CHAPTER

6

Distributed Systems

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.

— Leslie Lamport

6.1 Introduction

We've seen in the last few chapters how people can authenticate themselves to systems (and systems can authenticate themselves to each other) using security protocols; how access controls can be used to manage which principals can perform what operations in a system; and some of the mechanics of how crypto can be used to underpin access control in distributed systems. But there's much more to building a secure distributed system than just implementing access controls, protocols and crypto. When systems become large, the scale-up problems are not linear; there is often a qualitative change in complexity, and some things that are trivial to deal with in a network of only a few machines and principals (such as naming) suddenly become a big deal.

Over the last 40 years, computer science researchers have built many distributed systems and studied issues such as concurrency, failure recovery and naming. The theory is supplemented by a growing body of experience from industry, commerce and government. These issues are central to the design of effective secure systems but are often handled rather badly. I've already described attacks on security protocols that can be seen as concurrency failures. If we replicate data to make a system fault-tolerant then we may increase the risk of a compromise of confidentiality. Finally, naming is a particularly thorny problem. Many governments and organisations are trying to build larger, flatter namespaces — using identity cards to number citizens and using RFID to number objects — and yet naming problems undermined attempts during the 1990s to build useful public key infrastructures.

6.2 Concurrency

Processes are said to be *concurrent* if they run at the same time, and concurrency gives rise to a number of well-studied problems. Processes may use old data; they can make inconsistent updates; the order of updates may or may not matter; the system might deadlock; the data in different systems might never converge to consistent values; and when it's important to make things happen in the right order, or even to know the exact time, this can be harder than you might think.

Systems are now rapidly becoming more concurrent. First, the scale of online business has grown rapidly; Google may have started off with four machines but now its server farms have hundreds of thousands. Second, devices are becoming more complex; a luxury car can now contain over forty different processors. Third, the components are also getting more complex: the microprocessor in your PC may now have two, or even four, CPU cores, and will soon have more, while the graphics card, disk controller and other accessories all have their own processors too. On top of this, virtualization technologies such as VMware and Xen may turn a handful of real CPUs into hundreds or even thousands of virtual CPUs.

Programming concurrent systems is hard; and, unfortunately, most of the textbook examples come from the relatively rarefied world of operating system internals and thread management. But concurrency control is also a security issue. Like access control, it exists in order to prevent users interfering with each other, whether accidentally or on purpose. Also, concurrency problems can occur at many levels in a system, from the hardware right up to the business environment. In what follows, I provide a number of concrete examples of the effects of concurrency on security. These are by no means exhaustive.

6.2.1 Using Old Data Versus Paying to Propagate State

I've already described two kinds of concurrency problem. First, there are replay attacks on protocols, where an attacker manages to pass off out-of-date credentials. Secondly, there are race conditions. I mentioned the '`mkdir`' vulnerability from Unix, in which a privileged instruction that is executed in two phases could be attacked halfway through the process by renaming an object on which it acts. These problems have been around for a long time. In one of the first multiuser operating systems, IBM's OS/360, an attempt to open a file caused it to be read and its permissions checked; if the user was authorized to access it, it was read again. The user could arrange things so that the file was altered in between [774].

These are examples of a *time-of-check-to-time-of-use* (TOCTTOU) attack. There are systematic ways of finding such attacks in file systems [176], but as more of our infrastructure becomes concurrent, attacks crop up at other levels such as system calls in virtualised environments, which may require different approaches. (I'll discuss this specific case in detail in Chapter 18.) They also appear at the level of business logic. Preventing them isn't always economical, as propagating changes in security state can be expensive.

For example, the banking industry manages lists of all *hot* credit cards (whether stolen or abused) but there are millions of them worldwide, so it isn't possible to keep a complete hot card list in every merchant terminal, and it would be too expensive to verify all transactions with the bank that issued the card. Instead, there are multiple levels of stand-in processing. Terminals are allowed to process transactions up to a certain limit (the *floor limit*) offline; larger transactions need online verification with a local bank, which will know about all the local hot cards plus foreign cards that are being actively abused; above another limit there might be a reference to an organization such as VISA with a larger international list; while the largest transactions might need a reference to the card issuer. In effect, the only transactions that are checked immediately before use are those that are local or large.

Credit card systems are interesting as the largest systems that manage the global propagation of security state — which they do by assuming that most events are local, of low value, or both. They taught us that revoking compromised credentials quickly and on a global scale was expensive. In the 1990s, when people started to build infrastructures of public key certificates to support everything from web shopping to corporate networks, there was a fear that biggest cost would be revoking the credentials of principals who changed address, changed job, had their private key hacked, or got fired. This turned out not to be the case in general[1]. Another aspect of the costs of revocation can be seen in large web services, where it would be expensive to check a user's credentials against a database every time she visits any one of the service's thousands of machines. A common solution is to use cookies — giving the user an encrypted credential that her browser automatically presents on each visit. That way only the key has to be shared between the server farm's many machines. However, if revoking users quickly is important to the application, some other method needs to be found to do this.

