Chapter 6

Access Control

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

Optimisation consists of taking something that works and replacing it with something that almost works but is cheaper.
– ROGER NEEDHAM

6.1 Introduction

I first learned to program on an IBM mainframe whose input was punched cards and whose output was a printer. You queued up with a deck of cards, ran the job, and went away with printout. All security was physical. Then along came machines that would run more than one program at once, and the *protection problem* of preventing one program from interfering with another. You don't want a virus to steal the passwords from your browser, or patch a banking application so as to steal your money. And many reliability problems stem from applications misunderstanding each other, or fighting with each other. But it's tricky to separate applications when the customer wants them to share data. It would make phishing much harder if your email client and browser ran on separate machines, so you were unable to just click on URLs in emails, but that would make life too hard.

From the 1970s, access control became the centre of gravity of computer security. It's 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. It's become horrendously complex. If you start out by leafing through the 7000-plus pages of Arm's architecture reference manual or the equally complex arrangements for

Windows at the O/S level, your first reaction might be 'I wish I'd studied music instead!' In this chapter I try to help you make sense of it all.

Access control works at a number of different levels, including at least:

1. Access controls at the application level may express a very rich, domain-specific security policy. The call centre staff in a bank are typically not allowed to see your account details until you have answered a couple of security questions; this not only stops outsiders impersonating you, but also stops the bank staff looking up the accounts of celebrities, or their neighbours. Some transactions might also require approval from a supervisor. And that's nothing compared with the complexity of the access controls on a modern social networking site, which will have a thicket of rules about who can see, copy, and search what data from whom, and privacy options that users can set to modify these rules.

2. The applications may be written on top of middleware, such as a web browser, a bank's bookkeeping system or a social network's database management system. These enforce a number of protection properties. For example, bookkeeping systems ensure that a transaction that debits one account must credit another for the same amount, so that money cannot be created or destroyed, and allow the system's state at some past time to be reconstructed.

3. As the operating system constructs resources such as files and communications ports from lower level components, it has to provide ways to control access to them. Your Android phone treats apps written by different companies as different users and protects their data from each other. The same happens when a shared server separates the VMs, containers or other resources belonging to different users.

4. Finally, the operating system relies on hardware protection provided by the processor and its associated memory-management hardware, which control which memory addresses a given process or thread 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. And we find the same access-control functions being implemented at multiple layers. For example, the separation between different phone apps that is provided by Android is mirrored in your browser which separates web page material according to the domain name it came from (though this separation is often less thorough). And the access controls built at the application layer or the middleware layer may largely duplicate access controls in the underlying operating system or hardware. It can get very messy, and to make sense of it we need to understand the underlying principles, the common architectures, and how they have evolved.

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, as well as inspiring similar mechanisms at higher layers. They are often described as

discretionary access control (DAC) mechanisms, which leave protection to the machine operator, or *mandatory access control* (MAC) mechanisms which are typically under the control of the vendor and protect the operating system itself from being modified by malware. I'll give an introduction to software attacks and techniques for defending against them – MAC, ASLR, sandboxing, virtualisation and what can be done with hardware. Modern hardware not only provides CPU support for virtualisation and capabilities, but also hardware support such as TPM chips for trusted boot to stop malware being persistent. These help us tackle the toxic legacy of the old single-user PC operating systems such as DOS and Win95/98 which let any process modify any data, and constrain the many applications that won't run unless you trick them into thinking that they are running with administrator privileges.

6.2 Operating system access controls

The access controls provided with an operating system typically authenticate principals using a mechanism such as passwords or fingerprints in the case of phones, or passwords or security protocols in the case of servers, then authorise access to files, communications ports and other system resources.

Access controls can often be modeled as 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

Fig. 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 don't 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	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	r	r
Alice	rx	x	–	–
Accounts program	rx	rx	rw	w
Bob	rx	r	r	r

Fig. 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 that share access to the same resources.

Access control matrices (whether in two or three dimensions) can be used to implement protection mechanisms as well as just model them. But they don't scale well: a bank with 50,000 staff and 300 applications would have a matrix of 15,000,000 entries, which might not only impose a performance overhead but also be vulnerable to administrators' mistakes. We will need a better way of storing and managing this information, and the two main options are to compress the users and to compress the rights. With the first, we can use groups or roles to manage large sets of users simultaneously, while with the second we may store the access control matrix either by columns (access control lists) or rows (capabilities, also known as 'tickets' to protocol engineers and 'permissions' on mobile phones) [1326, 1631].

6.2.1 Groups and Roles

When we look at large organisations, we usually find that most staff fit into one of a small number of categories. A bank might have 40 or 50: teller, call centre operator, loan officer and so on. Only a few dozen people (security manager, chief foreign exchange dealer, ...) will need personally customised access rights.

So we need to design a set of groups, or functional roles, to which staff can be assigned. Some vendors (such as Microsoft) use the words *group* and *role* almost interchangeably, but a 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. 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 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 their 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, even an ordinary seaman may 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.

6.2.2 Access control lists

The traditional way to simplify 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.

User	Accounting Data
Sam	rw
Alice	rw
Bob	r

Fig. 4.4 – access control list (ACL)

ACLs have a number of advantages and disadvantages as a means of managing security state. They are a natural choice in environments where users manage their own file security, and became widespread in Unix systems from the 1970s. They are the basic access control mechanism in Unix-based systems such as Linux and Apple's macOS, as well as in derivatives such as Android and iOS. The access controls in Windows were 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 for 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 makes it tedious to find all the files to which a user has access. Verifying that no files have been left world-readable or even world-writable could involve checking ACLs on millions of user files; this is a real issue for large complex firms. Although you can write a script to check whether any file on a server has ACLs that breach a security policy, you can be tripped up by technology changes; the move to containers has led to many corporate data exposures as admins forgot to check the containers' ACLs too. (The containers themselves are often dreadful as it's a new technology being sold by dozens of clueless startups.) And revoking the access of an employee

who has just been fired will usually have to be done by cancelling their password or authentication token.

Let's look at an important example of ACLs – their implementation in Unix (plus its derivatives Android, MacOS and iOS).

6.2.3 Unix operating system security

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

```
drwxrwxrwx Alice Accounts
```

In our first example in Figure 4.2, the ACL of file 3 would be:

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

This records that the file is simply a file rather than a directory; that the file owner can read and write it; that group members (including Bob) can read it but not write it; that non-group members have no access at all; that the file owner is Alice; and that the group is Accounts.

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 in systems with discretionary access control.

This means that the system administrator can do anything, so we have difficulty implementing an audit trail as a file that they cannot modify. In our example, Sam could tinker with the accounts, and have difficulty defending himself if he were falsely accused of tinkering; what's more, a hacker who managed to become the administrator could remove all evidence of his intrusion. The traditional, and still the 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, it means sending it to another machine, or even to a third-party service. Increasingly, it may also involve mandatory access control, as we discuss later.

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 the file representing that program 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 that the accounts program needs.

But when you take an access control problem that has three dimensions – (user, program, data) – and implement it using two-dimensional mechanisms, the outcome is much less intuitive than triples and people are liable to make mistakes. 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 shocking security holes. The responsibility for making access control decisions is moved from the operating system environment to the application program, and most programmers are insufficiently experienced 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 could manipulate this in unexpected ways.)

