CHAPTER

4

Access Control

Going all the way back to early time-sharing systems, we systems people regarded the users, and any code they wrote, as the mortal enemies of us and each other. We were like the police force in a violent slum.

—ROGER NEEDHAM

Microsoft could have incorporated effective security measures as standard, but good sense prevailed. Security systems have a nasty habit of backfiring, and there is no doubt they would cause enormous problems.

—RICK MAYBURY

4.1 Introduction

Access control is the traditional center of gravity of computer security. It is where security engineering meets computer science. Its function is to control which principals (persons, processes, machines, . . .) have access to which resources in the system—which files they can read, which programs they can execute, how they share data with other principals, and so on.

NOTE

This chapter necessarily assumes more computer science background than previous chapters, but I try to keep it to a minimum.

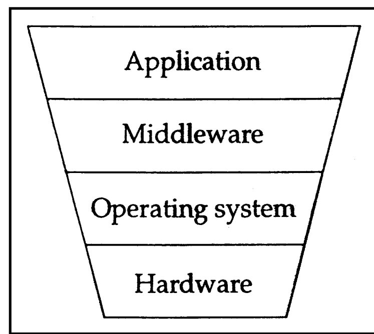

Figure 4.1 Access controls at different levels in a system.

Access control works at a number of levels, as shown in Figure 4.1, and described in the following:

1. *The access control mechanisms, which the user sees at the application level, may express a very rich and complex security policy.* A modern online business could assign staff to one of dozens of different roles, each of which could initiate some subset of several hundred possible transactions in the system. Some of these (such as credit card transactions with customers) might require online authorization from a third party while others (such as refunds) might require dual control.
2. *The applications may be written on top of middleware, such as a database management system or bookkeeping package, which enforces a number of protection properties.* For example, bookkeeping software may ensure that a transaction that debits one ledger for a certain amount must credit another ledger for the same amount.
3. *The middleware will use facilities provided by the underlying operating system.* As this constructs resources such as files and communications ports from lower-level components, it acquires the responsibility for providing ways to control access to them.
4. *Finally, the operating system access controls will usually rely on hardware features provided by the processor or by associated memory management hardware.* These control which memory addresses a given process can access.

As we work up from the hardware through the operating system and middleware to the application layer, the controls become progressively more complex and less reliable. Most actual computer frauds involve staff accidentally discovering features of the application code that they can exploit in an opportunistic way, or just abusing features of the application that they were trusted not to. But in this chapter, we will focus on the fundamentals: access control at the hardware and operating system level. (Application-level controls aren't different in principle, but I leave detailed discussion to Part 2 of this book.)

As with the other building blocks discussed so far, access control makes sense only in the context of a protection goal, typically expressed as a security policy. This puts us at a slight disadvantage when discussing PCs running single-user operating systems such as DOS and Win95/98, which have no overt security policy: any process can modify any data. People do have implicit protection goals, though; you don't expect a

shrink-wrap program to trash your hard disk. So an explicit security policy is a good idea, especially when products support some features that appear to provide protection, such as login IDs.

I mention one protection technique—*sandboxing*—later, but leave off a substantial discussion of viruses and the like to Section 18.4. In what follows, the focus will be on protection mechanisms for systems that support the isolation of multiple processes. I discuss operating system mechanisms first, as it is their requirements that usually drive hardware protection system design.

4.2 Operating System Access Controls

The access controls provided with an operating system typically authenticate principals using some mechanism such as passwords or Kerberos, then mediate their access to files, communications ports, and other system resources.

Their effect can often be modelled by a matrix of access permissions, with columns for files and rows for users. We'll write r for permission to read, w for permission to write, x for permission to execute a program, and (–) for no access at all, as shown in Figure 4.2.

In this simplified example, Sam is the system administrator, and has universal access (except to the audit trail, which even he should only be able to read). Alice, the manager, needs to execute the operating system and application, but only through the approved interfaces—she mustn't have the ability to tamper with them. She also needs to read and write the data. Bob, the auditor, can read everything.

	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	rw	r
Alice	x	x	rw	–
Bob	rx	r	r	r

Figure 4.2 Naive access control matrix.

User	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	r	r
Alice	rx	x	–	–
Accounts program	rx	r	rw	w
Bob	rx	r	r	r

Figure 4.3 Example access control matrix for bookkeeping.

This is often enough, but in the specific case of a bookkeeping system, it's not quite what we need. We want to ensure that transactions are well formed—that each debit is matched by a credit somewhere else—so we would not want Alice to have uninhibited write access to the account file. We would also prefer that Sam didn't have this access; so that all write access to the accounting data file was via the accounting program. The

access permissions might now look like those shown in Figure 4.3. (There is still an indirect vulnerability in that Sam could overwrite the accounts program with an unauthorised one of his own devising, but we'll leave off discussing that till Chapter 9.)

Another way of expressing a policy of this type would be with *access triples* of *user, program, file.* In the general case, our concern isn't with a program as much as a *protection domain*, which is a set of processes or threads that share access to the same resources (though at any given time they might have different files open or different scheduling priorities).

Access control matrices (whether in two or three dimensions) can be used to implement protection mechanisms, as well as just model them. But they do not scale well. For instance, a bank with 50,000 staff and 300 applications would have an access control matrix of 15 million entries. This is inconveniently large. It might not only impose a performance problem but also be vulnerable to administrators' mistakes. We will usually need a more compact way of storing and managing this information. The two main ways of doing this are to use groups or roles to manage the privileges of large sets of users simultaneously, or to store the access control matrix either by columns (access control lists) or rows (capabilities, sometimes known as "tickets") or certificates [662, 804].

4.2.1 Groups and Roles

When we look at large organizations, we usually find that most staff fit into one or other of a small number of categories. A bank might have 40 or 50 such categories: teller, chief teller, branch accountant, branch manager, and so on. The remainder (such as the security manager, and chief foreign exchange dealer,...), who need to have their access rights defined individually, may amount to only a few dozen people.

