Access control and recovery

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.

Access control works at a number of levels (Figure 4.1).

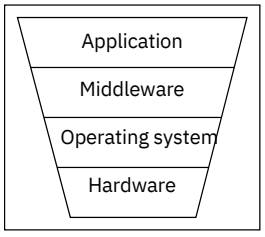

Figure 4.1: Access controls at different levels in a system

1. The access control mechanisms 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 refunds) might require dual control or approval from a supervisor. And that's nothing compared with the com- plexity of the access controls on a modern social networking site, which will have a thicket of rules and options about who can see, copy, and search what data from whom.
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 which debits one ledger for a certain amount must credit another ledger for the same amount, while database software typically has access controls specifying which dictionaries a given user can select, and which procedures they can run.
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. However, loopholes in access-control mechanisms — such as in database software used in a large number of web servers — can expose many systems simultaneously and compel large numbers of companies to patch or rewrite their products. So in this chapter, we will focus on the fundamentals: access control at the hardware, operating system and database levels. You have to understand the basic principles to design serviceable application-level controls too (I give many examples in Part II of how to combine access controls with the needs of specific applications).

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. PCs carry an unfortunate legacy, in that the old single-user operating

systems such as DOS and Win95/98 let any process modify any data; as a result many applications won't run unless they are run with administrator privileges and thus given access to the whole machine. Insisting that your software run as administrator is also more convenient for the programmer. But people do have implicit protection goals; you don't expect a shrink- wrap program to trash your hard disk. So an explicit security policy is a good idea.

Now one of the biggest challenges in computer security is preventing one program from interfering with another. You don't want a virus to be able to steal the passwords from your browser, or to patch a banking application so as to steal your money. In addition, many everyday reliability problems stem from applications interacting with each other or with the system configuration. However, it's difficult to separate applications when the customer wants to share data. It would make phishing much harder if people were simply unable to paste URLs from emails into a browser, but that would make everyday life much harder too. The single-user history of personal computing has got people used to all sorts of ways of working that are not really consistent with separating applications and their data in useful ways. Indeed, one senior manager at Microsoft took the view in 2000 that there was really nothing for operating-system access controls to do, as client PCs were single-user and server PCs were single-application.

The pendulum is now swinging back. Hosting centres make increasing use of virtualization; having machines exclusively for your own use costs several times what you pay for the same resource on shared machines. The Trusted Computing initiative, and Microsoft Vista, place more emphasis on separating applications from each other; even if you don't care about security, preventing your programs from overwriting each others' configuration files should make your PC much more reliable. More secure operating systems have led to ever more technical attacks on software other than the operating system; you don't want your brokerage account hacked via a computer game you downloaded that later truns out to be insecure. And employers would like ways of ensuring that employees' laptops don't pick up malware at home, so it makes sense to have one partition (or virtual machine) for work and another for play. Undoing the damage done by many years of information-sharing promiscuity will be hard, but in the medium term we might reasonably hope for a framework that enables interactions between applications to be restricted to controllable interfaces.

Many access control systems build on the mechanisms provided by the operating system. I will start off by discussing operating-system protection mechanisms that support the isolation of multiple processes. These came first historically — being invented along with the first time-sharing systems in the

1960s — and they remain the foundation on which many higher-layer mechanisms are built. I will then go on to discuss database systems, which provide broadly similar access control mechanisms that may or may not be tied to the operating-systems mechanisms. Finally I'll discuss three advanced protection techniques — *sandboxing, virtualization* and *'Trusted Computing'*. Sandboxing is an application-level control, run for example in a browser to restrict what mobile code can do; virtualization runs underneath the operating system, creating two or more independent virtual machines between which infor- mation flows can be controlled or prevented; and Trusted Computing is a project to create two virtual machines side-by-side, one being the 'old, inse- cure' version of an operating system and the second being a more restricted environment in which security-critical operations such as cryptography can be carried out.

The latest Microsoft system, Vista, is trying to move away from running all code with administrator privilege, and these three modern techniques are each, independently, trying to achieve the same thing — to get us back where we'd be if all applications had to run with user privileges rather than as the administrator. That is more or less where computing was in the 1970s, when people ran their code as unprivileged processes on time-shared minicomputers and mainframes. Only time will tell whether we can recapture the lost Eden of order and control alluded to in the quote from Roger Needham at the start of this chapter, and to escape the messy reality of today to which Rick Maybury's quote refers; but certainly the attempt is worth making.

4.2 OperatingSystemAccessControls

The access controls provided with an operating system typically authenticate principals using a 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.

	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

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.

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 rather that Sam didn't have this access. So we would prefer that write access to the accounting data file be possible only via the accounting program. The access permissions might now look like in Figure 4.3:

User	OperatingAccountsAccountingAudit			
Sam	System	Program	Data	Trail
Alice	rwx	rwx	r	r
Accountsprogram	rx	x	–	–
Bob	rx	r	rw	w
	rx	r	r	r

Figure 4.3: Access control matrix for bookkeeping

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 so much as a *protection domain* which is a set of processes or threads which 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 15,000,000 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 compress the users and to compress the rights. As for the first of these, the simplest is to use groups or roles to manage the privileges of large sets of users simultaneously, while in the second we may store the access control matrix either by columns (access control lists) or rows (capabilities, sometimes known as 'tickets') [1102, 1344]. (There are more complex approaches involving policy engines, but let's learn to walk before we try to run.)

4.2.1 GroupsandRoles

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

So we want a small number of pre-defined 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 government and business 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 organisation (and perhaps even a salary band) while the role *acting manager* might include an assistant accountant standing in while the manager, deputy manager, and branch accountant are all off sick.

Whether we need to be careful about this distinction is a matter for the application. In a warship, we want even an ordinary seaman to be allowed to stand watch if everyone more senior has been killed. In a bank, we might have a policy that 'transfers over $10m must be approved by two staff, one with rank at least manager and one with rank at least assistant accountant'. If the branch manager is sick, then the assistant accountant acting as manager might have to get the regional head office to provide the second signature on a large transfer.

