.g., /srv/www or /etc.) aren't visible or reachable at all.

Chrooting therefore helps contain the effects of a given daemon's being compromised or hijacked. The main disadvantage of this method is added complexity: Certain files, directories, and special files typically must be copied into the chroot jail, and determining just what needs to go into the jail for the daemon to work properly can be tricky, though detailed procedures for chrooting many different Linux applications are easy to find on the World Wide Web.

Troubleshooting a chrooted application can also be difficult: Even if an application explicitly supports this feature, it may behave in unexpected ways when run chrooted. Note also that if the chrooted process runs as root, it can "break out" of the chroot jail with little difficulty. Still, the advantages usually far outweigh the disadvantages of chrooting network services.

Modularity

If an application runs in the form of a single, large, multipurpose process, it may be more difficult to run it as an unprivileged user; it may be harder to locate and fix security bugs in its source code (depending on how well documented and structured the code is); and it may be harder to disable unnecessary areas of functionality. In modern network service applications, therefore, **modularity** is a highly prized feature.

Postfix, for example, consists of a suite of daemons and commands, each dedicated to a different mail-transfer-related task. Only a couple of these processes ever run as root, and they practically never run all at the same time. Postfix therefore has a much smaller **attack surface** than the monolithic Sendmail. The popular Web server Apache used to be monolithic, but it now supports code modules that can be loaded at startup time as needed; this both reduces Apache's memory footprint and reduces the threat posed by vulnerabilities in unused functionality areas.

Encryption

Sending logon credentials or application data over networks in clear text (i.e., unencrypted) exposes them to network eavesdropping attacks. Most Linux network applications therefore support encryption nowadays, most commonly via the OpenSSL library. Using application-level encryption is, in fact, the most effective way to ensure end-to-end encryption of network transactions.

The SSL and TLS protocols provided by OpenSSL require the use of **X.509 digital certificates**. These can be generated and signed by the user space **openssl** command. For optimal security, either a local or commercial (third-party) **Certificate Authority** (CA) should be used to sign all server certificates, but **self-signed** (that is, nonverifiable) certificates may also be used. [BAUE05] provides detailed instructions on how to create and use your own Certificate Authority with OpenSSL.

Logging

Most applications can be configured to log to whatever level of detail you want, ranging from "debugging" (maximum detail) to "none." Some middle setting is usually the best choice, but you should not assume that the default setting is adequate.

In addition, many applications allow you to specify either a dedicated file to write application event data to, or a syslog **facility** to use when writing log data to **/dev/log** (see Section 23.5). If you wish to handle system logs in a consistent, centralized manner, it's usually preferable for applications to send their log data to /dev/log. Note, however, that logrotate (also discussed in Section 23.5) can be configured to rotate *any* logs on the system, whether written by syslogd, Syslog-NG, or individual applications.

Analyzing Computer Security:
A Threat / Vulnerability / Countermeasure Approach
Charles P. Pfleeger and Shari Lawrence Pfleeger
ISBN: 978-0-13-283940-2

RELATED ATTACK: COVERT CHANNEL

A man-in-the-middle attack involves two unsuspecting parties whose data exchange is intercepted by an outsider without knowledge of the parties or the system. A closely related, but different, problem is called a covert channel, in which malicious code allows data to be delivered to an authorized receiver without knowledge of the system. Both attacks can be difficult to detect.

A covert channel involves transmission of data across a channel where it is unnoticed. As a simple example, consider students in a classroom taking a multiple-choice exam, each question offering three choices. The students select one student, who we call the leader, to learn the material for the exam very well. On test day the leader agrees to communicate the answers to the exam to the other students in the following way.

- If the answer is A, the leader coughs.
- If the answer is B, the leader sneezes.
- If the answer is C, the leader sighs.

The leader performs this sequence for each question on the exam, and all the other students obtain the answers. The leader uses a channel, sounds in the exam room, to transmit data without the exam proctor's knowledge. (After a series of these noises, the proctor is likely to sense something, so the actual protocol used would have to be more subtle.)

Covert Channels in Computers

How does this behavior relate to computers and their security, you might well ask. Covert channels are a means of breaking through a security policy. Suppose a system administrator had established a policy that no data were to flow outside the local network because the users were working on a very sensitive project, perhaps developing a new product for a company or doing classified work for the government. A firewall at the network boundary strictly controlled outbound traffic, and access controls on each computer prevented programs from sending data out.

You are writing malicious code for that computer, and your task is to create a stealthy agent that can get data out, even at a rate of one bit per millisecond. Are there things you can do that could be visible outside to signal protected data? The U.S. Defense Department has people concerned about this very problem to protect Defense Department computers against leaking classified data (see [NCS93]). Just as important, individual companies' research laboratories, law offices with sensitive materials for clients, and even doctors handling records of well-known patients, all need to protect the unintended transfer of information outside the protected environment.

Such a transfer can occur across a **covert channel**, which is a flow of data across an unprotected, unintended means of communications. The concept of a covert channel comes from a paper by Butler Lampson [LAM73]; Jonathon Millen [MIL88] presents a good taxonomy of covert channels.

We begin by describing how a programmer can create covert channels. The attack is more complex than one by a lone programmer who accesses a data source directly. A programmer who has access to data can usually just read the data and write it to another file or print it or figure some other way to get it out of the environment stealthily. However, if the programmer is one step removed from the data—for example, outside the organization owning the data—the programmer must figure how to get at the data. The real threat for covert channels is not insiders such as programmers; instead, outsiders, who cannot get to data any other way, may find a way to introduce a malicious program that exfiltrates data.

The typical case for a covert channel is a spy who wants to get sensitive data from a computer system. The spy is not an employee or authorized user, so she has to figure out how to cause the computer to leak the data in a way she can recover. One way is to supply a bona fide program with a built-in Trojan horse; once the horse is enabled, it finds and transmits the data. However, it would be too bold to generate a report labeled "Send this report to Jane Smith in Camden, Maine"; the spy has to arrange to extract the data more surreptitiously. Covert channels are a means of using a Trojan horse to extract data without users or administrators noticing that data are leaving. Thus, the Trojan horse is in the middle between an unsuspecting user and the spy.

A covert channel is a communications channel that piggybacks on other valid communications. As in the example at the start of this section, noises in a classroom are normal; someone with a cold may sneeze frequently, and students are known to groan involuntarily as they read different test questions. Thus, the covert communication works within the signal band of these normal sneezes and groans and other noises.

Figure 12-12 shows a "service program" containing a Trojan horse that tries to copy information from a legitimate user (who is allowed access to the information) to a spy (who ought not be allowed to access the information). The user may not know that a Trojan horse is running and may not be in collusion to leak information to the spy.

As we noted, a program that produces a specific output report or displays a value may be too obvious. For example, in some installations, a printed report might occasionally be scanned by security staff before it is delivered to its intended recipient.

If printing the data values themselves is too obvious, the spy can encode the data values in another innocuous report by varying the format of the output, changing the lengths of lines, or printing or not printing certain values. For example, changing the word "TOTAL" to "TOTALS" in a heading will seldom be noticed, but this creates a 1-bit covert channel. The absence or presence of the S conveys one bit of information. Numeric values can be inserted in insignificant positions of output fields, and the number of lines per page can be changed.

Storage Channels

Some covert channels are called **storage channels** because they pass information by using the presence or absence of objects in storage.

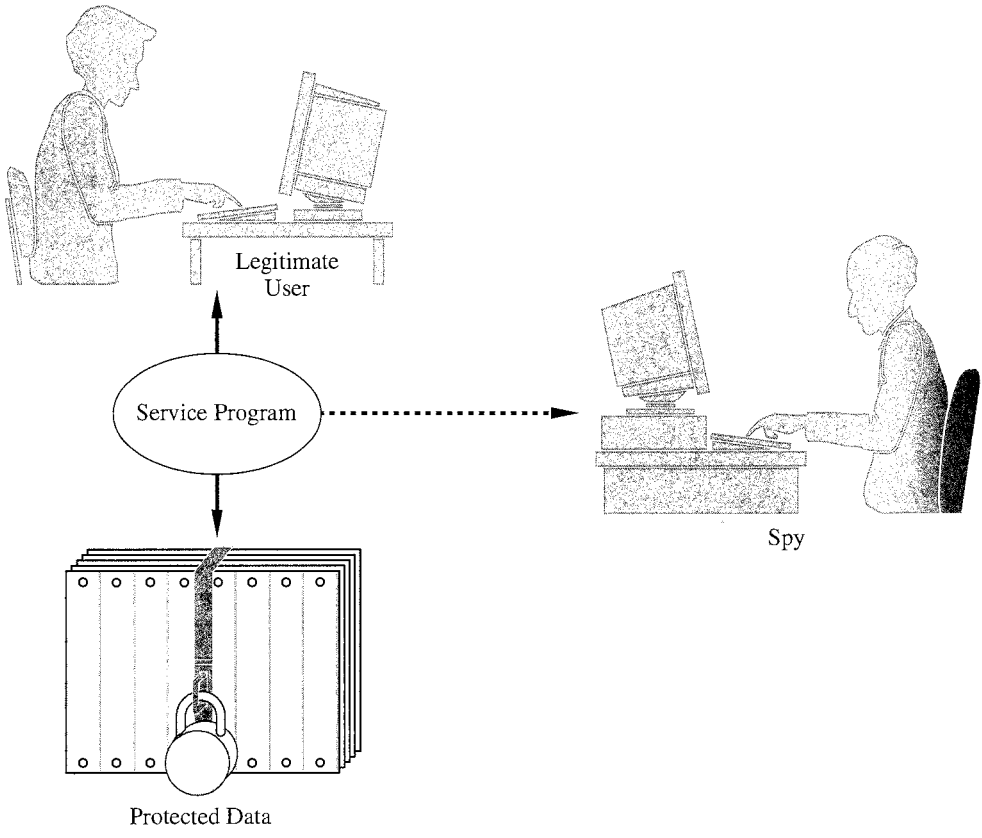


FIGURE 12-12 Covert Channel Service Program

A simple example of a covert channel is the **file lock channel**. In multiuser systems, files can be "locked" to prevent two people from writing to the same file at the same time (which could corrupt the file if one person writes over some of what the other wrote). The operating system or database management system allows only one program to write to a file at a time by blocking, delaying, or rejecting write requests from other programs. A covert channel can signal one bit of information by whether or not a file is locked.

Remember that the service program contains a Trojan horse written by the spy but run by the unsuspecting user. As shown in Figure 12-13, the service program reads confidential data (to which the spy should not have access) and signals the data one bit at a time by locking or not locking some file (any file, the contents of which are arbitrary and not even modified). The service program and the spy need a common timing source, broken into intervals. To signal a 1, the service program locks the file for the interval; for a 0, it does not lock. Later in the interval, the spy tries to lock the file itself. If the spy program cannot lock the file, it knows the service program must have locked the file, and thus the spy program concludes the service program is signaling a 1; if the spy program can lock the file, it knows the service program is signaling a 0.

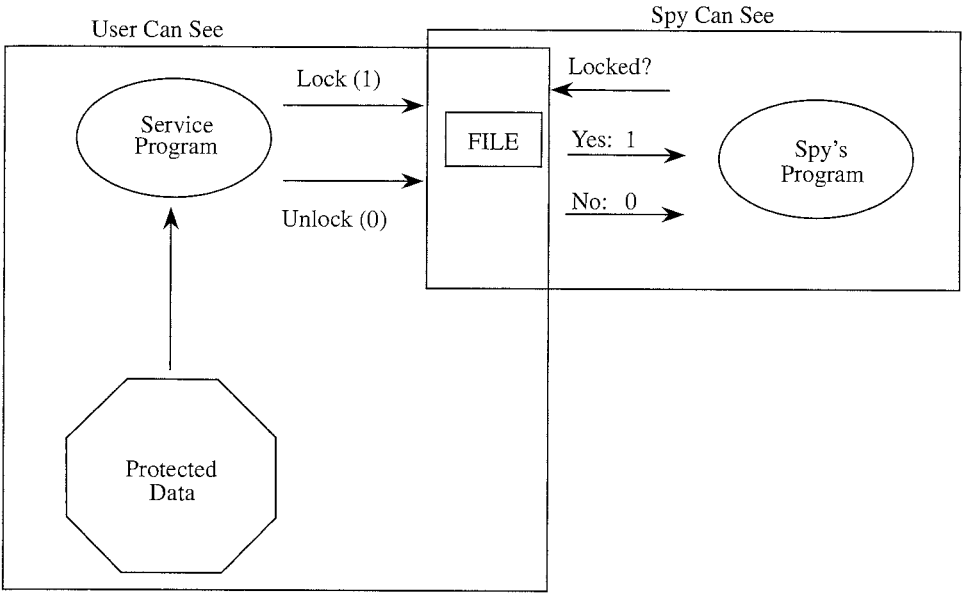


FIGURE 12-13 Service Program

This same approach can be used with disk storage quotas or other resources. With disk storage, the service program signals a 1 by creating an enormous file, so large that it consumes most of the available disk space. The spy program later tries to create a large file. If it succeeds, the spy program infers that the service program did not create a large file, and so the service program is signaling a 0; otherwise, the spy program infers a 1. Similarly the existence of a file or other resource of a particular name can be used to signal. Notice that the spy does not need access to a file itself; the mere existence of the file is adequate to signal. The spy can determine the existence of a file it cannot read by trying to create a file of the same name; if the request to create is rejected, the spy determines that the service program has such a file.

To signal more than one bit, the service program and the spy program signal one bit in each time interval. Figure 12-14 shows a service program signaling the string 100 by toggling the existence of a file.

In our final example, a storage channel uses a server of unique identifiers. Recall that some bakeries, banks, and other commercial establishments have a machine to distribute numbered tickets so that customers can be served in the order in which they arrived. Some computing systems provide a similar server of unique identifiers, usually numbers, used to name temporary files, to tag and track messages, or to record auditable events. Different processes can request the next unique identifier from the server. But two cooperating processes can use the server to send a signal: The spy process observes whether the numbers it receives are sequential or whether a number is missing. A missing number implies that the service program also requested a number, thereby signaling 1.

In all these examples, the service program and the spy need access to a shared resource (such as a file, or even knowledge of the existence of a file) and a shared

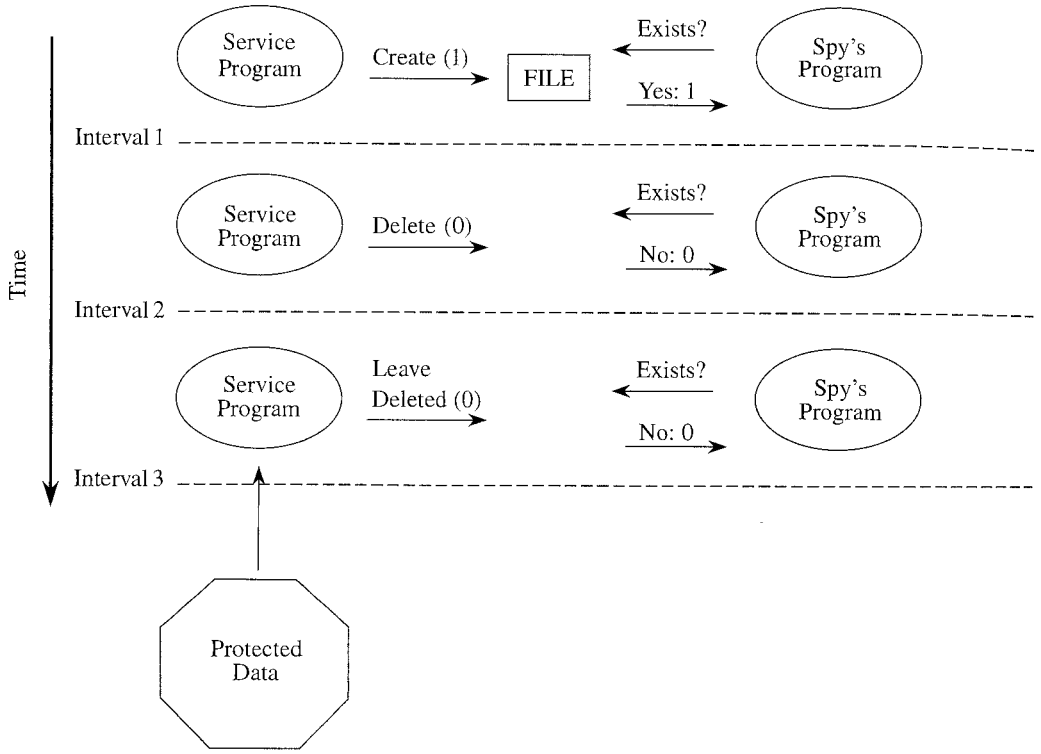


FIGURE 12-14 Signaling in a Covert Channel

sense of time. As shown, shared resources are common in multiuser environments, where the resource may be as seemingly innocuous as whether a file exists, a device is free, or space remains on disk. A source of shared time is also typically available since many programs need access to the current system time to set timers, to record the time at which events occur, or to synchronize activities. Karger and Wray [KAR91a] give a real-life example of a covert channel in the movement of a disk's arm and then describe ways to limit the potential information leakage from this channel.

Transferring data one bit at a time must seem awfully slow. But computers operate at such speeds that even the minuscule rate of 1 bit per millisecond (1/1000 second) would never be noticed but could easily be handled by two processes. At that rate of 1000 bits per second (which is unrealistically conservative), this entire book could be leaked in about two days. Increasing the rate by an order of magnitude or two, which is still quite conservative, reduces the transfer time to minutes.

Timing Channels

Other covert channels, called **timing channels**, pass information by using the speed at which things happen. Actually, timing channels are shared resource channels in which the shared resource is time.

A service program uses a timing channel to communicate by using or not using an assigned amount of computing time. In the simple case, a multiprogrammed system with two user processes divides time into blocks and allocates blocks of processing alternately to one process and the other. A process is offered processing time, but if the process is waiting for another event to occur and has no processing to do, it rejects the offer. The service process either uses its block (to signal a 1) or rejects its block (to signal a 0). Such a situation is shown in Figure 12-15, first with the service process and the spy's process alternating, and then with the service process communicating the string 101 to the spy's process. In the second part of the example, the service program wants to signal 0 in the third time block. It will do this by using just enough time to determine that it wants to send a 0 and then pause. The spy process then receives control for the remainder of the time block.

So far, all examples have involved just the service process and the spy's process. But in fact, multiuser computing systems typically have more than just two active processes. The only complications added by more processes are that the two cooperating processes must adjust their timings and deal with the possible interference from others. For example, with the unique identifier channel, other processes will also request identifiers. If on average, n other processes will request m identifiers each, then the service program will request more than $n*m$ identifiers for a 1 and no identifiers for a 0. The gap dominates the effect of all other processes. Also, the service process and the spy's process can use sophisticated coding techniques to compress their communication and detect and correct transmission errors caused by the effects of other unrelated processes.

Detecting Covert Channels

In this description of covert channels, ordinary things, such as the existence of a file or time used for a computation, have been the medium through which a covert channel communicates. Covert channels are not easy to find because these media are numerous

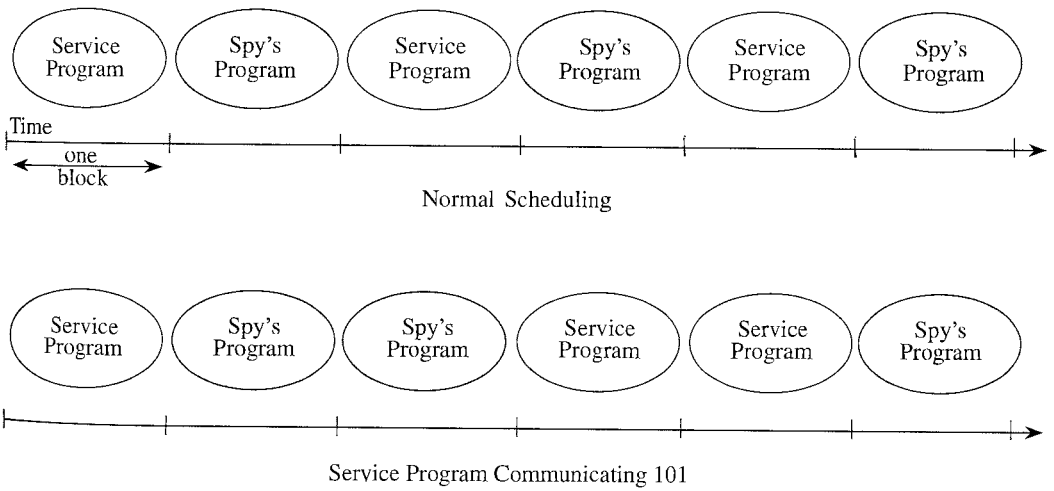


FIGURE 12-15 Signaling Using Time on a Covert Channel

and frequently used. Two relatively old techniques remain the standards for locating potential covert channels. One works by analyzing the resources of a system, and the other works at the source code level.

Shared Resource Matrix

Since the basis of a covert channel is a shared resource, the search for potential covert channels involves finding all shared resources and determining which processes can write to and read from the resources. The technique was introduced by Kemmerer [KEM83]. Although laborious, the technique can be automated.

To use this technique, you construct a matrix of resources (rows) and processes that can access them (columns). The matrix entries are R for "can read (or observe) the resource" and M for "can set (or modify, create, delete) the resource." For example, the file lock channel has the matrix shown in Table 12-1.

You then look for two columns and two rows having the following pattern:

	M		R	
	R			

This pattern identifies two resources and two processes such that the second process is not allowed to read from the second resource. However, the first process can pass the information to the second by reading from the second resource and signaling the data through the first resource. Thus, this pattern implies the potential information flow as shown here.

	M		R	
	R		R	

Next, you complete the shared resource matrix by adding these implied information flows, and analyzing the matrix for undesirable flows. Thus, you can tell that the spy's

TABLE 12-1 Shared Resource Matrix

	Service Process	Spy's Process
Locked	R, M	R, M
Confidential data	R	

TABLE 12-2 Complete Information Flow Matrix

	Service Process	Spy's Process
Locked	R, M	R, M
Confidential data	R	R

process can read the confidential data by using a covert channel through the file lock, as shown in Table 12-2.

Information Flow Method

Dorothy Denning [DEN76] derived a technique for flow analysis from a program's syntax. Conveniently, this analysis can be automated within a compiler so that information flow potentials can be detected while a program is under development.

Using this method, we can recognize nonobvious flows of information between statements in a program. For example, we know that the statement B := A, which assigns the value of A to the variable B, obviously supports an information flow from A to B. This type of flow is called an "explicit flow." Similarly, the pair of statements B := A; C := B indicates an information flow from A to C (by way of B). The conditional statement IF D=1 THEN B := A has two flows: from A to B because of the assignment, but also from D to B because the value of B can change if and only if the value of D is 1. This second flow is called an "implicit flow."

The statement B := *fcn(args)* supports an information flow from the function *fcn* to B. At a superficial level, we can say that there is a potential flow from the arguments *args* to B. However, we could more closely analyze the function to determine whether the function's value depended on all its arguments and whether any global values, not part of the argument list, affected the function's value. These information flows can be traced from the bottom up: At the bottom there must be functions that call no other functions, and we can analyze them and then use those results to analyze the functions that call them. By looking at the elementary functions first, we could say definitively whether there is a potential information flow from each argument to the function's result and whether there are any flows from global variables. Table 12-3 lists several examples of syntactic information flows.

Finally, we put all the pieces together to show which outputs are affected by which inputs. Although this analysis sounds frightfully complicated, it can be automated during the syntax analysis portion of compilation. This analysis can also be performed on the higher-level design specification.

Countermeasures against Covert Channels

As we just described, covert channels can exist only because of a shared resource, a physical object, that can be seen both inside and outside the protected environment. Control of access to these shared devices is one way to prevent covert channels. However, if it were possible to control access to all these objects, covert channels would not

TABLE 12-3 Syntactic Information Flows

Statement	Flow
B := A	From A to B
If C=1 THEN B := A	From A to B, from C to B
FOR K := 1 to N DO *stmts* END	From K to *stmts*
WHILE K > 0 DO *stmts* END	From K to *stmts*
CASE (*expr*) *val1*: *stmts*	From *expr* to *stmts*
B := *fcn*(*args*)	From *fcn* to B
OPEN FILE *f*	None
READ (*f*, X)	From file *f* to X
WRITE (*f*, X)	From X to file *f*

be an interest area. The fact that these channels use objects that are outside the normal control of an operating system is what makes them possible. The speed or slowness with which a particular result is obtained is evident to outsiders and uncontrollable inside.

The other countermeasure is recognizing shared resources and blocking sensitive processes from accessing them. Unfortunately, these resources are common, so preventing their access is not feasible.

The best control over covert channels is to slow down the channel to a rate at which leaking information would be unproductively slow. Reducing a modern computer to a slow access rate is similarly counterproductive.

Covert channels represent a real threat to secrecy in information systems. A covert channel attack is fairly sophisticated, but the basic concept is not beyond the capabilities of even an average programmer. Since the subverted program can be practically any user service, such as a printer utility, planting the compromise can be as easy as planting a virus or any other kind of Trojan horse. And recent experience has shown how readily viruses can be planted.

Capacity and speed are not problems; our estimate of 1000 bits per second is unrealistically low, but even at that rate much information leaks swiftly. With modern hardware architectures, certain covert channels inherent in the hardware design have capacities of millions of bits per second. And the attack does not require significant finance. Thus, the attack could be very effective in certain situations involving highly sensitive data.

For these reasons, security researchers have worked diligently to develop techniques for closing covert channels. The closure results have been bothersome; in ordinarily open environments, there is essentially no control over the subversion of a service program, nor is there an effective way of screening such programs for covert channels. And other than in a few very high security systems, operating systems cannot control

the flow of information from a covert channel. The hardware-based channels cannot be closed, given the underlying hardware architecture.

Although covert channel demonstrations are highly speculative—reports of actual covert channel attacks just do not exist—the analysis is sound. The mere possibility of their existence calls for more rigorous attention to other aspects of security, such as program development analysis, system architecture analysis, and review of output.

RELATED ATTACK: STEGANOGRAPHY

We conclude this chapter with a different kind of in-the-middle attack. This attack is related to covert channels because it is a means to pass information surreptitiously. It is also a form of in-the-middle attack because it can be used by a third party to piggyback on a communication between an unsuspecting sender and receiver.

We are about to describe **steganography**, the science of hidden writing. Cryptography is different, because the goal of encryption is to conceal meaning, but the output remains in plain sight. The goal of steganography is to conceal that anything is written, which naturally also conceals the content. Steganography applies not just to writing using letters and words, but also to communication using any recorded data; music, graphics, and video files are especially good for steganographic embedding.

Information Hiding

Steganography is part of the larger topic called **information hiding**, an activity that has been practiced for centuries. Fabien Petitcolas and colleagues at Cambridge [PET99] carefully define the various aspects of information hiding. Gus Simmons [SIM84] framed information hiding in the context of prisoners wanting to communicate in a way their guards would not notice. If you were a spy, being captured with a sheet of encrypted text would be incriminating, even if your captors could not decrypt the content. However, a handwritten diary might well pass as innocuous. Your diary contains your own thoughts, and you can write using incomplete sentences, misspellings, misused words, and incoherently meandering topics. You can also implement a nonobvious pattern to represent a sensitive thought you want to write so everything can reflect elusive topics. Did you notice that the first letters of the last phrase of the previous sentence spell the word s-e-c-r-e-t? That is an example of information hiding.

Spies are not the only information hiders. Steganography can be used to mark objects to show derivation or ownership. If you copy prose, anyone can compare two samples and deduce plagiarism. It is much more difficult to demonstrate digitally that a music file is a recording of the Berlin Philharmonic. If the Berlin Philharmonic embeds a recognizable string, sometimes called a **digital watermark**, in its recording, the string can help them identify and prove that a music file is actually theirs. This topic has received attention recently because of **digital rights management**, the goal of the owner of a copyright to a work of art (painting, film, sound recording) to control use and copying of the work.

We do not intend to cover the entire topic of information hiding, which is beyond the scope of this book. However, we furnish one technical example to give you a sense of the approach.

Technical Example

Consider a typical computer image file, such as a bitmap or photo file. These file formats are sets of bits representing the colors of individual pixels in an image. Colors can be represented in 8-bit or 24-bit encodings; 8-bit encodings can represent a 256-color palette, and 24 bits yield 16,777,216 colors. Both encodings go from white to black, passing through pink, gold, indigo, and green along the way. A 24-bit encoding of brown, for example, is 0xc08000, meaning 192 (0xc0) parts red, 128 (0x80) parts green, and 0 parts blue. (Each of these three values ranges from 0 to 255 or 0x00 to 0xff.)

Composed of filtered light, brown is all red and green, no blue, although browns come in many shades, such as tree bark, bear's fur, soil, coffee, caramel, or sand. The human eye can hardly detect minor changes of a single bit: (using decimal notation) 191, 128, 0 looks brown, as does 193, 128, 0, or 192, 127, 0, or 192, 128, 1. Suppose you have a picture of the bark of a tree, all brown, but with variations in shading and color showing light and shadow, depth, and varied materials. Several adjacent pixels will have the same color pattern, and then there is a shift as the feature becomes darker. In the first column of Table 12-4 we show such a transition. We represent the colors in hexadecimal notation, separated to show the red–green–blue components.

With steganography we can embed a message by co-opting the low order bit of each pixel. Suppose we want to embed the binary string 000 101 011 000 101 111 011 001 110 001 000. As shown in the last column for Table 12-4, we arbitrarily change the least significant bit of each color of each pixel to the bit of the message we want to pass, effectively ORing the strings. Notice that sometimes the original rightmost bits do not change, but this does not interfere with the communication.