So we want a small number of predefined groups, or functional roles, to which staff can be assigned. Some people use the words *group* and *role* interchangeably, and with many systems they are; but the more careful definition is that a group is a list of principals, while a role is a fixed set of access permissions that one or more principals may assume for a period of time using some defined procedure. The classic example of a role is the officer of the watch on a ship. There is exactly one watchkeeper at any one time, and there is a formal procedure whereby one officer relieves another when the watch changes. In fact, in most military applications, it's the role that matters rather than the individual.

Groups and roles can be combined. *The officers of the watch of all ships currently at sea* is a group of roles. In banking, the manager of the Cambridge branch might have his or her privileges expressed by membership of the group *manager* and assumption of the role *acting manager of Cambridge branch.* The group *manager* might express a rank in the organization (and perhaps even a salary scale) while the role *acting manager* might include an assistant accountant standing in while the manager, deputy manager, and branch accountant are all sick.

Whether we need to be careful about this distinction is a matter for the application. In a warship, we want even an able seaman to be allowed to stand watch if all the officers have been killed. In a bank, we might have a policy that "transfers over $10 million must be approved by two staff, one with the rank of manager and one with the rank of assistant accountant." In the event of sickness, the assistant accountant acting as manager would have to get the regional head office to provide the second signature on a large transfer.

Until recently, some support for groups and roles existed but was not very widely used. Developers either implemented this kind of functionality in their application code, or as custom middleware (in the 1980s, I worked on two bank projects where group support was hand-coded as extensions to the mainframe operating system). Recently, Windows 2000 (Win2K) has been launched with very extensive support for groups, while academic researchers have started working on *role-based access control* (RBAC), which I discuss further in Chapter 7. We will have to wait and see whether either of these has a major effect on application development practices.

User	Accounting Data
Sam	rw
Alice	rw
Bob	r

Figure 4.4 Access control list (ACL).

4.2.2 Access Control Lists

Another way of simplifying access rights management is to store the access control matrix a column at a time, along with the resource to which the column refers. This is called an *access control list*, or ACL. In the first of the examples, the ACL for file 3 (the account file) might look as shown in Figure 4.4.

ACLs have a number of advantages and disadvantages as a means of managing security state. These can be divided into general properties of ACLs and specific properties of particular implementations.

ACLs are widely used in environments where users manage their own file security, such as the Unix systems common in universities and science labs. Where access control policy is set centrally, they are suited to environments where protection is data-oriented; they are less suited where the user population is large and constantly changing, or where users want to be able to delegate their authority to run a particular program to another user for some set period of time. ACLs are simple to implement, but are not efficient as a means of doing security checking at runtime, as the typical operating system knows which user is running a particular program, rather than which files it has been authorized to access since it was invoked. The operating system must either check the ACL at each file access or keep track of the active access rights in some other way.

Finally, distributing the access rules into ACLs can make it tedious to find all the files to which a user has access. Revoking the access of an employee who has just been fired, for example, will usually have to be done by cancelling their password or other authentication mechanism. It may also be tedious to run systemwide checks, such as verifying that no files have been left world-writable. This could involve checking ACLs on millions of user files.

Let's look at two important examples of ACLs: their implementation in Unix and NT.

4.2.3 Unix Operating System Security

In Unix (and its popular variant Linux), files are not allowed to have arbitrary access control lists, but simply `rwx` attributes for the resource owner, the group, and the world. These attributes allow the file to be read, written, and executed. The access control list as normally displayed has a flag to show whether the file is a directory; then flags r, w, and x for owner, group, and world respectively; it then has the owner's name and the group name. A directory with all flags set would have the ACL:

`drwxrwxrwx Alice Accounts`

In the first example in Figure 4.4, the ACL of file 3 would be:

`-rw-r----- Alice Accounts`

This records that the file is not a directory; the file owner can read and write it; group members can read it but not write it; nongroup members have no access at all; the file owner is Alice; and the group is Accounts.

In Unix, the program that gets control when the machine is booted (the operating system kernel) runs as the supervisor, and has unrestricted access to the whole machine. All other programs run as users, and have their access mediated by the supervisor. Access decisions are made on the basis of the userid associated with the program. However if this is zero (`root`), then the access control decision is "yes." So root can do what it likes—access any file, become any user, or whatever. What's more, there are certain things that only root can do, such as starting certain communication processes. The root userid is typically made available to the system administrator.

This means that (with most flavors of Unix) the system administrator can do anything, so we have difficulty implementing an audit trail as a file that he cannot modify. This not only means that, in our example, Sam could tinker with the accounts, and have difficulty defending himself if he were falsely accused of tinkering, but that a hacker who managed to become the system administrator could remove all evidence of his intrusion. A common defense is to send the system log to a printer in a locked room or—if the volumes of data are too great—to another machine that is administered by somebody else.

The Berkeley distributions, including FreeBSD, go some way toward fixing the problem. Files can be set to be append-only, immutable or undeletable for user, system or both. When set by a user at a sufficient security level during the boot process, they cannot be overridden or removed later, even by root. Various military variants go to even greater trouble to allow separation of duty. However, the simplest and most common way to protect logs against root compromise is to keep them on a separate server.

Second, ACLs contain only the names of users, not of programs, so there is no straightforward way to implement access triples of (user, program, file). Instead, Unix provides an indirect method: the `suid` and `sgid` file attributes.

The owner of a program can mark it as suid. This enables it to run with the privilege of its owner rather than the privilege of the user who has invoked it; `sgid` does the same for groups. Thus, in order to achieve the functionality needed by Figure 4.3, we could create a user "`account-package`" to own file 2 (the accounts package), make the file `suid`, and place it in a directory to which Alice has access. This special user could then be given the access control attributes we want for the accounts program.

One way of looking at this is that an access control problem that is naturally modelled in three dimensions—the triples (user, program, data)—is being implemented using two-dimensional mechanisms. These mechanisms are much less intuitive than

triples, and people make many mistakes implementing them. Programmers are often lazy or facing tight deadlines; so they just make the application suid root, and it can do anything.