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`. Managing stateful access rules is difficult; they can complicate the revocation of users who have just been fired, as it can be hard to track down the files they've opened, and stuff can get stuck.

Fourth, the Unix ACL only names one user. If a resource will be used by more than one of them, and you want to do access control at the OS level, you have a couple of options. With older systems you had to use groups; newer systems implement the Posix system of extended ACLs, which may contain any number of named user and named group entities. In theory, the ACL and `suid` mechanisms can often be used to achieve the desired effect. In practice, programmers are often in too much of a hurry to figure out how to do this, and security interfaces are usually way too fiddly to use. So people design their code to require much more privilege than it strictly ought to have, as that seems to be the only way to get the job done.

6.2.4 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

Fig. 4.5 – a capability

The strengths and weaknesses of capabilities are roughly 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 becomes more tricky as it can be hard to find out which users have access. This can be tiresome when we have to investigate an incident or prepare evidence. In fact, scalable systems end up using de-facto capabilities internally, as instant system-wide revocation is just too expensive; in Unix, file descriptors are really capabilities, and continue to grant access for some time even after ACL permissions or even file owners change. In a distributed Unix, access may persist for the lifetime of Kerberos tickets.

Could we do away with ACLs entirely then? People built experimental machines in the 1970s that used capabilities throughout [1631]; the first commercial product was the Plessey System 250, a telephone-switch controller [1275]. The IBM AS/400 series systems brought capability-based protection to the mainstream computing market in 1988, and enjoyed some commercial success. The public key certificates used in cryptography are in effect capabilities, and became mainstream from the mid-1990s. Capabilities have started to supplement ACLs in operating systems, including more recent versions of Windows, FreeBSD and iOS, as I will describe later.

In some applications, they can be the natural way to express security policy. For example, a hospital may have access rules like 'a nurse shall have access to all the patients who are on his or her ward, or who have been there in the last 90 days'. In early systems based on traditional ACLs, each access control decision required a reference to administrative systems to find out which nurses and which patients were on which ward, when – but this made both the HR system and the patient administration system safety-critical, which hammered reliability. Matters were fixed by giving nurses ID cards with certificates that entitle them to access the files associated with a number of wards or hospital departments [442, 443]. If you can make the trust relationships in systems mirror the trust relationships in that part of the world you're trying to automate, you should. Working with the grain can bring advantages at all levels in the stack, making things more usable, supporting safer defaults, cutting errors, reducing engineering effort and saving money too.

6.2.5 DAC and MAC

In the old days, anyone with physical access to a computer controlled all of it: you could load whatever software you liked, inspect everything in memory or on disk and change anything you wanted to. This is the model behind *discretionary access control* (DAC): you start your computer in supervisor mode and then, as the administrator, you can make less-privileged accounts available for less-trusted tasks – such as running apps written by companies you don't entirely trust, or giving remote logon access to others. But this can make things hard to manage at scale, and in the 1970s the US military started a huge computer-security research program whose goal was to protect classified information: to ensure that a file marked 'Top Secret' would never be made available to a user with only a 'Secret' clearance, regardless of the actions of any ordinary user or even of the supervisor. In such a *multilevel secure* (MLS) system, the sysadmin is no longer the boss: ultimate control rests with a remote government authority that sets security policy. The mechanisms started to be described as *mandatory*

access control (MAC). The supervisor, or root access if you will, is under remote control. This drove development of technology for mandatory access control – a fascinating story, which I tell in Part 2 of the book.

From the 1980s, safety engineers also worked on the idea of *safety integrity levels*; roughly, that a more dependable system must not rely on a less dependable one. They started to realise they needed something similar to multilevel security, but for safety. Military system people also came to realise that the tamper-resistance of the protection mechanisms themselves was of central importance. In the 1990s, as computers and networks became fast enough to handle audio and video, the creative industries lobbied for *digital rights management* (DRM) in the hope of preventing people undermining their business models by sharing music and video. This is also a form of mandatory access control – stopping a subscriber sharing a song with a non-subscriber is in many ways like stopping a Top Secret user sharing an intelligence report with a Secret user.

In the early 2000s, these ideas came together as a number of operating-system vendors started to incorporate ideas and mechanisms from the MAC research programme into their products. The catalyst was an initiative by Microsoft and Intel to introduce cryptography into the PC platform to support DRM. Intel believed the business market for PCs was saturated, so growth would come from home sales where, they believed, DRM would be a requirement. Microsoft started with DRM and then realised that offering rights management for documents too might be a way of locking customers tightly into Windows and Office. They set up an industry alliance, now called the Trusted Computing Group, to introduce cryptography and MAC mechanisms into the PC platform. To do this, the operating system had to be made tamper-resistant, and this is achieved by means of a separate processor, the Trusted Platform Module (TPM), basically a smartcard chip mounted on the PC motherboard to support trusted boot and hard disk encryption. The TPM monitors the boot process, and at each stage a hash of everything loaded so far is needed to retrieve the key needed to decrypt the next stage. The real supervisor on the system is now no longer you, the machine owner – it's the operating-system vendor.

MAC, based on TPMs and trusted boot, was used in Windows 6 (Vista) from 2006 as a defence against persistent malware[1]. The TPM standards and architecture were adapted by other operating-system vendors and device OEMs, and there is now even a project for an open-source TPM chip, OpenTitan, based on Google's product. However the main purpose of such a design, whether its own design is open or closed, is to lock a hardware device to using specific software.

[1]Microsoft had had more ambitious plans; its project Palladium would have provided a new, more trusted world for rights-management apps, alongside the normal one for legacy software. They launched Information Rights Management – DRM for documents – in 2003 but corporates didn't buy it, seeing it as a lock-in play. A two-world implementation turned out to be too complex for Vista and after two separate development efforts it was was abandoned; but the vision persisted from 2004 in Arm's TrustZone, which I discuss below.

6.2.6 Apple's macOS

Apple's macOS operating system (formerly called OS/X or Mac OS X) 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 without having to reboot the system. On top of this Unix core are a number of graphics components, 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, macOS 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. If you are such a user, you can install programs (you are asked for the root password when you do so). Since version 10.5 (Leopard), it has been based on TrustedBSD, a variant of BSD that incorporates mandatory access control mechanisms, which are used to protect core system components against tampering.

6.2.7 iOS

Since 2008, Apple has led the smartphone revolution with the iPhone, which (along with other devices like the iPad) uses the iOS operating system. iOS is based on Unix; Apple took the Mach kernel from CMU and fused it with the FreeBSD version of Unix, making a number of changes for performance and robustness. For example, in vanilla Unix a filename can have multiple pathnames that lead to an inode representing a file object, which is what the operating system sees; in iOS, this has been simplified so that files have unique pathnames, which in turn are the subject of the file-level access controls. Again, there is a MAC component, where mechanisms from Domain and Type Enforcement (DTE) are used to tamper-proof core system components; we'll discuss DTE in more detail in chapter 9.

Apps also have *permissions*, which are capabilities; they request a capability to access device services such as the mobile network, the phone, SMSes, the camera, and the first time the app attempts to use such a service. This is granted if the user consents[2]. The many device services open up possible side-channel attacks; for example, an app that's denied access to the keyboard could deduce keypresses using the accelerometer and gyro. We'll discuss side channels in Part 2, in the chapter on that subject.