Operating-system level support is available for groups and roles, but its appearance has been fairly recent and its uptake is still slow. Developers used to implement this kind of functionality in 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). Windows 2000 introduced extensive support for groups, while academic researchers have done quite a lot of work since the mid-90s on *role-based access control* (RBAC), which I'll discuss further in Part II, and which is starting to be rolled out in some large applications.

4.2.2 Access Control Lists

Another way of simplifying the management of access rights 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 (pronounced 'ackle'). In the first of our above examples, the ACL for file 3 (the account file) might look as shown here 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 a natural choice in environments where users manage their own file security, and became widespread in the Unix systems common in universities and science labs from the 1970s. They are the basic access control mechanism in Unix-based systems such as GNU/Linux and Apple's OS/X; the access controls in Windows are also based on ACLs, but have become more complex over time. Where access control policy is set centrally, ACLs 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 what 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 means that it can be tedious to find all the files to which a user has access. Revoking the access of an employee who has just been fired will usually have to be done by cancelling their password or other authentication mechanism. It may also be tedious to run system-wide checks; for example, verifying that no files have been left world-writable could involve checking ACLs on millions of user files.

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

User	Accounting Data
Sam	rw
Alice	rw
Bob	r

Figure 4.4: Access control list (ACL)

4.2.3 UnixOperatingSystemSecurity

In Unix (including 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 world, group and owner 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 our first example in Figure 4.3, 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; non-group 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 flavours 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.

The Berkeley distributions, including FreeBSD and OS/X, go some way towards 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 separate. In the old days that meant sending the system log to a printer in a locked room; nowadays, given the volumes of data, it means sending it to another machine, administered by somebody else.

Second, ACLs only contain 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 *set-user-id* (suid) file attribute. The owner of a program can mark it as suid, which enables it to run with the privilege of its owner rather than the privilege of the user who has invoked it. So in order to achieve the functionality needed by our second example above, 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 can then be given the access control attributes the accounts program needs.

One way of looking at this is that an access control problem that is naturally modelled in three dimensions — by 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, so 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 they should. (It's hard to know what to check, as 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 mutable state. Suppose we want a transaction to be authorised by a manager and an accountant before it's acted on; we can either do this at the application level (say, by having queues of transactions awaiting a second signature) or by doing something fancy with suid. In general, managing stateful access rules is difficult; this can even complicate the revocation of users who have just been fired, as it can be hard to track down the files they might have open.

Fourth, the Unix ACL only names 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 suid mechanisms can often be used to achieve the desired effect. In practice, programmers often can't be bothered to figure out how to do this, and design their code to require much more privilege than it really ought to have.

4.2.4 Apple's OS/X

Apple's OS/X operating system is based on the FreeBSD version of Unix running on top of the Mach kernel. The BSD layer provides memory protection; applications cannot access system memory (or each others') unless running with advanced permissions. This means, for example, that you can kill a wedged application using the 'Force Quit' command; you usually do not have

to reboot the system. On top of this Unix core are a number of graphics compo- nents, including OpenGL, Quartz, Quicktime and Carbon, while at the surface the Aqua user interface provides an elegant and coherent view to the user. At the file system level, OS/X is almost a standard Unix. The default installation has the root account disabled, but users who may administer the system are in a group 'wheel' that allows them to su to root. The most visible implication is that if you are such a user, you can install programs (you are asked for the root password when you do so). This may be a slightly better approach than Windows (up till XP) or Linux, which in practice let only administrators install software but do not insist on an authentication step when they do so; the many Windows users who run as administrator for convenience do dreadful things by mistake (and malware they download does dreadful things deliberately). Although Microsoft is struggling to catch up with Vista, as I'll discuss below, Apple's advantage may be increased further by OS/X version 10.5 (Leopard), which is based on TrustedBSD, a variant of BSD developed for government systems that incorporates mandatory access control. (I'll discuss this in Chapter 8.)

4.2.5 Windows–BasicArchitecture

The most widespread PC operating system is Windows, whose protection has been largely based on access control lists since Windows NT. The current version of Windows (Vista) is fairly complex, so it's helpful to trace its antecedents. The protection in Windows NT (Windows v4) was very much like Unix, and was inspired by it, and has since followed the Microsoft philosophy of 'embrace and extend'.

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 suid 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 everyday configuration tasks, such as installing printers, don't require full administrator privileges. (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 personnel domain administered by Human Resources, while resources such as servers and printers could be in resource domains under departmental control; individual workstations may 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 can't do much damage outside his own domain. The individual workstations would in turn trust the department (but not vice versa) so that users can perform tasks that require local privilege (installing many software packages requires this). Administrators are still all-powerful (so you can't create truly tamper-resistant audit trails without using write-once storage devices or writing to machines controlled by others) but the damage they can do can be limited by suitable organisation. The data structure used to manage all this, and hide the ACL details from the user interface, is called the *Registry*.

Problems with designing a Windows security architecture in very large organisations include naming issues (which I'll explore in Chapter 6), 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 Windows is that *everyone* is a principal, not a default or an absence of control, so 'remove everyone' means just stop 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.

4.2.6 Capabilities

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

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

Figure 4.5: A capability

The strengths and weaknesses of capabilities are more or less the opposite of ACLs. Runtime security checking is more efficient, and we can delegate a right 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 9am to 1pm, 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 we have to investigate an incident or prepare 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 [1344]. 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 harder.) The IBM AS/400 series systems employed capability-based protection, and enjoyed some commercial success. Now capabilities have made a limited comeback in the form of *public key certificates*. I'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 Part II. 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, or a member of some group, or the holder of some privilege.

As an example of where certificate-based capabilities can be useful, consider a hospital. If we implemented a rule like 'a nurse shall 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 will require several references to administrative systems, to find out which nurses and which patients were on which ward, when. So a failure of the administrative systems can now affect patient safety much more directly than before, and this is clearly a bad thing. Matters can be much simplified by giving nurses certificates which entitle them to access the files associated with their current ward. Such a system has been used for several years at our university hospital.