This practice leads to some rather shocking security holes. The responsibility for making access control decisions is moved from the operating system environment to the program, and most programmers are insufficiently experienced and careful to check everything that they should. In particular, the person invoking a suid root program controls its environment and can often manipulate this to cause protection failures.

Third, ACLs are not very good at expressing changing state. Managing stateful access rules, such as dual control, becomes difficult; one either has to do it at the application level or use suid/sgid again. Also, it's hard to track the files that a user might have open (as you typically want to do when revoking their rights on a system).

Fourth, the Unix ACL names only one user. Older versions allow a process to hold only one group ID at a time and force it to use a privileged program to access other groups; newer Unix systems put a process in all groups that the user is in. This is still much less expressive than one might like. In theory, the ACL and su mechanisms can often be used to achieve the desired effect. In practice, programmers are often too lazy to figure out how to do this, and so design their code to require much more privilege than it really ought to.

4.2.4 Windows NT

Another important operating system whose protection is largely based on access control lists is Windows NT. The current version of NT (version 5, or Win2K) is fairly complex, so it's helpful to trace its antecedents. (This can also be useful if you have to help manage upgrades from NT4 to Win2K).

NT4 protection is very much like Unix, and appears to be inspired by it, so it's simpler to describe the main innovations.

First, rather than *just read, write*, and *execute*, there are separate attributes for *take ownership, change permissions*, and *delete*, which means that more flexible delegation can be supported. These attributes apply to groups as well as users, and group permissions allow you to achieve much the same effect as sgid programs in Unix. Attributes are not simply on or off, as in Unix, but have multiple values: you can set AccessDenied, AccessAllowed, or SystemAudit. These are parsed in that order. If an AccessDenied is encountered in an ACL for the relevant user or group, then no access is permitted, regardless of any conflicting AccessAllowed flags.

A benefit of the richer syntax is that you can arrange matters so that much less than full administrator privileges are required for everyday configuration tasks, such as installing printers. (This is rarely done, though.)

Second, users and resources can be partitioned into domains with distinct administrators, and trust can be inherited between domains in one direction or both. In a typical large company, you might put all the users into a domain administered by the personnel department, while resources such as servers and printers could be in resource domains under departmental control; individual workstations might even be administered by their users. Things would be arranged so that the departmental resource domains trust the user domain, but not vice versa—so a corrupt or careless departmental administrator couldn't do much damage outside his or her own domain. The individual workstations would in turn trust the department (but not vice versa) so that users could perform tasks that require local privilege (installing many software packages requires

this). Administrators are all-powerful (so you can't create truly tamper-resistant audit trails without using write-once storage devices), but the damage they can do can be limited by suitable organization. The data structure used to manage all this, and hide the ACL details from the user interface, is called the *Registry.*

Problems with designing an NT architecture in very large organizations include naming issues (which we'll explore later), the way domains scale as the number of principals increases (badly), and the restriction that a user in another domain can't be an administrator (which can cause complex interactions between local and global groups).

One peculiarity of NT is that everyone is a principal, not a default or an absence of control, so remove everyone means just prevent a file being generally accessible. A resource can be locked quickly by setting everyone to have no access. This brings us naturally to the subject of capabilities.

4.2.5 Capabilities

The next way to manage the access control matrix is to store it by rows. These are called *capabilities.* In the example in Figure 4.2, Bob's capabilities would be as shown in Figure 4.5.

The strengths and weaknesses of capabilities are more or less the opposite of ACLs. Runtime security checking is more efficient, and we can do delegation without much difficulty: Bob could create a certificate saying "Here is my capability, and I hereby delegate to David the right to read file 4 from 9 A.M. to 1 P.M.; signed Bob." On the other hand, changing a file's status can suddenly become more tricky, as it can be difficult to find out which users have access. This can be tiresome when investigating an incident or preparing evidence of a crime.

There were a number of experimental implementations in the 1970s, which were rather like file passwords; users would get hard-to-guess bitstrings for the various read, write, and other capabilities to which they were entitled. It was found that such an arrangement could give very comprehensive protection [804]. It was not untypical to find that almost all of an operating system could run in user mode, rather than as supervisor, so operating system bugs were not security critical. (In fact, many operating system bugs caused security violations, which made debugging the operating system much easier.)

The IBM AS/400 series systems employed capability-based protection, and enjoyed some commercial success. Now capabilities are making a comeback in the form of *public key certificates.* We'll discuss the mechanisms of public key cryptography in Chapter 5, and give more concrete details of certificate-based systems, such as SSL/TLS, in Section 19.5. For now, think of a public key certificate as a credential signed by some authority, which declares that the holder of a certain cryptographic key is a certain person, a member of some group, or the holder of some privilege.

User	Operating System	Accounts Program	Accounting Data	Audit Trail
Bob	rx	r	r	r

Figure 4.5 A capability.

As an example of where certificate-based capabilities can be useful, consider a hospital. If we implemented a rule stating "a nurse will have access to all the patients who are on her ward, or who have been there in the last 90 days," naively, each access control decision in the patient record system would require several references to administrative systems, to find out which nurses and which patients were on which ward, when. This means that a failure of the administrative systems can now affect patient safety much more directly than was previously the case, which is a clearly bad thing. Matters can be much simplified by giving nurses certificates that entitle them to access the files associated with their current ward. Such a system is starting to be fielded at our university hospital.

One point to bear in mind is that as public key certificates are often considered to be "crypto" rather than "access control," their implications for access control policies and architectures are not always thought through. The lessons that could have been learned from the capability systems of the 1970s are generally having to be rediscovered (the hard way). In general, the boundary between crypto and access control is a fault line where things can easily go wrong. The experts often come from different backgrounds, and the products from different suppliers.

4.2.6 Added Features in Windows 2000

A number of systems, from mainframe access control products to research systems, have combined ACLs and capabilities in an attempt to get the best of both worlds. But the most important application of capabilities is in Win2K.