The Apple ecosystem is closed in the sense that an iPhone will only run apps that Apple has signed[3]. This enables the company to extract a share of app

[2]The trust-on-first-use model goes back to the 1990s with the Java standard J2ME, popularised by Symbian, and the Resurrecting Duckling model that came into use about the same time. J2ME also supported trust-on-install and more besides. When Apple and Android came along, they initially made different choices. In each case, having an app store was a key innovation; Nokia failed to realise that this was important to get a two-sided market going. The app store does some of the access control by deciding what apps can run. This is hard power in Apple's case, and soft power in Android's; we'll discuss this in the chapter on phones.

[3]There are a few exceptions: corporates can get signing keys for internal apps, but these

revenue, and also to screen apps for malware or other undesirable behaviour, such as the exploitation of side channels to defeat access controls. The mechanisms that protect against malware (and also against users rooting their phones to run unauthorised software) include the encryption of memory; each phone has a 256-bit AES key burned into fusible links on the system-on-chip, and this is used to encrypt data kept in the phone's flash memory, to protect not just files but mechanisms such as the PIN retry counter[4].

Up to iPhone 5, Apple relied on TrustZone to protect biometrics and payment data; since then, they have their own secure enclave, which was added in the A7 processor from the iPhone 5S, when you could start to use your fingerprint to authorise payments. By implication, they decided to trust neither iOS nor TrustZone to protect such sensitive data. In addition, with file system encryption, iOS doesn't see the keys used to encrypt files; these are in the secure enclave, which also manages upgrades and prevents rollbacks. Such public information as there is can be found in the iOS Security white paper [107].

The security of mobile devices is a rather complex issue, involving not just access controls and tamper resistance, but the whole ecosystem – from the provision of SIM cards through the operation of app stores to the culture of how people use devices, how businesses try to manipulate them and how government agencies spy on them. I will discuss this in detail in the chapter on phones in Part 2.

6.2.8 Android

Android is the world's most widely used operating system, with 2.5 billion active Android devices in May 2019, according to Google's figures (there are just over a billion PCs and just under a billion iOS devices). Android is based on Linux; apps from different vendors run under different userids. The Linux mechanisms control access at the file level, preventing one app from reading another's data and exhausting shared resources such as memory and CPU. As in iOS, apps have *permissions*, which are in effect capabilities: they grant access to device services such as SMSes, the camera and the address book.

Apps come in signed packages, as .apk files, and while iOS apps are signed by Apple, the verification keys for Android come in self-signed certificates and function as the developer's name. This supports integrity of updates while maintaining an open ecosystem. Each package contains a manifest that demands a set of permissions, and users have to approve the 'dangerous' ones – roughly, those that can spend money or compromise personal data. In early versions of Android, the user would have to approve the lot on installation or not run the app. But experience showed that most users would just click on anything to get through the installation process, and you found even flashlight apps demanding access to your address book, as they could sell it for money. So Android 6 moved to the Apple model of trust on first use; apps compiled for earlier versions still demand capabilities on installation.

can be blacklisted if abused.

[4] I'll discuss fusible links in the chapter on tamper resistance, and iPhone PIN retry defeats in the chapter on surveillance and privacy.

Since Android 5, SELinux has been used to harden the operating system with mandatory access controls, so as not only to protect core system functions from attack but also to separate processes strongly and log violations. SELinux was developed by the NSA to support MAC in government systems; we'll discuss it further in chapter 9.

As with iOS (and indeed Windows), the security of Android is a matter of the whole ecosystem, not just of the access control mechanisms. The new phone ecosystem is sufficiently different from the old PC ecosystem, but inherits enough of the characteristics of the old wireline phone system, that I devote a whole chapter to it in Part 2. We'll consider app stores, side channels, the crime ecosystem, state surveillance and everything else there together there.

6.2.9 Windows

The current version of Windows (Windows 10) has a scarily complex access control system, and a quick canter through its evolution may make it easier to understand what's going on.

Early versions of Windows had no access control. A break came with Windows 4 (NT), which was very much like Unix, and was inspired by it, but with some extensions. First, rather than just *read*, *write* and *execute* there were separate attributes for *take ownership*, *change permissions* and *delete*, to support more flexible delegation. 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. The richer syntax lets you arrange matters so that everyday configuration tasks, such as installing printers, don't have to require full administrator privileges.

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 HR, while assets such as servers and printers may be in resource domains under departmental control; individual workstations may even be administered by their users. Things can be arranged so that the departmental resource domains trust the user domain, but not vice versa – so a hacked or careless departmental administrator can't do too much external damage. The individual workstations would in turn trust the department (but not vice versa) so that users can perform tasks that require local privilege (such as installing software packages). Limiting the damage a hacked administrator can do still needs careful organisation. The data structure used to manage all this, and hide the ACL details from the user interface, is called the *Registry*. Its core used to be the *Active Directory* which managed remote authentication – using either a Kerberos variant or TLS, encapsulated behind the *Security Support Provider Interface* (SSPI) which enables administrators to plug in other authentication services. Active Directory is essentially a database that organises users, groups, machines, and organisational units within a domain in a hierarchical namespace.

It lurked behind Exchange, but is now being phased out as Microsoft becomes a cloud-based company and moves its users to Office365.

Windows has added capabilities in two ways which can override or complement ACLs. First, users or groups can be either whitelisted or blacklisted by means of profiles. Security policy is set by groups rather than for the system as a whole; group policy overrides individual profiles, and can be associated with sites, domains or organisational units, so it can start to tackle complex problems. Policies can be created using standard tools or custom coded.

The second way in which capabilities insinuate their way into Windows is that in many applications, people use TLS for authentication, and TLS certificates provide another, capability-oriented, layer of access control outside the purview of the Active Directory.

I already mentioned that Windows Vista introduced trusted boot to make the operating system itself tamper-resistant, in the sense that it always boots into a known state, limiting the persistence of malware. It added three further protection mechanisms to get away from the previous default of all software running as root. First, the kernel was closed off to developers; second, the graphics subsystem and most drivers were removed from the kernel; and third, *User Account Control* (UAC) replaced the default administrator privilege with user defaults instead. Previously, so many routine tasks needed administrative privilege that many enterprises made all their users administrators, which made it difficult to contain malware; and many developers wrote their software on the assumption that it would have access to everything (for a hall of shame, see [694]). According to Microsoft engineers, this was a major reason for Windows' lack of robustness: applications monkey with system resources in incompatible ways. So they added an Application Information Service that launches applications which require elevated privilege and uses virtualisation to contain them: if they modify the registry, for example, they don't modify the 'real' registry but simply the version of it that they can see.

Since Vista, the desktop acts as the parent process for later user processes, so even administrators browse the web as normal users, and malware they download can't overwrite system files unless given later authorisation. When a task requires admin privilege, the user gets an *elevation prompt* asking them for an admin password. (Apple's macOS is similar although the details under the hood differ somewhat.) As admin users are often tricked into installing malicious software, Vista added mandatory access controls in the form of file integrity levels. 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) in the absence of some trusted process (such as verification of a signature by Microsoft on the code in question).