Public key certificates are often considered to be 'crypto' rather than 'access control', with the result that their implications for access control policies and architectures are not 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.

4.2.7 Windows–AddedFeatures

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 Windows.

Windows 2000 added capabilities in two ways which can override or complement the ACLs of Windows NT. 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 custom coded. Groups are defined in the *Active Directory*, an object-oriented database that organises users, groups, machines, and organisational units within a domain in a hierarchical namespace, indexing them so they can be searched for on any attribute. There are also finer grained access control lists on individual resources.

As already mentioned, Windows adopted Kerberos from Windows 2000 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.

This brings us to the second way in which capabilities insinuate their way into Windows: in many applications, people use the public key protocol 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.

The latest version of Windows, Vista, introduces a further set of protection mechanisms. Probably the most important is a package of measures aimed at getting away from the previous default situation of all software running as root. First, the kernel is closed off to developers; second, the graphics subsystem is removed from the kernel, as are most drivers; and third, User Account Control (UAC) replaces the default administrator privilege with user defaults instead. This involved extensive changes; in XP, many routine tasks required administrative privilege and this meant that enterprises usually made all their users administrators, which made it difficult to contain the effects of malware. Also, developers wrote their software on the assumption that it woulod have access to all system resources.

In Vista, when an administrator logs on, she is given two access tokens: a standard one and an admin one. The standard token is used to start the desktop, explorer.exe, which acts as the parent process for later user processes. This means, for example, that even administrators browse the web as normal users, and malware they download can't overwrite system files unless given later authorisation. When a task is started that requires admin privilege, then a user who has it gets an *elevation prompt* asking her to authorise it by entering an admin password. (This brings Windows into line with Apple's OS/X although the details under the hood differ somewhat.)

[1] It was in fact a proprietary variant, with changes to the ticket format which prevent Windows clients from working with existing Unix Kerberos infrastructures. The documentation for the changes was released on condition that it was not used to make compatible implementations. Microsoft's goal was to get everyone to install Win2K Kerberos servers. This caused an outcry in the open systems community [121]. Since then, the European Union prosecuted an antitrust case against Microsoft that resulted in interface specifications being made available in late 2006.

Of course, admin users are often tricked into installing malicious software, and so Vista provides further controls in the form of file integrity levels. I'll discuss these along with other mandatory access controls in Chapter 8 but the basic idea is that low-integrity processes (such as code you download from the Internet) should not be able to modify high-integrity data (such as system files). It remains to be seen how effective these measures will be; home users will probably want to bypass them to get stuff to work, while Microsoft is providing ever-more sophisticated tools to enable IT managers to lock down corporate networks — to the point, for example, of preventing most users from installing anything from removable media. UAC and mandatory integrity controls can certainly play a role in this ecology, but we'll have to wait and see how things develop.

The final problem with which the Vista developers grappled is the fact that large numbers of existing applications expect to run as root, so that they can fool about with registry settings (for a hall of shame, see [579]). According to the Microsoft folks, this is a major reason for Windows' lack of robustness: applications monkey with system resources in incompatible ways. So there is an Application Information Service that launches applications which require elevated privileges to run. Vista uses virtualization technology for legacy applications: if they modify the registry, for example, they don't modify the 'real' registry but simply the version of it that they can see. This is seen as a 'short-term fix' [885]. I expect it will be around for a long time, and I'm curious to see whether the added complexity will be worth the reduced malware risk.

Despite virtualisation, the bugbear with Vista is compatibility. As this book went to press in early January 2008, sales of Vista were still sluggish, with personal users complaining that games and other applications just didn't work, while business users were waiting for service pack 1 and postponing large-scale roll-out to late 2008 or even early 2009. It has clearly been expensive for Microsoft to move away from running everything as root, but it's clearly a necessary move and they deserve full credit for biting the bullet.

To sum up, Windows 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 require an external management infrastructure. Second, Windows is still (in its consumer 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 which are so inconvenient that few users will bother. (On my Mac, I can run two users simultaneously and switch between them quickly.) So Vista should be seen as the latest step on a journey, rather than a destination. The initial version also has some

undesirable implementation quirks. For example, it uses some odd heuristics to try to maintain backwards compatibility with programs that assume they'll run as administrator: if I compile a C

```
# program called Fred Installer.exe
```

then Vista will ask for elevated privilege to run it, and tell it that it's running on Windows XP, while if I call the program simply Fred.exe it will run as user and be told that it's running on Vista [797]. Determining a program's privileges on the basis of its filename is just bizarre.

And finally, there are serious usability issues. For example, most users will run still run administrator accounts all the time, and will be tempted to disable UAC; if they don't, they'll become habituated to clicking away the UAC dialog box that forever asks them if they really meant to do what they just tried to. For these reasons, UAC may be much less effective in practice than it might be in theory [555]. We will no doubt see in due course.

It's interesting to think about what future access controls could support, for example, an electronic banking application that would be protected from malware running on the same machine. Microsoft did come up with some ideas in the context of its 'Trusted Computing' project, which I'll describe below in section 4.2.11, but they didn't make it into Vista.

4.2.8 Middleware

Doing access control at the level of files and programs was all very well in the early days of computing, when these were the resources that mattered. Since about the 1980s, growing scale and complexity has meant led to access control being done at other levels instead of (sometimes as well as) at the operating system level. For example, a bank's branch bookkeeping system will typically run on top of a database product, and the database looks to the operating system as one large file. This means that the access control has to be done in the database; all the operating system supplies it may be an authenticated ID for each user who logs on.

4.2.8.1 DatabaseAccessControls

Until the dotcom boom, database security was largely a back-room concern. But it is now common for enterprises to have critical databases, that handle inventory, dispatch and e-commerce, fronted by web servers that pass trans- actions to the databases directly. These databases now contain much of the data of greatest relevance to our lives — such as bank accounts, vehicle regis- trations and employment records — and front-end failures sometimes expose the database itself to random online users.