Win2K adds capabilities in two ways that can override or complement the ACLs of NT4. First, users or groups can be either whitelisted or blacklisted by means of profiles. (Some limited blacklisting was also possible in NT4.) Security policy is set by groups rather than for the system as a whole. Groups are intended to be the primary method for centralized configuration management and control (group policy overrides individual profiles). Group policy can be associated with sites, domains, or organizational units, so it can start to tackle some of the real complexity problems with naming. Policies can be created using standard tools or by custom-coding (Microsoft has announced that group policy data will be exposed in a standard schema). Groups are defined in the *Active Directory*, an object-oriented database which organizes users, groups, machines, and organizational units within a domain in a hierarchical namespace, indexing them so they can searched for on any attribute. There are also finer-grained access control lists on individual resources.

As already mentioned, Win2K uses Kerberos as its main means of authenticating users across networks.[1] This is encapsulated behind the *Security Support Provider Interface* (SSPI), which enables administrators to plug in other authentication services.

---

[1]In fact, it's a proprietary variant, with changes to the ticket format, which prevent Win2K clients from working with existing Unix Kerberos infrastructures. The documentation for the changes is released on condition that it not be used to make compatible implementations. Microsoft's goal is to get everyone to install Win2K Kerberos servers. This has caused an outcry in the open systems community [76].

This brings us to the second way in which capabilities insinuate their way into Win2K: in many applications, people are likely to use the public key protocol SSL/TLS, which is widely used on the Web, and which is based on public key certificates. The management of these certificates can provide another, capability-oriented, layer of access control outside the purview of the Active Directory. (I discuss SSL/TLS in Section 19.5.)

There are various backward-compatibility issues. For example, high-security configurations of Win2K with full cryptographic authentication can't interwork with NT4 systems. This is because an active directory can exist alongside the registry of NT4, but the registry can't read it. So the deployment of Win2K's high-security features in large organizations is likely to be delayed until all the important applications have migrated.

Win2K provides a richer and more flexible set of access control tools than any system previously sold in mass markets. It does still have design limitations. Implementing roles whose requirements differ from those of groups could be tricky in some applications; SSL certificates are the obvious way to do this, but would require an external management infrastructure. Second, Windows is still (in most of its incarnations) a single-user operating system, in the sense that only one person can operate a PC at a time. Thus, if I want to run an unprivileged, sacrificial user on my PC for accessing untrustworthy Web sites that might contain malicious code, I have to log off and log on again, or use other techniques that are so inconvenient that few users will bother. So users still do not get the benefit from the operating system's protection properties that they might wish when browsing the Web.

4.2.7 Granularity

A practical problem with all current flavors of access control system is granularity. As the operating system works with files, this will usually be the smallest object with which its access control mechanisms can deal. So it will be application-level mechanisms that, for example, ensure that a bank customer at a cash machine can see his or her own balance but not anybody else's.

But it goes deeper than that. Many applications are built using database tools that give rise to some problems that are much the same whether running DB2 on MVS or Oracle on Unix. All the application data is bundled together in one file, and the operating system must either grant or deny a user access to the lot. So, if you developed your branch accounting system under a database product, then you'll probably have to manage one access mechanism at the operating system level and another at the database or application level. Many real problems result. For example, the administration of the operating system and the database system may be performed by different departments, which do not talk to each other; and often user pressure drives IT departments to put in crude hacks that make the various access control systems seem to work as one, but that open up serious holes.

Another granularity problem is *single sign-on.* Despite the best efforts of computer managers, most large companies accumulate systems of many different architectures, so users get more and more logons to different systems; consequently, the cost of administering them escalates. Many organizations want to give each employee a single logon to all the machines on the network. A crude solution is to endow their PCs with a menu of hosts to which a logon is allowed, and hide the necessary userids and passwords in scripts. More sophisticated solutions may involve a single security server

through which all logons must pass, or a smartcard to do multiple authentication protocols for different systems. Such solutions are hard to engineer properly. Whichever route one takes, the security of the best system can easily be reduced to that of the worst.

4.2.8 Sandboxing and Proof-Carrying Code

Another way of implementing access control is a software *sandbox.* Here users want to run some code that they have downloaded from the Web as an applet. Their concern is that the applet might do something nasty, such as taking a list of all their files and mailing it off to a software marketing company.

The designers of Java tackle this problem by providing a “sandbox” for such code—a restricted environment in which it has no access to the local hard disk (or at most only temporary access to a restricted directory), and is only allowed to communicate with the host it came from. These security objectives are met by having the code executed by an interpreter—the Java Virtual Machine (JVM)—which has only limited access rights [346]. Java is also used on smartcards, but (in current implementations at least) the JVM is, in effect, a compiler external to the card, which raises the issue of how the code it outputs can be gotten to the card in a trustworthy manner.

An alternative is proof-carrying code. Here, code to be executed must carry with it a proof that it doesn’t do anything that contravenes the local security policy. This way, rather than using an interpreter with the resulting speed penalty, one merely has to trust a short program that checks the proofs supplied by downloaded programs before allowing them to be executed. The huge overhead of a JVM is not necessary [585].

Both of these are less general alternatives to an architecture that supports proper supervisor-level confinement.

4.2.9 Object Request Brokers

There has been much interest of late in object-oriented software development, as it has the potential to cut the cost of software maintenance. An *object* consists of code and data bundled together, accessible only through specified externally visible *methods.* This also gives the potential for much more powerful and flexible access control. Much research is underway with the goal of producing a uniform security interface that is independent of the underlying operating system and hardware.

The idea is to base security functions on the *object request broker*, or ORB, a software component that mediates communications between objects. Many research efforts focus on the *Common Object Request Broker Architecture* (CORBA), which is an attempt at an industry standard for object-oriented systems. The most important aspect of this is that an ORB is a means of controlling calls that are made across protection domains. This approach appears promising but is still under development. (A book on CORBA security is [112].)

4.3 Hardware Protection

Most access control systems set out not just to control what users can do, but to limit what programs can do as well. In most systems, users can either write programs or download and install them. Programs may be buggy or even malicious.