TABLE 12-4 Embedding a Binary String through Steganography

Original Color	Message to Embed	Result
c0 80 00	0 0 0	c0 80 00
c0 80 00	1 0 1	c1 80 01
c0 80 00	0 1 1	c0 81 01
c0 80 00	0 0 0	c0 80 00
c0 80 00	1 0 1	c1 80 01
c0 80 00	1 1 1	c1 81 01
c0 80 01	0 1 1	c0 81 01
c0 80 02	0 0 1	c0 80 03
c0 80 04	1 1 0	c1 81 04
c0 80 03	0 0 1	c0 80 03
c0 80 07	0 0 0	c0 80 07

We chose this embedding scheme for easy explanation. As you can see, the changes between adjacent pixels are slightly abrupt. The changes would still be invisible to the human eye, but the frequency and discontinuity of the changes might be detected by a program that measures distortion to search specifically for unusual variation. Other embedding techniques are more subtle.

This method is also a good case for a form of in-the-middle attack. How many photos of famous landmarks are there on the Internet (and how many will be added each year)? An attacker can easily collect a virtual album of photos, doctor them steganographically, and repost them in a different album. Most people finding the album would see only travel photos; colleagues of the attacker, knowing the album and hiding algorithm, can download the photos and extract the information.

Charles Kurak and John McHugh [KUR92] present an interesting analysis of covert signaling through graphic images. In their work they show two different images combined by some rather simple arithmetic on the bit patterns of digitized pictures. The second image in a printed copy is undetectable to the human eye, but it can easily be separated and reconstructed by the spy receiving the digital version of the image. Simon Byers [BYE04] explores the topic in the context of data hidden within complex Word documents.

CONCLUSION

This chapter has given a name to our attacker behind the curtain, Mr. Man in the Middle. (We apologize that the terminology makes it seem as if only men perpetrate these attacks.) We have introduced him in this chapter to describe some complex attacks that involve both interception and fabrication. You will probably notice that our attacks have gotten more complex and sophisticated, and the interception step leads to deducing information, which in turn leads to being able to create new information.

As our examples have shown, man-in-the-middle attacks can occur in many different contexts, from computer-to-computer communications such as routing data, to person-to-computer or computer-to-person interchanges, such as the Syrian air defense example. Of the vulnerabilities we discussed, trust, identification and authentication, and access control have come up several times already, which just underscores how serious they are.

In this chapter we introduced one important new countermeasure: use of public key cryptography for key exchange. We have alluded to the technique in earlier chapters, but the full solution had to wait until this chapter to explore how to block attacks from the man in the middle. Here are the key points we presented in this chapter:

- In-the-middle attacks occur in many contexts: protocols, routing, addressing, web browsing, and applications.
- There is no single vulnerability that permits in-the-middle attacks; they seem to come from design and implementation flaws, poor or incomplete identification and authentication, and misplaced trust.
- Cryptography, applied in the right places, can counter such attacks, because it prevents the intermediary from seeing or modifying critical data. But cryptography

TABLE 12-5 Threat–Vulnerability–Countermeasure Chart for Man-in-the-Middle Attacks

Threat	Consequence	Severity	Ease of Exploitation
Man-in-the-middle	Modification	High	Fairly easy

Vulnerability	Exploitability	Prevalence
Identification and authentication	High	High
Unauthorized access	High	High
Program flaw	High	High

Countermeasure	Issues Addressed	Mitigation Type	Mitigation Effect	Effort
Protocols	Authentication	Prevent, detect	High	Difficult
Access control	Unauthorized access	Prevent	High	Moderate
Cryptography	Authentication	Prevent	High	High

has to be applied at the right time; otherwise, it only seals the attack that has already occurred.

- Covert channels are often a type of man-in-the-middle attack. A covert channel is a method by which an inside malicious process can signal sensitive data to an outside receiver using an existing baseline communication band.
- Steganography is another form of concealed communication. Instead of trying to hide the communication, steganography presents it in clear sight, but in a form that is not likely to be noticed.

In the next chapter we consider the related topic of forgeries: how to determine that something, particularly computer code or data, is real. Two issues involved are trust, how to determine whether the source of the object is authentic, and delivery, how to determine that no malicious man in the middle has modified the object during transmission.

Oh Look: The Easter Bunny! **Sidebar 3-2**

Microsoft's Excel spreadsheet program, in an old version, Excel 97, had the following feature.

- Open a new worksheet
- Press F5
- Type X97:L97 and press Enter
- Press Tab
- Hold <Ctrl-Shift> and click the Chart Wizard

A user who did that suddenly found that the spreadsheet disappeared and the screen filled with the image of an airplane cockpit! Using the arrow keys, the user could fly a simulated plane through space. With a few more keystrokes the user's screen seemed to follow down a corridor with panels on the sides, and on the panels were inscribed the names of the developers of that version of Excel.

Such a piece of code is called an **Easter egg**, for chocolate candy eggs filled with toys for children. This is not the only product with an Easter egg. An old version of Internet Explorer had something similar, and other examples can be found with an Internet search. Although most Easter eggs do not appear to be harmful, they raise a serious question: If such complex functionality can be embedded in commercial software products without being stopped by a company's quality control group, are there other holes, potentially with security vulnerabilities?

VULNERABILITY: UNDOCUMENTED ACCESS POINT

For our final vulnerability we describe a common programming situation. During program development and testing, the programmer needs a way to access the internals of a module. Perhaps a result is not being computed correctly so the programmer wants a way to interrogate data values during execution. Maybe flow of control is not proceeding as it should and the programmer needs to feed test values into a routine. It could be that the programmer wants a special debug mode to test conditions. For whatever reason the programmer creates an undocumented entry point or execution mode. Such an access point is called a **backdoor** or **trapdoor**.

These situations are understandable during program development. Sometimes, however, the programmer forgets to remove these entry points when the program moves from development to product. Or the programmer decides to leave them in to facilitate program maintenance later; the programmer may believe that nobody will find the special entry. Programmers can be naïve, because if there is a hole, someone is likely to find it. See Sidebar 3-2 for a description of an especially intricate backdoor.

The vulnerabilities we have presented here—incomplete mediation, race conditions, time-of-check to time-of-use, and undocumented access points—are flaws that can be exploited to cause a failure of security. Throughout this book we describe other sources of failures, because programmers have many process points to exploit and opportunities to create program flaws. Most of these flaws may have been created because the programmer failed to think clearly and carefully: simple human errors. Occasionally, however, the programmer maliciously planted an intentional flaw.

Next we consider how to prevent, or at least find and fix, program flaws.

Salami

With an interesting and apt name, a salami attack involves a different kind of substitution. In a financial institution, a criminal can perform what is known as a **salami attack**: The crook shaves a little from many accounts and puts these shavings together to form a valuable result, like the meat scraps joined in a salami. Suppose an account generates $10.32 in interest; would the account-holder notice if there were only $10.31? Or $10.21? Or $9.31? Highly unlikely for a difference of $0.01, and not very likely for $1.01. If the thief accumulates all the shaved-off interest "scraps" into a single account (of an accomplice), the total interest amount is right, and the attack is unlikely to cause any alarm.

TABLE 11-3 Contrast of Law vs. Ethics.

Law	Ethics
Described by formal, written documents	Described by unwritten principles
Interpreted by courts	Interpreted by each individual
Established by legislatures representing all people	Presented by philosophers, religions, professional groups
Applicable to everyone	Personal choice
Priority determined by courts if two laws conflict	Priority determined by an individual if two principles conflict
Court is final arbiter of "right"	No external arbiter
Enforceable by police and courts	Limited enforcement

we explore some of these problems and then consider how understanding ethics can help in dealing with issues of computer security.

Ethics and Religion

Ethics is a set of principles or norms for justifying what is right or wrong in a given situation. To understand what ethics *is* we may start by trying to understand what it *is not.* Ethical principles are different from religious beliefs. Religion is based on personal notions about the creation of the world and the existence of controlling forces or beings. Many moral principles are embodied in the major religions, and the basis of a personal morality is a matter of belief and conviction, much the same as for religions. However, two people with different religious backgrounds may develop the same ethical philosophy, while two exponents of the same religion might reach opposite ethical conclusions in a particular situation. Finally, we can analyze a situation from an ethical perspective and reach ethical conclusions without appealing to any particular religion or religious framework. Thus, it is important to distinguish ethics from religion.

Ethical Principles Are Not Universal

Ethical values vary by society, and from person to person within a society. For example, the concept of privacy is important in Western cultures. But in Eastern cultures, privacy is not desirable because people associate privacy with having something to hide. Not only is a Westerner's desire for privacy not understood but in fact it has a negative connotation. Therefore, the attitudes of people may be affected by culture or background.

Also, an individual's standards of behavior may be influenced by past events in life. A person who grew up in a large family may place greater emphasis on personal control and ownership of possessions than would an only child who seldom had to share. Major events or close contact with others can also shape one's ethical position. Despite

these differences, the underlying principles of how to make moral judgment are the same.

Although these aspects of ethics are quite reasonable and understandable, they lead people to distrust ethics because it is not founded on basic principles all can accept. Also, people from a scientific or technical background expect precision and universality.

Ethics Does Not Provide Answers

Ethical pluralism is recognizing or admitting that more than one position may be ethically justifiable—even equally so—in a given situation. Pluralism is another way of noting that two people may legitimately disagree on issues of ethics. We expect and accept disagreement in such areas as politics and religion.

However, in the scientific and technical fields, people expect to find unique, unambiguous, and unequivocal answers. In science, one answer must be correct or demonstrable in some sense. Science has provided life with fundamental explanations. Ethics is rejected or misunderstood by some scientists because it is "soft," meaning that it has no underlying framework or it does not depend on fundamental truths.

One need only study the history of scientific discovery to see that science itself is founded largely on temporary truths. For many years the earth was believed to be the center of the solar system. Ptolemy developed a complicated framework of epicycles, orbits within orbits of the planets, to explain the inconsistency of observed periods of rotation. Eventually his theory was superseded by the Copernican model of planets that orbit the sun. Similarly, Einstein's relativity theory opposed the traditional quantum basis of physics. Science is littered with theories that have fallen from favor as we learned or observed more and as new explanations were proposed. As each new theory is proposed, some people readily accept the new proposal, while others cling to the old.

But the basis of science is presumed to be "truth." A statement is expected to be provably true, provably false, or unproven, but a statement can never be both true and false. Scientists are uncomfortable with ethics because ethics does not provide these clean distinctions.

Worse, there is no higher authority of ethical truth. Two people may disagree on their opinion of the ethics of a situation, but there is no one to whom to appeal for a final determination of who is "right." Conflicting answers do not deter one from considering ethical issues in computer security. Nor do they excuse us from making and defending ethical choices.

Ethical Reasoning

Most people make ethical judgments often, perhaps daily. (Is it better to buy from a hometown merchant or from a nationwide chain? Should I spend time with a volunteer organization or with my friends? Is it acceptable to release sensitive data to someone who might not have justification for access to that data?) Because we all engage in ethical choice, we should clarify how we do this so that we can learn to apply the principles of ethics in professional situations, as we do in private life.

Study of ethics can yield two positive results. First, in situations in which we already know what is right and what is wrong, ethics should help us justify our choice.

Second, if we do not know the ethical action to take in a situation, ethics can help us identify the issues involved so that we can make reasoned judgments.

Examining a Case for Ethical Issues

How, then, can issues of ethical choice in computer security be approached? Here are several steps to making and justifying an ethical choice.

1. *Understand the situation.* Learn the facts of the situation. Ask questions of interpretation or clarification. Attempt to find out whether any relevant forces have not been considered.
2. *Know several theories of ethical reasoning.* To make an ethical choice, you have to know how those choices can be justified.
3. *List the ethical principles involved.* What different philosophies could be applied in this case? Do any of these include others?
4. *Determine which principles outweigh others.* This is a subjective evaluation. It often involves extending a principle to a logical conclusion or determining cases in which one principle clearly supersedes another.

The most important steps are the first and third. Too often people judge a situation on incomplete information, a practice that leads to judgments based on prejudice, suspicion, or misinformation. Considering all the different ethical issues raised forms the basis for evaluating the competing interests of step four.

Examples of Ethical Principles

There are two different schools of ethical reasoning: one based on the good that results from actions and one based on certain prima facie duties of people.

Consequence-Based Principles

The **teleological** theory of ethics focuses on the consequences of an action. The action to be chosen is that which results in the greatest future good and the least harm. For example, if a fellow student asks you to write a program he was assigned for a class, you might consider the good (he will owe you a favor) against the bad (you might get caught, causing embarrassment and possible disciplinary action, plus your friend will not learn the techniques to be gained from writing the program, leaving him deficient). The negative consequences clearly outweigh the positive, so you would refuse. *Teleology* is the general name applied to many theories of behavior, all of which focus on the goal, outcome, or consequence of the action.

There are two important forms of teleology. **Egoism** is the form that says a moral judgment is based on the positive benefits to the person taking the action. An egoist weighs the outcomes of all possible acts and chooses the one that produces the most personal good for him or her with the least negative consequence. The effects on other people are not relevant. For example, an egoist trying to justify the ethics of writing shoddy computer code when pressed for time might argue as follows. "If I complete the project quickly, I will satisfy my manager, which will bring me a raise and other good things. The customer is unlikely to know enough about the program to complain,

so I am not likely to be blamed. My company's reputation may be tarnished, but that will not be tracked directly to me. Thus, I can justify writing shoddy code."

The principle of **utilitarianism** is also an assessment of good and bad results, but the reference group is the entire universe. The utilitarian chooses that action that will bring the greatest collective good for all people with the least possible negative for all. In this situation, the utilitarian would assess personal good and bad, good and bad for the company, good and bad for the customer, and, perhaps, good and bad for society at large. For example, a developer designing software to monitor smokestack emissions would need to assess its effects on everyone breathing. The utilitarian might perceive greater good to everyone by taking the time to write high-quality code, despite the negative personal consequence of displeasing management.

Rule-Based Principles

Another ethical theory is **deontology**, which is founded in a sense of duty. This ethical principle states that certain things are good in and of themselves. These things that are naturally good are good rules or acts, which require no higher justification. Something just *is* good; it does not have to be judged for its effect.

Examples (from Frankena [FRA73]) of intrinsically good things are

- truth, knowledge, and true opinion of various kinds; understanding, wisdom
- just distribution of good and evil; justice
- pleasure, satisfaction; happiness; life, consciousness
- peace, security, freedom
- good reputation, honor, esteem; mutual affection, love, friendship, cooperation; morally good dispositions or virtues
- beauty, aesthetic experience

Rule-deontology is the school of ethical reasoning that believes certain universal, self-evident, natural rules specify our proper conduct. Certain basic moral principles are adhered to because of our responsibilities to one another; these principles are often stated as rights: the right to know, the right to privacy, the right to fair compensation for work. Sir David Ross [ROS30] lists various duties incumbent on all human beings:

- *fidelity,* or truthfulness
- *reparation,* the duty to recompense for a previous wrongful act
- *gratitude,* thankfulness for previous services or kind acts
- *justice,* distribution of happiness in accordance with merit
- *beneficence,* the obligation to help other people or to make their lives better
- *nonmaleficence,* not harming others
- *self-improvement,* to become continually better, both in a mental sense and in a moral sense (for example, by not committing a wrong a second time)

Another school of reasoning is based on rules derived by each individual. Religion, teaching, experience, and reflection lead each person to a set of personal moral principles. The answer to an ethical question is found by weighing values in terms of what a person believes to be right behavior.

TABLE 11-4 Taxonomy of Ethical Theories.

	Consequence-based	Rule-based
Individual	Based on consequences to individual	Based on rules acquired by the individual—from religion, experience, analysis
Universal	Based on consequences to all of society	Based on universal rules, evident to everyone

Summary of Ethical Theories

We have seen two bases of ethical theories, each applied in two ways. Simply stated, the two bases are consequence based and rule based, and the applications are either individual or universal. These theories are depicted in Table 11-4.

In the next section, we apply these theories to analyze certain situations that arise in the ethics of computer security.

11.7 CASE STUDIES OF ETHICS

To understand how ethics affects professional actions, ethicists often study example situations. The remainder of this section consists of several representative examples. These cases are modeled after ones developed by Parker [PAR79] as part of the AFIPS/NSF study of ethics in computing and technology. Each case study is designed to bring out certain ethical points, some of which are listed following the case. You should reflect on each case, determining for yourself what the most influential points are. These cases are suitable for use in a class discussion, during which other values will certainly be mentioned. Finally, each case reaches no conclusion because each individual must assess the ethical situation alone. In a class discussion it may be appropriate to take a vote. Remember, however, that ethics are not determined by majority rule. Those siding with the majority are not "right," and the rest are not "wrong."

Case I: Use of Computer Services

This case concerns deciding what is appropriate use of computer time. Use of computer time is a question both of access by one person and of availability of quality of service to others. The person involved is permitted to access computing facilities for a certain purpose. Many companies rely on an unwritten standard of behavior that governs the actions of people who have legitimate access to a computing system. The ethical issues involved in this case can lead to an understanding of that unwritten standard.

The Case

Dave works as a programmer for a large software company. He writes and tests utility programs such as compilers. His company operates two computing shifts: During the day program development and online applications are run; at night batch production jobs are completed. Dave has access to workload data and learns that the evening batch runs are complementary to daytime programming tasks; that is, adding programming

work during the night shift would not adversely affect performance of the computer to other users.

Dave comes back after normal hours to develop a program to manage his own stock portfolio. His drain on the system is minimal, and he uses very few expendable supplies, such as printer paper. Is Dave's behavior ethical?

Values Issues

Some of the ethical principles involved in this case are listed below.

- *Ownership of resources.* The company owns the computing resources and provides them for its own computing needs.
- *Effect on others.* Although unlikely, a flaw in Dave's program could adversely affect other users, perhaps even denying them service because of a system failure.
- *Universalism principle.* If Dave's action is acceptable, it should also be acceptable for others to do the same. However, too many employees working in the evening could reduce system effectiveness.
- *Possibility of detection, punishment.* Dave does not know whether his action would be wrong or right if discovered by his company. If his company decided it was improper use, Dave could be punished.

What other issues are involved? Which principles are more important than others?

Analysis

The utilitarian would consider the total excess of good over bad for all people. Dave receives benefit from use of computer time, although for this application the amount of time is not large. Dave has a possibility of punishment, but he may rate that as unlikely. The company is neither harmed nor helped by this. Thus, the utilitarian could argue that Dave's use is justifiable.

The universalism principle seems as if it would cause a problem because clearly if everyone did this, quality of service would degrade. A utilitarian would say that each new user has to weigh good and bad separately. Dave's use might not burden the machine, and neither might Ann's; but when Bill wants to use the machine, it is heavily enough used that Bill's use *would* affect other people.

Alternative Situations

Would it affect the ethics of the situation if any of the following actions or characteristics were considered?

- Dave began a business managing stock portfolios for many people for profit.
- Dave's salary was below average for his background, implying that Dave was due the computer use as a fringe benefit.
- Dave's employer knew of other employees doing similar things and tacitly approved by not seeking to stop them.
- Dave worked for a government office instead of a private company and reasoned that the computer belonged "to the people."

Case IV: Ownership of Programs

In this case we consider who owns programs: the programmer, the employer, the manager, or all. From a legal standpoint, most rights belong to the employer, as presented earlier in this chapter. However, this case expands on that position by presenting several competing arguments that might be used to support positions in this case. As described in the previous section, legal controls for secrecy of programs can be complicated, time consuming, and expensive to apply. In this case we search for individual ethical controls that can prevent the need to appeal to the legal system.

The Case

Greg is a programmer working for a large aerospace firm, Star Computers, which works on many government contracts; Cathy is Greg's supervisor. Greg is assigned to program various kinds of simulations.

To improve his programming abilities, Greg writes some programming tools, such as a cross-reference facility and a program that automatically extracts documentation

from source code. These are not assigned tasks for Greg; he writes them independently and uses them at work, but he does not tell anyone about them. Greg has written them in the evenings, at home, on his personal computer.

Greg decides to market these programming aids by himself. When Star's management hears of this, Cathy is instructed to tell Greg that he has no right to market these products since, when he was employed, he signed a form stating that all inventions become the property of the company. Cathy does not agree with this position because she knows that Greg has done this work on his own. She reluctantly tells Greg that he cannot market these products. She also asks Greg for a copy of the products.

Cathy quits working for Star and takes a supervisory position with Purple Computers, a competitor of Star. She takes with her a copy of Greg's products and distributes it to the people who work with her. These products are so successful that they substantially improve the effectiveness of her employees, and Cathy is praised by her management and receives a healthy bonus. Greg hears of this, and contacts Cathy, who contends that because the product was determined to belong to Star and because Star worked largely on government funding, the products were really in the public domain and therefore they belonged to no one in particular.

Analysis

This case certainly has major legal implications. Probably everyone could sue everyone else and, depending on the amount they are willing to spend on legal expenses, they could keep the cases in the courts for several years. Probably no judgment would satisfy all.

Let us set aside the legal aspects and look at the ethical issues. We want to determine who might have done what, and what changes might have been possible to prevent a tangle for the courts to unscramble.

First, let us explore the principles involved.

- *Rights.* What are the respective rights of Greg, Cathy, Star, and Purple?
- *Basis.* What gives Greg, Cathy, Star, and Purple those rights? What principles of fair play, business, property rights, and so forth are involved in this case?
- *Priority.* Which of these principles are inferior to which others? Which ones take precedence? (Note that it may be impossible to compare two different rights, so the outcome of this analysis may yield some rights that are important but that cannot be ranked first, second, third.)
- *Additional information.* What additional facts do you need in order to analyze this case? What assumptions are you making in performing the analysis?

Next, we want to consider what events led to the situation described and what alternative actions could have prevented the negative outcomes.

- What could Greg have done differently before starting to develop his product? After developing the product? After Cathy explained that the product belonged to Star?
- What could Cathy have done differently when she was told to tell Greg that his products belonged to Star? What could Cathy have done differently to avert this

Case VIII: Ethics of Hacking or Cracking

What behavior is acceptable in cyberspace? Who owns or controls the Internet? Does malicious or nonmalicious intent matter? Legal issues are involved in the answers to these questions, but as we have pointed out previously, laws and the courts cannot protect everything, nor should we expect them to. Some people separate investigating computer security vulnerabilities from exploiting them, calling the former "white hat" hacking and the latter "black hat." It is futile to try to stop people from learning nor should we even try, for the sake of society, as Cross [CRO06] points out. There is reasonable debate over publication or dissemination of knowledge: Is the world safer if only a few are allowed to know how to build sophisticated weapons? Or how to break certain security systems? Is the public better served by open knowledge of system vulnerabilities? We recommend students, researchers, faculty, and technologists, and certainly users, join in thoughtful debate of this issue, one of the largest ethical matters in our field.

In this final case study we consider ethical behavior in a shared-use computing environment, such as the Internet. The questions are similar to "what behavior is acceptable in outer space?" or "who owns the oceans?"

Goli is a computer security consultant; she enjoys the challenge of finding and fixing security vulnerabilities. Independently wealthy, she does not need to work, so she has ample spare time in which to test the security of systems.

In her spare time, Goli does three things: First, she aggressively attacks commercial products for vulnerabilities. She is quite proud of the tools and approach she has developed, and she is quite successful at finding flaws. Second, she probes accessible systems on the Internet, and when she finds vulnerable sites, she contacts the owners to offer her services repairing the problems. Finally, she is a strong believer in high-quality pastry, and she plants small programs to slow performance in the web sites of pastry shops that do not use enough butter in their pastries. Let us examine these three actions in order.

Vulnerabilities in Commercial Products

We have already described a current debate regarding the vulnerability reporting process. Now let us explore the ethical issues involved in that debate.

Clearly from a rule-based ethical theory, attackers are wrong to perform malicious attacks. The appropriate theory seems to be one of consequence: who is helped or hurt by finding and publicizing flaws in products? Relevant parties are attackers, the vulnerability finder, the vendor, and the using public. Notoriety or credit for finding the flaw is a small interest. And the interests of the vendor (financial, public relations) are less important than the interests of users to have secure products. But how are the interests of users best served?

- *Full disclosure* helps users assess the seriousness of the vulnerability and apply appropriate protection. But it also gives attackers more information with which to formulate attacks. Early full disclosure—before the vendor has countermeasures ready—may actually harm users by leaving them vulnerable to a now widely known attack.
- *Partial disclosure*—the general nature of the vulnerability but not a detailed exploitation scenario—may forestall attackers. One can argue that the vulnerability details are there to be discovered; when a vendor announces a patch for an unspecified flaw in a product, the attackers will test that product aggressively and study the patch carefully to try to determine the vulnerability. Attackers will then spread a complete description of the vulnerability to other attackers through an underground network, and attacks will start against users who may not have applied the vendor's fix.
- *No disclosure*. Perhaps users are best served by a scheme in which every so often new code is released, sometimes fixing security vulnerabilities, sometimes fixing things that are not security related, and sometimes adding new features. But without a sense of significance or urgency, users may not install this new code.

Searching for Vulnerabilities and Customers

What are the ethical issues involved in searching for vulnerabilities? Again, the party of greatest interest is the user community and the good or harm that can come from the search.

On the positive side, searching may find vulnerabilities. Clearly, it would be wrong for Goli to report vulnerabilities that were not there simply to get work, and it would also be wrong to report some but not all vulnerabilities to be able to use the additional vulnerabilities as future leverage against the client.

But suppose Goli does a diligent search for vulnerabilities and reports them to the potential client. Is that not similar to a service station owner's advising you that a headlight is not operating when you take your car in for gasoline? Not quite, you might say. The headlight flaw can be seen without any possible harm to your car; probing for vulnerabilities might cause your system to fail.

The ethical question seems to be which is greater: the potential for good or the potential for harm? And if the potential for good is stronger, how much stronger does it need to be to override the risk of harm?

This case is also related to the common practice of ostensible nonmalicious probing for vulnerabilities: Hackers see if they can access your system without your permission, perhaps by guessing a password. Spafford [SPA98] points out that many crackers

Sidebar 11-4 Is Cracking a Benign Practice?

Many people argue that cracking is an acceptable practice because lack of protection means that the owners of systems or data do not really value them. Spafford [SPA98] questions this logic by using the analogy of entering a house.

Consider the argument that an intruder who does no harm and makes no changes is simply learning about how computer systems operate. "Most of these people would never think to walk down a street, trying every door to find one unlocked, then search through the drawers or the furniture inside. Yet, these same people seem to give no second thought to making repeated attempts at guessing passwords to accounts they do not own, and once onto a system, browsing through the files on disk." How would you feel if you knew your home had been invaded, even if no harm was done?

Spafford notes that breaking into a house or a computer system constitutes trespassing. To do so in an effort to make security vulnerabilities more visible is "presumptuous and reprehensible." To enter either a home or a computer system in an unauthorized way, even with benign intent, can lead to unintended consequences. "Many systems have been damaged accidentally by ignorant (or careless) intruders."

We do not accept the argument that hackers make good security experts. There are two components to being a good security professional: knowledge and credibility. Diligent explorers, who may experiment with computer breaking in a benign setting like a closed laboratory network, can learn just as much about finding and exploiting vulnerabilities as a hacker. The key differentiator is trust. If you hire a hacker you will always have a nagging fear that your expert is gathering data to attack you or someone else. Comparing two otherwise equal candidates for a position, you choose the one with the lesser risk. To us, the hacker-turned-consultant is seeking to capitalize on a history of unethical behavior. See [PFL06b] for a longer discussion.

simply want to look around, without damaging anything. As discussed in Sidebar 11-4, Spafford compares this seemingly innocent activity with entry into your house when the door is unlocked. Even when done without malicious intent, cracking can be a serious offense; at its worst, it has caused millions of dollars in damage. Although crackers are prosecuted severely with harsh penalties, cracking continues to be an appealing crime, especially to juveniles.

Politically Inspired Attacks

Finally, consider Goli's interfering with operation of web sites whose actions she opposes. We have purposely phrased the issue in a situation that arouses perhaps only a few gourmands and pâtissiers. We can dismiss the interest of the butter fans as an insignificant minority on an insignificant issue. But you can certainly think of many other issues that have brought on wars. (See Denning's excellent article on cybercriminals [DEN99a] for real examples of politically motivated computer activity.)

The ethical issues abound in this scenario. Some people will see the (butter) issue as one of inherent good, but is butter use one of the fundamental good principles, such as honesty or fairness or not doing harm to others? Is there universal agreement that butter use is good? Probably there will be a division of the world into the butter advocates (x%), the unrestricted pastry advocates (y%), and those who do not take a position (z%). By how much does x have to exceed y for Goli's actions to be acceptable? What if the value of z is large? Greatest good for the greatest number requires a balance among these three percentages and some measure of benefit or harm.

Is butter use so patently good that it justifies harm to those who disagree? Who is helped and who suffers? Is the world helped if only good, but more expensive, pastries are available, so poor people can no longer afford pastry? Suppose we could determine that 99.9 percent of people in the world agreed that butter use was a good thing. Would that preponderance justify overriding the interests of the other 0.1 percent?

Codes of Ethics

Because of ethical issues such as these, various computer groups have sought to develop codes of ethics for their members. Most computer organizations, such as the Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE), and the Data Processing Management Association (DPMA), are voluntary organizations. Being a member of one of these organizations does not certify a level of competence, responsibility, or experience in computing. For these reasons, codes of ethics in these organizations are primarily advisory. Nevertheless, these codes are fine starting points for analyzing ethical issues.