Database products, such as Oracle, DB2 and MySQL, have their own access control mechanisms. As the database looks to the operating system as a single large file, the most the operating system can do is to identify users and to

separate the database from other applications running on the same machine. The database access controls are in general modelled on operating-system mechanisms, with privileges typically available for both users and objects (so the mechanisms are a mixture of access control lists and capabilities). However, the typical database access control architecture is more complex even than Windows: Oracle 10g has 173 system privileges of which at least six can be used to take over the system completely [804]. There are more privileges because a modern database product is very complex and some of the things one wants to control involve much higher levels of abstraction than files or processes. The flip side is that unless developers know what they're doing, they are likely to leave a back door open. Some products let developers bypass operating-system controls. For example, Oracle has both operating system accounts (whose users must be authenticated externally by the platform) and database accounts (whose users are authenticated directly by the Oracle software). It is often more convenient to use database accounts as it saves the developer from synchronising his work with the details of what other departments are doing. In many installations, the database is accessible directly from the outside; this raises all sorts of issues from default passwords to flaws in network protocols. Even where the database is shielded from the outside by a web service front-end, this often contains loopholes that let SQL code be inserted into the database. Database security failures can also cause problems directly. The Slammer worm in January 2003 propagated itself using a stack-overflow in Microsoft SQL Server 2000 and created large amounts of traffic and compromised machines sent large numbers of attack packets to random IP addresses.

Just as Windows is trickier to configure securely, because it's more complex, so the typical database system is trickier still, and it takes specialist knowledge that's beyond the scope of this book. Database security is now a discipline in its own right; if you have to lock down a database system — or even just review one as part of a broader assignment — I'd strongly recommend that you read a specialist text, such as David Litchfield's [804].

4.2.8.2 General Middleware Issues

There are a number of aspects common to middleware security and application- level controls. The first is granularity: as the operating system works with files, these are usually the smallest objects with which its access control mechanisms can deal. The second is state. An access rule such as 'a nurse can see the records of any patient on her ward' or 'a transaction over $100,000 must be authorised by a manager and an accountant' both involve managing state: in the first case the duty roster, and in the second the list of transactions that have so far been authorised by only one principal. The third is level: we may end up with separate access control systems at the machine, network and application

levels, and as these typically come from different vendors it may be difficult to keep them consistent.

Ease of administration is often a critical bottleneck. In companies I've advised, the administration of the operating system and the database system have been done by different departments, which do not talk to each other; and often user pressure drives IT departments to put in crude hacks which make the various access control systems seem to work as one, but which open up serious holes. An example 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 and the cost of administering them escalates. Many organisations want to give each employee a single logon to all the machines on the network. Commercial solutions may involve a single security server through which all logons must pass, and the use of a smartcard to do multiple authentication protocols for different systems. Such solutions are hard to engineer properly, and the security of the best system can very easily be reduced to that of the worst.

4.2.8.3 ORBsandPolicyLanguages

These problems led researchers to look for ways in which access control for a number of applications might be handled by standard middleware. Research in the 1990s focussed on *object request brokers* (ORBs). An ORB is a software component that mediates communications between objects (an object consists of code and data bundled together, an abstraction used in object-oriented as C++). A CORBA typically provides a means of controlling calls that are made across protection domains. The *Common Object Request Broker Architecture* (CORBA) is an attempt at an industry standard for object-oriented systems; a book on CORBA security is [182]. This technology is starting to be adopted in some industries, such as telecomms.

Research since 2000 has included work on languages to express security policy, with projects such as XACML (Sun), XrML (ISO) and SecPAL (Microsoft). They followed early work on 'Policymaker' by Matt Blaze and others [188], and vary in their expressiveness. XrML deals with subjects and objects but not relationships, so cannot easily express a concept such as 'Alice is Bob's manager'. XACML does relationships but does not support universally quantified variables, so it cannot easily express 'a child's guardian may sign its report card' (which we might want to program as 'if *x* is a child and *y* is *x*'s guardian and *z* is *x*'s report card, then *y* may sign *z*). The initial interest in these languages appears to come from the military and the rights-management industry, both of which have relatively simple state in their access control policies. Indeed, DRM engineers have already developed a number of specialised rights-management languages that are built into products such as Windows Media Player and can express concepts such as 'User *X* can play this file as

audio until the end of September and can burn it to a CD only once.' The push for interoperable DRM may create a demand for more general mechanisms that can embrace and extend the products already in the field.

If a suitably expressive policy language emerges, and is adopted as a standard scripting language on top of the access-control interfaces that major applications provide to their administrators, it might provide some value by enabling people to link up access controls when new services are constructed on top of multiple existing services. There are perhaps two caveats. First, people who implement access control when customizing a package are not likely to do so as a full-time job, and so it may be better to let them use a language with which they are familiar, in which they will be less likely to make mistakes. Second, security composition is a hard problem; it's easy to come up with examples of systems that are secure in isolation but that break horribly when joined up together. We'll see many examples in Part II.

Finally, 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 — so the less dependable they are likely to prove. Platform vendors such as Microsoft have more security PhDs, and more experience in security design, than almost any application vendor; and a customer who customises an application package usually has less experience still. Code written by users is most likely to have glaring flaws. For example, the fatal accidents that happened in healthcare as a result of the Y2K bug were not platform failures, but errors in spreadsheets developed by individual doctors, to do things like processing lab test results and calculating radiology dosages. Letting random users write security-critical code carries the same risks as letting them write safety-critical code.

4.2.9 Sandboxing and Proof-Carrying Code

The late 1990s saw the emergence of yet another way of implementing access control: the software *sandbox*. This was introduced by Sun with its Java programming language. The model is that a user wants to run some code that she has downloaded from the web as an applet, but is concerned that the applet might do something nasty, such as taking a list of all her files and mailing it off to a software marketing company.