In 2012, Windows 8 added *dynamic access control* which lets you control user access by context, such as their work PC versus their home PC and their phone; this is done via account attributes in Active Directory, which appear as claims about a user, or in Kerberos tickets as claims about a domain. In 2016, Windows 8.1 added a cleaner abstraction with *principals*, which can be a user, computer, process or thread running in a security context or a group to which such a principal belongs, and *security identifiers* (SIDs) which represent such

principals. When a user signs in, they get tickets with the SIDs to which they belong. Windows 8.1 also prepared for the move to cloud computing by adding *Microsoft accounts* (formerly LiveID), whereby a user signs in to a Microsoft cloud service rather than to a local server. Where credentials are stored locally, it protects them using virtualisation. Finally, Windows 10 added a number of features to support the move to cloud computing with a diversity of client devices, ranging from certificate pinning (which we'll discuss in the chapter on Network Security) to the abolition of the old secure attention sequence ctrl-alt-del (which is hard to do on touch-screen devices and which users didn't understand anyway).

To sum up, Windows evolved to provide a richer and more flexible set of access control tools than any system previously sold in mass markets. It was driven by corporate customers who need to manage tens of thousands of staff performing hundreds of different job roles across hundreds of different sites, providing internal controls to limit the damage that can be done by small numbers of dishonest staff or infected machines. (How such controls are actually designed will be our topic in the chapter on Banking and Bookkeeping.) The driver for this development was the fact that Microsoft made over half of its revenue from firms that licensed more than 25,000 seats; but the cost of the flexibility that corporate customers demanded is complexity. Setting up access control for a big Windows shop is a highly skilled job.

6.2.10 Middleware

Doing access control at the level of files and programs was fine in the early days of computing, when these were the resources that mattered. Since the 1980s, growing scale and complexity has led to access control being done at other levels instead of (or as well as) at the operating system level. For example, bookkeeping systems often run on top of a database product such as Oracle, which looks to the operating system as one large file. So most of the access control has to be done in the database; all the operating system supplies may be an authenticated ID for each user who logs on. And since the 1990s, a lot of the work at the client end has been done by the web browser.

6.2.10.1 Database access controls

Before people started using websites for shopping, database security was largely a back-room concern. But enterprises now have critical databases to handle inventory, dispatch and e-commerce, fronted by web servers that pass transactions to the databases directly. These databases now contain much of the data that matter to our lives – bank accounts, vehicle registrations and employment records – and failures sometimes expose them to random online users.

Database products, such as Oracle, DB2 and MySQL, have their own access control mechanisms, which are 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 comparable in complexity with Windows; modern databases are intrinsically complex, as are the things they support – typically

business processes involving higher levels of abstraction than files or domains. There may be access controls aimed at preventing any user learning too much about too many customers; these tend to be stateful, and may deal with possible statistical inference rather than simple yes-no access rules. I devote a whole chapter in Part 2 to exploring the topic of Inference Control.

Ease of administration is often a bottleneck. In companies I've advised, the operating-system and database access controls have been managed by different departments, which don't talk to each other; and often IT departments have to put in crude hacks to make the various access control systems seem to work as one, but which open up serious holes.

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 convenient to use the latter, to save the effort of synchronising with what other departments are doing. In many installations, the database is accessible directly from the outside; and even where it's shielded by a web service front-end, this often contains loopholes that let SQL code be inserted into the database.

Database security failures can thus cause problems directly. The Slammer worm in 2003 propagated itself using a stack-overflow exploit against Microsoft SQL Server 2000 and created large amounts of traffic as compromised machines sent floods of attack packets to random IP addresses.

Just as Windows is tricky to configure securely, because it's so complicated, the same goes for the typical database system. If you ever have to lock one down – or even just understand what's going on – you had better read a specialist textbook, such as [955], or get in an expert.

6.2.10.2 Browsers

The web browser is another middleware platform on which we rely for access control and whose complexity often lets us down. In previous editions of this book, we considered web security to be a matter of how the servers were configured, and whether this led to vulnerabilities such as cross-site scripting. By now there's a growing realisation that we should probably have treated browsers as access control devices all along. After all, the browser is the place on your machine were you run code written by people you don't want to trust and who will occasionally be malicious. Even in the absence of malice, you don't want to have to reboot your browser if it hangs because of a script in one of the tabs. (Chrome tried to ensure this by running each tab in a separate operating-system process.)

The main access control rule is the *same-origin policy* whereby javascript or other active content on a web page is only allowed to communicate with the IP address that it originally came from; such code is run in a *sandbox* to prevent it altering the host system, as I'll describe in section 6.2.11 below. But many things can go wrong. Bugs in browsers are exploited in *drive-by download* attacks, where visiting an attack web page can infect your machine, and even without this the modern web environment is extremely difficult to

control. Many web pages are full of trackers and other bad things, supplied by multiple ad networks and data brokers, which make a mockery of the intent behind the same-origin policy; I'll discuss web security in more detail in 25.3.

6.2.11 Sandboxing

The late 1990s saw the emergence of yet another type of access control: the software *sandbox*, introduced by Sun with its Java programming language. The model is that a user wants to run some code that she has downloaded as an applet, but is concerned that the applet might do something nasty, such as stealing her address book and mailing it off to a marketing company, or just hogging the CPU and running down the battery.

The designers of Java tackled this problem by providing a 'sandbox' – a restricted environment in which the code 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 (the *same-origin policy*). This is enforced by having the code executed by an interpreter – the Java Virtual Machine (JVM) – with only limited access rights [637]. This idea was adapted to javascript, the main scripting language used in web pages, though it's actually a different language; and other active content too. A version of Java is also used on smartcards so they can support applets written by different firms.

6.2.12 Virtualisation

Virtualisation is what powers cloud computing; it enables a single machine to emulate a number of machines independently, so that you can rent a *virtual machine* (VM) in a data centre for a few tens of dollars a month rather than having to pay maybe a hundred for a whole server. Virtualisation was invented in the 1960s by IBM [413]; a single machine could be partitioned using VM/370 into multiple virtual machines. Initially this was about enabling a new mainframe to run legacy apps from several old machine architectures; it soon became normal for a company that bought two computers to use one for its production environment and the other as a series of logically separate machines for development, testing, and minor applications. It's not enough to run a virtual machine monitor (VMM) on top of a host operating system, and then run other operating systems on top; you have to deal with sensitive instructions that reveal processor state such as absolute addresses and the processor clock. Working VMMs appeared for Intel platforms with VMware ESX Server in 2003 and (especially) Xen in 2003, which accounted for resource usage well enough to enable AWS and the cloud computing revolution. Things can be done more cleanly with processor support, which Intel has provided since 2006 with VT-x, and whose details I'll discuss below. VM security claims rest to some extent on the argument that a VMM hypervisor's code can be much smaller than an operating system and thus easier to code-review and secure; whether there are actually fewer vulnerabilities is of course an empirical question [1275].

At the client end, virtualisation allows people to run a guest operating system on top of a host (for example, Windows on top of macOS), which offers not just

flexibility but the prospect of better containment. For example, an employee might have two copies of Windows running on their laptop – a locked-down version with the office environment, and another for use at home. Samsung offers Knox, which creates a virtual machine on a mobile phone that an employer can lock down and manage remotely, while the user enjoys a normal Android as well on the same device.