IEEE

The IEEE has produced a code of ethics for its members. The IEEE is an organization of engineers, not limited to computing. Thus, their code of ethics is a little broader than might be expected for computer security, but the basic principles are applicable in computing situations. The IEEE Code of Ethics is shown in Figure 11-1.

ACM

The ACM code of ethics recognizes three kinds of responsibilities of its members: general moral imperatives, professional responsibilities, and leadership responsibilities, both inside the association and in general. The code of ethics has three sections (plus a fourth commitment section), as shown in Figure 11-2.

Computer Ethics Institute

The Computer Ethics Institute is a nonprofit group that aims to encourage people to consider the ethical aspects of their computing activities. The organization has been in existence since the mid-1980s, founded as a joint activity of IBM, the Brookings Institution, and the Washington Theological Consortium. The group has published its ethical guidance as ten commandments of computer ethics, listed in Figure 11-3.

Many organizations take ethics seriously and produce a document guiding the behavior of its members or employees. Some corporations require new employees to read

We, the members of the IEEE, in recognition of the importance of our technologies in affecting the quality of life throughout the world, and in accepting a personal obligation to our profession, its members, and the communities we serve, do hereby commit ourselves to conduct of the highest ethical and professional manner and agree

1. to accept responsibility in making engineering decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;
2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;
3. to be honest and realistic in stating claims or estimates based on available data;
4. to reject bribery in all of its forms;
5. to improve understanding of technology, its appropriate application, and potential consequences;
6. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;
7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;
8. to treat fairly all persons regardless of such factors as race, religion, gender, disability, age, or national origin;
9. to avoid injuring others, their property, reputation, or employment by false or malicious action;
10. to assist colleagues and coworkers in their professional development and to support them in following this code of ethics.

FIGURE 11-1 IEEE Code of Ethics. (Reprinted courtesy of the Institute of Electrical and Electronics Engineers © 1996.)

its code of ethics and sign a promise to abide by it. Others, especially at universities and research centers, have special boards that must approve proposed research and ensure that projects and team members act ethically. As an individual professional, it may be useful for you to review these codes of ethics and compose a code of your own, reflecting your ideas about appropriate behavior in likely situations. A code of ethics can help you assess situations quickly and act in a consistent, comfortable, and ethical manner.

Conclusion of Computer Ethics

In this study of ethics, we have tried not to decide right and wrong, or even to brand certain acts as ethical or unethical. The purpose of this section is to stimulate thinking about ethical issues concerned with confidentiality, integrity, and availability of data and computations.

1. Thou shalt not use a computer to harm other people.
2. Thou shalt not interfere with other people's computer work.
3. Thou shalt not snoop around in other people's computer files.
4. Thou shalt not use a computer to steal.
5. Thou shalt not use a computer to bear false witness.
6. Thou shalt not copy or use proprietary software for which you have not paid.
7. Thou shalt not use other people's computer resources without authorization or proper compensation.
8. Thou shalt not appropriate other people's intellectual output.
9. Thou shalt think about the social consequences of the program you are writing or the system you are designing.
10. Thou shalt always use a computer in ways that insure consideration and respect for your fellow humans.

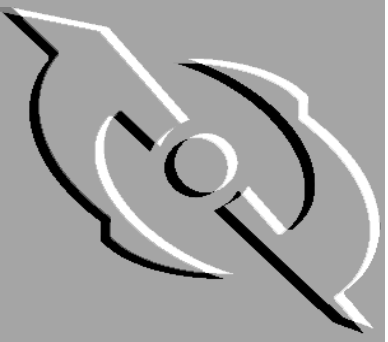
FIGURE 11-3 The Ten Commandments of Computer Ethics. (Reprinted with permission, Computer Ethics Institute, Washington, D.C.)

The cases presented show complex, conflicting ethical situations. The important first step in acting ethically in a situation is to obtain the facts, ask about any uncertainties, and acquire any additional information needed. In other words, first we must understand the situation.

The second step is to identify the ethical principles involved. Honesty, fair play, proper compensation, and respect for privacy are all ethical principles. Sometimes these conflict, and then we must determine which principles are more important than others. This analysis may not lead to one principle that obviously overshadows all others. Still, a ranking to identify the major principles involved is needed.

The third step is choosing an action that meets these ethical principles. Making a decision and taking action are difficult, especially if the action has evident negative consequences. However, taking action based on a *personal* ranking of principles is necessary. The fact that other equally sensible people may choose a different action does not excuse us from taking some action.

This section is not trying to force the development of rigid, inflexible principles. Decisions may vary, based on fine differences between two situations. Or a person's views can change over time in response to experience and changing context. Learning to reason about ethical situations is not quite the same as learning "right" from "wrong." Terms such as *right* and *wrong* or *good* and *bad* imply a universal set of values. Yet we know that even widely accepted principles are overridden by some people in some situations. For example, the principle of not killing people may be violated in the case of war or capital punishment. Few, if any, values are held by everyone or in all cases. Therefore, our purpose in introducing this material has been to stimulate you to recognize and think about ethical principles involved in cases related to computer security. Only by recognizing and analyzing principles can you act consistently, thoughtfully, and responsibly.



An Introduction to Cryptography

The Basics of Cryptography 1

When Julius Caesar sent messages to his generals, he didn't trust his messengers. So he replaced every A in his messages with a D, every B with an E, and so on through the alphabet. Only someone who knew the “shift by 3” rule could decipher his messages.

And so we begin.

Encryption and decryption

Data that can be read and understood without any special measures is called *plaintext* or *cleartext*. The method of disguising plaintext in such a way as to hide its substance is called *encryption*. Encrypting plaintext results in unreadable gibberish called *ciphertext*. You use encryption to ensure that information is hidden from anyone for whom it is not intended, even those who can see the encrypted data. The process of reverting ciphertext to its original plaintext is called *decryption*.

Figure 1-1 illustrates this process.

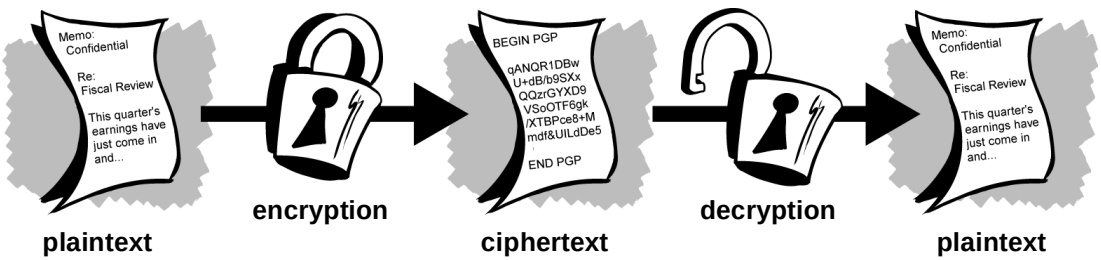


Figure 1-1. Encryption and decryption

What is cryptography?

Cryptography is the science of using mathematics to encrypt and decrypt data. Cryptography enables you to store sensitive information or transmit it across insecure networks (like the Internet) so that it cannot be read by anyone except the intended recipient.

While cryptography is the science of securing data, *cryptanalysis* is the science of analyzing and breaking secure communication. Classical cryptanalysis involves an interesting combination of analytical reasoning, application of mathematical tools, pattern finding, patience, determination, and luck. Cryptanalysts are also called *attackers*.

Cryptology embraces both cryptography and cryptanalysis.

Strong cryptography

"There are two kinds of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files. This book is about the latter."

--Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C.

PGP is also about the latter sort of cryptography.

Cryptography can be *strong* or *weak*, as explained above. Cryptographic strength is measured in the time and resources it would require to recover the plaintext. The result of *strong cryptography* is ciphertext that is very difficult to decipher without possession of the appropriate decoding tool. How difficult? Given all of today's computing power and available time—even a billion computers doing a billion checks a second—it is not possible to decipher the result of strong cryptography before the end of the universe.

One would think, then, that strong cryptography would hold up rather well against even an extremely determined cryptanalyst. Who's really to say? No one has proven that the strongest encryption obtainable today will hold up under tomorrow's computing power. However, the strong cryptography employed by PGP is the best available today. Vigilance and conservatism will protect you better, however, than claims of impenetrability.

How does cryptography work?

A *cryptographic algorithm*, or *cipher*, is a mathematical function used in the encryption and decryption process. A cryptographic algorithm works in combination with a *key*—a word, number, or phrase—to encrypt the plaintext. The same plaintext encrypts to different ciphertext with different keys. The security of encrypted data is entirely dependent on two things: the strength of the cryptographic algorithm and the secrecy of the key.

A cryptographic algorithm, plus all possible keys and all the protocols that make it work comprise a *cryptosystem*. PGP is a cryptosystem.

Conventional cryptography

In conventional cryptography, also called *secret-key* or *symmetric-key* encryption, one key is used both for encryption and decryption. The Data Encryption Standard (DES) is an example of a conventional cryptosystem that is widely employed by the Federal Government. *Figure 1-2* is an illustration of the conventional encryption process.

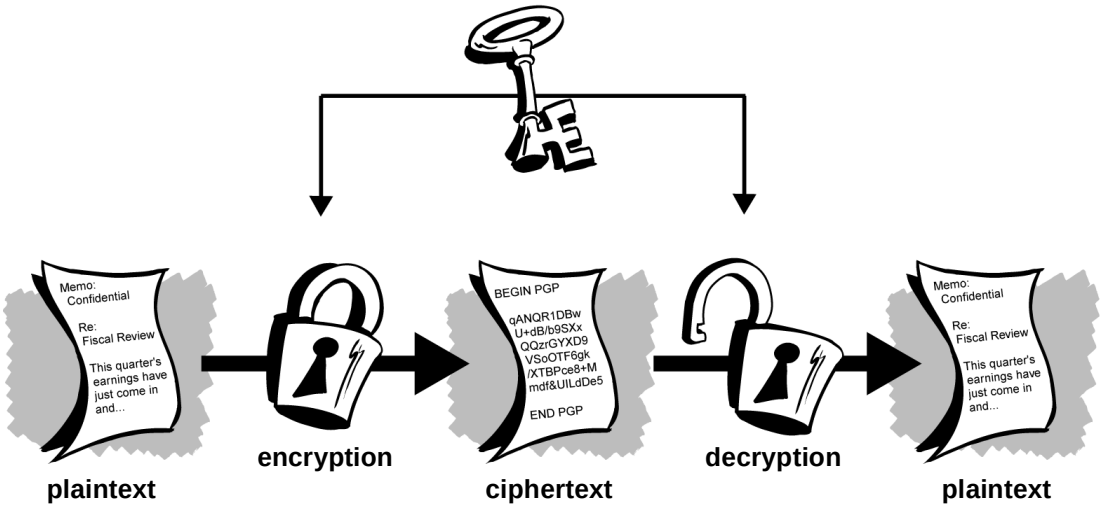


Figure 1-2. Conventional encryption

Caesar's Cipher

An extremely simple example of conventional cryptography is a substitution cipher. A substitution cipher substitutes one piece of information for another. This is most frequently done by offsetting letters of the alphabet. Two examples are Captain Midnight's Secret Decoder Ring, which you may have owned when you were a kid, and Julius Caesar's cipher. In both cases, the algorithm is to offset the alphabet and the key is the number of characters to offset it.

For example, if we encode the word "SECRET" using Caesar's key value of 3, we offset the alphabet so that the 3rd letter down (D) begins the alphabet.

So starting with

ABCDEFGHIJKLMNOPQRSTUVWXYZ

and sliding everything up by 3, you get

DEFGHIJKLMNOPQRSTUVWXYZABC

where D=A, E=B, F=C, and so on.

Using this scheme, the plaintext, "SECRET" encrypts as "VHFUHW." To allow someone else to read the ciphertext, you tell them that the key is 3.

Obviously, this is exceedingly weak cryptography by today's standards, but hey, it worked for Caesar, and it illustrates how conventional cryptography works.

Key management and conventional encryption

Conventional encryption has benefits. It is very fast. It is especially useful for encrypting data that is not *going* anywhere. However, conventional encryption alone as a means for transmitting secure data can be quite expensive simply due to the difficulty of secure key distribution.

Recall a character from your favorite spy movie: the person with a locked briefcase handcuffed to his or her wrist. What is in the briefcase, anyway? It's probably not the missile launch code / biotoxin formula / invasion plan itself. It's the *key* that will decrypt the secret data.

For a sender and recipient to communicate securely using conventional encryption, they must agree upon a key and keep it secret between themselves. If they are in different physical locations, they must trust a courier, the Bat Phone, or some other secure communication medium to prevent the disclosure of the secret key during transmission. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all information encrypted or authenticated with that key. From DES to Captain Midnight's Secret Decoder Ring, the persistent problem with conventional encryption is *key distribution*: how do you get the key to the recipient without someone intercepting it?

Public key cryptography

The problems of key distribution are solved by *public key cryptography*, the concept of which was introduced by Whitfield Diffie and Martin Hellman in 1975. (There is now evidence that the British Secret Service invented it a few years before Diffie and Hellman, but kept it a military secret—and did nothing with it.)[1]

Public key cryptography is an asymmetric scheme that uses a *pair* of keys for encryption: a *public key*, which encrypts data, and a corresponding *private*, or *secret key* for decryption. You publish your public key to the world while keeping your private key secret. Anyone with a copy of your public key can then encrypt information that only you can read. Even people you have never met.

---

1. J H Ellis, The Possibility of Secure Non-Secret Digital Encryption, CESG Report, January 1970. [CESG is the UK's National Authority for the official use of cryptography.]

It is computationally infeasible to deduce the private key from the public key. Anyone who has a public key can encrypt information but cannot decrypt it. Only the person who has the corresponding private key can decrypt the information.

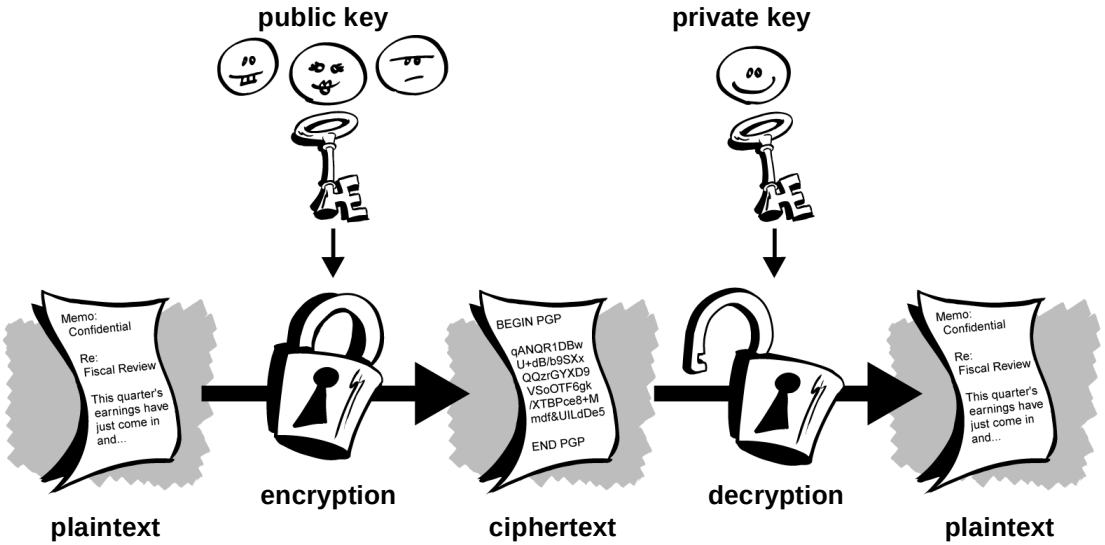


Figure 1-3. Public key encryption

The primary benefit of public key cryptography is that it allows people who have no preexisting security arrangement to exchange messages securely. The need for sender and receiver to share secret keys via some secure channel is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. Some examples of public-key cryptosystems are Elgamal (named for its inventor, Taher Elgamal), RSA (named for its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman), Diffie-Hellman (named, you guessed it, for its inventors), and DSA, the Digital Signature Algorithm (invented by David Kravitz).

Because conventional cryptography was once the only available means for relaying secret information, the expense of secure channels and key distribution relegated its use only to those who could afford it, such as governments and large banks (or small children with secret decoder rings). Public key encryption is the technological revolution that provides strong cryptography to the adult masses. Remember the courier with the locked briefcase handcuffed to his wrist? Public-key encryption puts him out of business (probably to his relief).

How PGP works

PGP combines some of the best features of both conventional and public key cryptography. PGP is a *hybrid cryptosystem*.

When a user encrypts plaintext with PGP, PGP first compresses the plaintext. Data compression saves modem transmission time and disk space and, more importantly, strengthens cryptographic security. Most cryptanalysis techniques exploit patterns found in the plaintext to crack the cipher. Compression reduces these patterns in the plaintext, thereby greatly enhancing resistance to cryptanalysis. (Files that are too short to compress or which don't compress well aren't compressed.)

PGP then creates a *session key*, which is a one-time-only secret key. This key is a random number generated from the random movements of your mouse and the keystrokes you type. This session key works with a very secure, fast conventional encryption algorithm to encrypt the plaintext; the result is ciphertext. Once the data is encrypted, the session key is then encrypted to the recipient's public key. This public key-encrypted session key is transmitted along with the ciphertext to the recipient.

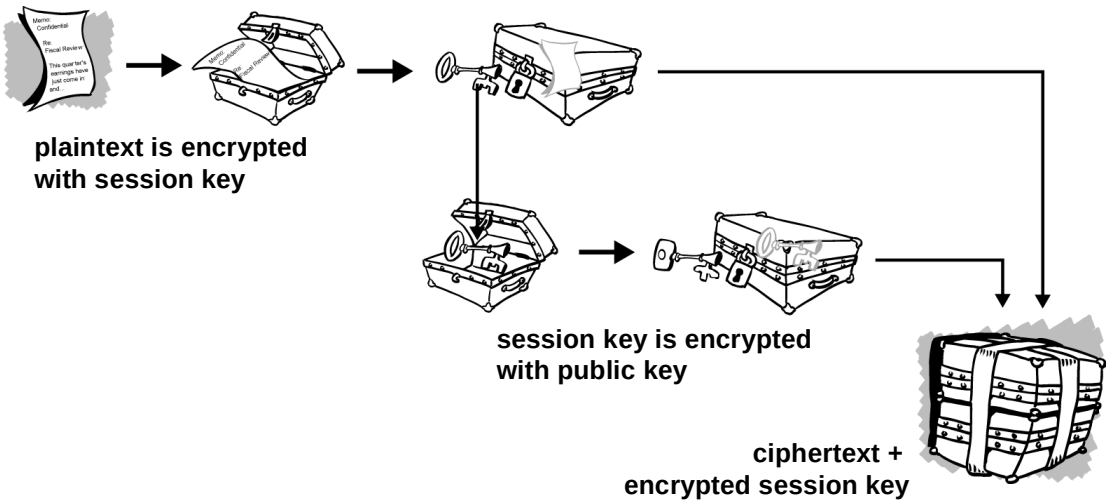


Figure 1-4. How PGP encryption works

Decryption works in the reverse. The recipient's copy of PGP uses his or her private key to recover the temporary session key, which PGP then uses to decrypt the conventionally-encrypted ciphertext.

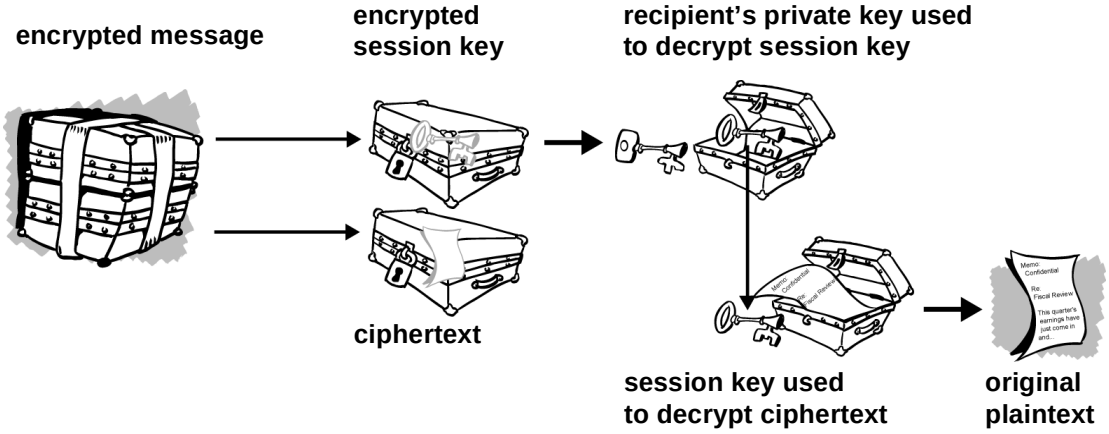


Figure 1-5. How PGP decryption works

The combination of the two encryption methods combines the convenience of public key encryption with the speed of conventional encryption. Conventional encryption is about 1,000 times faster than public key encryption. Public key encryption in turn provides a solution to key distribution and data transmission issues. Used together, performance and key distribution are improved without any sacrifice in security.

Keys

A key is a value that works with a cryptographic algorithm to produce a specific ciphertext. Keys are basically really, really, really big numbers. Key size is measured in bits; the number representing a 1024-bit key is darn huge. In public key cryptography, the bigger the key, the more secure the ciphertext.

However, public key size and conventional cryptography's secret key size are totally unrelated. A conventional 80-bit key has the equivalent strength of a 1024-bit public key. A conventional 128-bit key is equivalent to a 3000-bit public key. Again, the bigger the key, the more secure, but the algorithms used for each type of cryptography are very different and thus comparison is like that of apples to oranges.

While the public and private keys are mathematically related, it's very difficult to derive the private key given only the public key; however, deriving the private key is always possible given enough time and computing power. This makes it very important to pick keys of the right size; large enough to be secure, but small enough to be applied fairly quickly. Additionally, you need to consider who might be trying to read your files, how determined they are, how much time they have, and what their resources might be.

Larger keys will be cryptographically secure for a longer period of time. If what you want to encrypt needs to be hidden for many years, you might want to use a very large key. Of course, who knows how long it will take to determine your key using tomorrow's faster, more efficient computers? There was a time when a 56-bit symmetric key was considered extremely safe.

Keys are stored in encrypted form. PGP stores the keys in two files on your hard disk; one for public keys and one for private keys. These files are called *keyrings.* As you use PGP, you will typically add the public keys of your recipients to your public keyring. Your private keys are stored on your private keyring. If you lose your private keyring, you will be unable to decrypt any information encrypted to keys on that ring.

Digital signatures

A major benefit of public key cryptography is that it provides a method for employing *digital signatures*. Digital signatures enable the recipient of information to verify the authenticity of the information's origin, and also verify that the information is intact. Thus, public key digital signatures provide *authentication* and data *integrity*. A digital signature also provides *non-repudiation*, which means that it prevents the sender from claiming that he or she did not actually send the information. These features are every bit as fundamental to cryptography as privacy, if not more.

A digital signature serves the same purpose as a handwritten signature. However, a handwritten signature is easy to counterfeit. A digital signature is superior to a handwritten signature in that it is nearly impossible to counterfeit, plus it attests to the contents of the information as well as to the identity of the signer.

Some people tend to use signatures more than they use encryption. For example, you may not care if anyone knows that you just deposited $1000 in your account, but you do want to be darn sure it was the bank teller you were dealing with.

The basic manner in which digital signatures are created is illustrated in *Figure 1-6*. Instead of encrypting information using someone else's public key, you encrypt it with your private key. If the information can be decrypted with your public key, then it must have originated with you.

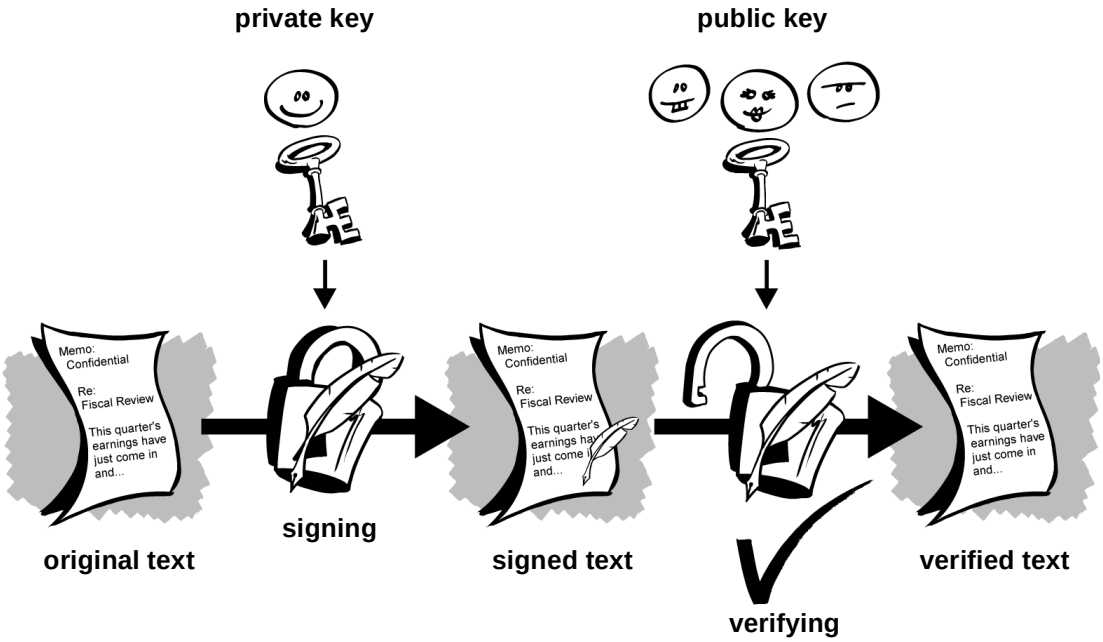


Figure 1-6. Simple digital signatures

Hash functions

The system described above has some problems. It is slow, and it produces an enormous volume of data—at least double the size of the original information. An improvement on the above scheme is the addition of a one-way *hash function* in the process. A one-way hash function takes variable-length input—in this case, a message of any length, even thousands or millions of bits—and produces a fixed-length output; say, 160-bits. The hash function ensures that, if the information is changed in any way—even by just one bit—an entirely different output value is produced.

PGP uses a cryptographically strong hash function on the plaintext the user is signing. This generates a fixed-length data item known as a *message digest*. (Again, any change to the information results in a totally different digest.)

Then PGP uses the digest and the private key to create the “signature.” PGP transmits the signature and the plaintext together. Upon receipt of the message, the recipient uses PGP to recompute the digest, thus verifying the signature. PGP can encrypt the plaintext or not; signing plaintext is useful if some of the recipients are not interested in or capable of verifying the signature.

As long as a secure hash function is used, there is no way to take someone's signature from one document and attach it to another, or to alter a signed message in any way. The slightest change in a signed document will cause the digital signature verification process to fail.

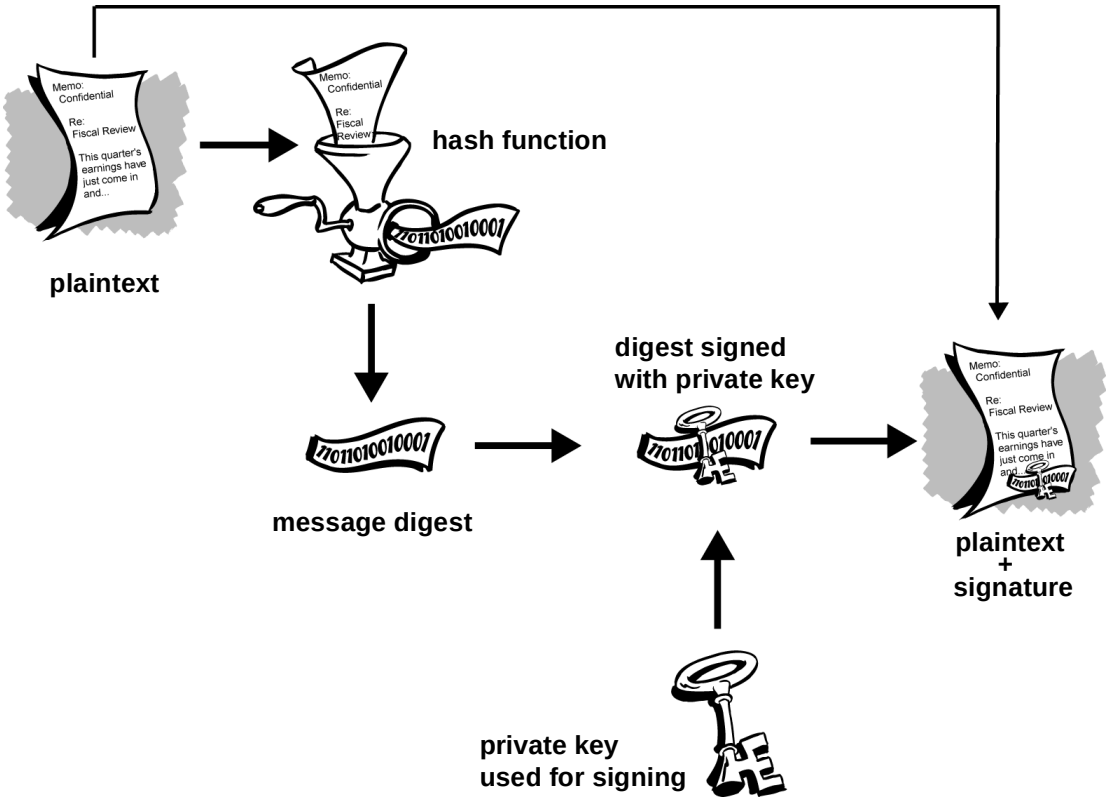


Figure 1-7. Secure digital signatures

Digital signatures play a major role in authenticating and *validating* other PGP users’ keys.