The designers of Java tackled 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 [539]. 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 got to the card in a trustworthy manner. Another application is in the new Blu-ray format for high-definition DVDs; players have virtual machines that execute rights-management code bundled with the disk. (I describe the mechanisms in detail in section 22.2.6.2.)

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 overhead of a JVM is not necessary [956].

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

4.2.10 Virtualization

This refers to systems that enable a single machine to emulate a number of machines independently. It was invented in the 1960s by IBM [336]; back when CPUs were very expensive, a single machine could be partitioned using VM/370 into multiple virtual machines, so that a company that bought two mainframes could use one for its production environment and the other as a series of logically separate machines for development, testing, and minor applications.

The move to PCs saw the emergence of virtual machine software for this platform, with offerings from various vendors, notably VMware and (in open- source form) the Xen project. Virtualization is very attractive to the hosting industry, as clients can be sold a share of a machine in a hosting centre for much less than a whole machine. In the few years that robust products have been available, their use has become extremely widespread.

At the client end, virtualization allows people to run a host operating system on top of a guest (for example, Windows on top of Linux or OS/X) and this offers not just flexibility but the prospect of better containment. For example, an employee might have two copies of Windows running on his laptop — a locked-down version with her office environment, and another for use at home. The separation can be high-grade from the technical viewpoint; the usual problem is operational. People may feel the need to share data between the two virtual machines and resort to ad-hoc mechanisms, from USB sticks to webmail accounts, that undermine the separation. Military system designers are nonetheless very interested in virtualization; I discuss their uses of it in section 8.5.3.

4.2.11 Trusted Computing

The 'Trusted Computing' initiative was launched by Microsoft, Intel, IBM, HP and Compaq to provide a more secure PC. Their stated aim was to provide

software and hardware add-ons to the PC architecture that would enable people to be sure that a given program was running on a machine with a given specification; that is, that software had not been patched (whether by the user or by other software) and was running on a identifiable type and configuration of PC rather than on an emulator. The initial motivation was to support digital rights management. The problem there was this: if Disney was going to download a copy of a high-definition movie to Windows Media Player on your PC, how could they be sure it wasn't a hacked version, or running on a copy of Windows that was itself running on top of Xen? In either case, the movie might be ripped and end up on file-sharing systems.

The hardware proposal was to add a chip, the Trusted Platform Module or TPM, which could monitor the PC at boot time and report its state to the operating system; cryptographic keys would be made available depending on this state. Thus if a platform were modified — for example, by changing the boot ROM or the hard disk controller — different keys would be derived and previously encrypted material would not be available. A PC would also be able to use its TPM to certify to other PCs that it was in an 'approved' configuration, a process called *remote attestation*. Of course, individual PCs might be hacked in less detectable ways, such as by installing dual-ported memory or interfering with the bus from the TPM to the CPU — but the idea was to exclude low-cost break-once-run-anywhere attacks. Then again, the operating system will break from time to time, and the media player; so the idea was to make the content protection depend on as little as possible, and have revocation mechanisms that would compel people to upgrade away from broken software. Thus a vendor of digital content might only serve premium products to a machine in an approved configuration. Furthermore, data-based access control policies could be enforced. An example of these is found in the 'Information Rights Management' mechanisms introduced with Office 2003; here, a file can be marked with access controls in a rights expression language which can state, for example, that it may only be read by certain named users and only for a certain period of time. Word-processing documents (as well as movies) could be marked 'view three times only'; a drug dealer could arrange for the spreadsheet with November's cocaine shipments to be unreadable after December, and so on.

There are objections to data-based access controls based on competition policy, to which I'll return in Part III. For now, my concern is the mechanisms. The problem facing Microsoft was to maintain backwards compatibility with the bad old world where thousands of buggy and insecure applications run as administrator, while creating the possibility of new access domains to which the old buggy code has no access. One proposed architecture, Palladium, was unveiled in 2002; this envisaged running the old, insecure, software in parallel with new, more secure components.

In addition to the normal operating system, Windows would have a 'Nexus', a security kernel small enough to be verified, that would talk directly to the TPM hardware and monitor what went on in the rest of the machine; and each application would have a Nexus Control Program (NCP) that would run on top of the Nexus in the secure virtual machine and manage critical things like cryptographic keys. NCPs would have direct access to hardware. In this way, a DRM program such as a media player could keep its crypto keys in its NCP and use them to output content to the screen and speakers directly — so that the plaintext content could not be stolen by spyware.

At the time of writing, the curtained memory features are not used in Vista; presentations at Microsoft research workshops indicated that getting fine-grained access control and virtualization to work at the middle layers of such a complex operating system has turned out to be a massive task. Meanwhile the TPM is available for secure storage of root keys for utilities such as hard disk encryption; this is available as 'BitLocker' in the more expensive versions of Vista. It remains to be seen whether the more comprehensive vision of Trusted Computing can be made to work; there's a growing feeling in the industry that it was too hard and, as it's also politically toxic, it's likely to be quietly abandoned. Anyway, TPMs bring us to the more general problem of the hardware protection mechanisms on which access controls are based.

4.3 HardwareProtection

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. So 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. There are several flavours of each. The goal may be to prevent active interference, such as memory overwriting, or 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, or subjects, or processes — so that, for example, a user can't find out what other users are logged on to the system or what processes they're running. In some applica- tions, 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.

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 which points to a segment of memory, and another address register which 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. So later processors offered more complex hardware mechanisms. Multics, an operating system developed at MIT in the 1960's and which 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 [1139]. Its features have to some extent been adopted in more recent processors, such as the Intel main processor line from the 80286 onwards.

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 which executed a normal and an authorized task concurrently could make the former authorized too [774]. Also, performance may depend quite drastically on whether routines at different privilege levels are called by reference or by value [1139].

4.3.1 Intel Processors, and 'Trusted Computing'

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 rings of protection are supported by a number of mechanisms. The current privilege level can only be changed by a process in ring 0 (the kernel). Procedures cannot access objects in lower level rings directly but there are *gates* which allow execution of code at a different privilege level and which manage the supporting infrastructure, such as multiple stack segments for different privilege levels and exception handling. For more details, see [646].