But using virtualisation to separate security domains on clients is harder than it looks. People need to share data between multiple VMs and if they use ad-hoc mechanisms, such as USB sticks and webmail accounts, this undermines the separation. Safe data sharing is far from trivial. For example, Bromium[5] offers VMs tailored to specific apps on corporate PCs, so you have one VM for Office, one for Acrobat reader, one for your browser and so on. This enables firms to work reasonably securely with old, unsupported software. So how do you download an Office document? Well, the browser exports the file from its VM to the host hard disc, marking it 'untrusted', so when the user tries to open it they're given a new VM which holds that document plus Office and nothing else. When they then email this untrusted document, there's an Outlook plugin that stops it being rendered in the 'sent mail' pane. Things get even more messy with network services integrated into apps; the rules on what sites can access which cookies are complicated, and it's hard to deal with single signon and workflows that cross multiple domains. The clipboard also needs a lot more rules to control it. Many of the rules change from time to time, and are heuristics rather than hard, verifiable access logic. In short, using VMs for separation at the client requires deep integration with the OS and apps if it's to appear transparent to the user, and there are plenty of tradeoffs made between security and usability. In effect, you're retrofitting virtualisation on to an existing OS and apps that were not built for it.

Containers have been the hot new topic in the late 2010s. They evolved as a lightweight alternative to virtualisation in cloud computing and are often confused with it, especially by the marketing people. My definition is that while a VM has a complete operating system, insulated from the hardware by a hypervisor, a container is an isolated guest process that shares a kernel with other containers. Container implementations separate groups of processes by virtualising a subset of operating-system mechanisms, including process identifiers, interprocess communication, and namespaces; they also use techniques such as sandboxing and system call filtering. The business incentive is to minimise the guests' size, their interaction complexity and the costs of managing them, so they are deployed along with orchestration tools. Like any other new technology, there are many startups with more enthusiasm than experience. A 2019 survey by Jerry Gamblin disclosed that of the top 1000 containers available to developers on Docker Hub, 194 were setting up blank root passwords [**?**]. If you're going to use cloud systems, you need to pay serious attention to your choice of tools, and also learn yet another set of access control mechanisms – those offered by the service provider, such as the Amazon AWS Identity and Access Management (IAM). This adds another layer of complexity, which people can get wrong. For example, in 2019 a security firm providing biometric identification services to banks and the police left its entire database unprotected;

[5]Now owned by HP

two researchers found it using Elasticsearch and discovered millions of people's photos, fingerprints, passwords and security clearance levels on a database that they could not only read but write [1509].

But even if you tie down a cloud system properly, there are hardware limits on what the separation mechanisms can achieve. In 2018, two classes of powerful side-channel attacks were published: Meltdown and Spectre, which I discuss in the following section and at greater length in the chapter on side channels. These Banks that use containers to deploy payment processing rely, at least implicitly, on their containers being difficult to target in a cloud the size of Amazon's or Google's. For a comprehensive survey of the evolution of virtualisation and containers, see Randal [1275].

6.3 Hardware Protection

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

Preventing one process from interfering with another is the *protection problem*. The *confinement problem* is 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.

Unless one uses sandboxing techniques (which are too restrictive for general programming environments), solving the protection 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 to allow interprocess communication; but programs must be protected from accidental or deliberate modification, and 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 classic mechanism is *segment addressing*. Memory is addressed by two registers, a segment register that points to a segment of memory, and an address register that points to a location within that segment. The segment registers are controlled by the operating system, often by a component of it called the *reference monitor* which links the access control mechanisms with the hardware.

The implementation has become more complex as 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 write to 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 depended 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 1960s and which inspired 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 [1368]. Many of its features have been adopted in more recent processors.

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 (e.g. a device driver). The mechanisms for doing this need to be designed with care, or security bugs can be expected. Also, performance may depend quite drastically on whether routines at different privilege levels are called by reference or by value [1368].

6.3.1 Intel processors

The Intel 8088/8086 processors used in early PCs had no distinction between system and user mode, and thus any running program controlled the whole machine[6]. The 80286 added protected segment addressing and rings, so for the first time a PC 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 additional instructions such as 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* that allow execution of code at a different privilege level and manage the supporting infrastructure, such as multiple stack segments.

From 2006, Intel added hardware support for x86 virtualisation, known as Intel VT, which helped drive the adoption of cloud computing. Some processor architectures such as S/370 and PowerPC are easy to virtualise, and the theoretical requirements for this had been established in 1974 by Gerald Popek and Robert Goldberg [1240]; they include that all sensitive instructions that expose raw processor state be privileged instructions. The native Intel instruction set, however, has sensitive user-mode instructions, requiring messy workarounds such as application code rewriting and patches to hosted operating systems. Adding VMM support in hardware means that you can run the VMM in a new 'ring -1' or root mode, so you can run an operating system in ring 0 as it was designed. You still have to trap sensitive opcodes, but system calls don't automatically require VMM intervention, you can run unmodified

[6] They had been developed on a crash programme to save market share following the advent of RISC processors and the market failure of the iAPX432.

operating systems, things go faster and systems are generally more robust.

In 2015, Intel released Software Guard eXtensions (SGX), which lets trusted code run in an *enclave* – an encrypted section of the memory – while the rest of the code is executed as usual. The company had worked on such architectures in the early years of the Trusted Computing initiative, but let things slide until it needed an enclave architecture to compete with TrustZone, which I discuss in the next section. The encryption is performed by a Memory Encryption Engine (MEE), while SGX also introduces new instructions and memory-access checks to ensure non-enclave processes cannot access enclave memory (not even root processes). SGX has been promoted for DRM and securing cloud VMs, particularly those containing crypto keys, credentials or sensitive personal information; this is under threat from Spectre and similar attacks, which I discuss in detail in the chapter on side channels. Another drawback used to be that SGX code had to be signed by Intel. The company has now delegated signing (so bad people can get code signed) and from SGXv2 will open up the root of trust to others. So people are experimenting with SGX malware, which can remain undetectable by anti-virus software. As SGX apps cannot issue syscalls, it had been hoped that enclave malware couldn't do much harm, yet Michael Schwarz, Samuel Weiser and Daniel Gruss have now worked out how to mount stealthy return-oriented programming (ROP) attacks from an enclave on a host app; they argue that the problem is a lack of clarity about what enclaves are supposed to do, and that any reasonable threat model must include untrusted enclaves [1370]. This simple point may force a rethink of enclave architectures; Arm people mutter about doing better with the next release while Intel told me 'In the future, Intel's control-flow enforcement technology (CET) should help address this threat inside SGX'.

As well as the access-control vulnerabilities, there are crypto issues: the compromise of one secret key in any CPU anywhere beaks the whole architecture. This happened in 2019 for AMD's equivalent of SGX [289], and Intel appears vulnerable in the same way. I'll discuss this in the chapter on Advanced Cryptographic Engineering.

6.3.2 Arm processors

The Arm is the processor core most commonly used in phones, tablets and IoT devices; billions have been used in mobile phones alone, with a high-end device having several dozen Arm cores of various sizes in its chipset. The original Arm (which stood for *Acorn Risc Machine*) was the first commercial RISC design; it was released in 1985, just before MIPS. In 1991, Arm became a separate firm which, unlike Intel, does not own or operate any fabs: it licenses a range of processor cores, which chip designers include in their products. Early cores had a 32-bit datapath and contained fifteen registers, of which seven were shadowed by banked registers for system processes to cut the cost of switching context on interrupt. There are multiple supervisor modes, dealing with fast and normal interrupts, the system mode entered on reset, and various kinds of exception handling. The core initially contained no memory management, so Arm-based designs could have their hardware protection extensively customized; there are now variants with *memory protection units* (MPUs), and others with *memory*

management units (MMUs) that handle virtual memory as well.