Digital certificates

One issue with public key cryptosystems is that users must be constantly vigilant to ensure that they are encrypting to the correct person's key. In an environment where it is safe to freely exchange keys via public servers, *man-in-the-middle* attacks are a potential threat. In this type of attack, someone posts a phony key with the name and user ID of the user's intended recipient. Data encrypted to— and intercepted by—the true owner of this bogus key is now in the wrong hands.

In a public key environment, it is vital that you are assured that the public key to which you are encrypting data is in fact the public key of the intended recipient and not a forgery. You could simply encrypt only to those keys which have been physically handed to you. But suppose you need to exchange information with people you have never met; how can you tell that you have the correct key?

Digital certificates, or *certs*, simplify the task of establishing whether a public key truly belongs to the purported owner.

A certificate is a form of credential. Examples might be your driver's license, your social security card, or your birth certificate. Each of these has some information on it identifying you and some authorization stating that someone else has confirmed your identity. Some certificates, such as your passport, are important enough confirmation of your identity that you would not want to lose them, lest someone use them to impersonate you.

A digital certificate is data that functions much like a physical certificate. A digital certificate is information included with a person's public key that helps others verify that a key is genuine or *valid*. Digital certificates are used to thwart attempts to substitute one person's key for another.

A digital certificate consists of three things:

- A public key.
- Certificate information. ("Identity" information about the user, such as name, user ID, and so on.)
- One or more digital signatures.

The purpose of the digital signature on a certificate is to state that the certificate information has been attested to by some other person or entity. The digital signature does not attest to the authenticity of the certificate as a whole; it vouches only that the signed identity information goes along with, or *is bound to*, the public key.

Thus, a certificate is basically a public key with one or two forms of ID attached, plus a hearty stamp of approval from some other trusted individual.

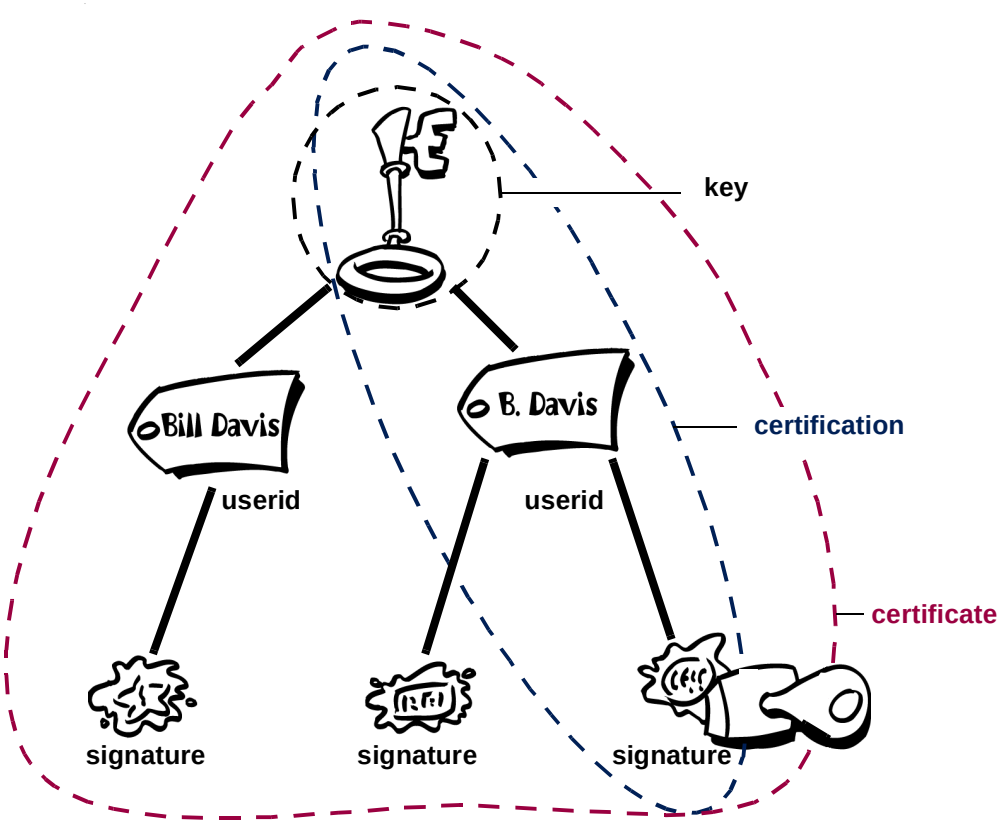


Figure 1-8. Anatomy of a PGP certificate

Certificate distribution

Certificates are utilized when it's necessary to exchange public keys with someone else. For small groups of people who wish to communicate securely, it is easy to manually exchange diskettes or emails containing each owner's public key. This is *manual public key distribution*, and it is practical only to a certain point. Beyond that point, it is necessary to put systems into place that can provide the necessary security, storage, and exchange mechanisms so coworkers, business partners, or strangers could communicate if need be. These can come in the form of storage-only repositories called *Certificate Servers*, or more structured systems that provide additional key management features and are called *Public Key Infrastructures (PKIs)*.

Certificate servers

A *certificate server*, also called a *cert server* or a *key server*, is a database that allows users to submit and retrieve digital certificates. A cert server usually provides some administrative features that enable a company to maintain its security policies—for example, allowing only those keys that meet certain requirements to be stored.

Public Key Infrastructures

A PKI contains the certificate storage facilities of a certificate server, but also provides certificate management facilities (the ability to issue, revoke, store, retrieve, and trust certificates). The main feature of a PKI is the introduction of what is known as a *Certification Authority*, or *CA*, which is a human entity—a person, group, department, company, or other association—that an organization has authorized to issue certificates to its computer users. (A CA's role is analogous to a country's government's Passport Office.) A CA creates certificates and digitally signs them using the CA's private key. Because of its role in creating certificates, the CA is the central component of a PKI. Using the CA's public key, anyone wanting to verify a certificate's authenticity verifies the issuing CA's digital signature, and hence, the integrity of the contents of the certificate (most importantly, the public key and the identity of the certificate holder).

Certificate formats

A digital certificate is basically a collection of identifying information bound together with a public key and signed by a trusted third party to prove its authenticity. A digital certificate can be one of a number of different *formats*.

PGP recognizes two different certificate formats:

- PGP certificates
- X.509 certificates

PGP certificate format

A PGP certificate includes (but is not limited to) the following information:

- **The PGP version number**—this identifies which version of PGP was used to create the key associated with the certificate.
- **The certificate holder's public key**—the public portion of your key pair, together with the algorithm of the key: RSA, DH (Diffie-Hellman), or DSA (Digital Signature Algorithm).

- **The certificate holder's information**—this consists of "identity" information about the user, such as his or her name, user ID, photograph, and so on.
- **The digital signature of the certificate owner**—also called a *self-signature*, this is the signature using the corresponding private key of the public key associated with the certificate.
- **The certificate's validity period**—the certificate's start date / time and expiration date / time; indicates when the certificate will expire.
- **The preferred symmetric encryption algorithm for the key**—indicates the encryption algorithm to which the certificate owner prefers to have information encrypted. The supported algorithms are CAST, IDEA or Triple-DES.

You might think of a PGP certificate as a public key with one or more labels tied to it (see Figure 1-9). On these 'labels' you'll find information identifying the owner of the key and a signature of the key's owner, which states that the key and the identification go together. (This particular signature is called a *self-signature*; every PGP certificate contains a self-signature.)

One unique aspect of the PGP certificate format is that a single certificate can contain multiple signatures. Several or many people may sign the key / identification pair to attest to their own assurance that the public key definitely belongs to the specified owner. If you look on a public certificate server, you may notice that certain certificates, such as that of PGP's creator, Phil Zimmermann, contain many signatures.

Some PGP certificates consist of a public key with several labels, each of which contains a different means of identifying the key's owner (for example, the owner's name and corporate email account, the owner's nickname and home email account, a photograph of the owner—all in one certificate). The list of signatures of each of those identities may differ; signatures attest to the authenticity that one of the labels belongs to the public key, not that all the labels on the key are authentic. (Note that 'authentic' is in the eye of its beholder—signatures are opinions, and different people devote different levels of due diligence in checking authenticity before signing a key.)

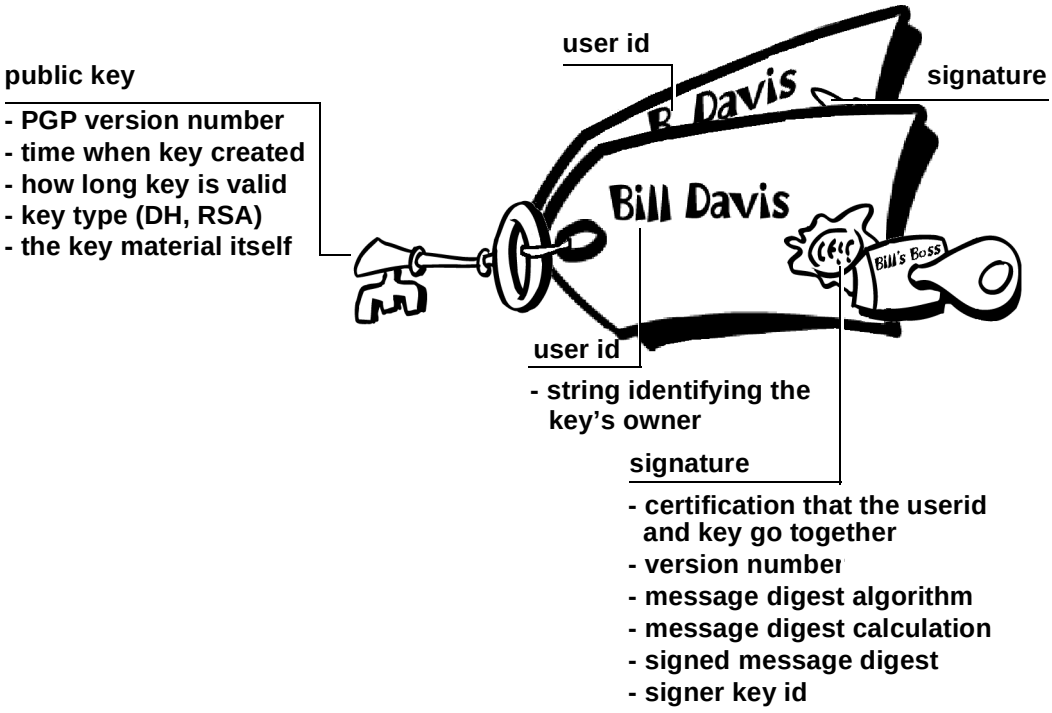


Figure 1-9. A PGP certificate

X.509 certificate format

X.509 is another very common certificate format. All X.509 certificates comply with the ITU-T X.509 international standard; thus (theoretically) X.509 certificates created for one application can be used by any application complying with X.509. In practice, however, different companies have created their own extensions to X.509 certificates, not all of which work together.

A certificate requires someone to validate that a public key and the name of the key's owner go together. With PGP certificates, anyone can play the role of validator. With X.509 certificates, the validator is always a Certification Authority or someone designated by a CA. (Bear in mind that PGP certificates also fully support a hierarchical structure using a *CA* to validate certificates.)

An X.509 certificate is a collection of a standard set of fields containing information about a user or device and their corresponding public key. The X.509 standard defines what information goes into the certificate, and describes how to encode it (the data format). All X.509 certificates have the following data:

- **The X.509 version number**—this identifies which version of the X.509 standard applies to this certificate, which affects what information can be specified in it. The most current is version 3.
- **The certificate holder's public key**—the public key of the certificate holder, together with an algorithm identifier which specifies which cryptosystem the key belongs to and any associated key parameters.
- **The serial number of the certificate**—the entity (application or person) that created the certificate is responsible for assigning it a unique serial number to distinguish it from other certificates it issues. This information is used in numerous ways; for example when a certificate is revoked, its serial number is placed in a *Certificate Revocation List* or *CRL*.
- **The certificate holder's unique identifier—** (or *DN—distinguished name*). This name is intended to be unique across the Internet. This name is intended to be unique across the Internet. A DN consists of multiple subsections and may look something like this:

  CN=Bob Allen, OU=Total Network Security Division, O=Network Associates, Inc., C=US

  (These refer to the subject's **C**ommon **N**ame, **O**rganizational **U**nit, **O**rganization, and **C**ountry.)
- **The certificate's validity period**—the certificate's start date/time and expiration date/time; indicates when the certificate will expire.
- **The unique name of the certificate issuer**—the unique name of the entity that signed the certificate. This is normally a CA. Using the certificate implies trusting the entity that signed this certificate. (Note that in some cases, such as *root* or *top-level* CA certificates, the issuer signs its own certificate.)
- **The digital signature of the issuer**—the signature using the private key of the entity that issued the certificate.
- **The signature algorithm identifier**—identifies the algorithm used by the CA to sign the certificate.

There are many differences between an X.509 certificate and a PGP certificate, but the most salient are as follows:

- you can create your own PGP certificate; you must request and be issued an X.509 certificate from a Certification Authority
- X.509 certificates natively support only a single name for the key's owner
- X.509 certificates support only a single digital signature to attest to the key's validity

To obtain an X.509 certificate, you must ask a CA to issue you a certificate. You provide your public key, proof that you possess the corresponding private key, and some specific information about yourself. You then digitally sign the information and send the whole package—the certificate *request*—to the CA. The CA then performs some due diligence in verifying that the information you provided is correct, and if so, generates the certificate and returns it.

You might think of an X.509 certificate as looking like a standard paper certificate (similar to one you might have received for completing a class in basic First Aid) with a public key taped to it. It has your name and some information about you on it, plus the signature of the person who issued it to you.

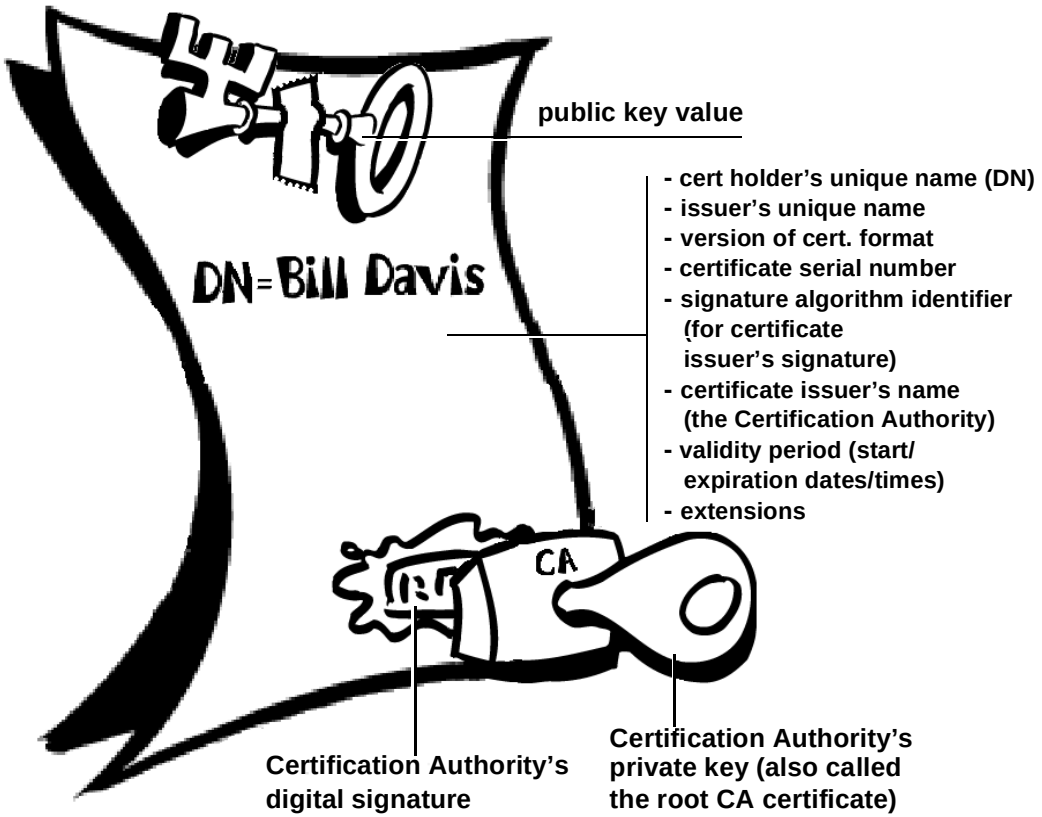


Figure 1-10. An X.509 certificate

Probably the most widely visible use of X.509 certificates today is in web browsers.

Validity and trust

Every user in a public key system is vulnerable to mistaking a phony key (certificate) for a real one. *Validity* is confidence that a public key certificate belongs to its purported owner. Validity is essential in a public key environment where you must constantly establish whether or not a particular certificate is authentic.

When you've assured yourself that a certificate belonging to someone else is valid, you can sign the copy on your keyring to attest to the fact that you've checked the certificate and that it's an authentic one. If you want others to know that you gave the certificate your stamp of approval, you can export the signature to a certificate server so that others can see it.

As described in the section, "Public Key Infrastructures." some companies designate one or more Certification Authorities (CAs) to indicate certificate validity. In an organization using a PKI with X.509 certificates, it is the job of the CA to *issue* certificates to users—a process which generally entails responding to a user's request for a certificate. In an organization using PGP certificates without a PKI, it is the job of the CA to check the authenticity of all PGP certificates and then sign the good ones. Basically, the main purpose of a CA is to bind a public key to the identification information contained in the certificate and thus assure third parties that some measure of care was taken to ensure that this binding of the identification information and key is valid.

The CA is the Grand Pooh-bah of validation in an organization; someone whom everyone trusts, and in some organizations, like those using a PKI, no certificate is considered valid unless it has been signed by a trusted CA.

Checking validity

One way to establish validity is to go through some manual process. There are several ways to accomplish this. You could require your intended recipient to physically hand you a copy of his or her public key. But this is often inconvenient and inefficient.

Another way is to manually check the certificate's *fingerprint*. Just as every human's fingerprints are unique, every PGP certificate's fingerprint is unique. The fingerprint is a hash of the user's certificate and appears as one of the certificate's properties. In PGP, the fingerprint can appear as a hexadecimal number or a series of so-called *biometric words*, which are phonetically distinct and are used to make the fingerprint identification process a little easier.

You can check that a certificate is valid by calling the key's owner (so that you originate the transaction) and asking the owner to read his or her key's fingerprint to you and verifying that fingerprint against the one you believe to be the real one. This works if you know the owner's voice, but, how do you manually verify the identity of someone you don't know? Some people put the fingerprint of their key on their business cards for this very reason.

Another way to establish validity of someone's certificate is to *trust* that a third individual has gone through the process of validating it.

A CA, for example, is responsible for ensuring that prior to issuing to a certificate, he or she carefully checks it to be sure the public key portion really belongs to the purported owner. Anyone who trusts the CA will automatically consider any certificates signed by the CA to be valid.

Another aspect of checking validity is to ensure that the certificate has not been revoked. For more information, see the section, "Certificate Revocation".

Establishing trust

You validate *certificates*. You trust *people*. More specifically, you trust people to validate other people' certificates. Typically, unless the owner hands you the certificate, you have to go by someone else's word that it is valid.

Meta and trusted introducers

In most situations, people completely trust the CA to establish certificates' validity. This means that everyone else relies upon the CA to go through the whole manual validation process for them. This is fine up to a certain number of users or number of work sites, and then it is not possible for the CA to maintain the same level of quality validation. In that case, adding other validators to the system is necessary.

A CA can also be a *meta-introducer*. A meta-introducer bestows not only validity on keys, but bestows the *ability to trust keys* upon others. Similar to the king who hands his seal to his trusted advisors so they can act on his authority, the meta-introducer enables others to act as *trusted introducers*. These trusted introducers can validate keys to the same effect as that of the meta-introducer. They cannot, however, create new trusted introducers.

Meta-introducer and trusted introducer are PGP terms. In an X.509 environment, the meta-introducer is called the *root Certification Authority* (*root CA*) and trusted introducers *subordinate* Certification Authorities.

The root CA uses the private key associated with a special certificate type called a *root CA certificate* to sign certificates. Any certificate signed by the root CA certificate is viewed as valid by any other certificate signed by the root. This validation process works even for certificates signed by other CAs in the system—as long as the root CA certificate signed the subordinate CA's certificate, any certificate signed by the CA is considered valid to others within the hierarchy. This process of checking back up through the system to see who signed whose certificate is called tracing a *certification path* or *certification chain*.

Trust models

In relatively closed systems, such as within a small company, it is easy to trace a certification path back to the root CA. However, users must often communicate with people outside of their corporate environment, including some whom they have never met, such as vendors, customers, clients, associates, and so on. Establishing a line of trust to those who have not been explicitly trusted by your CA is difficult.

Companies follow one or another *trust model*, which dictates how users will go about establishing certificate validity. There are three different models:

- Direct Trust
- Hierarchical Trust
- A Web of Trust

Direct Trust

Direct trust is the simplest trust model. In this model, a user trusts that a key is valid because he or she knows where it came from. All cryptosystems use this form of trust in some way. For example, in web browsers, the root Certification Authority keys are directly trusted because they were shipped by the manufacturer. If there is any form of hierarchy, it extends from these directly trusted certificates.

In PGP, a user who validates keys herself and never sets another certificate to be a trusted introducer is using direct trust.

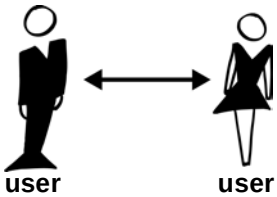


Figure 1-11. Direct trust

Hierarchical Trust

In a hierarchical system, there are a number of “root” certificates from which trust extends. These certificates may certify certificates themselves, or they may certify certificates that certify still other certificates down some chain. Consider it as a big trust “tree.” The “leaf” certificate's validity is verified by tracing backward from its certifier, to other certifiers, until a directly trusted root certificate is found.

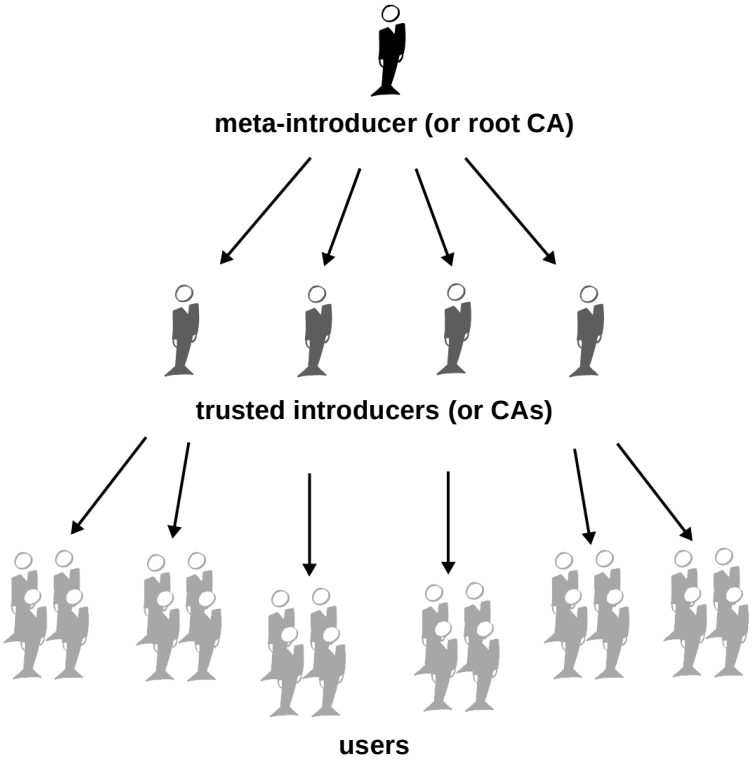


Figure 1-12. Hierarchical trust

Web of Trust

A web of trust encompasses both of the other models, but also adds the notion that trust is in the eye of the beholder (which is the real-world view) and the idea that more information is better. It is thus a cumulative trust model. A certificate might be trusted directly, or trusted in some chain going back to a directly trusted root certificate (the meta-introducer), or by some group of introducers.

Perhaps you've heard of the term *six degrees of separation*, which suggests that any person in the world can determine some link to any other person in the world using six or fewer other people as intermediaries. This is a web of introducers.

It is also the PGP view of trust. PGP uses digital signatures as its form of introduction. When any user signs another's key, he or she becomes an introducer of that key. As this process goes on, it establishes a *web of trust.*

In a PGP environment, *any* user can act as a certifying authority. Any PGP user can validate another PGP user's public key certificate. However, such a certificate is only valid to another user if the relying party recognizes the validator as a trusted introducer. (That is, you trust my opinion that others' keys are valid only if you consider me to be a trusted introducer. Otherwise, my opinion on other keys' validity is moot.)

Stored on each user's public keyring are indicators of

- whether or not the user considers a particular key to be valid
- the level of trust the user places on the key that the key's owner can serve as certifier of others' keys

You indicate, on your copy of my key, whether you think my judgement counts. It's really a reputation system: certain people are reputed to give good signatures, and people trust them to attest to other keys' validity.

Levels of trust in PGP

The highest level of trust in a key, *implicit* trust, is trust in your own key pair. PGP assumes that if you own the private key, you must trust the actions of its related public key. Any keys signed by your implicitly trusted key are valid.

There are three levels of trust you can assign to someone else's public key:

- *Complete* trust
- *Marginal* trust
- No trust (or *Untrusted*)

To make things confusing, there are also three levels of validity:

- Valid
- Marginally valid
- Invalid

To define another's key as a trusted introducer, you

1. Start with a valid key, one that is either

- signed by you or
- signed by another trusted introducer

and then

2. Set the level of trust you feel the key's owner is entitled.

For example, suppose your key ring contains Alice's key. You have validated Alice's key and you indicate this by signing it. You know that Alice is a real stickler for validating others' keys. You therefore assign her key with Complete trust. This makes Alice a Certification Authority. If Alice signs another's key, it appears as Valid on your keyring.

PGP requires one Completely trusted signature or two Marginally trusted signatures to establish a key as valid. PGP's method of considering two Marginals equal to one Complete is similar to a merchant asking for two forms of ID. You might consider Alice fairly trustworthy and also consider Bob fairly trustworthy. Either one alone runs the risk of accidentally signing a counterfeit key, so you might not place complete trust in either one. However, the odds that both individuals signed the same phony key are probably small.

Certificate Revocation

Certificates are only useful while they are valid. It is unsafe to simply assume that a certificate is valid forever. In most organizations and in all PKIs, certificates have a restricted lifetime. This constrains the period in which a system is vulnerable should a certificate compromise occur.

Certificates are thus created with a scheduled *validity period*: a start date / time and an expiration date / time. The certificate is expected to be usable for its entire validity period (its *lifetime*). When the certificate expires, it will no longer be valid, as the authenticity of its key / identification pair are no longer assured. (The certificate can still be safely used to reconfirm information that was encrypted or signed within the validity period—it should not be trusted for cryptographic tasks moving forward, however.)

There are also situations where it is necessary to invalidate a certificate prior to its expiration date, such as when an the certificate holder terminates employment with the company or suspects that the certificate's corresponding private key has been compromised. This is called *revocation*. A revoked certificate is *much* more suspect than an expired certificate. Expired certificates are unusable, but do not carry the same threat of compromise as a revoked certificate.

Anyone who has signed a certificate can revoke his or her signature on the certificate (provided he or she uses the same private key that created the signature). A revoked signature indicates that the signer no longer believes the public key and identification information belong together, or that the certificate's public key (or corresponding private key) has been compromised. A revoked signature should carry nearly as much weight as a revoked certificate.

With X.509 certificates, a revoked signature is practically the same as a revoked certificate given that the only signature on the certificate is the one that made it valid in the first place—the signature of the CA. PGP certificates provide the added feature that you can revoke your entire certificate (not just the signatures on it) if you yourself feel that the certificate has been compromised.

Only the certificate's owner (the holder of its corresponding private key) or someone whom the certificate's owner has *designated* as a revoker can revoke a PGP certificate. (Designating a revoker is a useful practice, as it's often the loss of the passphrase for the certificate's corresponding private key that leads a PGP user to revoke his or her certificate—a task that is only possible if one has access to the private key.) Only the certificate's issuer can revoke an X.509 certificate.

Communicating that a certificate has been revoked

When a certificate is revoked, it is important to make potential users of the certificate aware that it is no longer valid. With PGP certificates, the most common way to communicate that a certificate has been revoked is to post it on a certificate server so others who may wish to communicate with you are warned not to use that public key.

In a PKI environment, communication of revoked certificates is most commonly achieved via a data structure called a *Certificate Revocation List*, or *CRL*, which is published by the CA. The CRL contains a time-stamped, validated list of all revoked, unexpired certificates in the system. Revoked certificates remain on the list only until they expire, then they are removed from the list—this keeps the list from getting too long.

The CA distributes the CRL to users at some regularly scheduled interval (and potentially off-cycle, whenever a certificate is revoked). Theoretically, this will prevent users from unwittingly using a compromised certificate. It is possible, though, that there may be a time period between CRLs in which a newly compromised certificate is used.

What is a passphrase?