The Pentium 3 finally added a new security feature — a processor serial number. This caused a storm of protest because privacy advocates feared it could be used for all sorts of 'big brother' purposes, which may have been irrational as computers have all sorts of unique numbers in them that software can use to tell which machine it's running on (examples range from MAC addresses to the serial numbers of hard disk controllers). At the time the serial number was launched, Intel had planned to introduce cryptographic support into the Pentium by 2000 in order to support DRM. Their thinking was that as they earned 90% of their profits from PC microprocessors, where they had 90% of the market, they could only grow their business by expanding the market for PCs; and since the business market was saturated, that meant sales to homes where, it was thought, DRM would be a requirement.

Anyway, the outcry against the Pentium serial number led Intel to set up an industry alliance, now called the Trusted Computing Group, to introduce cryptography into the PC platform by means of a separate processor, the Trusted Platform Module (TPM), which is a smartcard chip mounted on the PC motherboard. The TPM works together with *curtained memory* features introduced in the Pentium to enable operating system vendors to create memory spaces isolated from each other, and even against a process in one memory space running with administrator privileges. The mechanisms proposed by Microsoft are described above, and have not been made available in commercial releases of Windows at the time of writing.

One Intel hardware feature that has been implemented and used is the x86 virtualization support, known as Intel VT (or its development name, Vanderpool). AMD has an equivalent offering. Processor architectures such as S/370 and M68000 are easy to virtualize, and the theoretical requirements for this have been known for many years [1033]. The native Intel instruction set, however, had instructions that were hard to virtualize, requiring messy workarounds, such as patches to hosted operating systems. Processors with these extensions can use products such as Xen to run unmodified copies of guest operating systems. (It does appear, though, that if the Trusted Computing

mechanisms are ever implemented, it will be complex to make them work alongside virtualization.)

4.3.2 ARM Processors

The ARM is the 32-bit processor core most commonly licensed to third party vendors of embedded systems; hundreds of millions are used in mobile phones and other consumer electronic devices. The original ARM (which stood for *Acorn Risc Machine*) was the first commercial RISC design. ARM chips are also used in many security products, from the Capstone chips used by the US government to protect secret data, to the crypto accelerator boards from firms like nCipher that do cryptography for large web sites. A fast multiply- and-accumulate instruction and low power consumption made the ARM very attractive for embedded applications doing crypto and/or signal processing. The standard reference is [508].

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 manage- ment, 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 gives fast context switching. Standard product ARM CPU chips, from the model 600 onwards, have this memory support built in. There is a version, the Amulet, which uses self-timed logic. Eliminating the clock saved power and reduces RF interference, but made it necessary to introduce hardware protection features, such as register locking, into the main processor itself so that contention between different hardware processes could be managed. This is an interesting example of protection techniques typical of an operating system being recycled in main-line processor design.

4.3.3 Security Processors

Specialist security processors range from the chips in smartcards, through the TPM chips now fixed to most PC motherboards (which are basically smartcard chips with parallel interfaces) and crypto accelerator boards, to specialist crypto devices.

Many of the lower-cost smartcards 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 was 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.

Many of the encryption devices used in banking to handle ATM PINs have a further layer of application-level access control in the form of an 'authorized state' which must be set (by two console passwords, or a physical key) when PINs are to be printed. This is reminiscent of the old IBM mainframes, but is used for manual rather than programmatic control: it enables a shift supervisor to ensure that he is present when this job is run. Similar devices are used by the military to distribute keys. I'll discuss cryptoprocessors in more detail in Chapter 16.

4.4 WhatGoesWrong

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 circumstances. Consequently, many 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 are circulating on the net. A vulnerability has a typical lifecycle whereby it is discovered; reported to CERT or to the vendor; a patch is shipped; the patch is reverse-engineered, and an exploit is produced for the vulnerability; and people who did not apply the patch in time find that their machines have been recruited to a botnet when their ISP cuts them off for sending spam. There is a variant in which the vulnerability is exploited at once rather than reported — often called a *zero-day* exploit as attacks happen from day zero of the vulnerability's known existence. The economics, and the ecology, of the vulnerability lifecycle are the subject of intensive study by security economists; I'll discuss their findings in Part III.

The traditional goal of an attacker was to get a normal account on the system and then become the system administrator, so he could take over the system completely. The first step might have involved guessing, or social-engineering, a password, and then using one of the many known operating-system bugs that allow the transition from user to root. A taxonomy of such technical flaws was compiled in 1993 by Carl Landwehr [774]. These involved failures in the technical implementation, of which I will give examples in the next two sections, and also in the higher level design; for example, the user interface might induce people to mismanage access rights or do other stupid things which cause the access control to be bypassed. I will give some examples in section 4.4.3 below.

The user/root distinction has become less important in the last few years for two reasons. First, Windows PCs have predominated, running applications that insist on being run as administrator, so any application that can be compromised gives administrator access. Second, attackers have come to focus on compromising large numbers of PCs, which they can organise into a botnet in order to send spam or phish and thus make money. Even if your mail client were not running as administrator, it would still be useful to a spammer who could control it. However, botnet herders tend to install *rootkits* which, as their name suggests, run as root; and the user/root distinction does still matter in business environments, where you do not want a compromised web server or database application to expose your other applications as well. Perhaps if large numbers of ordinary users start running Vista with User Account Control enabled, it will make the botnet herders' lives a bit harder. We may at least hope.

In any case, the basic types of technical attack have not changed hugely since the early 1990s and I'll now consider them briefly.

4.4.1 Smashing the Stack

About half 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'. The proportion was even higher in the late 1990s and early 2000s but is now dropping slowly.

The basic idea behind the stack-smashing attack is that programmers are often careless about checking the size of arguments, so an attacker who passes a long argument to a program may find that some of it gets treated as code rather than data. 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. When an attacker used the command with a longer argument, the trailing bytes of the argument ended up being executed by the system.