Preventing one process from interfering with another is the *protection problem.* The *confinement problem* is usually defined as that of preventing programs communicating outward other than through authorized channels. This comes in several flavors. The goal may be to prevent active interference, such as memory overwriting, and to stop one process reading another's memory directly. This is what commercial operating systems set out to do. Military systems may also try to protect *metadata*—data about other data, subjects, or processes—so that, for example, a user can't find out which other users are logged on to the system or which processes they are running. In some applications, such as processing census data, confinement means allowing a program to read data but not release anything about it other than the results of certain constrained queries; this is covered further in Chapter 7.

Unless one uses sandboxing techniques (which are too restrictive for general programming environments), solving the confinement problem on a single processor means, at the very least, having a mechanism that will stop one program from overwriting another's code or data. There may be areas of memory that are shared in order to allow interprocess communication; but programs must be protected from accidental or deliberate modification, and they must have access to memory that is similarly protected.

This usually means that hardware access control must be integrated with the processor's memory management functions. A typical mechanism is *segment addressing.* Memory is addressed by two registers, a segment register that points to a segment of memory, and another address register that points to a location within that segment. The segment registers are controlled by the operating system, and often by a special component of it called the *reference monitor*, which links the access control mechanisms with the hardware.

The actual implementation has become more complex as the processors themselves have. Early IBM mainframes had a two-state CPU: the machine was either in authorized state or it was not. In the latter case, the program was restricted to a memory segment allocated by the operating system. In the former, it could alter the segment registers at will. An authorized program was one that was loaded from an authorized library.

Any desired access control policy can be implemented on top of this, given suitable authorized libraries, but this is not always efficient; and system security depends on keeping bad code (whether malicious or buggy) out of the authorized libraries. Later processors have offered more complex hardware mechanisms. Multics, an operating system developed at MIT in the 1960s and that inspired the development of Unix, introduced *rings of protection* which express differing levels of privilege: ring 0 programs had complete access to disk, supervisor states ran in ring 2, and user code at various less privileged levels [687]. Its features have to some extent been adopted in more recent processors, such as the Intel main processor line from the 80286 onward.

There are a number of general problems with interfacing hardware and software security mechanisms. For example, it often happens that a less privileged process such as application code needs to invoke a more privileged process such as a device driver. The mechanisms for doing this need to be designed with some care, or security bugs can be expected. The IBM mainframe operating system MVS, for example, had a bug in which a program that executed a normal and an authorized task concurrently could make the former authorized too [493]. Also, performance may depend quite drastically on whether routines at different privilege levels are called by reference or by value [687].

4.3.1 Intel 80_86/Pentium Processors

Early Intel processors, such as the 8088/8086 used in early PCs, had no distinction between system and user mode, and thus no protection at all—any running program controlled the whole machine. The 80286 added protected segment addressing and rings, so for the first time it could run proper operating systems. The 80386 had built-in virtual memory and large enough memory segments (4 Gb) that they could be ignored and the machine treated as a 32-bit flat-address machine. The 486 and Pentium series chips added more performance (caches, out-of-order execution and MMX). The Pentium 3 finally added a new security feature—a processor serial number. This caused such a storm of protest, driven by privacy advocates who feared it could be used for all sorts of "big brother" purposes, that it will apparently be discontinued in future Pentium products. (But identifying a PC will remain easy, as there are many other serial numbers in disk controllers and other components that a snooping program can read.)

The rings of protection are supported by a number of mechanisms. The current privilege level can be changed only by a process in ring 0 (the kernel). Procedures cannot access objects in lower-level rings directly; but there are *gates* that allow execution of code at a different privilege level and that manage the supporting infrastructure, such as multiple stack segments for different privilege levels and exception handling. (For more details, see [404].)

The Pentium's successor architecture, the IA-64, was not yet available at the time of writing. According to the advance publicity, its memory management is based on dividing the virtual address space of each process into several *regions* whose identifiers specify the set of translations belonging to a process, and provide a unique intermediate virtual address. This is to help avoid thrashing problems in caches and in translation lookaside buffers. Regions also provide efficient shared areas between processes. Like the Pentium, the IA-64 has four protection rings [382].

4.3.2 ARM Processors

The ARM is the 32-bit processor core most commonly licensed to third-party vendors of embedded systems. The original ARM (which stood for Acorn Rise Machine) was the first commercial RISC design. Its modern day successors are important because they are incorporated in all sorts of security-sensitive applications from mobile phones to the Capstone chips used by the U.S. government to protect secret data. A fast multiply-and-accumulate instruction and low-power consumption make the ARM very attractive for embedded applications doing public key cryptography and/or signal processing. (The standard reference is [325].)

The ARM is licensed as a processor core, which chip designers can include in their products, plus a number of optional add-ons. The basic core contains separate banks of registers for user and system processes, plus a software-interrupt mechanism that puts the processor in supervisor mode and transfers control to a process at a fixed address. The core contains no memory management, so ARM-based designs can have their hardware protection extensively customized. A system control coprocessor is available to help with this. It can support domains of processes that have similar access rights (and thus share the same translation tables) but that retain some protection from each other. This enables fast context switching. Standard product ARM CPU chips, from the model 600 onward, have this memory support built in.

One version, the Amulet, uses self-timed logic. Eliminating the clock saves power and reduces RF interference, but makes it necessary to introduce hardware protection features, such as register locking, into the main processor itself so that contention between different hardware processes can be managed. This is an interesting example of protection techniques typical of an operating system being recycled in mainline processor design.

4.3.3 Security Processors

Some modern smartcards are based on ARM processors, and the preceding remarks apply (though memory limitations mean that only basic hardware protection may be used). But the great majority of the microprocessor smartcards in the field still have 8-bit processors. Some of them have memory management routines that let certain addresses be read only when passwords are entered into a register in the preceding few instructions. The goal is that the various principals with a stake in the card—perhaps a card manufacturer, an OEM, a network, and a bank—can all have their secrets on the card and yet be protected from each other. This may be a matter of software; but some cards have small, hardwired access control matrices to enforce this protection.

There are other kinds of specialized hardware security support for cryptography and access control. Some of the encryption devices used in banking to handle ATM PINs have an *authorized state*, which must be set (by two console passwords or a physical key) when PINs are to be printed. This enables a shift supervisor to control when this job is run. Similar devices are used by the military to distribute keys. We'll discuss cryptoprocessors in more detail in Chapter 14, "Physical Tamper Resistance."