Most people are familiar with restricting access to computer systems via a *password*, which is a unique string of characters that a user types in as an identification code.

A *passphrase* is a longer version of a password, and in theory, a more secure one. Typically composed of multiple words, a passphrase is more secure against standard *dictionary attacks*, wherein the attacker tries all the words in the dictionary in an attempt to determine your password. The best passphrases are relatively long and complex and contain a combination of upper and lowercase letters, numeric and punctuation characters.

PGP uses a passphrase to encrypt your private key on your machine. Your private key is encrypted on your disk using a hash of your passphrase as the secret key. You use the passphrase to decrypt and use your private key. A passphrase should be hard for you to forget and difficult for others to guess. It should be something already firmly embedded in your long-term memory, rather than something you make up from scratch. Why? Because **if you forget your passphrase, you are out of luck**. Your private key is totally and absolutely useless without your passphrase and nothing can be done about it. Remember the quote earlier in this chapter? PGP is cryptography that will keep major governments out of your files. It will certainly keep you out of your files, too. Keep that in mind when you decide to change your passphrase to the punchline of that joke you can never quite remember.

Key splitting

They say that a secret is not a secret if it is known to more than one person. Sharing a private key pair poses such a problem. While it is not a recommended practice, sharing a private key pair is necessary at times. *Corporate Signing Keys*, for example, are private keys used by a company to sign—for example—legal documents, sensitive personnel information, or press releases to authenticate their origin. In such a case, it is worthwhile for multiple members of the company to have access to the private key. However, this means that any single individual can act fully on behalf of the company.

In such a case it is wise to *split* the key among multiple people in such a way that more than one or two people must present a piece of the key in order to reconstitute it to a usable condition. If too few pieces of the key are available, then the key is unusable.

Some examples are to split a key into three pieces and require two of them to reconstitute the key, or split it into two pieces and require both pieces. If a secure network connection is used during the reconstitution process, the key's shareholders need not be physically present in order to rejoin the key.

Technical details

This chapter provided a high-level introduction to cryptographic concepts and terminology. In Chapter 2, Phil Zimmermann, the creator of PGP, provides a more in-depth discussion of privacy, the technical details of how PGP works, including the various algorithms it uses, as well as various attacks and how to protect yourself against them.

For more information on cryptography, please refer to some of the books listed in the "Related reading" section of the Preface.

The GNU Privacy Handbook

Original at
https://www.gnupg.org/gph/en/manual.pdf

Chapter 1. Getting Started

GnuPG is a tool for secure communication. This chapter is a quick-start guide that covers the core functionality of GnuPG. This includes keypair creation, exchanging and verifying keys, encrypting and decrypting documents, and authenticating documents with digital signatures. It does not explain in detail the concepts behind public-key cryptography, encryption, and digital signatures. This is covered in Chapter 2. It also does not explain how to use GnuPG wisely. This is covered in Chapters 3 and 4.

GnuPG uses public-key cryptography so that users may communicate securely. In a public-key system, each user has a pair of keys consisting of a *private key* and a *public key*. A user's private key is kept secret; it need never be revealed. The public key may be given to anyone with whom the user wants to communicate. GnuPG uses a somewhat more sophisticated scheme in which a user has a primary keypair and then zero or more additional subordinate keypairs. The primary and subordinate keypairs are bundled to facilitate key management and the bundle can often be considered simply as one keypair.

Generating a new keypair

The command-line option `-gen-key` is used to create a new primary keypair.

```
alice% gpg -gen-key
gpg (GnuPG) 0.9.4; Copyright (C) 1999 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Please select what kind of key you want:
   (1) DSA and ElGamal (default)
   (2) DSA (sign only)
   (4) ElGamal (sign and encrypt)
Your selection?
```

GnuPG is able to create several different types of keypairs, but a primary key must be capable of making signatures. There are therefore only three options. Option 1 actually creates two keypairs. A DSA keypair is the primary keypair usable only for making signatures. An ElGamal subordinate keypair is also created for encryption. Option 2 is similar but creates only a DSA keypair. Option 4[1] creates a single ElGamal keypair usable for both making signatures and performing encryption. In all cases it is possible to later add additional subkeys for encryption and signing. For most users the default option is fine.

You must also choose a key size. The size of a DSA key must be between 512 and 1024 bits, and an ElGamal key may be of any size. GnuPG, however, requires that keys be no smaller than 768 bits. Therefore, if Option 1 was chosen and you choose a keysize larger than 1024 bits, the ElGamal key will have the requested size, but the DSA key will be 1024 bits.

1. Option 3 is to generate an ElGamal keypair that is not usable for making signatures.

```
About to generate a new ELG-E keypair.
              minimum keysize is  768 bits
              default keysize is 1024 bits
    highest suggested keysize is 2048 bits
What keysize do you want? (1024)
```

The longer the key the more secure it is against brute-force attacks, but for almost all purposes the default keysize is adequate since it would be cheaper to circumvent the encryption than try to break it. Also, encryption and decryption will be slower as the key size is increased, and a larger keysize may affect signature length. Once selected, the keysize can never be changed.

Finally, you must choose an expiration date. If Option 1 was chosen, the expiration date will be used for both the ElGamal and DSA keypairs.

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

For most users a key that does not expire is adequate. The expiration time should be chosen with care, however, since although it is possible to change the expiration date after the key is created, it may be difficult to communicate a change to users who have your public key.

You must provide a user ID in addition to the key parameters. The user ID is used to associate the key being created with a real person.

```
You need a User-ID to identify your key; the software constructs the user id
from Real Name, Comment and Email Address in this form:
    "Heinrich Heine (Der Dichter) <heinrichh@duesseldorf.de>"

Real name:
```

Only one user ID is created when a key is created, but it is possible to create additional user IDs if you want to use the key in two or more contexts, e.g., as an employee at work and a political activist on the side. A user ID should be created carefully since it cannot be edited after it is created.

GnuPG needs a passphrase to protect the primary and subordinate private keys that you keep in your possession.

```
You need a Passphrase to protect your private key.

Enter passphrase:
```

There is no limit on the length of a passphrase, and it should be carefully chosen. From the perspective of security, the passphrase to unlock the private key is one of the weakest points in GnuPG (and other public-key encryption systems as well) since it is the only protection you have if another individual gets your private key. Ideally, the passphrase should not use words from a dictionary and should mix the case of alphabetic characters as well as use non-alphabetic characters. A good passphrase is crucial to the secure use of GnuPG.

Generating a revocation certificate

After your keypair is created you should immediately generate a revocation certificate for the primary public key using the option `-gen-revoke`. If you forget your passphrase or if your private key is compromised or lost, this revocation certificate may be published to notify others that the public key should no longer be used. A revoked public key can still be used to verify signatures made by you in the past, but it cannot be used to encrypt future messages to you. It also does not affect your ability to decrypt messages sent to you in the past if you still do have access to the private key.

```
alice% gpg -output revoke.asc -gen-revoke mykey
[...]
```

The argument **`mykey`** must be a *key specifier*, either the key ID of your primary keypair or any part of a user ID that identifies your keypair. The generated certificate will be left in the file *`revoke.asc`*. If the `-output` option is omitted, the result will be placed on standard output. Since the certificate is short, you may wish to print a hardcopy of the certificate to store somewhere safe such as your safe deposit box. The certificate should not be stored where others can access it since anybody can publish the revocation certificate and render the corresponding public key useless.

Exchanging keys

To communicate with others you must exchange public keys. To list the keys on your public keyring use the command-line option `-list-keys`.

```
alice% gpg -list-keys
/users/alice/.gnupg/pubring.gpg
-------------------------
pub  1024D/BB7576AC 1999-06-04 Alice (Judge) <alice@cyb.org>
sub  1024g/78E9A8FA 1999-06-04
```

Exporting a public key

To send your public key to a correspondent you must first export it. The command-line option `-export` is used to do this. It takes an additional argument identifying the public key to export. As with the `-gen-revoke` option, either the key ID or any part of the user ID may be used to identify the key to export.

```
alice% gpg -output alice.gpg -export alice@cyb.org
```

The key is exported in a binary format, but this can be inconvenient when the key is to be sent though email or published on a web page. GnuPG therefore supports a command-line option `-armor`[2] that causes output to be

2. Many command-line options that are frequently used can also be set in a configuration file.

generated in an ASCII-armored format similar to uuencoded documents. In general, any output from GnuPG, e.g., keys, encrypted documents, and signatures, can be ASCII-armored by adding the `-armor` option.

```
alice% gpg -armor -export alice@cyb.org
---BEGIN PGP PUBLIC KEY BLOCK---
Version: GnuPG v0.9.7 (GNU/Linux)
Comment: For info see http://www.gnupg.org

[...]
---END PGP PUBLIC KEY BLOCK---
```

Importing a public key

A public key may be added to your public keyring with the `-import` option.

```
alice% gpg -import blake.gpg
gpg: key 9E98BC16: public key imported
gpg: Total number processed: 1
gpg:               imported: 1
alice% gpg -list-keys
/users/alice/.gnupg/pubring.gpg
-------------------------
pub  1024D/BB7576AC 1999-06-04 Alice (Judge) <alice@cyb.org>
sub  1024g/78E9A8FA 1999-06-04

pub  1024D/9E98BC16 1999-06-04 Blake (Executioner) <blake@cyb.org>
sub  1024g/5C8CBD41 1999-06-04
```

Once a key is imported it should be validated. GnuPG uses a powerful and flexible trust model that does not require you to personally validate each key you import. Some keys may need to be personally validated, however. A key is validated by verifying the key's fingerprint and then signing the key to certify it as a valid key. A key's fingerprint can be quickly viewed with the `-fingerprint` command-line option, but in order to certify the key you must edit it.

```
alice% gpg -edit-key blake@cyb.org

pub  1024D/9E98BC16  created: 1999-06-04 expires: never      trust: -/q
sub  1024g/5C8CBD41  created: 1999-06-04 expires: never
(1)  Blake (Executioner) <blake@cyb.org>

Command> fpr
pub  1024D/9E98BC16 1999-06-04 Blake (Executioner) <blake@cyb.org>
             Fingerprint: 268F 448F CCD7 AF34 183E  52D8 9BDE 1A08 9E98 BC16
```

A key's fingerprint is verified with the key's owner. This may be done in person or over the phone or through any other means as long as you can guarantee that you are communicating with the key's true owner. If the fingerprint you get is the same as the fingerprint the key's owner gets, then you can be sure that you have a correct copy of the key.

After checking the fingerprint, you may sign the key to validate it. Since key verification is a weak point in public-key cryptography, you should be extremely careful and *always* check a key's fingerprint with the owner before signing the key.

```
Command> sign

pub  1024D/9E98BC16  created: 1999-06-04 expires: never      trust: -/q
             Fingerprint: 268F 448F CCD7 AF34 183E  52D8 9BDE 1A08 9E98 BC16

     Blake (Executioner) <blake@cyb.org>

Are you really sure that you want to sign this key
with your key: "Alice (Judge) <alice@cyb.org>"

Really sign?
```

Once signed you can check the key to list the signatures on it and see the signature that you have added. Every user ID on the key will have one or more self-signatures as well as a signature for each user that has validated the key.

```
Command> check
uid  Blake (Executioner) <blake@cyb.org>
sig!       9E98BC16 1999-06-04   [self-signature]
sig!       BB7576AC 1999-06-04   Alice (Judge) <alice@cyb.org>
```

Encrypting and decrypting documents

A public and private key each have a specific role when encrypting and decrypting documents. A public key may be thought of as an open safe. When a correspondent encrypts a document using a public key, that document is put in the safe, the safe shut, and the combination lock spun several times. The corresponding private key is the combination that can reopen the safe and retrieve the document. In other words, only the person who holds the private key can recover a document encrypted using the associated public key.

The procedure for encrypting and decrypting documents is straightforward with this mental model. If you want to encrypt a message to Alice, you encrypt it using Alice's public key, and she decrypts it with her private key. If Alice wants to send you a message, she encrypts it using your public key, and you decrypt it with your private key.

To encrypt a document the option `-encrypt` is used. You must have the public keys of the intended recipients. The software expects the name of the document to encrypt as input; if omitted, it reads standard input. The encrypted result is placed on standard output or as specified using the option `-output`. The document is compressed for additional security in addition to encrypting it.

```
alice% gpg –output doc.gpg –encrypt –recipient blake@cyb.org doc
```

The `-recipient` option is used once for each recipient and takes an extra argument specifying the public key to which the document should be encrypted. The encrypted document can only be decrypted by someone

with a private key that complements one of the recipients' public keys. In particular, you cannot decrypt a document encrypted by you unless you included your own public key in the recipient list.

To decrypt a message the option `-decrypt` is used. You need the private key to which the message was encrypted. Similar to the encryption process, the document to decrypt is input, and the decrypted result is output.

```
blake% gpg -output doc -decrypt doc.gpg

You need a passphrase to unlock the secret key for
user: "Blake (Executioner) <blake@cyb.org>"
1024-bit ELG-E key, ID 5C8CBD41, created 1999-06-04 (main key ID 9E98BC16)

Enter passphrase:
```

Documents may also be encrypted without using public-key cryptography. Instead, you use a symmetric cipher to encrypt the document. The key used to drive the symmetric cipher is derived from a passphrase supplied when the document is encrypted, and for good security, it should not be the same passphrase that you use to protect your private key. Symmetric encryption is useful for securing documents when the passphrase does not need to be communicated to others. A document can be encrypted with a symmetric cipher by using the `-symmetric` option.

```
alice% gpg -output doc.gpg -symmetric doc
Enter passphrase:
```

Making and verifying signatures

A digital signature certifies and timestamps a document. If the document is subsequently modified in any way, a verification of the signature will fail. A digital signature can serve the same purpose as a hand-written signature with the additional benefit of being tamper-resistant. The GnuPG source distribution, for example, is signed so that users can verify that the source code has not been modified since it was packaged.

Creating and verifying signatures uses the public/private keypair in an operation different from encryption and decryption. A signature is created using the private key of the signer. The signature is verified using the corresponding public key. For example, Alice would use her own private key to digitally sign her latest submission to the Journal of Inorganic Chemistry. The associate editor handling her submission would use Alice's public key to check the signature to verify that the submission indeed came from Alice and that it had not been modified since Alice sent it. A consequence of using digital signatures is that it is difficult to deny that you made a digital signature since that would imply your private key had been compromised.

The command-line option `-sign` is used to make a digital signature. The document to sign is input, and the signed document is output.

```
alice% gpg -output doc.sig -sign doc

You need a passphrase to unlock the private key for
user: "Alice (Judge) <alice@cyb.org>"
1024-bit DSA key, ID BB7576AC, created 1999-06-04
```

```
Enter passphrase:
```

The document is compressed before being signed, and the output is in binary format.

Given a signed document, you can either check the signature or check the signature and recover the original document. To check the signature use the `-verify` option. To verify the signature and extract the document use the `-decrypt` option. The signed document to verify and recover is input and the recovered document is output.

```
blake% gpg –output doc –decrypt doc.sig
gpg: Signature made Fri Jun  4 12:02:38 1999 CDT using DSA key ID BB7576AC
gpg: Good signature from "Alice (Judge) <alice@cyb.org>"
```

Clearsigned documents

A common use of digital signatures is to sign usenet postings or email messages. In such situations it is undesirable to compress the document while signing it. The option `-clearsign` causes the document to be wrapped in an ASCII-armored signature but otherwise does not modify the document.

```
alice% gpg –clearsign doc

You need a passphrase to unlock the secret key for
user: "Alice (Judge) <alice@cyb.org>"
1024-bit DSA key, ID BB7576AC, created 1999-06-04

---BEGIN PGP SIGNED MESSAGE---
Hash: SHA1

[...]
---BEGIN PGP SIGNATURE---
Version: GnuPG v0.9.7 (GNU/Linux)
Comment: For info see http://www.gnupg.org

iEYEARECAAYFAjdYCQoACgkQJ9S6ULt1dqz6IwCfQ7wP6i/i8HhbcOSKF4ELyQB1
oCoAoOuqpRqEzr4kOkQqHRLE/b8/Rw2k
=y6kj
---END PGP SIGNATURE---
```

Detached signatures

A signed document has limited usefulness. Other users must recover the original document from the signed version, and even with clearsigned documents, the signed document must be edited to recover the original. Therefore, there is a third method for signing a document that creates a detached signature, which is a separate file. A detached signature is created using the `-detach-sig` option.

```
alice% gpg –output doc.sig –detach–sig doc

You need a passphrase to unlock the secret key for
user: "Alice (Judge) <alice@cyb.org>"
```

```
1024-bit DSA key, ID BB7576AC, created 1999-06-04

Enter passphrase:
```

Both the document and detached signature are needed to verify the signature. The `-verify` option can be to check the signature.

```
blake% gpg –verify doc.sig doc
gpg: Signature made Fri Jun  4 12:38:46 1999 CDT using DSA key ID BB7576AC
gpg: Good signature from "Alice (Judge) <alice@cyb.org>"
```

Chapter 2. Concepts

GnuPG makes uses of several cryptographic concepts including *symmetric ciphers*, *public-key ciphers*, and *one-way hashing*. You can make basic use GnuPG without fully understanding these concepts, but in order to use it wisely some understanding of them is necessary.

This chapter introduces the basic cryptographic concepts used in GnuPG. Other books cover these topics in much more detail. A good book with which to pursue further study is Bruce Schneier (http://www.counterpane.com/schneier.htm "Applied Cryptography" (http://www.counterpane.com/applied.html).

Symmetric ciphers

A symmetric cipher is a cipher that uses the same key for both encryption and decryption. Two parties communicating using a symmetric cipher must agree on the key beforehand. Once they agree, the sender encrypts a message using the key, sends it to the receiver, and the receiver decrypts the message using the key. As an example, the German Enigma is a symmetric cipher, and daily keys were distributed as code books. Each day, a sending or receiving radio operator would consult his copy of the code book to find the day's key. Radio traffic for that day was then encrypted and decrypted using the day's key. Modern examples of symmetric ciphers include 3DES, Blowfish, and IDEA.

A good cipher puts all the security in the key and none in the algorithm. In other words, it should be no help to an attacker if he knows which cipher is being used. Only if he obtains the key would knowledge of the algorithm be needed. The ciphers used in GnuPG have this property.

Since all the security is in the key, then it is important that it be very difficult to guess the key. In other words, the set of possible keys, i.e., the *key space*, needs to be large. While at Los Alamos, Richard Feynman was famous for his ability to crack safes. To encourage the mystique he even carried around a set of tools including an old stethoscope. In reality, he used a variety of tricks to reduce the number of combinations he had to try to a small number and then simply guessed until he found the right combination. In other words, he reduced the size of the key space.

Britain used machines to guess keys during World War 2. The German Enigma had a very large key space, but the British built specialized computing engines, the Bombes, to mechanically try keys until the day's key was found. This meant that sometimes they found the day's key within hours of the new key's use, but it also meant that on some days they never did find the right key. The Bombes were not general-purpose computers but were precursors to modern-day computers.

Today, computers can guess keys very quickly, and this is why key size is important in modern cryptosystems. The cipher DES uses a 56-bit key, which means that there are 2^{56} possible keys. 2^{56} is 72,057,594,037,927,936 keys. This is a lot of keys, but a general-purpose computer can check the entire key space in a matter of days. A specialized computer can check it in hours. On the other hand, more recently designed ciphers such as 3DES, Blowfish, and IDEA all use 128-bit keys, which means there are 2^{128} possible keys. This is many, many more keys, and even if all the computers on the planet cooperated, it could still take more time than the age of the universe to find the key.

Public-key ciphers

The primary problem with symmetric ciphers is not their security but with key exchange. Once the sender and receiver have exchanged keys, that key can be used to securely communicate, but what secure communication channel was used to communicate the key itself? In particular, it would probably be much easier for an attacker to work to intercept the key than it is to try all the keys in the key space. Another problem is the number of keys needed. If there are *n* people who need to communicate, then *n(n-1)/2* keys are needed for each pair of people to communicate privately. This may be OK for a small number of people but quickly becomes unwieldy for large groups of people.

Public-key ciphers were invented to avoid the key-exchange problem entirely. A public-key cipher uses a pair of keys for sending messages. The two keys belong to the person receiving the message. One key is a *public key* and may be given to anybody. The other key is a *private key* and is kept secret by the owner. A sender encrypts a message using the public key and once encrypted, only the private key may be used to decrypt it.

This protocol solves the key-exchange problem inherent with symmetric ciphers. There is no need for the sender and receiver to agree upon a key. All that is required is that some time before secret communication the sender gets a copy of the receiver's public key. Furthermore, the one public key can be used by anybody wishing to communicate with the receiver. So only *n* keypairs are needed for *n* people to communicate secretly with one another.

Public-key ciphers are based on one-way trapdoor functions. A one-way function is a function that is easy to compute, but the inverse is hard to compute. For example, it is easy to multiply two prime numbers together to get a composite, but it is difficult to factor a composite into its prime components. A one-way trapdoor function is similar, but it has a trapdoor. That is, if some piece of information is known, it becomes easy to compute the inverse. For example, if you have a number made of two prime factors, then knowing one of the factors makes it easy to compute the second. Given a public-key cipher based on prime factorization, the public key contains a composite number made from two large prime factors, and the encryption algorithm uses that composite to encrypt the message. The algorithm to decrypt the message requires knowing the prime factors, so decryption is easy if you have the private key containing one of the factors but extremely difficult if you do not have it.

As with good symmetric ciphers, with a good public-key cipher all of the security rests with the key. Therefore, key size is a measure of the system's security, but one cannot compare the size of a symmetric cipher key and a public-key cipher key as a measure of their relative security. In a brute-force attack on a symmetric cipher with a key size of 80 bits, the attacker must enumerate up to 2^{80} keys to find the right key. In a brute-force attack on a public-key cipher with a key size of 512 bits, the attacker must factor a composite number encoded in 512 bits (up to 155 decimal digits). The workload for the attacker is fundamentally different depending on the cipher he is attacking. While 128 bits is sufficient for symmetric ciphers, given today's factoring technology public keys with 1024 bits are recommended for most purposes.

Hybrid ciphers

Public-key ciphers are no panacea. Many symmetric ciphers are stronger from a security standpoint, and public-key encryption and decryption are more expensive than the corresponding operations in symmetric systems. Public-key ciphers are nevertheless an effective tool for distributing symmetric cipher keys, and that is how they are used in hybrid cipher systems.

A hybrid cipher uses both a symmetric cipher and a public-key cipher. It works by using a public-key cipher to share a key for the symmetric cipher. The actual message being sent is then encrypted using the key and sent to the recipient. Since symmetric key sharing is secure, the symmetric key used is different for each message sent. Hence it is sometimes called a session key.

Both PGP and GnuPG use hybrid ciphers. The session key, encrypted using the public-key cipher, and the message being sent, encrypted with the symmetric cipher, are automatically combined in one package. The recipient uses his private-key to decrypt the session key and the session key is then used to decrypt the message.

A hybrid cipher is no stronger than the public-key cipher or symmetric cipher it uses, whichever is weaker. In PGP and GnuPG, the public-key cipher is probably the weaker of the pair. Fortunately, however, if an attacker could decrypt a session key it would only be useful for reading the one message encrypted with that session key. The attacker would have to start over and decrypt another session key in order to read any other message.

Digital signatures

A hash function is a many-to-one function that maps its input to a value in a finite set. Typically this set is a range of natural numbers. A simple hash function is $f(x) = 0$ for all integers x. A more interesting hash function is $f(x) = x \bmod 37$, which maps x to the remainder of dividing x by 37.

A document's digital signature is the result of applying a hash function to the document. To be useful, however, the hash function needs to satisfy two important properties. First, it should be hard to find two documents that hash to the same value. Second, given a hash value it should be hard to recover the document that produced that value.

Some public-key ciphers[1] could be used to sign documents. The signer encrypts the document with his *private* key. Anybody wishing to check the signature and see the document simply uses the signer's public key to decrypt the document. This algorithm does satisfy the two properties needed from a good hash function, but in practice, this algorithm is too slow to be useful.

An alternative is to use hash functions designed to satisfy these two important properties. SHA and MD5 are examples of such algorithms. Using such an algorithm, a document is signed by hashing it, and the hash value is the signature. Another person can check the signature by also hashing their copy of the document and comparing the hash value they get with the hash value of the original document. If they match, it is almost certain that the documents are identical.

Of course, the problem now is using a hash function for digital signatures without permitting an attacker to interfere with signature checking. If the document and signature are sent unencrypted, an attacker could modify the document and generate a corresponding signature without the recipient's knowledge. If only the document is encrypted, an attacker could tamper with the signature and cause a signature check to fail. A third option is to use a hybrid public-key encryption to encrypt both the signature and document. The signer uses his private key, and anybody can use his public key to check the signature and document. This sounds good but is actually nonsense. If this algorithm truly secured the document it would also secure it from tampering and there would be no need for the signature. The more serious problem, however, is that this does not protect either the

1. The cipher must have the property that the actual public key or private key could be used by the encryption algorithm as the public key. RSA is an example of such an algorithm while ElGamal is not an example.

signature or document from tampering. With this algorithm, only the session key for the symmetric cipher is encrypted using the signer's private key. Anybody can use the public key to recover the session key. Therefore, it is straightforward for an attacker to recover the session key and use it to encrypt substitute documents and signatures to send to others in the sender's name.

An algorithm that does work is to use a public key algorithm to encrypt only the signature. In particular, the hash value is encrypted using the signer's private key, and anybody can check the signature using the public key. The signed document can be sent using any other encryption algorithm including none if it is a public document. If the document is modified the signature check will fail, but this is precisely what the signature check is supposed to catch. The Digital Signature Standard (DSA) is a public key signature algorithm that works as just described. DSA is the primary signing algorithm used in GnuPG.

Chapter 3. Key Management

Key tampering is a major security weakness with public-key cryptography. An eavesdropper may tamper with a user's keyrings or forge a user's public key and post it for others to download and use. For example, suppose Chloe wants to monitor the messages that Alice sends to Blake. She could mount what is called a *man in the middle* attack. In this attack, Chloe creates a new public/private keypair. She replaces Alice's copy of Blake's public key with the new public key. She then intercepts the messages that Alice sends to Blake. For each intercept, she decrypts it using the new private key, reencrypts it using Blake's true public key, and forwards the reencrypted message to Blake. All messages sent from Alice to Blake can now be read by Chloe.

Good key management is crucial in order to ensure not just the integrity of your keyrings but the integrity of other users' keyrings as well. The core of key management in GnuPG is the notion of signing keys. Key signing has two main purposes: it permits you to detect tampering on your keyring, and it allows you to certify that a key truly belongs to the person named by a user ID on the key. Key signatures are also used in a scheme known as the *web of trust* to extend certification to keys not directly signed by you but signed by others you trust. Responsible users who practice good key management can defeat key tampering as a practical attack on secure communication with GnuPG.

Managing your own keypair

A keypair has a public key and a private key. A public key consists of the public portion of the master signing key, the public portions of the subordinate signing and encryption subkeys, and a set of user IDs used to associate the public key with a real person. Each piece has data about itself. For a key, this data includes its ID, when it was created, when it will expire, etc. For a user ID, this data includes the name of the real person it identifies, an optional comment, and an email address. The structure of the private key is similar, except that it contains only the private portions of the keys, and there is no user ID information.

The command-line option `-edit-key` may be used to view a keypair. For example,

```
chloe% gpg -edit-key chloe@cyb.org
Secret key is available.

pub  1024D/26B6AAE1  created: 1999-06-15 expires: never      trust: -/u
sub  2048g/0CF8CB7A  created: 1999-06-15 expires: never
sub  1792G/08224617  created: 1999-06-15 expires: 2002-06-14
sub   960D/B1F423E7  created: 1999-06-15 expires: 2002-06-14
(1)  Chloe (Jester) <chloe@cyb.org>
(2)  Chloe (Plebian) <chloe@tel.net>
Command>
```

The public key is displayed along with an indication of whether or not the private key is available. Information about each component of the public key is then listed. The first column indicates the type of the key. The keyword `pub` identifies the public master signing key, and the keyword `sub` identifies a public subordinate key. The second column indicates the key's bit length, type, and ID. The type is `D` for a DSA key, `g` for an encryption-only ElGamal key, and `G` for an ElGamal key that may be used for both encryption and signing. The creation date and expiration date are given in columns three and four. The user IDs are listed following the keys.

More information about the key can be obtained with interactive commands. The command **toggle** switches between the public and private components of a keypair if indeed both components are available.

```
Command> toggle

sec  1024D/26B6AAE1  created: 1999-06-15 expires: never
sbb  2048g/0CF8CB7A  created: 1999-06-15 expires: never
sbb  1792G/08224617  created: 1999-06-15 expires: 2002-06-14
sbb   960D/B1F423E7  created: 1999-06-15 expires: 2002-06-14
(1)  Chloe (Jester) <chloe@cyb.org>
(2)  Chloe (Plebian) <chloe@tel.net>
```

The information provided is similar to the listing for the public-key component. The keyword `sec` identifies the private master signing key, and the keyword `sbb` identifies the private subordinates keys. The user IDs from the public key are also listed for convenience.

Key integrity

When you distribute your public key, you are distributing the public components of your master and subordinate keys as well as the user IDs. Distributing this material alone, however, is a security risk since it is possible for an attacker to tamper with the key. The public key can be modified by adding or substituting keys, or by adding or changing user IDs. By tampering with a user ID, the attacker could change the user ID's email address to have email redirected to himself. By changing one of the encryption keys, the attacker would also be able to decrypt the messages redirected to him.