[1]Frauds against web-based banking and shopping services don't generally involve compromised certificates. However, one application where revocation is a problem is the Deparatment of Defense, which has issued 16 million certificates to military personnel since 1999 and now has a list of 10 million revoked certificates that must be downloaded to all security servers every day [878].

6.2.2 Locking to Prevent Inconsistent Updates

When a number of people are working concurrently on a document, they may use a version control system to ensure that only one person has write access at any one time to any given part of it. This illustrates the importance of *locking* as a way to manage contention for resources such as filesystems and to reduce the likelihood of conflicting updates. Another mechanism is *callback*; a server may keep a list of all those clients which rely on it for security state, and notify them when the state changes.

Locking and callback also matter in secure distributed systems. Credit cards again provide an example. If I own a hotel, and a customer presents a credit card on checkin, I ask the card company for a *pre-authorization* which records the fact that I will want to make a debit in the near future; I might register a claim on 'up to $500' of her available credit. If the card is cancelled the following day, her bank can call me and ask me to contact the police, or to get her to pay cash. (My bank might or might not have guaranteed me the money; it all depends on what sort of contract I've managed to negotiate with it.) This is an example of the *publish-register-notify* model of how to do robust authorization in distributed systems (of which there's a more general description in [105]).

Callback mechanisms don't provide a universal solution, though. The credential issuer might not want to run a callback service, and the customer might object on privacy grounds to the issuer being told all her comings and goings. Consider passports as an example. In many countries, government ID is required for many transactions, but governments won't provide any guarantee, and most citizens would object if the government kept a record of every time an ID document was presented. Indeed, one of the frequent objections to the British government's proposal for biometric ID cards is that checking citizens' fingerprints against a database whenever they show their ID would create an audit trail of all the places where the card was used.

In general, there is a distinction between those credentials whose use gives rise to some obligation on the issuer, such as credit cards, and the others, such as passports. Among the differences is the importance of the order in which updates are made.

6.2.3 The Order of Updates

If two transactions arrive at the government's bank account — say a credit of $500,000 and a debit of $400,000 — then the order in which they are applied may not matter much. But if they're arriving at my bank account, the order will have a huge effect on the outcome! In fact, the problem of deciding the order in which transactions are applied has no clean solution. It's closely related to the problem of how to parallelize a computation, and much of the art of building efficient distributed systems lies in arranging matters so that processes are either simple sequential or completely parallel.

The usual algorithm in retail checking account systems is to batch the transactions overnight and apply all the credits for each account before applying all the debits. Inputs from devices such as ATMs and check sorters are first batched up into journals before the overnight reconciliation. The inevitable side-effect of this is that payments which bounce then have to be reversed out — and in the case of ATM and other transactions where the cash has already been dispensed, you can end up with customers borrowing money without authorization. In practice, chains of failed payments terminate, though in theory this isn't necessarily so. Some interbank payment mechanisms are moving to *real time gross settlement* in which transactions are booked in order of arrival. The downside here is that the outcome can depend on network vagaries. Some people thought this would limit the *systemic risk* that a non-terminating payment chain might bring down the world's banking system, but there is no real agreement on which practice is better. Credit cards operate a mixture of the two strategies, with credit limits run in real time or near real time (each authorization reduces the available credit limit) while settlement is run just as in a checking account. The downside here is that by putting through a large pre-authorization, a merchant can tie up your card.

The checking-account approach has recently been the subject of research in the parallel systems community. The idea is that disconnected applications propose tentative update transactions that are later applied to a master copy. Various techniques can be used to avoid instability; mechanisms for tentative update, such as with bank journals, are particularly important [553]. Application-level sanity checks are important; banks know roughly how much they expect to pay each other each day to settle net payments, and large cash flows get verified.

In other systems, the order in which transactions arrive is much less important. Passports are a good example. Passport issuers only worry about their creation and expiration dates, not the order in which visas are stamped on them. (There are exceptions, such as the Arab countries that won't let you in if you have an Israeli stamp on your passport, but most pure identification systems are stateless.)

6.2.4 Deadlock

Deadlock is another problem. Things may foul up because two systems are each waiting for the other to move first. A famous exposition of deadlock is the *dining philosophers' problem* in which a number of philosophers are seated round a table. There is a chopstick between each philosopher, who can only eat when he can pick up the two chopsticks on either side. Deadlock can follow if they all try to eat at once and each picks up (say) the chopstick on his right.