In 2011, Arm launched version 8, which supports 64-bit processing and enables multiple 32-bit operating systems to be virtualised. Hypervisor support added yet another supervisor mode. The cores come in all sizes, from large 64-bit superscalar processors with pipelines over a dozen stages deep, to tiny ones for cheap embedded devices.

TrustZone is a security extension that supports the 'two worlds' model mentioned above; in fact, it's the earliest such extension, having been made available secretly to mobile phone makers in 2004. Phones were the 'killer app' for enclaves as operators wanted to lock subsidised phones and regulators wanted to make the baseband software that controls the RF functions to be tamper-resistant. TrustZone supports an open world for a normal operating system and general-purpose applications, plus a closed enclave to handle sensitive operations such as cryptography and critical I/O (in a mobile phone, this can include the SIM card and the fingerprint reader). Whether the processor is in a secure or non-secure state is orthogonal to whether it's in user mode or a supervisor mode (though it must choose between secure and hypervisor mode). The closed world hosts a single *trusted execution environment* (TEE) with separate stacks, a simplified operating system, and typically runs only trusted code signed by the OEM – although Samsung's Knox, which sets out to provide 'home' and 'work' environments on your mobile phone, allows regular rich apps to execute in the secure environment.

Although TrustZone was released in 2004, it was kept closed until 2015; OEMs used it to protect their own interests and didn't open it up to app developers, except occasionally under NDA. That is slowly changing, but even firms that write banking apps are reluctant to adopt it: it's hard enough to make an app run robustly on enough versions of Android without also having to cope with multiple customised versions of TrustZone. It's also hard to assess security claims that vendors make about closed platforms. For the gory details, see Sandro Pinto and Nuno Santos [1221]. As with Intel SGX, there appears to be no way yet to deal with malicious enclave apps, which might come bundled as DRM with gaming apps or be mandated by authoritarian states; and, as with Intel SGX, enclave apps created with TrustZone can raise issues of transparency and control, which can spill over into auditability, privacy and much else. Again, company insiders mutter 'wait and see'; no doubt we shall.

Arm's latest offering is CHERI[7] which adds fine-grained capability support to Arm CPUs. At present, browsers such as Chrome put tabs in different processes, so that one webpage can't slow down the other tabs if its scripts run slowly. It would be great if each object in each web page could be sandboxed separately, but this isn't possible because of the large cost, in terms of CPU cycles, of each inter-process context switch. CHERI enables a process spawning a subthread to allocate it read and write accesses to specific ranges of memory, so that multiple sandboxes can run in the same process. This was announced as a product in 2018 and we expect too see silicon in 2021.

[7]Full disclosure: this was developed by a team of my colleagues at Cambridge and elsewhere, led by Robert Watson.

6.4 What Goes Wrong

Popular operating systems such as Android, Linux and Windows are very large and complex, with their features tested daily by billions of users under very diverse circumstances. Many bugs are found, some of which give rise to vulnerabilities, which have a typical lifecycle. After discovery, a bug is reported to a CERT or to the vendor; a patch is shipped; the patch is reverse-engineered, and an exploit may be produced; and people who did not apply the patch in time may find that their machines have been compromised. In a minority of cases, the vulnerability is exploited at once rather than reported – 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 this 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 they could take over the system completely. The first step might have involved guessing, or social-engineering, a password, and then using an operating-system bug to escalate from user to root [922].

The user/root distinction became less important in the twenty-first century for two reasons. First, Windows PCs were the most common attack targets, and ran many applications as administrator, so any application that could be compromised gave administrator access. Second, attackers come in two basic types: targeted attackers, who want to spy on a specific individual and whose goal is typically to acquire access to that person's accounts; and scale attackers, whose goal is typically to compromise large numbers of PCs, which they can organise into a botnet in order to make money. This, too, doesn't require administrator access. Even if your mail client does not run as administrator, it can still be useful to a spammer who takes control.

However, botnet herders do prefer to install *rootkits* which, as their name suggests, run as root; they are also known as *remote access trojans* or RATs. The user/root distinction does still matter in business environments, where you do not want such a kit installed as an *advanced persistent threat* by a hostile intelligence agency, or corporate espionage firm, or by a crime gang doing reconnaissance to set you up for a large fraud.

A separate distinction is whether an exploit is *wormable* – whether it can be used to spread malware quickly online from one machine to another without human intervention. The Morris worm was the first large-scale case of this, and there have been many since. I mentioned Wannacry and NotPetya in chapter 2; these used a vulnerability developed by the NSA and then leaked to other state actors. Operating system vendors react quickly to wormable exploits, typically releasing out-of-sequence patches, because of the scale of the damage they can do. The most troublesome wormable exploits at the time of writing are variants of Mirai, a worm used to take over IoT devices that use known root passwords. This appeared in October 2016 to exploit CCTV cameras, and hundreds of versions have been produced since, adapted to take over different vulnerable devices and recruit them into botnets. Wormable exploits often use root access but don't have to; it is sufficient that the exploit be capable of automatic onward

transmission[8].

In any case, the basic types of technical attack have not changed hugely in a generation and I'll now consider them briefly.

6.4.1 Smashing the stack

The classic software exploit is the memory overwriting attack, colloquially known as 'smashing the stack', as used by the Morris worm in 1988; this infected so many Unix machines that it disrupted the Internet and brought malware forcefully to the attention of the mass media [1462]. Attacks involving violations of memory safety accounted for well over half the exploits against operating systems in the late 1990s and early 2000s [404] but the proportion has been dropping slowly since then.

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. The classic example, used in the Morris worm, was a vulnerability in the Unix `finger` command. A common 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 overwriting the stack and being executed by the system.

The usual exploit technique was 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 didn't change the control flow, and whose task was to catch the processor if it executed any of them. The landing pad delivered the processor to the attack code which will do something like creating a shell with administrative privilege directly (see Figure 6.6).

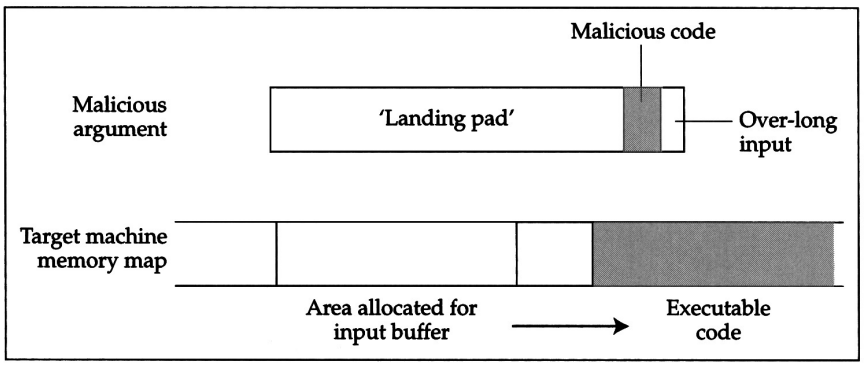

Figure 6.6: – stack smashing attack

Stack-overwriting attacks were not invented in 1988. Most of the early 1960s time-sharing systems suffered from this vulnerability, and fixed it [652]. Pen-

[8]In rare cases even human transmission can make malware spread quickly: an example was the ILoveYou worm which spread itself in 2000 via an email with that subject line, which caused enough people to open it, running a script that caused it to be sent to everyone in the new victim's address book.

etration testing in the early '70s showed that one of the most frequently-used attack strategies was still "unexpected parameters" [948]. 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. Stack overwriting attacks have been found against all sorts of programmable devices – even against things like smartcards and hardware security modules, whose designers really should have known better.