Using digital signatures is a solution to this problem. When data is signed by a private key, the corresponding public key is bound to the signed data. In other words, only the corresponding public key can be used to verify the signature and ensure that the data has not been modified. A public key can be protected from tampering by using its corresponding private master key to sign the public key components and user IDs, thus binding the components to the public master key. Signing public key components with the corresponding private master signing key is called *self-signing*, and a public key that has self-signed user IDs bound to it is called a *certificate*.

As an example, Chloe has two user IDs and three subkeys. The signatures on the user IDs can be checked with the command **check** from the key edit menu.

```
chloe% gpg -edit-key chloe
Secret key is available.

pub  1024D/26B6AAE1  created: 1999-06-15 expires: never      trust: -/u
sub  2048g/0CF8CB7A  created: 1999-06-15 expires: never
sub  1792G/08224617  created: 1999-06-15 expires: 2002-06-14
sub   960D/B1F423E7  created: 1999-06-15 expires: 2002-06-14
(1)  Chloe (Jester) <chloe@cyb.org>
(2)  Chloe (Plebian) <chloe@tel.net>

Command> check
uid  Chloe (Jester) <chloe@cyb.org>
sig!    26B6AAE1 1999-06-15  [self-signature]
uid  Chloe (Plebian) <chloe@tel.net>
```

```
sig!    26B6AAE1 1999-06-15  [self-signature]
```

As expected, the signing key for each signature is the master signing key with key ID `0x26B6AAE1`. The self-signatures on the subkeys are present in the public key, but they are not shown by the GnuPG interface.

Adding and deleting key components

Both new subkeys and new user IDs may be added to your keypair after it has been created. A user ID is added using the command **adduid**. You are prompted for a real name, email address, and comment just as when you create an initial keypair. A subkey is added using the command **addkey**. The interface is similar to the interface used when creating an initial keypair. The subkey may be a DSA signing key, and encrypt-only ElGamal key, or a sign-and-encrypt ElGamal key. When a subkey or user ID is generated it is self-signed using your master signing key, which is why you must supply your passphrase when the key is generated.

Additional user IDs are useful when you need multiple identities. For example, you may have an identity for your job and an identity for your work as a political activist. Coworkers will know you by your work user ID. Coactivists will know you by your activist user ID. Since those groups of people may not overlap, though, each group may not trust the other user ID. Both user IDs are therefore necessary.

Additional subkeys are also useful. The user IDs associated with your public master key are validated by the people with whom you communicate, and changing the master key therefore requires recertification. This may be difficult and time consuming if you communicate with many people. On the other hand, it is good to periodically change encryption subkeys. If a key is broken, all the data encrypted with that key will be vulnerable. By changing keys, however, only the data encrypted with the one broken key will be revealed.

Subkeys and user IDs may also be deleted. To delete a subkey or user ID you must first select it using the **key** or **uid** commands respectively. These commands are toggles. For example, the command **key *2*** selects the second subkey, and invoking **key *2*** again deselects it. If no extra argument is given, all subkeys or user IDs are deselected. Once the user IDs to be deleted are selected, the command **deluid** actually deletes the user IDs from your key. Similarly, the command **delkey** deletes all selected subkeys from both your public and private keys.

For local keyring management, deleting key components is a good way to trim other people's public keys of unnecessary material. Deleting user IDs and subkeys on your own key, however, is not always wise since it complicates key distribution. By default, when a user imports your updated public key it will be merged with the old copy of your public key on his ring if it exists. The components from both keys are combined in the merge, and this effectively restores any components you deleted. To properly update the key, the user must first delete the old version of your key and then import the new version. This puts an extra burden on the people with whom you communicate. Furthermore, if you send your key to a keyserver, the merge will happen regardless, and anybody who downloads your key from a keyserver will never see your key with components deleted. Consequently, for updating your own key it is better to revoke key components instead of deleting them.

Revoking key components

To revoke a subkey it must be selected. Once selected it may be revoked with the **revkey** command. The key is revoked by adding a revocation self-signature to the key. Unlike the command-line option `-gen-revoke`, the effect of revoking a subkey is immediate.

```
Command> revkey
Do you really want to revoke this key? y

You need a passphrase to unlock the secret key for
user: "Chloe (Jester) <chloe@cyb.org>"
1024-bit DSA key, ID B87DBA93, created 1999-06-28


pub  1024D/B87DBA93  created: 1999-06-28 expires: never      trust: -/u
sub  2048g/B7934539  created: 1999-06-28 expires: never
sub  1792G/4E3160AD  created: 1999-06-29 expires: 2000-06-28
rev! subkey has been revoked: 1999-06-29
sub   960D/E1F56448  created: 1999-06-29 expires: 2000-06-28
(1)  Chloe (Jester) <chloe@cyb.org>
(2)  Chloe (Plebian) <chloe@tel.net>
```

A user ID is revoked differently. Normally, a user ID collects signatures that attest that the user ID describes the person who actually owns the associated key. In theory, a user ID describes a person forever, since that person will never change. In practice, though, elements of the user ID such as the email address and comment may change over time, thus invalidating the user ID.

The OpenPGP specification does not support user ID revocation, but a user ID can effectively be revoked by revoking the self-signature on the user ID. For the security reasons described previously, correspondents will not trust a user ID with no valid self-signature.

A signature is revoked by using the command **revsig**. Since you may have signed any number of user IDs, the user interface prompts you to decide for each signature whether or not to revoke it.

```
Command> revsig
You have signed these user IDs:
     Chloe (Jester) <chloe@cyb.org>
   signed by B87DBA93 at 1999-06-28
     Chloe (Plebian) <chloe@tel.net>
   signed by B87DBA93 at 1999-06-28
user ID: "Chloe (Jester) <chloe@cyb.org>"
signed with your key B87DBA93 at 1999-06-28
Create a revocation certificate for this signature? (y/N)n
user ID: "Chloe (Plebian) <chloe@tel.net>"
signed with your key B87DBA93 at 1999-06-28
Create a revocation certificate for this signature? (y/N)y
You are about to revoke these signatures:
     Chloe (Plebian) <chloe@tel.net>
   signed by B87DBA93 at 1999-06-28
Really create the revocation certificates? (y/N)y

You need a passphrase to unlock the secret key for
user: "Chloe (Jester) <chloe@cyb.org>"
1024-bit DSA key, ID B87DBA93, created 1999-06-28


pub  1024D/B87DBA93  created: 1999-06-28 expires: never      trust: -/u
sub  2048g/B7934539  created: 1999-06-28 expires: never
sub  1792G/4E3160AD  created: 1999-06-29 expires: 2000-06-28
```

```
rev! subkey has been revoked: 1999-06-29
sub   960D/E1F56448  created: 1999-06-29 expires: 2000-06-28
(1)  Chloe (Jester) <chloe@cyb.org>
(2)  Chloe (Plebian) <chloe@tel.net>
```

A revoked user ID is indicated by the revocation signature on the ID when the signatures on the key's user IDs are listed.

```
Command> check
uid  Chloe (Jester) <chloe@cyb.org>
sig!    B87DBA93 1999-06-28  [self-signature]
uid  Chloe (Plebian) <chloe@tel.net>
rev!    B87DBA93 1999-06-29  [revocation]
sig!    B87DBA93 1999-06-28  [self-signature]
```

Revoking both subkeys and self-signatures on user IDs adds revocation self-signatures to the key. Since signatures are being added and no material is deleted, a revocation will always be visible to others when your updated public key is distributed and merged with older copies of it. Revocation therefore guarantees that everybody has a consistent copy of your public key.

Updating a key's expiration time

The expiration time of a key may be updated with the command **expire** from the key edit menu. If no key is selected the expiration time of the primary key is updated. Otherwise the expiration time of the selected subordinate key is updated.

A key's expiration time is associated with the key's self-signature. The expiration time is updated by deleting the old self-signature and adding a new self-signature. Since correspondents will not have deleted the old self-signature, they will see an additional self-signature on the key when they update their copy of your key. The latest self-signature takes precedence, however, so all correspondents will unambiguously know the expiration times of your keys.

Validating other keys on your public keyring

In Chapter 1 a procedure was given to validate your correspondents' public keys: a correspondent's key is validated by personally checking his key's fingerprint and then signing his public key with your private key. By personally checking the fingerprint you can be sure that the key really does belong to him, and since you have signed they key, you can be sure to detect any tampering with it in the future. Unfortunately, this procedure is awkward when either you must validate a large number of keys or communicate with people whom you do not know personally.

GnuPG addresses this problem with a mechanism popularly known as the *web of trust*. In the web of trust model, responsibility for validating public keys is delegated to people you trust. For example, suppose

- Alice has signed Blake's key, and
- Blake has signed Chloe's key and Dharma's key.

If Alice trusts Blake to properly validate keys that he signs, then Alice can infer that Chloe's and Dharma's keys are valid without having to personally check them. She simply uses her validated copy of Blake's public key to check that Blake's signatures on Chloe's and Dharma's are good. In general, assuming that Alice fully trusts everybody to properly validate keys they sign, then any key signed by a valid key is also considered valid. The root is Alice's key, which is axiomatically assumed to be valid.

Trust in a key's owner

In practice trust is subjective. For example, Blake's key is valid to Alice since she signed it, but she may not trust Blake to properly validate keys that he signs. In that case, she would not take Chloe's and Dharma's key as valid based on Blake's signatures alone. The web of trust model accounts for this by associating with each public key on your keyring an indication of how much you trust the key's owner. There are four trust levels.

unknown
: Nothing is known about the owner's judgment in key signing. Keys on your public keyring that you do not own initially have this trust level.

none
: The owner is known to improperly sign other keys.

marginal
: The owner understands the implications of key signing and properly validates keys before signing them.

full
: The owner has an excellent understanding of key signing, and his signature on a key would be as good as your own.

A key's trust level is something that you alone assign to the key, and it is considered private information. It is not packaged with the key when it is exported; it is even stored separately from your keyrings in a separate database.

The GnuPG key editor may be used to adjust your trust in a key's owner. The command is **trust**. In this example Alice edits her trust in Blake and then updates the trust database to recompute which keys are valid based on her new trust in Blake.

```
alice% gpg –edit-key blake

pub  1024D/8B927C8A  created: 1999-07-02 expires: never      trust: q/f
sub  1024g/C19EA233  created: 1999-07-02 expires: never
(1)  Blake (Executioner) <blake@cyb.org>

Command> trust
pub  1024D/8B927C8A  created: 1999-07-02 expires: never      trust: q/f
sub  1024g/C19EA233  created: 1999-07-02 expires: never
(1)  Blake (Executioner) <blake@cyb.org>

Please decide how far you trust this user to correctly
```

```
verify other users' keys (by looking at passports,
checking fingerprints from different sources...)?

 1 = Don't know
 2 = I do NOT trust
 3 = I trust marginally
 4 = I trust fully
 s = please show me more information
 m = back to the main menu

Your decision? 3

pub  1024D/8B927C8A  created: 1999-07-02 expires: never      trust: m/f
sub  1024g/C19EA233  created: 1999-07-02 expires: never
(1)  Blake (Executioner) <blake@cyb.org>

Command> quit
[...]
```

Trust in the key's owner and the key's validity are indicated to the right when the key is displayed. Trust in the owner is displayed first and the key's validity is second[1]. The four trust/validity levels are abbreviated: unknown (`q`), none (`n`), marginal (`m`), and full (`f`). In this case, Blake's key is fully valid since Alice signed it herself. She initially has an unknown trust in Blake to properly sign other keys but decides to trust him marginally.

Using trust to validate keys

The web of trust allows a more elaborate algorithm to be used to validate a key. Formerly, a key was considered valid only if you signed it personally. A more flexible algorithm can now be used: a key *K* is considered valid if it meets two conditions:

1. it is signed by enough valid keys, meaning
   - you have signed it personally,
   - it has been signed by one fully trusted key, or
   - it has been signed by three marginally trusted keys; and
2. the path of signed keys leading from *K* back to your own key is five steps or shorter.

The path length, number of marginally trusted keys required, and number of fully trusted keys required may be adjusted. The numbers given above are the default values used by GnuPG.

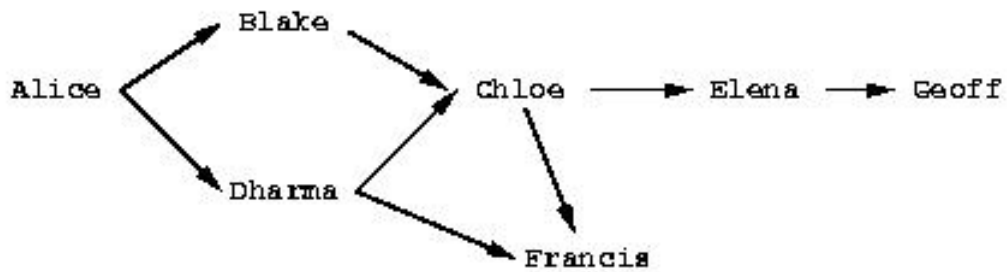
1. GnuPG overloads the word "trust" by using it to mean trust in an owner and trust in a key. This can be confusing. Sometimes trust in an owner is referred to as *owner-trust* to distinguish it from trust in a key. Throughout this manual, however, "trust" is used to mean trust in a key's owner, and "validity" is used to mean trust that a key belongs to the human associated with the key ID.

Figure 3-1 shows a web of trust rooted at Alice. The graph illustrates who has signed who's keys. The table shows which keys Alice considers valid based on her trust in the other members of the web. This example assumes that two marginally-trusted keys or one fully-trusted key is needed to validate another key. The maximum path length is three.

When computing valid keys in the example, Blake and Dharma's are always considered fully valid since they were signed directly by Alice. The validity of the other keys depends on trust. In the first case, Dharma is trusted fully, which implies that Chloe's and Francis's keys will be considered valid. In the second example, Blake and Dharma are trusted marginally. Since two marginally trusted keys are needed to fully validate a key, Chloe's key will be considered fully valid, but Francis's key will be considered only marginally valid. In the case where Chloe and Dharma are marginally trusted, Chloe's key will be marginally valid since Dharma's key is fully valid. Francis's key, however, will also be considered marginally valid since only a fully valid key can be used to validate other keys, and Dharma's key is the only fully valid key that has been used to sign Francis's key. When marginal trust in Blake is added, Chloe's key becomes fully valid and can then be used to fully validate Francis's key and marginally validate Elena's key. Lastly, when Blake, Chloe, and Elena are fully trusted, this is still insufficient to validate Geoff's key since the maximum certification path is three, but the path length from Geoff back to Alice is four.

The web of trust model is a flexible approach to the problem of safe public key exchange. It permits you to tune GnuPG to reflect how you use it. At one extreme you may insist on multiple, short paths from your key to another key *K* in order to trust it. On the other hand, you may be satisfied with longer paths and perhaps as little as one path from your key to the other key *K*. Requiring multiple, short paths is a strong guarantee that *K* belongs to whom your think it does. The price, of course, is that it is more difficult to validate keys since you must personally sign more keys than if you accepted fewer and longer paths.

Figure 3-1. A hypothetical web of trust



	trust	validity	
marginal	full	marginal	full
	Dharma		Blake, Chloe, Dharma, Francis
Blake, Dharma		Francis	Blake, Chloe, Dharma
Chloe, Dharma		Chloe, Francis	Blake, Dharma

	trust	validity	
marginal	full	marginal	full
Blake, Chloe, Dharma		Elena	Blake, Chloe, Dharma, Francis
	Blake, Chloe, Elena		Blake, Chloe, Elena, Francis

Distributing keys

Ideally, you distribute your key by personally giving it to your correspondents. In practice, however, keys are often distributed by email or some other electronic communication medium. Distribution by email is good practice when you have only a few correspondents, and even if you have many correspondents, you can use an alternative means such as posting your public key on your World Wide Web homepage. This is unacceptable, however, if people who need your public key do not know where to find it on the Web.

To solve this problem public key servers are used to collect and distribute public keys. A public key received by the server is either added to the server's database or merged with the existing key if already present. When a key request comes to the server, the server consults its database and returns the requested public key if found.

A keyserver is also valuable when many people are frequently signing other people's keys. Without a keyserver, when Blake sign's Alice's key then Blake would send Alice a copy of her public key signed by him so that Alice could add the updated key to her ring as well as distribute it to all of her correspondents. Going through this effort fulfills Alice's and Blake's responsibility to the community at large in building tight webs of trust and thus improving the security of PGP. It is nevertheless a nuisance if key signing is frequent.

Using a keyserver makes the process somewhat easier. When Blake signs Alice's key he sends the signed key to the key server. The key server adds Blake's signature to its copy of Alice's key. Individuals interested in updating their copy of Alice's key then consult the keyserver on their own initiative to retrieve the updated key. Alice need never be involved with distribution and can retrieve signatures on her key simply by querying a keyserver.

One or more keys may be sent to a keyserver using the command-line option `-send-keys`. The option takes one or more key specifiers and sends the specified keys to the key server. The key server to which to send the keys is specified with the command-line option `-keyserver`. Similarly, the option `-recv-keys` is used to retrieve keys from a keyserver, but the option `-recv-keys` requires a key ID be used to specify the key. In the following example Alice updates her public key with new signatures from the keyserver *`certserver.pgp.com`* and then sends her copy of Blake's public key to the same keyserver to contribute any new signatures she may have added.

```
alice% gpg -keyserver certserver.pgp.com -recv-key 0xBB7576AC
gpg: requesting key BB7576AC from certserver.pgp.com ...
gpg: key BB7576AC: 1 new signature
```

```
gpg: Total number processed: 1
gpg:      new signatures: 1
alice% gpg -keyserver certserver.pgp.com -send-key blake@cyb.org
gpg: success sending to 'certserver.pgp.com' (status=200)
```

There are several popular keyservers in use around the world. The major keyservers synchronize themselves, so it is fine to pick a keyserver close to you on the Internet and then use it regularly for sending and receiving keys.

Cryptography and Network Security Fourth Edition

Principles and Practices

William Stallings

Fully updated to reflect the latest trends and technologies, this is the definitive guide to encryption and network security principles, techniques, and effective usage.

Renowned author and consultant William Stallings systematically explains encryption concepts and standards; ciphers; symmetric and public key encryption; digital signatures; and much more. Next, Stallings turns to the *practice* of network security, introducing state-of-the-art applications for authentication, and for email, IP, and Web security. Finally, Stallings reviews the challenges of system security, covering leading attacks and today's best countermeasures. As always, Stallings provides unsurpassed support, including extensive supplements and online resources. For students, instructors, and working professionals alike, this Fourth Edition remains the field's best resource.

KEY FEATURES

Improved – **Simplified AES:** A clearer, simpler, easier-to-understand introduction to the Advanced Encryption Standard.
New – **Whirlpool:** Detailed treatment of the important new secure hash algorithm based on the use of symmetric block ciphers.
New – **CMAC:** Complete introduction to the Cipher-based Message Authentication Code for message authentication.
Improved – **PKI:** Enhanced, comprehensive coverage of Public Key Infrastructure.
New – **DDoS:** Understanding Distributed Denial of Service attacks, one of today's most significant threats to networks and the Internet.
New – **CCITSE:** Common Criteria for Information Technology Security Evaluation: A complete guide to today's emerging international framework for expressing security requirements and evaluating products and implementations.
Improved – Includes more than 100 new homework problems.
Hands-on experience: Provides unparalleled resources, including a course projects component containing research reports, programming projects, and reading/report assignments.
Internet resources: The author's website at **http://williamstallings.com/Crypto/Crypto4e.html** provides comprehensive support for students, instructors, and professionals.

ABOUT THE AUTHOR

William Stallings has made a unique contribution to understanding the broad sweep of technical developments in computer networking and architecture. He is a six-time winner of the annual Best Computer Science and Engineering Textbook award from the Textbook and Academic Authors Association. Stallings' 17 books include *Operating Systems, Computer Organization & Architecture,* and *Data and Computer Communications* (all from Prentice Hall). He is a member of the editorial board of *Cryptologia*, a scholarly journal devoted to all aspects of cryptology. Now an independent consultant, his clients include hardware and software providers, customers, and leading-edge government research institutions. Stallings holds a Ph.D. in computer science from MIT.

PEARSON
Prentice Hall

Upper Saddle River, NJ 07458
www.prenhall.com

ISBN 0-13-187316-4
9 780131 873162 90000

Stallings
Cryptography and Network Security
Principles and Practices
Fourth Edition
PEARSON
Prentice Hall

We cannot enter into alliance with neighboring princes until we are acquainted with their designs.

—*The Art of War*, Sun Tzu

KEY POINTS

- Kerberos is an authentication service designed for use in a distributed environment.
- Kerberos makes use of a trusted third-part authentication service that enables clients and servers to establish authenticated communication.
- X.509 defines the format for public-key certificates. This format is widely used in a variety of applications.
- A public key infrastructure (PKI) is defined as the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificates based on asymmetric cryptography.
- Typically, PKI implementations make use of X.509 certificates.

This chapter examines some of the authentication functions that have been developed to support application-level authentication and digital signatures.

We begin by looking at one of the earliest and also one of the most widely used services: Kerberos. Next, we examine the X.509 directory authentication service. This standard is important as part of the directory service that it supports, but is also a basic building block used in other standards, such as S/MIME, discussed in Chapter 15. Finally, this chapter examines the concept of a public-key infrastructure (PKI).

14.1 KERBEROS

Kerberos[1] is an authentication service developed as part of Project Athena at MIT. The problem that Kerberos addresses is this: Assume an open distributed environment in which users at workstations wish to access services on servers distributed throughout the network. We would like for servers to be able to restrict access to authorized users and to be able to authenticate requests for service. In this environment, a workstation cannot be trusted to identify its users correctly to network services. In particular, the following three threats exist:

- A user may gain access to a particular workstation and pretend to be another user operating from that workstation.

[1]"In Greek mythology, a many headed dog, commonly three, perhaps with a serpent's tail, the guardian of the entrance of Hades." From *Dictionary of Subjects and Symbols in Art*, by James Hall, Harper & Row, 1979. Just as the Greek Kerberos has three heads, the modern Kerberos was intended to have three components to guard a network's gate: authentication, accounting, and audit. The last two heads were never implemented.

- A user may alter the network address of a workstation so that the requests sent from the altered workstation appear to come from the impersonated workstation.
- A user may eavesdrop on exchanges and use a replay attack to gain entrance to a server or to disrupt operations.

In any of these cases, an unauthorized user may be able to gain access to services and data that he or she is not authorized to access. Rather than building in elaborate authentication protocols at each server, Kerberos provides a centralized authentication server whose function is to authenticate users to servers and servers to users. Unlike most other authentication schemes described in this book, Kerberos relies exclusively on symmetric encryption, making no use of public-key encryption.

Two versions of Kerberos are in common use. Version 4 [MILL88, STEI88] implementations still exist. Version 5 [KOHL94] corrects some of the security deficiencies of version 4 and has been issued as a proposed Internet Standard (RFC 1510).[2]

We begin this section with a brief discussion of the motivation for the Kerberos approach. Then, because of the complexity of Kerberos, it is best to start with a description of the authentication protocol used in version 4. This enables us to see the essence of the Kerberos strategy without considering some of the details required to handle subtle security threats. Finally, we examine version 5.

Motivation

If a set of users is provided with dedicated personal computers that have no network connections, then a user's resources and files can be protected by physically securing each personal computer. When these users instead are served by a centralized time-sharing system, the time-sharing operating system must provide the security. The operating system can enforce access control policies based on user identity and use the logon procedure to identify users.

Today, neither of these scenarios is typical. More common is a distributed architecture consisting of dedicated user workstations (clients) and distributed or centralized servers. In this environment, three approaches to security can be envisioned:

1. Rely on each individual client workstation to assure the identity of its user or users and rely on each server to enforce a security policy based on user identification (ID).
2. Require that client systems authenticate themselves to servers, but trust the client system concerning the identity of its user.
3. Require the user to prove his or her identity for each service invoked. Also require that servers prove their identity to clients.

In a small, closed environment, in which all systems are owned and operated by a single organization, the first or perhaps the second strategy may suffice.[3] But in a more open environment, in which network connections to other machines are supported, the third approach is needed to protect user information and resources

[2]Versions 1 through 3 were internal development versions. Version 4 is the "original" Kerberos.

[3]However, even a closed environment faces the threat of attack by a disgruntled employee.

housed at the server. Kerberos supports this third approach. Kerberos assumes a distributed client/server architecture and employs one or more Kerberos servers to provide an authentication service.

The first published report on Kerberos [STEI88] listed the following requirements:

- **Secure:** A network eavesdropper should not be able to obtain the necessary information to impersonate a user. More generally, Kerberos should be strong enough that a potential opponent does not find it to be the weak link.
- **Reliable:** For all services that rely on Kerberos for access control, lack of availability of the Kerberos service means lack of availability of the supported services. Hence, Kerberos should be highly reliable and should employ a distributed server architecture, with one system able to back up another.
- **Transparent:** Ideally, the user should not be aware that authentication is taking place, beyond the requirement to enter a password.
- **Scalable:** The system should be capable of supporting large numbers of clients and servers. This suggests a modular, distributed architecture.

To support these requirements, the overall scheme of Kerberos is that of a trusted third-party authentication service that uses a protocol based on that proposed by Needham and Schroeder [NEED78], which was discussed in Chapter 7. It is trusted in the sense that clients and servers trust Kerberos to mediate their mutual authentication. Assuming the Kerberos protocol is well designed, then the authentication service is secure if the Kerberos server itself is secure.[4]

Kerberos Version 4

Version 4 of Kerberos makes use of DES, in a rather elaborate protocol, to provide the authentication service. Viewing the protocol as a whole, it is difficult to see the need for the many elements contained therein. Therefore, we adopt a strategy used by Bill Bryant of Project Athena [BRYA88] and build up to the full protocol by looking first at several hypothetical dialogues. Each successive dialogue adds additional complexity to counter security vulnerabilities revealed in the preceding dialogue.

After examining the protocol, we look at some other aspects of version 4.

A Simple Authentication Dialogue In an unprotected network environment, any client can apply to any server for service. The obvious security risk is that of impersonation. An opponent can pretend to be another client and obtain unauthorized privileges on server machines. To counter this threat, servers must be able to confirm the identities of clients who request service. Each server can be required

[4]Remember that the security of the Kerberos server should not automatically be assumed but must be guarded carefully (e.g., in a locked room). It is well to remember the fate of the Greek Kerberos, whom Hercules was ordered by Eurystheus to capture as his Twelfth Labor: "Hercules found the great dog on its chain and seized it by the throat. At once the three heads tried to attack, and Kerberos lashed about with his powerful tail. Hercules hung on grimly, and Kerberos relaxed into unconsciousness. Eurystheus may have been surprised to see Hercules alive—when he saw the three slavering heads and the huge dog they belonged to he was frightened out of his wits, and leapt back into the safety of his great bronze jar." From *The Hamlyn Concise Dictionary of Greek and Roman Mythology*, by Michael Stapleton, Hamlyn, 1982.

to undertake this task for each client/server interaction, but in an open environment, this places a substantial burden on each server.

An alternative is to use an authentication server (AS) that knows the passwords of all users and stores these in a centralized database. In addition, the AS shares a unique secret key with each server. These keys have been distributed physically or in some other secure manner. Consider the following hypothetical dialogue:[5]

$$\begin{aligned}
&\textbf{(1)}\ \mathrm{C} \rightarrow \mathrm{AS}: \quad ID_C \| P_C \| ID_V \\
&\textbf{(2)}\ \mathrm{AS} \rightarrow \mathrm{C}: \quad Ticket \\
&\textbf{(3)}\ \mathrm{C} \rightarrow \mathrm{V}: \quad ID_C \| Ticket \\
&Ticket = \mathrm{E}(K_v, [ID_C \| AD_C \| ID_V])
\end{aligned}$$

where

C = client
AS = authentication server
V = server
ID_C = identifier of user on C
ID_V = identifier of V
P_C = password of user on C
AD_C = network address of C
K_v = secret encryption key shared by AS and V

In this scenario, the user logs on to a workstation and requests access to server V. The client module C in the user's workstation requests the user's password and then sends a message to the AS that includes the user's ID, the server's ID, and the user's password. The AS checks its database to see if the user has supplied the proper password for this user ID and whether this user is permitted access to server V. If both tests are passed, the AS accepts the user as authentic and must now convince the server that this user is authentic. To do so, the AS creates a ticket that contains the user's ID and network address and the server's ID. This ticket is encrypted using the secret key shared by the AS and this server. This ticket is then sent back to C. Because the ticket is encrypted, it cannot be altered by C or by an opponent.

With this ticket, C can now apply to V for service. C sends a message to V containing C's ID and the ticket. V decrypts the ticket and verifies that the user ID in the ticket is the same as the unencrypted user ID in the message. If these two match, the server considers the user authenticated and grants the requested service.

Each of the ingredients of message (3) is significant. The ticket is encrypted to prevent alteration or forgery. The server's ID (ID_V) is included in the ticket so that the server can verify that it has decrypted the ticket properly. ID_C is included in the ticket to indicate that this ticket has been issued on behalf of C. Finally, AD_C serves to counter the following threat. An opponent could capture the ticket transmitted in message (2), then use the name ID_C and transmit a message of form (3) from another

[5]The portion to the left of the colon indicates the sender and receiver; the portion to the right indicates the contents of the message, the symbol ‖ indicates concatenation.

workstation. The server would receive a valid ticket that matches the user ID and grant access to the user on that other workstation. To prevent this attack, the AS includes in the ticket the network address from which the original request came. Now the ticket is valid only if it is transmitted from the same workstation that initially requested the ticket.