4.3.4 Other Processors

Some research systems in the 1970s implemented very extensive security checking in the hardware, from Multics to various capability systems. Some systems have a *fence address*, a boundary in the hardware below which only the operating system has access. More recent work has looked at quality of service (QoS) issues, and for ways in which one can guarantee that no process will hog the CPU to the extent that other processes are blocked. Such mechanisms are now starting to be introduced commercially ('Quality of Service Technology is promised by Microsoft for 'the Win2K timeframe'.) The interaction of such features with access control and protection generally is one of the things to watch out for in the future.

4.4 What Goes Wrong

Popular operating systems such as Unix/Linux and Windows are very large and complex, so they have many bugs. They are used in a huge range of systems, so their features are tested daily by millions of users under very diverse of circumstances. Consequently, many of the bugs are found and reported. Thanks to the Net, knowledge spreads widely and rapidly. Thus, at any one time, there may be dozens of security flaws that are known and for which attack scripts may be circulating on the Net. Until recently, this problem was limited. The banking industry used mainframes that ran less widely understood operating systems, while the military used custom "multilevel secure" operating systems, which were not available to outsiders at all. Nowadays, both of these industries are being forced by cost pressures to adopt commodity operating systems, so the publication of attack scripts has the potential to undermine a great range of systems.

The usual goal of an attacker is to get a normal account on the system and then become the system administrator, in order to take over the system completely. A surprising number of operating system bugs allow the transition from user to root. Such flaws can be classified in a number of ways, such as by the type of programming error, by the stage in the development process at which it was introduced, or by the level in the system at which things go wrong [493]. The failure might not even be in the technical implementation, but in the higher-level design. The user interface might induce people to mismanage access rights or do other stupid things that cause the access control to be bypassed (see Section 4.4.3 for some examples).

In general, the higher in a system we build the protection mechanisms, the more complex they'll be, the more other software they'll rely on, and the closer they'll be to the error-prone mark 1 human being, thus, the less dependable they are likely to be.

4.4.1 Smashing the Stack

Many, if not most, of the technical attacks on operating systems that are reported in *Computer Emergency Response Team* (CERT) bulletins and security mailing lists involve memory-overwriting attacks, colloquially known as "smashing the stack" (see Figure 4.6).

Programmers are often careless about checking the size of arguments. A classic example was a vulnerability in the Unix ~~`finger`~~ command. A widespread implementation of this would accept an argument of any length, although only 256 bytes had been allocated for this argument by the program. The result was that when an attacker used the command with a longer argument, the trailing bytes of the argument ended up being executed by the CPU.

The usual technique is to arrange for the trailing bytes of the argument to have a *landing pad*, a long space of *no-operation* (NOP) commands or other register commands that don't change the control flow, and whose task is to catch the processor if it executes any of them. The landing pad delivers the processor to the attack code, which will do something like creating a root account with no password or starting a shell with administrative privilege directly.

Many of the vulnerabilities reported routinely by CERT and bugtraq are variants on this theme. There is really no excuse for the problem to continue, as it has been well known for a generation. Most of the early 1960s time-sharing systems suffered from it,

and fixed it [349]. Penetration analysis efforts at the System Development Corporation in the early 1970s showed that the problem of "unexpected parameters" was still one of the most frequently used attack strategies [503]. Intel's 80286 processor introduced explicit parameter-checking instructions—verify read, verify write, and verify length—in 1982, but they were avoided by most software designers to prevent architecture dependencies. In 1988, large numbers of Unix computers were brought down simultaneously by the "Internet worm," which used the finger vulnerability just described, and thus brought memory-overwriting attacks to the notice of the mass media [724]. Yet programmers still don't check the size of arguments, and holes continue to be found. The attack isn't even limited to networked computer systems: at least one smartcard could be defeated by passing it a message longer than its programmer had anticipated.

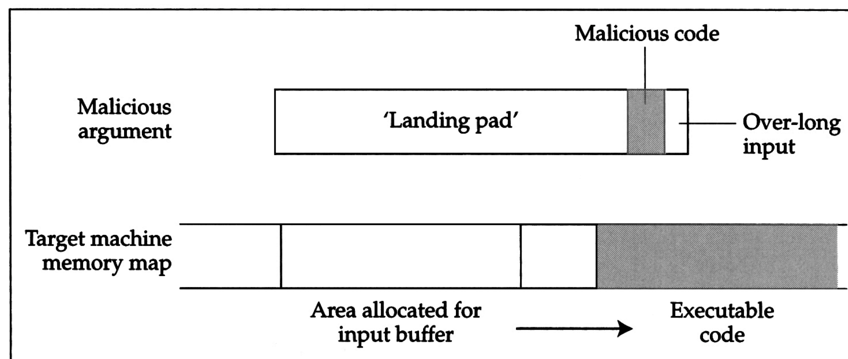

Figure 4.6 Stack Smashing Attack.

A recent survey paper describes memory-overwriting attacks as the "attack of the decade" [207].

4.4.2 Other Technical Attacks

After memory-overwriting attacks, *race conditions* are probably next. These are where a transaction is carried out in two or more stages, and it is possible for someone to alter it after the stage that involves verifying access rights.

For example, the Unix command to create a directory, mkdir, formerly worked in two steps: the storage was allocated, then ownership was transferred to the user. Since these steps were separate, a user could initiate a mkdir in background; and if this completed orly the first step before being suspended, a second process could be used to replace the newly created directory with a link to the password file. Then the original process would resume, and change ownership of the password file to the user. The /tmp directory, used for temporary files, can often be abused in this way; the trick is to wait until an application run by a privileged user writes a file here, then change it to a symbolic link to another file somewhere else—which will be removed when the privileged user's application tries to delete the temporary file.