6.4.2 Other technical attacks

Many vulnerabilities are variations on the same general theme, in that they occur when data in grammar A is interpreted as being code 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. These are failures of *type safety*. In fact, a stack overflow can be seen either as a memory safety failure or as a failure to sanitise user input, but there are purer examples of each type.

The *use after free* type of safety failure is now the most common cause of remote execution vulnerabilities and has provided a lot of attacks on browsers in recent years. It can happen when a chunk of memory is freed and then still used, perhaps because of confusion over which part of a program is responsible for freeing it. If a malicious chunk is now allocated, it may end up taking its place on the heap, and when an old innocuous function is called a new, malicious function may be invoked instead. There are many other variants on the memory safety 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* [1025] for a taxonomy.

SQL injection attacks are the most common attack based on failure to sanitise input, and arise when a careless web developer passes user input to a backend 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. There are similar command-injection problems afflicting other languages used by web developers, such as PHP. The remedy is to treat all user input as suspicious and validate it. Where possible, one should only act on user input in a safe context.

Once such type-safety and input-sanitisation attacks are dealt with, *race conditions* are probably next. These occur when a transaction is carried out in two or more stages, where access rights are verified at the first stage and something sensitive is done at the second. If someone can alter the state in between the two stages, this can lead to an attack. A classic example arose in early versions of Unix, where the command to create a directory, '`mkdir`', 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.

A more modern example arises with the wrappers used in containers to intercept system calls made by applications to the operating system, parse them, and modify them if need be. These wrappers execute in the kernel's address space, inspect the enter and exit state on all system calls, and encapsulate only security logic. They generally assume that system calls are atomic, but modern operating system kernels are highly concurrent. System calls are not atomic with respect to each other; there are many possibilities for two system calls to race each other for access to shared memory, which gives rise to *time-of-check-to-time-of-use* (TOCTTOU) attacks. An early (2007) example calls a path whose name spills over a page boundary by one byte, causing the kernel to sleep while the page is fetched; it then replaces the path in memory [1609]. There have been others since, and as more processors ship in each CPU chip as time passes, and containers become an ever more common way of deploying applications, this sort of attack may become more and more of a problem. Some operating systems have features specifically to deal with concurrency attacks, but this field is still in flux.

A different type of timing attack can come from 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? People can reacquire access rights that were revoked, and play even more subtle tricks.

One attack that has attracted a lot of research effort recently is *return-oriented programming* (ROP) [1386]. Many modern systems try to prevent type safety attacks by *data execution prevention* – marking memory as either code or data, a measure that goes back to the Burroughs 5000; and if all the code is signed, surely you'd think that unauthorised code cannot be executed? Wrong! An attacker can look for *gadgets* – sequences of instructions with some useful effect, ending in a return. By collecting enough gadgets, it's possible to assemble a machine that's Turing powerful, and implement our attack code as a chain of ROP gadgets. Then all one has to do is seize control of the call stack. This evolved from the *return-to-libc attack* which uses the common shared library `libc` to provide well-understood gadgets; many variants have been developed since, including an attack that enables malware in an SGX enclave to mount stealthy attacks on host apps [1370]. The latest attack variant, *block-oriented programming* (BOP), can often generate attacks automatically from crashes discovered by program fuzzing, defeating current control-flow integrity controls [783]. This coevolution of attack and defence will no doubt continue.

Finally there are *side channels*. The most recent major innovation in attack technology targets CPU pipeline behaviour. In early 2018, two game-changing attacks pioneered the genre: *Meltdown*, which exploits side-channels created by out-of-order execution on Intel processors [954], and *Spectre*, which exploits speculative execution on Intel, AMD and Arm processors [871]. The basic idea is that large modern CPUs' pipelines are so long and complex that they look ahead and anticipate the next dozen instructions, even if these are instructions that the current process wouldn't be allowed to execute (imagine the access check is two instructions in the future and the read operation it will forbid is two instructions after that). The path not taken can still load information into a cache and thus leak information in the form of delays. With some cunning,

one process can arrange things to read the memory of another. I will discuss Spectre and Meltdown in more detail in the chapter on side channels. Although mitigations have been published, further attacks of the same general kind keep on being discovered, and it may take several years and a new generation of processors before they are brought entirely under control. It all reminds me of the saying by Roger Needham at the head of this chapter. Optimisation consists of replacing something that works with something that almost works, but is cheaper; and modern CPUs are so heavily optimised that we're bound to see more variants on the Spectre theme. Such attacks limit the protection that can be offered not just by containers and VMs, but also by enclave mechanisms such as TrustZone and SGX. In particular, they may stop careful firms from entrusting high-value cryptographic keys to enclaves and prolong the service life of old-fashioned hardware cryptography.

6.4.3 User interface failures

A common way to attack a fortress is to deceive the guards into helping you, and operating systems are no exception. One of the earliest attacks was the *Trojan Horse*, a program the administrator is invited to run but which contains a nasty surprise. People would write games that checked whether the player was the system administrator, and if so would create another administrator account with a known password. A variant was to write a program with the same name as a common 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. You then complain to the administrator that something's wrong with the directory. When they enter the directory and type `ls` to see what's there, the damage is done. The fix was simple: an administrator's 'PATH' variable (the list of directories to be searched for a suitably-named program when a command is invoked) should not contain '.' (the symbol for the current directory). Modern Unix versions ship with this as a default; but it's still an example of how you have to get lots of little details right for access control to be robust, and these details aren't always obvious in advance.

Perhaps the most serious example of user interface failure, in terms of the number of systems historically attacked, consists of two facts: first, Windows is forever popping up confirmation dialogues, which trained people to click boxes away to get their work done; and second, that until 2006 a user needed to be the administrator to install anything. The idea was that restricting software installation to admins enabled Microsoft's big corporate customers, such as banks and government departments, to lock down their systems so that staff couldn't run games or other unauthorised software. But in most environments, ordinary people need to install software to get their work done. So hundreds of millions of people had administrator privileges who shouldn't have needed them, and installed malicious code when a website simply popped up a box telling them to do something. This was compounded by the many application developers who insisted that their code run as root, either out of laziness or because they wanted to collect data that they really shouldn't have had. Windows Vista started to move away from this, but a malware ecosystem is now well established in the PC world, and one is starting to take root in the Android ecosystem as businesses

pressure people to install apps rather than using websites, and the apps demand access to all sorts of data and services that they really shouldn't have. We'll discuss this later in the chapter on phones.

6.4.4 Remedies

Software security is not all doom and gloom; things got substantially better during the 2000s. At the turn of the century, 90% of vulnerabilties were buffer overflows; by the time the second edition of this book came out in 2008, it was just under half, and now it's even less. Several things made a difference.