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 (see Figure 4.6).

Many of the vulnerabilities reported routinely by CERT and bugtraq are variants on this theme. I wrote in the first edition of this book, in 2001, 'There is really no excuse for the problem to continue, as it's been well known for a generation'. Yet it remains a problem.

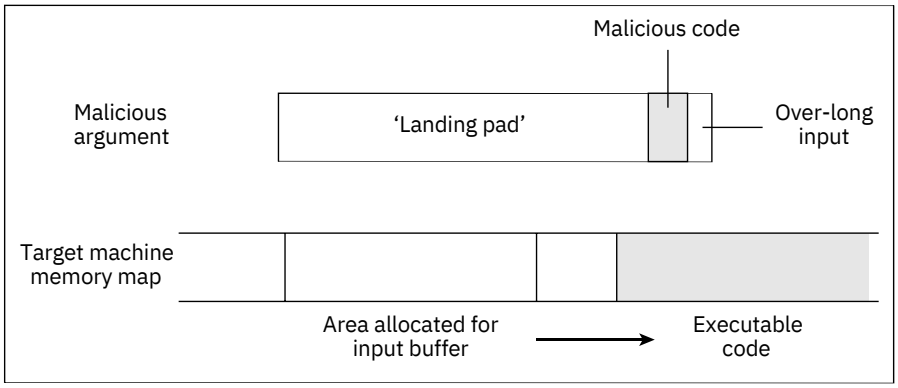

Figure 4.6: Stack smashing attack

Most of the early 1960's time sharing systems suffered from it, and fixed it[549]. Penetration analysis efforts at the System Development Corporation intheearly'70sshowedthattheproblemof'unexpectedparameters'wasstill one of the most frequently used attack strategies[799]. 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', whichusedthefingervulnerabilitydescribedabove,andthusbroughtmemory overwriting attacks to the notice of the mass media[1206]. A 1998 survey paperdescribedmemoryoverwritingattacksasthe'attackofthedecade'[329]. Yet programmersstilldon't check the sizeof arguments, and holes keepon being found. The attack isn't evenlimitedto networked computer systems: at least one smartcard could be defeated by passing it a message longer than its programmerhad anticipated.

4.4.2 OtherTechnicalAttacks

In 2002, Microsoft announced a security initiative that involved every programmerbeingtrainedinhowtowritesecurecode.(Thebooktheyproduced for this, *'Writing Secure Code'* by Michael Howard and David LeBlanc, is good; I recommend it to my students[627].) Other tech companies have launched similar training programmes. Despite the training and the tools, memory overwriting attacks are still appearing, to the great frustration of software companymanagers.However,theyareperhapshalfofallnewvulnerabilities nowrather than the 90% they were in 2001. The other new vulnerabilities are mostly variations on the same general theme, in that they occur when data in grammar A is interpreted as being in grammar B. A stack overflow is when data are accepted as input (e.g. a URL)

and end up being executed as machine code. They are essentially failures of type safety.

A **format string vulnerability** arises when a machine accepts input data as a formatting instruction (e.g. %n in the C command printf()). These commonly arise when a programmer tries to print user-supplied data and carelessly allows the print command to interpret any formatting instructions in the string; this may allow the string's author to write to the stack. There are many other variants on the theme; buffer overflows can be induced by improper string termination, passing an inadequately sized buffer to a path manipulation function, and many other subtle errors. See Gary McGraw's book *'Software Security'* [858] for a taxonomy.

SQL insertion attacks commonly arise when a careless web developer passes user input to a back-end database without checking to see whether it contains SQL code. The game is often given away by error messages, from which a capable and motivated user may infer enough to mount an attack. (Indeed, a survey of business websites in 2006 showed that over 10% were potentialy vulnerable [1234].) There are similar command-injection problems afflicting other languages used by web developers, such as PHP and perl. The remedy in general is to treat all user input as suspicious and validate it.

Checking data sizes is all very well when you get the buffer size calculation correct, but when you make a mistake — for example, if you fail to consider all the edge cases — you can end up with another type of attack called an **integer manipulation attack**. Here, an overflow, underflow, wrap-around or truncation can result in the 'security' code writing an inappropriate number of bytes to the stack.

Once such type-safety attacks are dealt with, **race conditions** are probably next. These occur when a transaction is carried out in two or more stages, and it is possible for someone to alter it after the stage which involves verifying access rights. I mentioned in Chapter 2 how a race condition can allow users to log in as other users if the userid can be overwritten while the password validation is in progress. Another classic example arose in the Unix command to create a directory, 'mkdir', which used to work in two steps: the storage was allocated, and then ownership was transferred to the user. Since these steps were separate, a user could initiate a 'mkdir' in background, and if this completed only 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 supervisor 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. In more modern systems, the most intractable user-to-root problems tend to be feature interactions. For example, we've struggled with backup and recovery systems. It's convenient if you can let users recover their own files, rather than having to call a sysadmin — but how do you protect information assets from a time traveller, especially if the recovery system allows him to compose parts of pathnames to get access to directories that were always closed to him? And what if the reovery functionality is buried in an application to which he needs access in order to do his job, and can be manipulated to give root access?

There have also been many 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. So a user could exhaust this limit and lock the system so that not even the administrator could log on [774]. And until the late 1990's, 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*, which I'll discuss in Part II.

The most recently discovered family of attacks of this kind are on *system call wrappers*. These are software products that modify software behaviour by intercepting the system calls it makes and performing some filtering or manipulation. Some wrapper products do virtualization; others provide secu- rity extensions to operating systems. However Robert Watson has discovered that such products may have synchronization bugs and race conditions that allow an attacker to become root [1325]. (I'll describe these in more detail in section 18.3.) The proliferation of concurrency mechanisms everywhere, with multiprocessor machines suddenly becoming the norm after many years in which they were a research curiosity, may lead to race conditions being the next big family of attacks.