A wide variety of other bugs have enabled users to assume root status and take over the system. For example, the PDP-10 TENEX operating system had the bug that the program address could overflow into the next bit of the process state word, which was the privilege-mode bit; this meant that a program overflow could put a program in su-

pervisor state. In another example, some Unix implementations had the feature that if a user tried to execute the command su when the maximum number of files were open, then su was unable to open the password file, and responded by giving the user root status.

There have also been a number of bugs that allowed service denial attacks. For example, Multics had a global limit on the number of files that could be open at once, but no local limits. A user could exhaust this limit and lock the system so that not even the administrator could log on [493]. And until the late 1990s, most implementations of the Internet protocols allocated a fixed amount of buffer space to process the SYN packets with which TCP/IP connections are initiated. The result was *SYN flooding attacks.* By sending a large number of SYN packets, an attacker could exhaust the available buffer space and prevent the machine accepting any new connections. This is now fixed using *syncookies*, discussed in Chapter 18, in Part 2.

4.4.3 User Interface Failures

One of the earliest attacks to be devised was the *Trojan Horse*, a program that the administrator is invited to run and that will do some harm if he does so. People would write games that checked occasionally whether the player was the system administrator, and if so would create another administrator account with a known password.

Another trick is to write a program that has the same name as a commonly used system utility, such as the ls command which lists all the files in a Unix directory, and design it to abuse the administrator privilege (if any) before invoking the genuine utility. The next step is to complain to the administrator that something is wrong with this directory. When the administrator enters the directory and types ls to see what's there, the damage is done. The fix is simple: an administrator's PATH variable (the list of directories that will be searched for a suitably named program when a command is invoked) shouldn't contain '.' (the symbol for the current directory). Recent Unix versions are shipped with this as a default; but it's still an unnecessary trap for the unwary.

Perhaps the most serious example of user interface failure, in terms of the number of systems at risk, is in Windows NT. In this operating system, a user must be the system administrator to install anything. This might be useful, as a configuration option, to prevent staff in a bank branch from running games on their PCs at lunchtime, and picking up viruses. However, most environments are much less controlled, and people need to be able to install software to get their work done. In practice, this means that millions of people have administrator privileges who shouldn't need them, and are vulnerable to attacks in which malicious code simply pops up a box telling them to do something. Microsoft's response to this has been the one-way trust mechanism already discussed, which makes it possible to configure systems so that people can administer their own machines without having too much power to damage other IT resources in the company. However, this requires some care to implement properly. It also provides no protection where applications such as Web servers must run as root, are visible to the outside world, and contain software bugs that enable them to be taken over.

Another example, which might be argued is an interface failure, comes from the use of active content of various kinds such as ActiveX controls. These can be a menace because users have no intuitively clear way of controlling them, and they can be used to launch serious attacks. Even Java, for all its supposed security, has suffered a number of attacks that exploited careless implementations [226]. However, many people

(and many companies) are unwilling to forgo the bells and whistles that active content can provide.

4.4.4 Why So Many Things Go Wrong

We've already mentioned the basic problems faced by operating system security designers: their products are huge and therefore buggy, and are tested by large numbers of users in parallel, some of whom will publicize their discoveries rather than reporting them to the vendor. There are other structural problems, too.

One of the more serious causes of failure is *kernel bloat.* Under Unix, all device drivers, filesystems, and so on must be in the kernel. The Windows 2000 kernel contains drivers for a large number of smartcards, card readers, and the like, many of which were written by equipment vendors. So large quantities of code are trusted, in that they are put inside the security perimeter. It can't really be a good idea for software companies to enable so many of their suppliers to break their systems, whether on purpose or accidentally. Some other systems, such as MVS, introduced mechanisms that decrease the level of trust needed by many utilities. However, the means to do this in the most common operating systems are few and relatively nonstandard.

Even more seriously, application developers often make their programs run as root. This may be easier, as it avoids permission problems. It also often introduces horrible vulnerabilities where more limited privilege could have been used with only a modicum of thought and a minor redesign. There are many systems—such as lpr / lpd, the Unix lineprinter subsystem—that do not need to run as root but do anyway on most systems. This has also been a source of security failures in the past (e.g., getting the printer to spool to the password file).

Some applications need a certain amount of privilege. For example, mail delivery agents must be able to deal with user mailboxes. But while a prudent designer would restrict this privilege to a small part of the application, most agents are written so that the whole program needs to run as root. The classic example is sendmail, which has a long history of serious security holes; but many other MTAs also have problems. The general effect is that a bug that ought to compromise only one person's mail may end up giving root privilege to an outside attacker.

Sometimes the cure is almost as bad as the disease. Some programmers avoid *root bloat* and the difficulty of getting non-root software installed and working securely by leaving important shared data structures and resources accessible to all users. Many systems store mail in a file per user in a world-writeable directory, which makes mail forgery easy. The Unix file utmp—the list of users logged in—is frequently used for security checking of various kinds, but is also frequently world-writeable! This should have been built as a service rather than a file, but fixing problems like these once the initial design decisions have been made can be difficult.

4.4.5 Remedies

Some classes of vulnerability can be fixed using automatic tools. Stack-overwriting attacks, for example, are largely due to the lack of proper bounds checking in C (the language most commonly used to write operating systems). Various tools are available on the Net for checking C programs for potential problems; there is even a compiler patch called StackGuard, which puts a *canary* next to the return address on the stack. This can be a random 32-bit value chosen when the program is started, and checked when a function is torn down. If the stack has been overwritten meanwhile, then with high probability the canary will change [207].

But, in general, much more effort needs to be put into design, coding, and testing. Rather than overusing powerful tools such as ~~setuid~~ in Unix and administrator privilege in NT, designers should create groups with limited powers, and be clear about what the compromise of that group should mean for the rest of the system. Programs should have only as much privilege as necessary: the *principle of least privilege* [662].

Software should also be designed so that the default configuration, and in general, the easiest way of doing something, is safe. But, many systems are shipped with dangerous defaults.