A More Secure Authentication Dialogue Although the foregoing scenario solves some of the problems of authentication in an open network environment, problems remain. Two in particular stand out. First, we would like to minimize the number of times that a user has to enter a password. Suppose each ticket can be used only once. If user C logs on to a workstation in the morning and wishes to check his or her mail at a mail server, C must supply a password to get a ticket for the mail server. If C wishes to check the mail several times during the day, each attempt requires reentering the password. We can improve matters by saying that tickets are reusable. For a single logon session, the workstation can store the mail server ticket after it is received and use it on behalf of the user for multiple accesses to the mail server.

However, under this scheme it remains the case that a user would need a new ticket for every different service. If a user wished to access a print server, a mail server, a file server, and so on, the first instance of each access would require a new ticket and hence require the user to enter the password.

The second problem is that the earlier scenario involved a plaintext transmission of the password [message (1)]. An eavesdropper could capture the password and use any service accessible to the victim.

To solve these additional problems, we introduce a scheme for avoiding plaintext passwords and a new server, known as the ticket-granting server (TGS). The new but still hypothetical scenario is as follows:

Once per user logon session:

(1) C → AS: $ID_C \| ID_{tgs}$

(2) AS → C: $E(K_c, Ticket_{tgs})$

Once per type of service:

(3) C → TGS: $ID_C \| ID_V \| Ticket_{tgs}$

(4) TGS → C: $Ticket_v$

Once per service session:

(5) C → V: $ID_C \| Ticket_v$

$$Ticket_{tgs} = E(K_{tgs}, [ID_C \| AD_C \| ID_{tgs} \| TS_1 \| Lifetime_1])$$
$$Ticket_v = E(K_v, [ID_C \| AD_C \| ID_v \| TS_2 \| Lifetime_2])$$

The new service, TGS, issues tickets to users who have been authenticated to AS. Thus, the user first requests a ticket-granting ticket ($Ticket_{tgs}$) from the AS. The client module in the user workstation saves this ticket. Each time the user requires access to a new service, the client applies to the TGS, using the ticket to authenticate itself. The TGS then grants a ticket for the particular service. The client

saves each service-granting ticket and uses it to authenticate its user to a server each time a particular service is requested. Let us look at the details of this scheme:

1. The client requests a ticket-granting ticket on behalf of the user by sending its user's ID and password to the AS, together with the TGS ID, indicating a request to use the TGS service.
2. The AS responds with a ticket that is encrypted with a key that is derived from the user's password. When this response arrives at the client, the client prompts the user for his or her password, generates the key, and attempts to decrypt the incoming message. If the correct password is supplied, the ticket is successfully recovered.

Because only the correct user should know the password, only the correct user can recover the ticket. Thus, we have used the password to obtain credentials from Kerberos without having to transmit the password in plaintext. The ticket itself consists of the ID and network address of the user, and the ID of the TGS. This corresponds to the first scenario. The idea is that the client can use this ticket to request multiple service-granting tickets. So the ticket-granting ticket is to be reusable. However, we do not wish an opponent to be able to capture the ticket and use it. Consider the following scenario: An opponent captures the login ticket and waits until the user has logged off his or her workstation. Then the opponent either gains access to that workstation or configures his workstation with the same network address as that of the victim. The opponent would be able to reuse the ticket to spoof the TGS. To counter this, the ticket includes a timestamp, indicating the date and time at which the ticket was issued, and a lifetime, indicating the length of time for which the ticket is valid (e.g., eight hours). Thus, the client now has a reusable ticket and need not bother the user for a password for each new service request. Finally, note that the ticket-granting ticket is encrypted with a secret key known only to the AS and the TGS. This prevents alteration of the ticket. The ticket is reencrypted with a key based on the user's password. This assures that the ticket can be recovered only by the correct user, providing the authentication.

Now that the client has a ticket-granting ticket, access to any server can be obtained with steps 3 and 4:

1. The client requests a service-granting ticket on behalf of the user. For this purpose, the client transmits a message to the TGS containing the user's ID, the ID of the desired service, and the ticket-granting ticket.
2. The TGS decrypts the incoming ticket and verifies the success of the decryption by the presence of its ID. It checks to make sure that the lifetime has not expired. Then it compares the user ID and network address with the incoming information to authenticate the user. If the user is permitted access to the server V, the TGS issues a ticket to grant access to the requested service.

The service-granting ticket has the same structure as the ticket-granting ticket. Indeed, because the TGS is a server, we would expect that the same elements are needed to authenticate a client to the TGS and to authenticate a client to an application server. Again, the ticket contains a timestamp and lifetime. If the user wants access to the same service at a later time, the client can simply use

the previously acquired service-granting ticket and need not bother the user for a password. Note that the ticket is encrypted with a secret key (K_v) known only to the TGS and the server, preventing alteration.

Finally, with a particular service-granting ticket, the client can gain access to the corresponding service with step 5:

5. The client requests access to a service on behalf of the user. For this purpose, the client transmits a message to the server containing the user's ID and the service-granting ticket. The server authenticates by using the contents of the ticket.

This new scenario satisfies the two requirements of only one password query per user session and protection of the user password.

The Version 4 Authentication Dialogue Although the foregoing scenario enhances security compared to the first attempt, two additional problems remain. The heart of the first problem is the lifetime associated with the ticket-granting ticket. If this lifetime is very short (e.g., minutes), then the user will be repeatedly asked for a password. If the lifetime is long (e.g., hours), then an opponent has a greater opportunity for replay. An opponent could eavesdrop on the network and capture a copy of the ticket-granting ticket and then wait for the legitimate user to log out. Then the opponent could forge the legitimate user's network address and send the message of step (3) to the TGS. This would give the opponent unlimited access to the resources and files available to the legitimate user.

Similarly, if an opponent captures a service-granting ticket and uses it before it expires, the opponent has access to the corresponding service.

Thus, we arrive at an additional requirement. A network service (the TGS or an application service) must be able to prove that the person using a ticket is the same person to whom that ticket was issued.

The second problem is that there may be a requirement for servers to authenticate themselves to users. Without such authentication, an opponent could sabotage the configuration so that messages to a server were directed to another location. The false server would then be in a position to act as a real server and capture any information from the user and deny the true service to the user.

We examine these problems in turn and refer to Table 14.1, which shows the actual Kerberos protocol.

First, consider the problem of captured ticket-granting tickets and the need to determine that the ticket presenter is the same as the client for whom the ticket was issued. The threat is that an opponent will steal the ticket and use it before it expires. To get around this problem, let us have the AS provide both the client and the TGS with a secret piece of information in a secure manner. Then the client can prove its identity to the TGS by revealing the secret information, again in a secure manner. An efficient way of accomplishing this is to use an encryption key as the secure information; this is referred to as a session key in Kerberos.

Table 14.1a shows the technique for distributing the session key. As before, the client sends a message to the AS requesting access to the TGS. The AS responds with a message, encrypted with a key derived from the user's password (K_c), that contains the ticket. The encrypted message also contains a copy of the session key, $K_{c,tgs}$, where the subscripts indicate that this is a session key for C and TGS. Because this session key is

Table 14.1 Summary of Kerberos Version 4 Message Exchanges

(1) C → AS $ID_c \| ID_{tgs} \| TS_1$

(2) AS → C $E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$

$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$

(a) Authentication Service Exchange to obtain ticket-granting ticket

(3) C → TGS $ID_v \| Ticket_{tgs} \| Authenticator_c$

(4) TGS → C $E(K_{c,tgs}, [K_{c,v} \| ID_v \| TS_4 \| Ticket_v])$

$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$

$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$

$Authenticator_c = E(K_{c,tgs}, [ID_C \| AD_C \| TS_3])$

(b) Ticket-Granting Service Exchange to obtain service-granting ticket

(5) C → V $Ticket_v \| Authenticator_c$

(6) V → C $E(K_{c,v}, [TS_5 + 1])$ (for mutual authentication)

$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$

$Authenticator_c = E(K_{c,v}, [ID_C \| AD_C \| TS_5])$

(c) Client/Server Authentication Exchange to obtain service

inside the message encrypted with K_c, only the user's client can read it. The same session key is included in the ticket, which can be read only by the TGS. Thus, the session key has been securely delivered to both C and the TGS.

Note that several additional pieces of information have been added to this first phase of the dialogue. Message (1) includes a timestamp, so that the AS knows that the message is timely. Message (2) includes several elements of the ticket in a form accessible to C. This enables C to confirm that this ticket is for the TGS and to learn its expiration time.

Armed with the ticket and the session key, C is ready to approach the TGS. As before, C sends the TGS a message that includes the ticket plus the ID of the requested service (message (3) in Table 14.1b). In addition, C transmits an authenticator, which includes the ID and address of C's user and a timestamp. Unlike the ticket, which is reusable, the authenticator is intended for use only once and has a very short lifetime. The TGS can decrypt the ticket with the key that it shares with the AS. This ticket indicates that user C has been provided with the session key $K_{c,tgs}$. In effect, the ticket says, "Anyone who uses $K_{c,tgs}$ must be C." The TGS uses the session key to decrypt the authenticator. The TGS can then check the name and address from the authenticator with that of the ticket and with the network address of the incoming message. If all match, then the TGS is assured that the sender of the ticket is indeed the ticket's real owner. In effect, the authenticator says, "At time TS_3, I hereby use $K_{c,tgs}$." Note that the ticket does not prove anyone's identity but is a way to distribute keys securely. It is the authenticator that proves the client's identity. Because the authenticator can be used only once and has a short lifetime, the threat of an opponent stealing both the ticket and the authenticator for presentation later is countered.

The reply from the TGS, in message (4), follows the form of message (2). The message is encrypted with the session key shared by the TGS and C and includes a session key to be shared between C and the server V, the ID of V, and the timestamp of the ticket. The ticket itself includes the same session key.

C now has a reusable service-granting ticket for V. When C presents this ticket, as shown in message (5), it also sends an authenticator. The server can decrypt the ticket, recover the session key, and decrypt the authenticator.

If mutual authentication is required, the server can reply as shown in message (6) of Table 14.1. The server returns the value of the timestamp from the authenticator, incremented by 1, and encrypted in the session key. C can decrypt this message to recover the incremented timestamp. Because the message was encrypted by the session key, C is assured that it could have been created only by V. The contents of the message assure C that this is not a replay of an old reply.

Finally, at the conclusion of this process, the client and server share a secret key. This key can be used to encrypt future messages between the two or to exchange a new random session key for that purpose.

Table 14.2 summarizes the justification for each of the elements in the Kerberos protocol, and Figure 14.1 provides a simplified overview of the action.

Kerberos Realms and Multiple Kerberi A full-service Kerberos environment consisting of a Kerberos server, a number of clients, and a number of application servers requires the following:

1. The Kerberos server must have the user ID and hashed passwords of all participating users in its database. All users are registered with the Kerberos server.
2. The Kerberos server must share a secret key with each server. All servers are registered with the Kerberos server.

Such an environment is referred to as a **Kerberos realm**. The concept of *realm* can be explained as follows. A Kerberos realm is a set of managed nodes that share the same Kerberos database. The Kerberos database resides on the Kerberos master computer system, which should be kept in a physically secure room. A read-only copy of the Kerberos database might also reside on other Kerberos computer systems. However, all changes to the database must be made on the master computer system. Changing or accessing the contents of a Kerberos database requires the Kerberos master password. A related concept is that of a **Kerberos principal**, which is a service or user that is known to the Kerberos system. Each Kerberos principal is identified by its principal name. Principal names consist of three parts: a service or user name, an instance name, and a realm name

Networks of clients and servers under different administrative organizations typically constitute different realms. That is, it generally is not practical, or does not conform to administrative policy, to have users and servers in one administrative domain registered with a Kerberos server elsewhere. However, users in one realm may need access to servers in other realms, and some servers may be willing to provide service to users from other realms, provided that those users are authenticated.

Table 14.2 Rationale for the Elements of the Kerberos Version 4 Protocol

Message (1)	Client requests ticket-granting ticket
ID_C	Tells AS identity of user from this client
ID_{tgs}	Tells AS that user requests access to TGS
TS_1	Allows AS to verify that client's clock is synchronized with that of AS
Message (2)	AS returns ticket-granting ticket
K_c	Encryption is based on user's password, enabling AS and client to verify password, and protecting contents of message (2)
$K_{c,tgs}$	Copy of session key accessible to client created by AS to permit secure exchange between client and TGS without requiring them to share a permanent key
ID_{tgs}	Confirms that this ticket is for the TGS
TS_2	Informs client of time this ticket was issued
$Lifetime_2$	Informs client of the lifetime of this ticket
$Ticket_{tgs}$	Ticket to be used by client to access TGS

(a) Authentication Service Exchange

Message (3)	Client requests service-granting ticket
ID_V	Tells TGS that user requests access to server V
$Ticket_{tgs}$	Assures TGS that this user has been authenticated by AS
$Authenticator_c$	Generated by client to validate ticket
Message (4)	TGS returns service-granting ticket
$K_{c,tgs}$	Key shared only by C and TGS protects contents of message (4)
$K_{c,v}$	Copy of session key accessible to client created by TGS to permit secure exchange between client and server without requiring them to share a permanent key
ID_V	Confirms that this ticket is for server V
TS_4	Informs client of time this ticket was issued
$Ticket_V$	Ticket to be used by client to access server V
$Ticket_{tgs}$	Reusable so that user does not have to reenter password
K_{tgs}	Ticket is encrypted with key known only to AS and TGS, to prevent tampering
$K_{c,tgs}$	Copy of session key accessible to TGS used to decrypt authenticator, thereby authenticating ticket
ID_C	Indicates the rightful owner of this ticket
AD_C	Prevents use of ticket from workstation other than one that initially requested the ticket
ID_{tgs}	Assures server that it has decrypted ticket properly
TS_2	Informs TGS of time this ticket was issued
$Lifetime_2$	Prevents replay after ticket has expired
$Authenticator_c$	Assures TGS that the ticket presenter is the same as the client for whom the ticket was issued has very short lifetime to prevent replay
$K_{c,tgs}$	Authenticator is encrypted with key known only to client and TGS, to prevent tamperig

ID_C	Must match ID in ticket to authenticate ticket
AD_C	Must match address in ticket to authenticate ticket
TS_3	Informs TGS of time this authenticator was generated

(b) Ticket-Granting Service Exchange

Message (5)	Client requests service
$Ticket_V$	Assures server that this user has been authenticated by AS
$Authenticator_c$	Generated by client to validate ticket
Message (6)	Optional authentication of server to client
$K_{c,v}$	Assures C that this message is from V
$TS_5 + 1$	Assures C that this is not a replay of an old reply
$Ticket_v$	Reusable so that client does not need to request a new ticket from TGS for each access to the same server
K_v	Ticket is encrypted with key known only to TGS and server, to prevent tampering
$K_{c,v}$	Copy of session key accessible to client; used to decrypt authenticator, thereby authenticating ticket
ID_C	Indicates the rightful owner of this ticket
AD_C	Prevents use of ticket from workstation other than one that initially requested the ticket
ID_V	Assures server that it has decrypted ticket properly
TS_4	Informs server of time this ticket was issued
$Lifetime_4$	Prevents replay after ticket has expired
$Authenticator_c$	Assures server that the ticket presenter is the same as the client for whom the ticket was issued; has very short lifetime to prevent replay
$K_{c,v}$	Authenticator is encrypted with key known only to client and server, to prevent tampering
ID_C	Must match ID in ticket to authenticate ticket
AD_c	Must match address in ticket to authenticate ticket
TS_5	Informs server of time this authenticator was generated

(c) Client/Server Authentication Exchange

Kerberos provides a mechanism for supporting such interrealm authentication. For two realms to support interrealm authentication, a third requirement is added:

3. The Kerberos server in each interoperating realm shares a secret key with the server in the other realm. The two Kerberos servers are registered with each other.

The scheme requires that the Kerberos server in one realm trust the Kerberos server in the other realm to authenticate its users. Furthermore, the participating servers in the second realm must also be willing to trust the Kerberos server in the first realm.

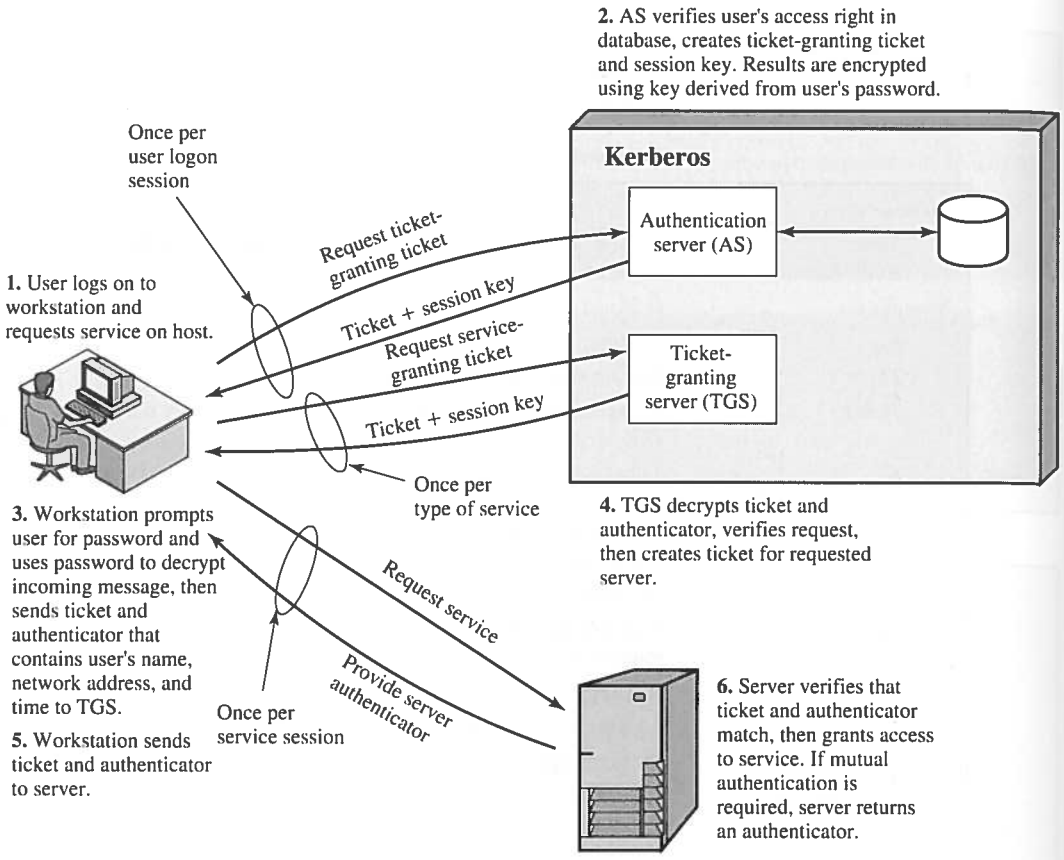


Figure 14.1 Overview of Kerberos

With these ground rules in place, we can describe the mechanism as follows (Figure 14.2): A user wishing service on a server in another realm needs a ticket for that server. The user's client follows the usual procedures to gain access to the local TGS and then requests a ticket-granting ticket for a remote TGS (TGS in another realm). The client can then apply to the remote TGS for a service-granting ticket for the desired server in the realm of the remote TGS.

The details of the exchanges illustrated in Figure 14.2 are as follows (compare Table 14.1):

(1) C → AS: $ID_c \| ID_{tgs} \| TS_1$

(2) AS → C: $E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$

(3) C → TGS: $ID_{tgsrem} \| Ticket_{tgs} \| Authenticator_c$

(4) TGS → C: $E(K_{c,tgs}, [K_{c,tgsrem} \| ID_{tgsrem} \| TS_4 \| Ticket_{tgsrem}])$

(5) C → TGS_{rem}: $ID_{vrem} \| Ticket_{tgsrem} \| Authenticator_c$

(6) TGS_{rem} → C: $E(K_{c,tgsrem}, [K_{c,vrem} \| ID_{vrem} \| TS_6 \| Ticket_{vrem}])$

(7) C → V_{rem}: $Ticket_{vrem} \| Authenticator_c$

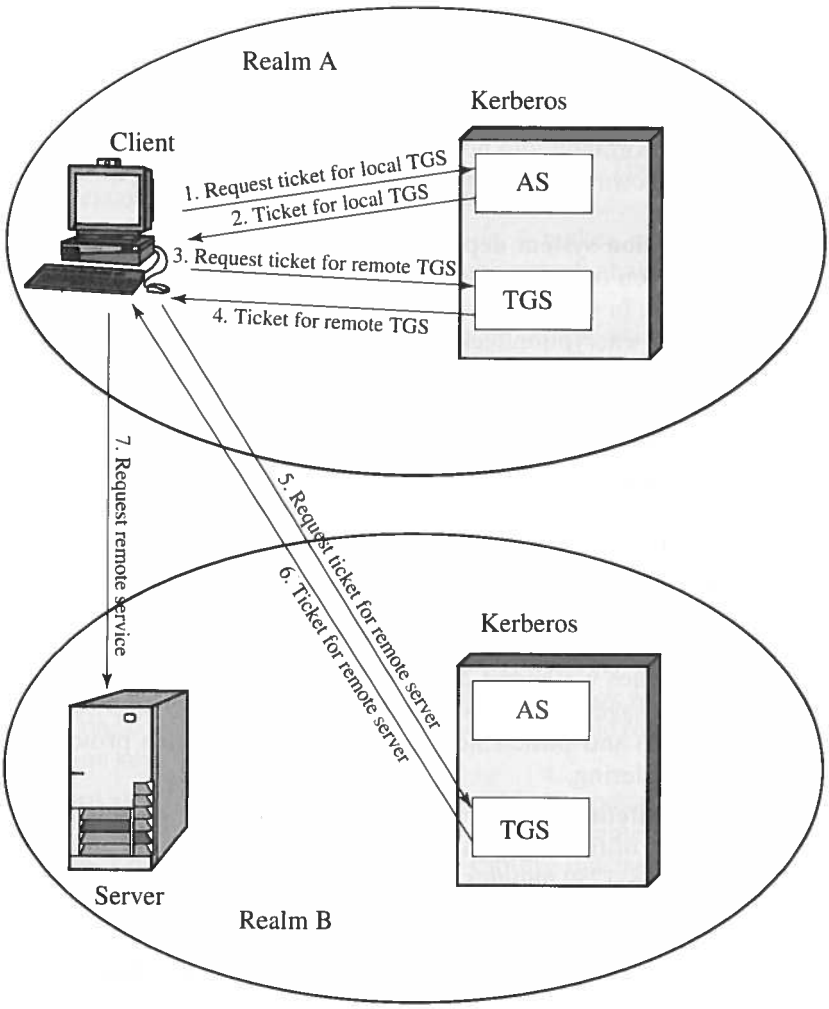


Figure 14.2 Request for Service in Another Realm

The ticket presented to the remote server (V_{rem}) indicates the realm in which the user was originally authenticated. The server chooses whether to honor the remote request.

One problem presented by the foregoing approach is that it does not scale well to many realms. If there are N realms, then there must be $N(N-1)/2$ secure key exchanges so that each Kerberos realm can interoperate with all other Kerberos realms.

Kerberos Version 5

Kerberos Version 5 is specified in RFC 1510 and provides a number of improvements over version 4 [KOHL94]. To begin, we provide an overview of the changes from version 4 to version 5 and then look at the version 5 protocol.

Differences between Versions 4 and 5 Version 5 is intended to address the limitations of version 4 in two areas: environmental shortcomings and technical deficiencies. Let us briefly summarize the improvements in each area.[6]

Kerberos Version 4 was developed for use within the Project Athena environment and, accordingly, did not fully address the need to be of general purpose. This led to the following **environmental shortcomings:**

1. **Encryption system dependence:** Version 4 requires the use of DES. Export restriction on DES as well as doubts about the strength of DES were thus of concern. In version 5, ciphertext is tagged with an encryption type identifier so that any encryption technique may be used. Encryption keys are tagged with a type and a length, allowing the same key to be used in different algorithms and allowing the specification of different variations on a given algorithm.
2. **Internet protocol dependence:** Version 4 requires the use of Internet Protocol (IP) addresses. Other address types, such as the ISO network address, are not accommodated. Version 5 network addresses are tagged with type and length, allowing any network address type to be used.
3. **Message byte ordering:** In version 4, the sender of a message employs a byte ordering of its own choosing and tags the message to indicate least significant byte in lowest address or most significant byte in lowest address. This techniques works but does not follow established conventions. In version 5, all message structures are defined using Abstract Syntax Notation One (ASN.1) and Basic Encoding Rules (BER), which provide an unambiguous byte ordering.
4. **Ticket lifetime:** Lifetime values in version 4 are encoded in an 8-bit quantity in units of five minutes. Thus, the maximum lifetime that can be expressed is $2^8 \times 5 = 1280$ minutes, or a little over 21 hours. This may be inadequate for some applications (e.g., a long-running simulation that requires valid Kerberos credentials throughout execution). In version 5, tickets include an explicit start time and end time, allowing tickets with arbitrary lifetimes.
5. **Authentication forwarding:** Version 4 does not allow credentials issued to one client to be forwarded to some other host and used by some other client. This capability would enable a client to access a server and have that server access another server on behalf of the client. For example, a client issues a request to a print server that then accesses the client's file from a file server, using the client's credentials for access. Version 5 provides this capability.
6. **Interrealm authentication:** In version 4, interoperability among N realms requires on the order of N^2 Kerberos-to-Kerberos relationships, as described earlier. Version 5 supports a method that requires fewer relationships, as described shortly.

Apart from these environmental limitations, there are **technical deficiencies** in the version 4 protocol itself. Most of these deficiencies were documented

[6]The following discussion follows the presentation in [KOHL94].

in [BELL90], and version 5 attempts to address these. The deficiencies are the following:

1. **Double encryption:** Note in Table 14.1 [messages (2) and (4)] that tickets provided to clients are encrypted twice, once with the secret key of the target server and then again with a secret key known to the client. The second encryption is not necessary and is computationally wasteful.
2. **PCBC encryption:** Encryption in version 4 makes use of a nonstandard mode of DES known as propagating cipher block chaining (PCBC).[7] It has been demonstrated that this mode is vulnerable to an attack involving the interchange of ciphertext blocks [KOHL89]. PCBC was intended to provide an integrity check as part of the encryption operation. Version 5 provides explicit integrity mechanisms, allowing the standard CBC mode to be used for encryption. In particular, a checksum or hash code is attached to the message prior to encryption using CBC.
3. **Session keys:** Each ticket includes a session key that is used by the client to encrypt the authenticator sent to the service associated with that ticket. In addition, the session key may subsequently be used by the client and the server to protect messages passed during that session. However, because the same ticket may be used repeatedly to gain service from a particular server, there is the risk that an opponent will replay messages from an old session to the client or the server. In version 5, it is possible for a client and server to negotiate a subsession key, which is to be used only for that one connection. A new access by the client would result in the use of a new subsession key.
4. **Password attacks:** Both versions are vulnerable to a password attack. The message from the AS to the client includes material encrypted with a key based on the client's password.[8] An opponent can capture this message and attempt to decrypt it by trying various passwords. If the result of a test decryption is of the proper form, then the opponent has discovered the client's password and may subsequently use it to gain authentication credentials from Kerberos. This is the same type of password attack described in Chapter 18, with the same kinds of countermeasures being applicable. Version 5 does provide a mechanism known as preauthentication, which should make password attacks more difficult, but it does not prevent them.

The Version 5 Authentication Dialogue Table 14.3 summarizes the basic version 5 dialogue. This is best explained by comparison with version 4 (Table 14.1).

First, consider the **authentication service exchange**. Message (1) is a client request for a ticket-granting ticket. As before, it includes the ID of the user and the TGS. The following new elements are added:

- **Realm:** Indicates realm of user
- **Options:** Used to request that certain flags be set in the returned ticket

[7]This is described in Appendix 14A.

[8]Appendix 14A describes the mapping of passwords to encryption keys.

Chapter 13

Security

http://link.springer.com/chapter/10.1007/978-3-642-84696-0_13

Källa: D. Powell (ed.): "Delta-4: A Generic Architecture for Dependable Distributed Computing". Springer-Verlag. ISBN 3-540-54985-4.

Since Delta-4 is open and compatible with current operating systems and applications, the basic security is provided by the local operating system, e.g., UNIX. However, in some distributed contexts, this basic security is not sufficient, e.g., because some individual sites or operators cannot be trusted, and/or because the information processed by the distributed system is very sensitive. This chapter presents some solutions to provide secure distributed application services, such as a file archiving service, and to manage the security of a Delta-4 system. The solutions are such that an intrusion into a part of the distributed system will not endanger its security, i.e., they are *intrusion-tolerant* solutions [Deswarte et al. 1991].

Computing system dependability has been classically considered as being the ability of a computing system to deal with faults that are, implicitly, seen as being accidental. Security, i.e., the ability of a computing system to deal with intentional attacks, was historically a somewhat later worry than "classical" dependability. Such intentional attacks are becoming more and more numerous and subsequent losses are ever-increasing, causing growing problems in computing centres, banking systems, etc. This is particularly true in large network environments. The numerous cases of computer fraud that are related in the newspapers underline the fact that although quite many techniques for achieving security have been developed, the problems of computing system security have not really been satisfactorily solved.