1. The first consists of specific defences. *Stack canaries* are a random number inserted by the compiler next to the return address on the stack. If the stack is overwritten, then with high probability the canary will change [404]. *Data execution prevention* (DEP) marks all memory as either data or code, and prevents the former being executed; it appeared in 2003 with Windows XP. *Address space layout randomisation* (ASLR) arrived at the same time; by making the memory layout different in each instance of a system, it makes it harder for an attacker to predict target addresses. This is particularly important now that there are toolkits to do ROP attacks, which bypass DEP. *Control flow integrity* mechanisms involve analysing the possible control-flow graph at compile time and enforcing this at runtime by validating indirect control-flow transfers; this appeared in 2005 and was incorporated in various products over the following decade [299]. However the analysis is not precise, and block-oriented programming attacks are among the tricks that have evolved to exploit the gaps [783].

2. The second consists of better general-purpose tools. Static-analysis programs such as Coverity can find large numbers of potential software bugs and highlight ways in which code deviates from best practice; if used from the start of a project, they can make a big difference. (If added later, they can throw up thousands of alerts that are a pain to deal with.) The radical solution is to use a better language; my colleagues increasingly write systems code in Rust rather than in C or C++[9].

3. The third is better training. In 2002, Microsoft announced a security initiative that involved every programmer being trained in how to write secure code. (The book they produced for this, *'Writing Secure Code'* [755], is still worth a read.) Other companies followed suit.

4. The latest approach is DevSecOps, which I discuss in Part 3. Agile development methodology is extended to allow very rapid deployment of patches and response to incidents; it may enable the effort put into design, coding and testing to be aimed at the most urgent problems.

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

[9]Rust emerged from Mozilla research in 2010 and has been used to redevelop Firefox; it's been voted the most favourite language in the Stack Overflow annual survey from 2016–2019.

security of the product, can make a huge difference. Programs should only have as much privilege as they need: the *principle of least privilege* [1326]. 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, exposing all sorts of interfaces to attacks like SQL injection that just shouldn't happen. These involve failures of incentives, personal and corporate, as well as inadequate education and the poor usability of security tools.

6.4.5 Environmental creep

Many security failures result when environmental change undermines a security model. Mechanisms that worked adequately in an initial environment often fail in a wider 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'. Over the 50 years since Ken Thomson started work on it at Bell Labs in 1969, 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. The Internet mechanisms (telnet, ftp, DNS, SMTP) were originally written for mainframes on a secure network. Mainframes were autonomous, the network was outside the security protocols, and there was no transfer of authorisation. So remote authentication, which the Berkeley model really needed, was simply not supported. The Sun extensions such as NFS added to the party, assuming a single firm with multiple trusted LANs. We've had to retrofit protocols like Kerberos, TLS and SSH as duct tape to hold the world together. The arrival of billions of phones, which communicate sometimes by wifi and sometimes by a mobile network, and which run apps from millions of authors (most of them selfish, some of them actively malicious), has left security engineers running ever faster to catch up.

Mixing many different models of computation together has been a factor in the present chaos. Some of their initial assumptions still apply partially, but none of them apply globally any more. The Internet now has billions of phones, billions of IoT devices, maybe a billion PCs, millions of organisations, and managements which are not just independent but may be in conflict. There are companies that compete; political groups that despise each other, and nation states that are at war with each other. Users, instead of being trustworthy but occasionally incompetent, are now largely unskilled – but some are both capable and hostile. Code used to be simply buggy – but now there is a lot of malicious code out there. Attacks on communications used to be the purview of

intelligence agencies – now they can be done by youngsters who've downloaded attack tools from the net and launched them without any real idea of how they work.

6.5 Summary

Access control mechanisms operate at a number of levels in a system, from the hardware up through the operating system and middleware like browsers to the applications. 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.

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 widely fielded examples are Android and Windows at the client end and Linux at the server end; they have a common lineage and many architectural similarities. The basic mechanisms (and their problems) are pervasive. Most attacks involve the opportunistic exploitation of bugs; products that are complex, widely used, or both are particularly likely to have vulnerabilities found and turned into exploits. Many techniques have been developed to push back on the number of implementation errors, to make it less likely that the resulting bugs give rise to vulnerabilties, and harder to turn the vulnerabilities into exploits; but the overall dependability of large software systems improves only slowly.

Research Problems

Most of the issues in access control were identified by the 1960s or early 1970s and were worked out on experimental systems such as Multics [1368] and the CAP [1631]. Much of the research in access control systems since has involved reworking the basic themes in new contexts, such as mobile phones.

Recent threads of research include enclaves, and the CHERI mechanisms for adding finer-grained access control. Another question is: how will developers use such tools effectively?

In the second edition I predicted that 'a useful research topic for the next few years will be how to engineer access control mechanisms that are not just robust but also usable – by both programmers and end users.' Recent work by Yasemin Acar and others has picked that up and developed it into one of the most rapidly-growing fields of security research [9]. Many if not most technical security failures appear due at least in part to the poor usability of the protection mechanisms that developers are expected to use. I already mention in the chapter on cryptography how crypto APIs often induce people to use really unsafe defaults, such as encrypting long messages with ECB mode; access control is just as bad, as anyone coming cold to the access control mechanisms in a Windows system or either an Intel or Arm CPU will find.

As a teaser, here's a new problem. Can we extend what we know about access control at the technical level – whether hardware, OS or app – to the

organisational level? Is there a more systematic way of thinking about firms' internal controls? In the 20th century, there were a number of security policies proposed, from Bell-LaPadula to Clark-Wilson, which we discuss at greater length in Part 2. Is it time to revisit this for a world of deep outsourcing and virtual organisations, now that we have interesting technical analogues?

Further Reading

There's a history of virtualisation and containers by Allison Randal at [1275], and a reference book for Java security written by its architect Li Gong [637]. The Cloud Native Security Foundation is trying to move people towards better open-source practices around containers and other technologies for deploying and managing cloud-native software. Going back a bit, the classic descriptions of Unix security are by Fred Grampp and Robert Morris in 1984 [653] and by Simson Garfinkel and Eugene Spafford in 1996 [608], while the classic on Internet security by Bill Cheswick and Steve Bellovin [189] gives many examples of network attacks on Unix systems.

Carl Landwehr gives a useful reference to many of the flaws found in operating systems in the 1960s through the 1980s [922]. One of the earliest reports on the subject (and indeed on computer security in general) is by Willis Ware in 1970 [1603]; Butler Lampson's seminal paper on the confinement problem appeared in 1970s [918] and three years later, another influential early paper was written by Jerry Saltzer and Mike Schroeder [1326]. The textbook we get our students to read on access control issues is Dieter Gollmann's *'Computer Security'* [634].

The field of software security is fast-moving; the attacks change significantly (at least in their details) from one year to the next. The classic starting point is Gary McGraw's 2006 book [1025]. Since then we've had ROP attacks, Spectre and much else; a short but useful update is Matthias Payer's *Software Security* [1216]. But to really keep up, it's not enough to just read textbooks; you need to follow security conferences such as Usenix and CCS as well as the security blogs such as Bruce Schneier, Brian Krebs and – dare I say it – our own `lightbluetouchpaper.org`.