Finally, there's a contrarian view, of which you should be aware, as it's held by some senior Microsoft people: that access control doesn't matter. Computers are becoming single-purpose or single-user devices. Single-purpose devices, such as Web servers that deliver a single service, don't need much in the way of access control as there's nothing for operating system access controls to do; the job of separating users from each other is best left to the application code. As for the PC on your desk, if all the software on it comes from a single source, then again there's no need for the operating system to provide separation [588]. Not everyone agrees with this: the NSA view is at the other extreme, with deep distrust of application-level security and heavy emphasis on using the mechanisms of trusted operating systems [510]. But one way or another, it's remarkable how little effective use is made of the access control mechanisms shipped with modern operating systems.

4.4.6 Environmental Creep

I have pointed out repeatedly that many security failures result from environmental change undermining a security model. Mechanisms that were adequate in a restricted environment often fail in a more general one.

Access control mechanisms are no exception. Unix, for example, was originally designed as a "single-user Multics" (hence the name). It then became an operating system to be used by a number of skilled and trustworthy people in a laboratory who were sharing a single machine. In this environment, the function of the security mechanisms is primarily to contain mistakes, to prevent one user's typing errors or program crashes from deleting or overwriting another user's files. The original security mechanisms were quite adequate for this purpose.

But Unix security became a classic "success disaster." Unix was repeatedly extended without proper consideration being given to how the protection mechanisms also needed to be extended. The Berkeley extensions (~~rsh, rhosts, etc.~~) were based on an extension from a single machine to a network of machines that were all on one LAN

and all under one management. Mechanisms such as `rhosts` were based on a tuple *(username,hostname)* rather than just a user name, and saw the beginning of the transfer of trust.

The Internet mechanisms (telnet, ftp, DNS, SMTP), which grew out of Arpanet in the 1970s, were written for mainframes on what was originally a secure WAN. Mainframes were autonomous, the network was outside the security protocols, and there was no transfer of authorization. Remote authentication, which the Berkeley model was starting to make prudent, was simply not supported. The Sun contributions (NFS, NIS, RPC, etc.) were based on a workstation model of the universe, with a multiple LAN environment with distributed management, but still usually in a single organization. (A proper tutorial on topics such as DNS and NFS is beyond the scope of this book, but there is some more detailed background material in Chapter 18, "Network Attack and Defense," Section 18.2.)

Mixing all these different models of computation together has resulted in chaos. Some of their initial assumptions still apply partially, but none of them applies globally any more. The Internet now has hundreds of millions of PCs and workstations, millions of LANs, thousands of interconnected WANs, and managements that are not just independent but may be in conflict (including nation states and substate groups at war with each other). Many workstations have no management at all.

Users, instead of being trustworthy but occasionally incompetent, are now largely incompetent—but some are both competent and hostile. Code used to be simply buggy—but now there is a significant amount of malicious code out there. Attacks on communications networks used to be the purview of national intelligence agencies—now they can be done by *script kiddies*, a term used to refer to relatively unskilled people who have downloaded attack tools from the Net and launched them without any real idea of how they work.

Unix and Internet security gives us yet another example of a system that started out reasonably well designed but that was undermined by a changing environment.

Win2K and its predecessors in the NT product series have more extensive protection mechanisms than Unix, but have been around for much less time. Realistically, all we can say is that the jury is still out.

4.5 Summary

Access control mechanisms operate at a number of levels in a system, from applications down through the operating system to the hardware. Higher-level mechanisms can be more expressive, but also tend to be more vulnerable to attack, for a variety of reasons ranging from intrinsic complexity to implementer skill levels. Most attacks involve the opportunistic exploitation of bugs; and software that is very large, very widely used, or both (as with operating systems) is particularly likely to have security bugs found and publicized. Operating systems are also vulnerable to environmental changes that undermine the assumptions used in their design.

The main function of access control in computer operating systems is to limit the damage that can be done by particular groups, users, and programs whether through error or malice. The most important fielded examples are Unix and NT, which are similar in many respects, though NT is more expressive. Access control is also an important part of the design of special-purpose hardware such as smartcards and other

encryption devices. New techniques are being developed to cope with object-oriented systems and mobile code. But implementation remains generally awful.

The general concepts of access control from read, write, and execute permissions to groups and roles will crop up again and again. In some distributed systems, they may not be immediately obvious, as the underlying mechanisms can be quite different. An example comes from public key infrastructures, which are a reimplementation of an old access control concept, the capability.

Research Problems

Most of the issues in access control were identified by the 1960s or early 1970s, and were worked out on experimental systems such as Multics [687] and the CAP [804]. Much of the research in access control systems since has involved reworking the basic themes in new contexts, such as object-oriented systems and mobile code.

A recent thread of research is how to combine access control with the admission control mechanisms used to provide quality of service guaranteed in multimedia operating systems. Another topic is how to implement and manage access control efficiently in large complex systems, using techniques such as roles.

Further Reading

The best textbook to use for a more detailed introduction to access control issues is Dieter Gollmann's *Computer Security* [344]. A technical report from U.S. Navy Labs gives a useful reference to many of the flaws found in operating systems over the last 30 years or so [493]. One of the earliest reports on the subject (and indeed on computer security in general) is by Willis Ware [791]. One of the most influential early papers is by Jerry Saltzer and Mike Schroeder [662]; Butler Lampson's influential paper on the confinement problem is at [488].

The classic description of Unix security is in the paper by Fred Grampp and Bob Morris [350]. The most comprehensive textbook on this subject is Simson Garfinkel and Gene Spafford's *Practical Unix and Internet Security* [331]; the classic on the Internet side of things is Bill Cheswick and Steve Bellovin's *Firewalls and Internet Security* [94], with many examples of network attacks on Unix systems.

The protection mechanisms of Windows NT4 are described briefly in Gollmann, but much more thoroughly in Karanjit Siyan's reference book, *Windows NT Server 4* [711]. For Win2K, I've used the Microsoft online documentation; no doubt a number of textbooks will appear very soon. There is a history of microprocessor architectures at [79], and a reference book for Java security written by its architect Li Gong [346].

All these topics are fast-moving; the attacks that are making the headlines change significantly (at least in their details) from one year to the next. To keep up, you should not just read textbooks, but follow the latest notices from CERT, and mailing lists such as bugtraq.