Until now, security and dependability have had their own meetings, terminologies, standardization committees... and very few research groups work on both aspects. The quite evident relationship that exists between security and dependability has not been clearly formalized. Nevertheless, recent attempts at relating these two domains tend to consider dependability as a general concept that embraces security, reliability and safety as different attributes. This viewpoint is not only conceptually interesting but should lead to the application of concepts originally intended for accidental faults to intentional faults, and vice versa. This approach implies the designing and implementation of secure and reliable systems using homogeneous solutions rather than viewing security and reliability as two opposing requirements.

Dependability can be more rigourously defined as that property of a computing system that allows reliance to be justifiably placed on the service it delivers (cf. chapter 4). This general but precise definition does not need any specification of the impairments or faults that may lead to a system failure. These faults may be either accidental ("classical" dependability) or intentional (security). This is why dependability has to be considered as a general concept, involving all kinds of faults. Reliability and safety may also require security in the sense that an intentional action may alter the mechanisms intended, for example, to ensure system reliability. It is also true that security requires reliability because an accidental fault in a security mechanism may allow intrusions that are usually not possible. In fact, it is not possible to integrate one aspect into the other as is sometimes attempted, but on the contrary these concepts should be

considered at the same level with tight relations linking one with each other. Figure 1 shows a conceptual organization of dependability as a generic concept (cf. chapter 4).

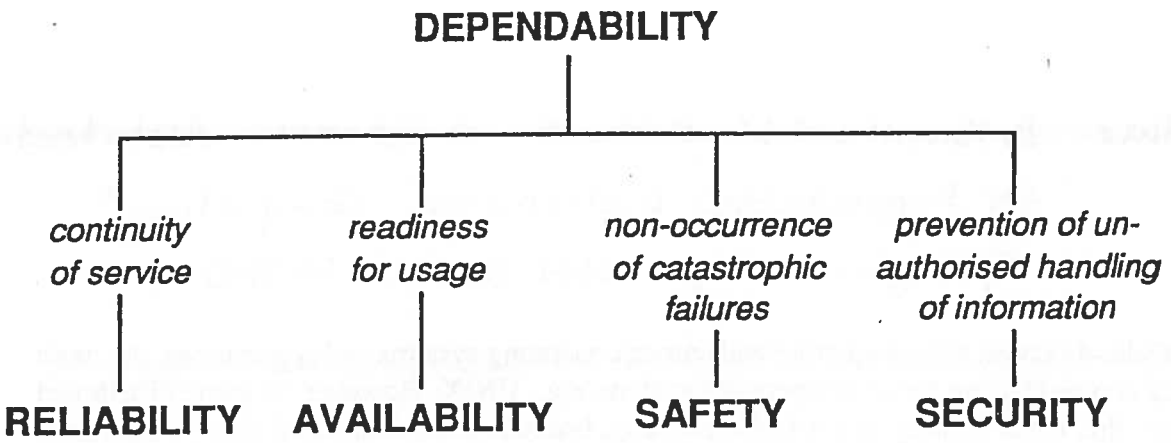


Fig. 1 - Perceptive Attributes of Dependability

Impairments to security come primarily from intentional faults, which may be either design faults or interaction faults. Intentional design faults (usually called *malicious logic*) are introduced into the system so that it delivers a service that is not as described in the specifications in order to mislead the user about the operations performed by the system. These illicit functions are often carried out without being detected by the user who may continue to work with a corrupted system for a long time before discovering the faults. Interaction faults are performed while the system is already in use, taking advantage of the weaknesses in the security mechanisms or policy. They are usually called intrusions. Both classes of intentional faults may cause a failure of the system. Their effects depend on the efficiency of the mechanisms designed to ensure security.

13.1. Principles of Intrusion-Tolerance

13.1.1. General Concept

An *intrusion* can be defined as an *intentional operational external fault* (cf. chapter 4). Different types of intrusions can be classified according to who makes the intrusion:

- It can be somebody outside the system who tries to access it. This is the most well known kind of intruder, but not the most important. In this case the intruder has to *bypass* physical, procedural and logical protections.
- The second kind of intruder is a user of the system who tries to access information or services for which he has no right of access. The intruder tries to *extend his privileges*. This is the most common intrusion. The intruder has "only" to bypass the logical protection.
- The third — and the most dangerous — type of intruder is a security administrator who *uses his rights* to perform illegitimate actions. In this later case, the administrator has enough access-rights to do these actions but, according to the security policy, is not supposed to do them.

Intrusions can be treated with the same means as for other faults, i.e., by means of fault-avoidance and fault-tolerance. Intrusion avoidance techniques are the most common in secure systems and in particular are the basis of the notion of *trust*. For instance, when the "*Orange*

Book" [DoD 5200.28] states that a reference monitor must be *tamperproof*, it means that you must prevent intrusions into it. In such systems, the protection mechanisms must prevent unauthorized actions. If an intruder succeeds in bypassing these protections, the security of the system is no longer ensured. If a security administrator decides to carry out illegal actions, there is no logical protection to prevent him from doing so. He could only be detected by using an intrusion-detection model [Denning 1986] which is able to detect intrusions by monitoring audit records for abnormal patterns of system usage.

The principles of intrusion tolerance are different [Deswarte et al. 1991, Fraga 1985, Fray et al. 1986]. The system tolerates a bounded number of misuses. If one or more intruders bypass the protection mechanisms and if the number of misuses is less than a given threshold, the security properties of the system (confidentiality, integrity and availability) are still ensured.

Three types of intrusion tolerance can be formulated:

- for *confidentiality*: read access to a subset of confidential data gives no information about the data,
- for *integrity*: the change of a subset of data does not change the data perceived by legitimate users,
- for *availability*: the change or deletion of a subset of data or of a server does not produce a denial of service to legitimate users.

For each property, a tolerance threshold is defined. If the reading, modification or destruction is done on a part D' of data/server D such that |D'| (size of D') is less than the threshold, the properties are always verified (figure 2).

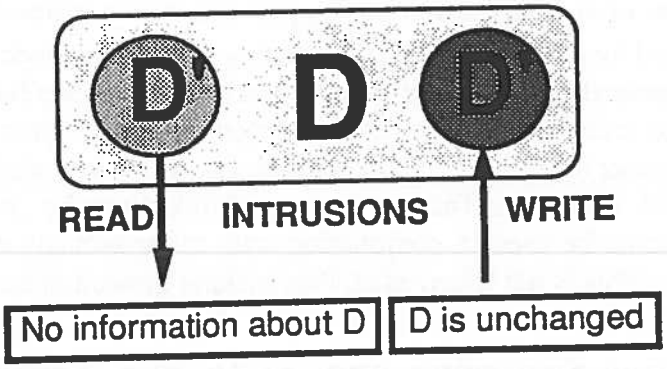


Fig. 2 - Intrusion Tolerance

13.1.2. Fragmentation-and-Scattering

Fragmentation-and-scattering is an intrusion-tolerance method that can be compared with redundancy in a classical fault-tolerance context. Redundancy is used to ensure that the occurrence of a fault in one copy will be of no consequence because this fault will not appear in the other copies. Fragmentation consists of defining different fragments of the data to ensure that once isolated, every fragment is of no interest due to the lack of sufficient information. Scattering refers to the way by which each fragment is isolated from the others. Once this operation is carried out, an intrusion into a part of the system has no consequence due to the lack of significant information involved in the fragments to which the intruder has access. The number of intrusions that can be tolerated without delivery of significant information is dependent on the way the operations are defined. This number is a parameter that can be chosen to determine the tradeoff between security and performance.

Fragmentation may be carried out in many ways, with or without a secret parameter (fragmentation key). It can be performed with the help of other techniques such as cryptography to increase the level of security. It may also be performed with various granularities of the data such as bit level or byte level. The granularity of fragmentation directly influences the security and the performance of the method.

Scattering may be carried out in different ways, depending on which intrusions one wants to tolerate. *Geographical scattering* may be used for both communications and data storage. One example of using different communication links for sending different fragments has been proposed for meshed networks [Koga et al. 1982]. As another example, Rutledge discusses the use of different telephone lines to transmit the different fragments to the receiver [Rutledge 1987]. Geographical scattering for data storage may take place in a network environment where several archive sites are available. This solution will be described later.

Another form of scattering is *temporal scattering*: fragments are sent over a communication channel at different and unpredictable moments. In this case, fragments coming from different sources must be the same length in order to hide their origin.

Frequential or spread-spectrum scattering consists of the use of different frequencies to transmit the fragments in radio-communication applications or in broad-band local area networks.

Yet another scattering technique is *privilege scattering*, which is a way to implement the separation of duties proposed by the Clark-Wilson policy [Clark and Wilson 1987]. With this technique, certain sensitive operations need the cooperation of several users to be performed.

The main idea that governs all forms of scattering is the difficulty that an intruder would have in retrieving all the information. A successful intruder would have to carry out several intrusions in different places or at different times or frequencies in a consistent manner.

Data cannot be protected by fragmentation-and-scattering all the time since it must exist as a whole when it is to be processed. It is possible to apply intrusion-tolerance for communication security or for data storage security but at the time of processing, data must be reassembled. Once data is reassembled, other forms of protection are needed. There is a similar problem with classical error-masking in which the final vote procedures must be trusted. Although fragmentation-scattering must be used in conjunction with other security means, it greatly restrains the locations where data is not fragmented, thus making protection easier.

13.2. Fragmentation-Scattering applied to File Archiving

13.2.1. Overall Framework

In Delta-4, a *secure data archiving system* has been designed to deal with different types of intrusions that could occur either in the storage devices (archive sites) or in the communication channel. Data security is provided by intrusion-tolerance and more precisely by geographical fragmentation-scattering (see figure 3).

The basis of the fragmentation and scattering technique is to cut every sensitive file into several fragments in such a way that one or more fragments (where the number of fragments is less than the total number of fragments) are not sufficient to reconstitute the file. These fragments are then stored in geographically distributed archive sites. An intruder accessing some sites cannot obtain all the fragments of a given file unless he has almost overall control of the complete distributed system. On the other hand, to ensure availability, several copies of each fragment are stored on different archive sites. This service thus meets all three security requirements: confidentiality, integrity and availability. Confidentiality and integrity are directly provided by fragmentation-scattering while availability is obtained by the replication of fragmented data. This section, after a presentation of the environment needed for implementing

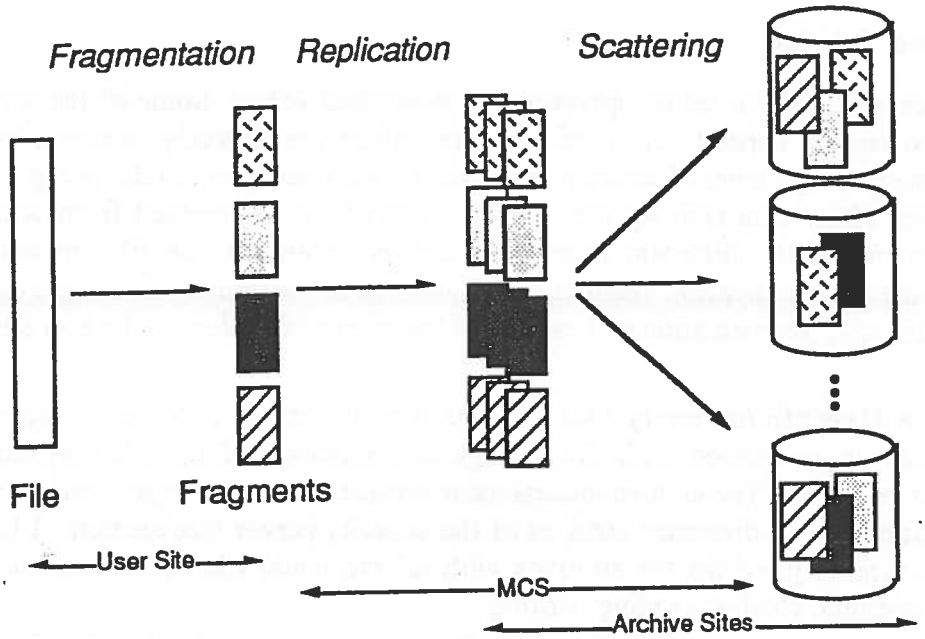


Fig. 3 - General Principle of Fragmentation-Scattering Applied to File Archiving

fragmentation-scattering, gives the basis of the different operations that will take place to secure sensitive data.

To describe the file archiving service, only two kinds of sites are considered, interconnected by the multipoint communication system (figure 4):

- *user sites*, which are personal workstations that constitute the physical domains of their users,
- *archive sites*, designed for long-term data storage, while the user of the data is not logged-in,

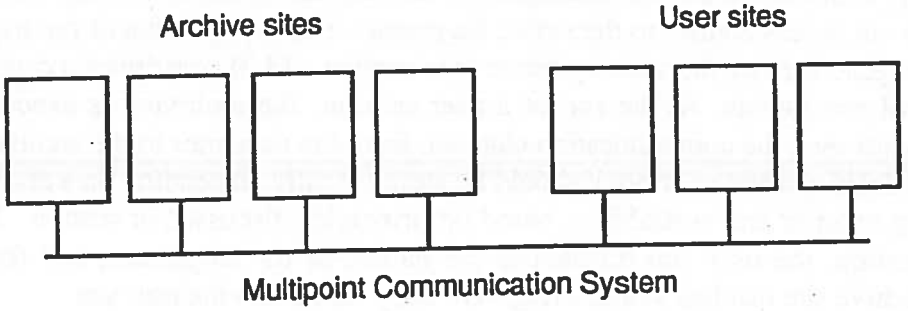


Fig. 4 - System Environment

A user has full authority over the objects stored in the machine he/she is working on. During the user session (log-in time), nobody else can work on the same machine. Key management and access right management will be addressed in the next section (see section §13.3).

13.2.2. Archive Service

An archive is processed under a set of operations as described below. Some of the archive management operations are carried out in the user site, others are remotely executed by the archive sites. To ensure a high level of security, complete files are never available, except in the connected-user site. Thus, data sent by the user is always in a fragmented form with no possibility of recognising the different fragments derived from a given file (otherwise, eavesdropping of the communication channel could annihilate the added advantages of the technique). Consequently, fragmentation and naming of fragments take place at the user site.

13.2.2.1. User Site Operations. Every user site has a well-defined archive-management service. This service is concerned with the storage and retrieval of the files within the distributed file archive system. The archive management-service uses the storage services of the different archive sites and the directory services of the security server (see section 13.3) to build the file structure. It provides the archives with all the usual file operations such as creation, deletion, opening, closing, reading, writing...

As stated earlier, two basic operations related to file security are provided in the user site: fragmentation and naming of the fragments. The fragmentation operation uses a fragmentation key that is distributed by the security server. This operation is based on fast and simple algorithms that give flexible access to any file whilst ensuring a high level of security due to the scattering of information. The names given to the different fragments are generated by cryptographic methods using the fragmentation key, such that no information can be derived from these names. The naming is carried out in such a way that fragments have a unique identifier, derived from the fragmentation key, the name of the file and some other parameters. During the read operation, the original file is reconstituted with the same key that was used for fragmentation.

13.2.2.2. Archive Site Operations. A storage service is provided by the archive sites. The operations of these sites correspond to simple space allocations on the physical storage devices and data transfers between the storage device and the network. These storage operations are only available to the file-management service embedded in the user sites. The archive sites carry out access control to the stored fragments. The presentation of the fragment names and a ticket generated by the security server (see section 13.3) constitutes a capability-type access control mechanism. At the end of a user session, file archiving is executed by sending the fragments over the communication channel, from the user sites to the archive sites. Every archive site decides whether or not it should be stored locally, depending on a distributed algorithm ensuring security and availability, based on principles discussed in section 13.2.5. For the read operation, the user site broadcasts the names of the fragments, and for each fragment, every archive site that had stored a fragment copy sends it to the user site.

Each archive site acts as a file server that would only store fixed length files, with a "flat directory" structure. The operations managed by these archive sites are fragment reading, fragment writing, and fragment deletion. Only fragment names are visible to the archive site; thus, it cannot determine where a fragment comes from or to which file a fragment belongs.

13.2.3. File Access Session

The aim of this section is to clarify the different actions needed to access a file either for writing or reading from a user site, where the user is already logged-in, and thus recognised as authorized.

13.2.3.1. Write Operation. The write operation is achieved with the following steps:

W1: The user request for file opening with write-access is transmitted to the security server, which checks the access rights according to the archive descriptor.

W2: If this user has write-access rights to this file, the fragmentation key is transmitted by the security server to the user site.

W3: The file is fragmented and the fragments are named, using the fragmentation key in the user site.

W4: The fragments are broadcasted from the user site to the archive classes in a random order (temporal scattering). An intruder who is eavesdropping on the communication channel or who controls an archive site does not know the order in which the fragments have been sent. The intruder would have to run a large number of trials to reconstitute a file page (see §13.2.4.2).

W5: Each archive site that receives a fragment decides whether it should be stored locally depending on a specific, pseudo-random, distributed algorithm that takes into account the relative available storage space on each archive site (the need to take into account the available space at each site is necessary to maintain a good balance among the different archive sites).

W6: The user-site copy of the file must be deleted.

13.2.3.2. Read Operation. The read operation entails a similar sequence:

R1: The user request for file opening with read-access is transmitted to the security server, which checks the access rights according to the archive descriptor.

R2: If this user has read-access rights to this file, the fragmentation key is transmitted by the security server to the user site thus allowing re-computation of the names of the fragments.

R3: Fragment requests (transmitted in a random order) are broadcast to the archive sites that send the fragments back to the user site.

R4: Once all the fragments have been received, the file is reassembled at the user site.

The following sections will discuss more accurately the way in which both fragmentation and scattering have been implemented.

13.2.4. Fragmentation Principles

A general approach is proposed for the fragmentation operation. This operation consists of defining all the fragments of a file. The file may be of any length and of any type. The fragmentation operation must ensure that, once the fragments are isolated, no information can be obtained from them. Moreover, it must be impossible to guess their origin, this requirement implying that all the fragments from a given file must be of fixed length and that their names do not allow any information to be deduced. Finally, an important requirement concerns data integrity: modification of a fragment must be easily detected at the time of reuse.

13.2.4.1. Partitioning. A method has to be defined that is suitable for producing identical-length fragments from files with very different lengths. The solution proposed is to first cut every file into pages of fixed size (partitioning). Every page may then be fragmented into an identical number of fragments. The files are padded out to reach a size equal to a multiple of a page size (figure 5). All the fragments so obtained will have the same length, which may be equal to that of a packet sent on the communication channel, or a quantum in the mass storage,

for example. These choices may also improve the speed of access to information. Another advantage is that one does not need to get the whole file: pages can be retrieved independently. So, a user does not need to reassemble a whole file if he only needs a single page. A signature, built as a cryptographic checksum, is added to each page; this signature is checked by the read operation to verify the integrity of the page.

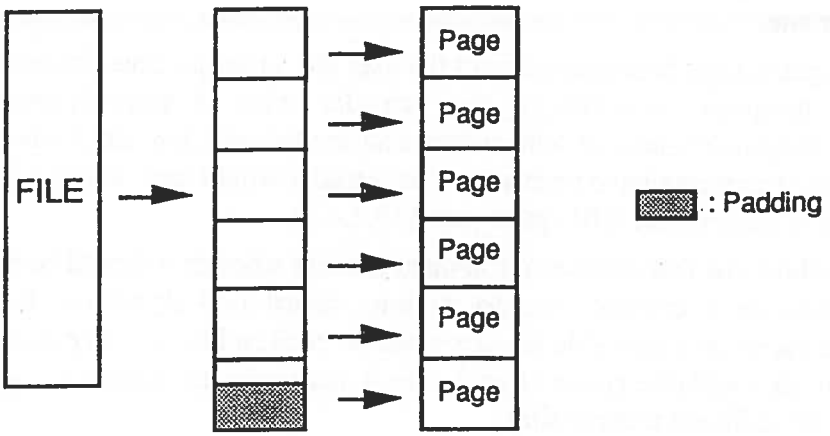


Fig. 5 - A File Partitioned into Pages

The mean overhead due to padding information is half a page, and is of course a large overhead for small files. The shorter a page is, the smaller the overhead is, but the longer the management time, mainly due to the fragment storage time.

13.2.4.2. Fragmentation. A very simple fragmentation scheme is to scatter the data symbols among the fragments, without any real ciphering. However, in this case, an intruder having several fragments could possibly derive the whole page by guessing the missing data. At the very least, he could obtain a good idea of the symbol vocabulary used in the non-fragmented file. Cryptographic methods must thus be used in conjunction with the fragmentation. The question that one may then ask is: why use fragmentation since cryptographic techniques must still be employed? There are two good reasons:

- first, the geographical scattering of fragments makes theft of individual storage media of no avail to the intruder — even if he possesses the cipher key,
- second, the added security of scattering means that the ciphers employed can be much simpler and thus faster than conventional ones.

One may imagine different schemes to realize ciphering and fragmentation together. Our choice consists of the fragmentation of ciphered pages (figure 6). Each page is first ciphered and the fragments are obtained from this ciphered page. The fragmentation itself uses a fixed scheme wherein each successive quantum of data is put into one of the fragments according to a fixed distribution that does not depend on the key. This operation leads to a fine-grain scattering of the data among all the fragments.

13.2.4.3. Choice of an Appropriate Cipher. To make it as difficult as possible for an intruder to decipher an individual fragment or sub-set of fragments, it is preferable to choose a cipher scheme that makes the ciphertext of each data-quantum, and thus each fragment, dependent on the others. This may be achieved by using a stream cipher. With such a cipher, the key used to cipher a quantum of plaintext changes for each quantum: the preceding ciphered

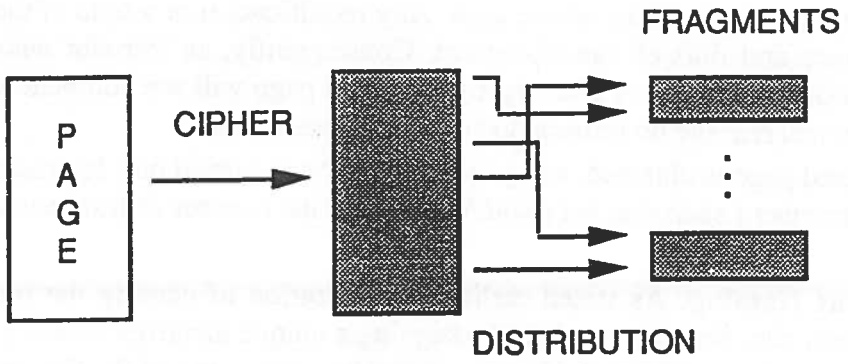


Fig. 6 - Ciphering/Fragmentation

text is necessary to determine the following plaintext. This introduces secrecy if any quantum of text is missing.

For this application, the stream cipher is based on a random number generator (RNG) chosen to make the ciphering very fast. The keystream is generated by two 32 bit random number generators (figure 7). Before each page ciphering, both RNGs are initialized with the two halves of the 64 bit key K (used as the seeds of the RNGs), then step by step the first one is reinitialized by the ciphertext quanta. The two results are exclusive-ored to generate the keystream K_i. Then, this keystream is exclusive-ored with 32 bit quanta of the plaintext. The first RNG is only used as a one-way-function applied to the previous ciphered quantum. The second RNG is equivalent to a large sized Linear Feedback Shift Register. Thus, the keystream K_i depends on the properties of the RNGs and on the previous ciphered text.

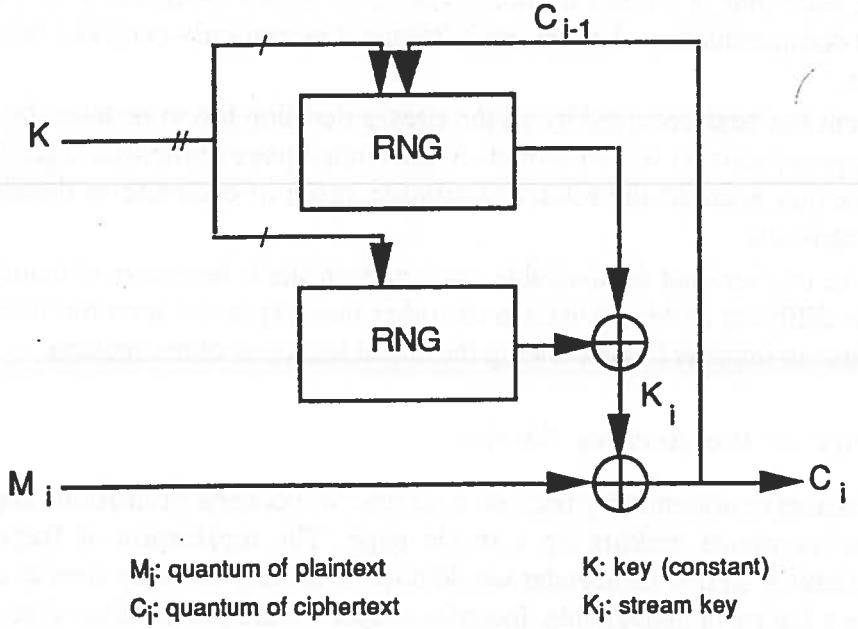


Fig. 7 - Random Number Generator Stream Cipher

This method enables the integrity of the fragments to be checked, because if one quantum of the ciphertext is modified, all the following deciphered text is changed. However, if one of the last plaintext quanta is modified, only the last ciphertext quanta will be changed. It is more interesting that the modification of any quantum changes all the ciphertext. To do so, the beginning of the plaintext is constituted by a signature issued by a hash (cryptographic)

function applied to the content of the whole page. Any modification of a byte of the plaintext changes the signature and thus all the ciphertext. Consequently, an intruder who observes successive releases of the fragments of a slightly modified page will see completely different fragments, and then will retrieve no information on the fragment order.

Once the ciphered page is obtained, a regular distribution is carried out: fragment number j receives the bytes number i such that $j=i \ (mod\ N)$, N being the number of fragments.

13.2.4.4. Fragment Naming. As stated earlier, the operation of naming the fragments is carried out on the user site. Naming consists of assigning a unique identifier to every fragment; this unique identifier is derived from the fragmentation key, the name of the file, the index of the page and the index of the fragment. The naming algorithm is based on one-way cryptographic functions such that no information concerning one fragment can be derived from its name.

13.2.5. Scattering Principles

The fragment write, read or delete requests that are transmitted by the user site to the fragment server sites, are sent in a random order for each page. This means that if an intruder is eavesdropping on the network or controls a fragment server site, he can receive all the fragments of a given page, but he is not able to ascertain the order in which he has to select the fragments before attempting a cryptanalysis of the page. For instance, if a page is cut in 16 fragments, an intruder may need to attempt about $16!/2 \approx 10^{13}$ trials before finding the correct arrangement. Thus, the confidentiality depends more on this random order than on the efficiency of the cipher.

Once the file is fragmented in the user site, the fragments are broadcasted to the archive sites to be stored, each one in a fixed number of sites to ensure availability by redundancy. Using a broadcast communication channel, each fragment is sent only once and is received by all the archive sites.

Once a fragment has been received by all the sites, a decision has to be taken by these sites to ensure that R copies (exactly) will be stored. A distributed pseudo-random algorithm is then required that takes into account the relative available space at each site to decide the final locations of the fragments.

The need to take into account the available space at each site is necessary to maintain a good balance among the different archive sites. On the other hand, (pseudo-)random behaviour can be applied to prevent an intruder from knowing the actual locations of the replicas.

13.2.6. Security of the Archive Service

Scattering increases confidentiality because a number of concerted intrusions are necessary to retrieve all the fragments making up a single page. The replication of fragments also increases data availability in that an intruder would have to destroy as many sites as the number of replicas to make a fragment unavailable. Integrity properties are provided because an intruder would have to modify all the replicas and so must carry out several intrusions. Moreover, it is very unlikely that an intruder could modify even one byte of a fragment without modifying the cryptographic signature of the page.

Other techniques can be used to implement a secure file archiving server. The first one consists of ciphering the file on the user site, and storing several copies of the ciphered file on different archive servers. In that case, confidentiality relies on the efficiency of the cipher algorithm. An intruder who is eavesdropping on the network or who gets a copy of the ciphered file (e.g., a file server back-up tape) can take all the time and all the computer power

he wants to cryptanalyse the file; with one intrusion, he gets all the information he needs. With the fragmentation-scattering technique, the intruder who gets all the fragments of a page, would have to try 10^{13} arrangements before cryptanalysing the ciphered page. That means that the cipher can be $\approx 10^{13}$ times less strong, and can be much faster. For integrity, the two techniques are comparable. For availability, the two techniques are equivalent for accidental server failures. However, the fragmentation-replication-scattering technique is less robust against simultaneous destruction of storage servers: if an intruder is able to destroy R (out of N) fragment server sites at the same time, he will make many more files unavailable than if he destroys R ciphered file servers. The overhead of the two techniques is equivalent for the communications and the storage space, but fragmentation-scattering can be made much less CPU-consuming than the ciphered file approach.

Another technique has been proposed by Rabin for fault-tolerant file servers: the "Information Dispersal" approach [Rabin 1989]. This technique consists of coding the file with a special error-correcting code and then splitting the coded file in n pieces, such that m out of n of these pieces are sufficient to rebuild the complete file. The n file pieces are stored by different storage servers. The coded file is longer than the original file, but the redundancy is much smaller than replication with $(n - m + 1)$ copies, for nearly the same availability and integrity. This technique is much more CPU consuming than the fragmentation and it does not ensure file confidentiality: to prevent information disclosure to eavesdroppers and storage server intruders, the file has to be ciphered before coding. To summarise, the storage and communication overhead is much less important in the Rabin's technique, but it consumes much more CPU time.