4.4.3 UserInterfaceFailures

One of the earliest attacks to be devised was the *Trojan Horse*, a program that the administrator is invited to run and which will do some harm if he does so. People would write games which 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 which will be searched for a suitably named program when a command is invoked) should not 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. I refer to the fact that, until Vista came along, a user needed to be the system administrator to install anything2. In theory this might have helped a bank preventing its branch staff from running games on their PCs at lunchtime and picking up viruses. But most environments are much less controlled, and many people need to be able to install software to get their work done. So 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. Thank goodness Vista is moving away from this, but UAC 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. These can be a menace because users have no intuitively clear way of controlling them. Javascript and ActiveX in web pages, macros in Office documents and executables in email attachments have all been used to launch serious attacks. Even Java, for all its supposed security, has suffered a number of attacks that exploited careless implementations [360]. However, many people (and many companies) are unwilling to forego the bells and whistles which active content can provide, and we saw in Chapter 2 how the marketing folks usually beat the security folks (even in applications like banking).

4.4.4 WhySoManyThingsGoWrong

We've already mentioned the basic problem faced by operating system security designers: their products are huge and therefore buggy, and are tested by large

[2] In theory a member of the Power Users Group in XP could but that made little difference.

numbers of users in parallel, some of whom will publicize their discoveries rather than reporting them to the vendor. Even if all bugs were reported responsibly, this wouldn't make much difference; almost all of the widely exploited vulnerabilities over the last few years had already been patched. (Indeed, Microsoft's 'Patch Tuesday' each month is followed by 'Exploit Wednesday' as the malware writers reverse the new vulnerabilities and attack them before everyone's patched them.)

There are other structural problems too. One of the more serious causes of failure is *kernel bloat*. Under Unix, all device drivers, filesystems etc. must be in the kernel. Until Vista, the Windows kernel used to contain drivers for a large number of smartcards, card readers and the like, many of which were written by the equipment vendors. So large quantities of code were trusted, in that they are put inside the security perimeter. 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, most application developers make their programs run as root. The reasons for this are economic rather than technical, and are easy enough to understand. A company trying to build market share for a platform, such as an operating system, must appeal to its complementers — its application developers — as well as to its users. It is easier for developers if programs can run as root, so early Windows products allowed just that. Once the vendor has a dominant position, the business logic is to increase the security, and also to customise it so as to lock in both application developers and users more tightly. This is now happening with Windows Vista as the access control mechanisms become ever more complex, and different from Linux and OS/X. A similar pattern, or too little security in the early stages of a platform lifecycle and too much (of the wrong kind) later, has been observed in other platforms from mainframes to mobile phones.

Making many applications and utilities run as root has repeatedly introduced 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 — which does not need to run as root but does 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 which

ought to compromise only one person's mail may end up giving root privilege to an outside attacker.

So we're going to have some interesting times as developers come to grips with UAC. The precedents are not all encouraging. Some programmers histori- cally avoided the difficulty of getting non-root software installed and working securely by simply leaving important shared data structures and resources accessible to all users. Many old systems stored 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 — was 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 hard. I expect to see all the old problems of 1970s multiuser systems come back again, as the complex- ity of using the Vista mechanisms properly just defeats many programmers who aren't security specialists and are just desparate to get something sort of working so they can end the assignment, collect their bonus and move on.

4.4.5 Remedies

Some classes of vulnerability can be fixed using automatic tools. Stack over- writing attacks, for example, are largely due to the lack of proper bounds checking in C (the language most operating systems are written in). There are various tools (including free tools) available for checking C programs for potential problems, and 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 [329]. The availability of these tools, and training initiatives such as Microsoft's, have slowly reduced the number of stack overflow errors. However, attack tools also improve, and attackers are now finding bugs such as format string vulnerabilities and integer overflows to which no-one paid much attention in the 1990s.

In general, much more effort needs to be put into design, coding and testing. Architecture matters; having clean interfaces that evolve in a controlled way, under the eagle eye of someone experienced who has a long-term stake in the security of the product, can make a huge difference. (I'll discuss this at greater length in Part III.) Programs should only have as much privilege as they need: the *principle of least privilege* [1102]. Software should also be designed so that the default configuration, and in general, the easiest way of doing something, should be safe. Sound architecture is critical in achieving safe defaults and using least privilege. However, many systems are shipped with dangerous defaults and messy code that potentially exposes all sorts of interfaces to attacks like SQL injection that just shouldn't happen.

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 mostly 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 versions assumed 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 username, and saw the beginning of the transfer of trust.

The Internet mechanisms (telnet, ftp, DNS, SMTP), which grew out of Arpanet in the 1970's, 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. Thus 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 organisation. (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 the section on Vulnerabilities in Network Protocols in Chapter 21.)

Mixing all these different models of computation together has resulted in chaos. Some of their initial assumptions still apply partially, but none of them apply globally any more. The Internet now has hundreds of millions of PCs, millions of LANs, thousands of interconnected WANs, and managements which are not just independent but may be in conflict (including nation states and substate groups that are at war with each other). Many PCs have no management at all, and there's a growing number of network-connected Windows and Linux boxes in the form of fax machines, routers and other embedded products that don't ever get patched.

> 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*, relatively unskilled people who have downloaded attack tools from the net and launched them without any real idea of how they work.

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

Windows Vista and its predecessors in the NT product series have more extensive protection mechanisms than Unix, but have been around for much less time. The same goes for database products such as Oracle. 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 middleware to the operating system and 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 products that are very large, very widely used, or both (as with operating systems and databases) are particularly likely to have security bugs found and publicized. Systems at all levels are also vulnerable to environmental changes which undermine the assumptions used in their design.

The main function of access control 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 Windows, which are similar in many respects, though Windows is more expressive. Database products are often more expressive still (and thus even harder to implement securely.) 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 push back on the number of implementation errors, such as stack overflow attacks; but new attacks are being found continually, and the overall dependability of large software systems improves only slowly.

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. However, the basic mechanisms (and their problems) are pervasive.