This problem, and the algorithms that can be used to avoid it, are presented in a classic paper by Dijkstra [388].

This can get horribly complex when you have multiple hierarchies of locks, and they're distributed across systems some of which fail (especially where failures can mean that the locks aren't reliable). There's a lot written on the problem in the distributed systems literature [104]. But it is not just a technical matter; there are many Catch-22 situations in business processes. So long as the process is manual, some fudge may be found to get round the catch, but when it is implemented in software, this option may no longer be available.

Sometimes it isn't possible to remove the fudge. In a well known business problem — the *battle of the forms* — one company issues an order with its own terms attached, another company accepts it subject to its own terms, and trading proceeds without any agreement about whose conditions govern the contract. The matter may only be resolved if something goes wrong and the two companies end up in court; even then, one company's terms might specify an American court while the other's specify a court in England. This kind of problem looks set to get worse as trading becomes more electronic.

6.2.5 Non-Convergent State

When designing protocols that update the state of a distributed system, the 'motherhood and apple pie' is ACID — that transactions should be *atomic, consistent, isolated and durable*. A transaction is atomic if you 'do it all or not at all' — which makes it easier to recover the system after a failure. It is consistent if some invariant is preserved, such as that the books must still balance. This is common in banking systems, and is achieved by insisting that each credit to one account is matched by an equal and opposite debit to another (I'll discuss this more in Chapter 10, 'Banking and Bookkeeping'). Transactions are isolated if they look the same to each other, that is, are serializable; and they are durable if once done they can't be undone.

These properties can be too much, or not enough, or both. On the one hand, each of them can fail or be attacked in numerous obscure ways; on the other, it's often sufficient to design the system to be *convergent*. This means that, if the transaction volume were to tail off, then eventually there would be consistent state throughout [912]. Convergence is usually achieved using semantic tricks such as timestamps and version numbers; this can often be enough where transactions get appended to files rather than overwritten.

However, in real life, you also need ways to survive things that go wrong and are not completely recoverable. The life of a security or audit manager can be a constant battle against entropy: apparent deficits (and surpluses) are always turning up, and sometimes simply can't be explained. For example, different national systems have different ideas of which fields in bank transaction records are mandatory or optional, so payment gateways often have to guess data in order to make things work. Sometimes they guess wrong; and sometimes people see and exploit vulnerabilities which aren't understood until much later (if ever). In the end, things get fudged by adding a correction factor, called something like 'branch differences', and setting a target for keeping it below a certain annual threshold.

Durability is a subject of debate in transaction processing. The advent of phishing and keylogging attacks has meant that some small proportion of bank accounts will at any time be under the control of criminals; money gets moved both from and through them. When an account compromise is detected, the bank moves to freeze it and to reverse any payments that have recently been made from it. The phishermen naturally try to move funds through institutions, or jurisdictions, that don't do transaction reversal, or do it at best slowly and grudgingly [55]. This sets up a tension between the recoverability and thus the resilience of the payment system on the one hand, and transaction durability and finality on the other. The solution may lie at the application level, namely charging customers a premium for irrevocable payments and letting the market allocate the associated risks to the bank best able to price it.

The battle of the forms mentioned in the above section gives an example of a distributed non-electronic system that doesn't converge.

In military systems, there is the further problem of dealing with users who request some data for which they don't have a clearance. For example, someone at a dockyard might ask the destination of a warship that's actually on a secret mission carrying arms to Iran. If she isn't allowed to know this, the system may conceal the ship's real destination by making up a *cover story*. Search may have to be handled differently from specific enquiries; the joining-up of intelligence databases since 9/11 has forced system builders to start sending clearances along with search queries, otherwise sorting the results became unmanageable. This all raises difficult engineering problems, with potentially severe conflicts between atomicity, consistency, isolation and durability (not to mention performance), which will be discussed at more length in Chapter 8, 'Multilevel Security'.

6.2.6 Secure Time

The final kind of concurrency problem with special interest to the security engineer is the provision of accurate time. As authentication protocols such as Kerberos can be attacked by inducing an error in the clock, it's not enough to simply trust a time source on the network. A few years ago, the worry was a *Cinderella attack*: if a security critical program such as a firewall has a license with a timelock in it, an attacker might wind your clock forward 'and cause your software to turn into a pumpkin'. Things have become more acute since the arrival of operating systems such as Vista with hardware security support, and of media players with built-in DRM; the concern now is that someone might do a large-scale service-denial attack by convincing millions of machines that their owners had tampered with the clock, causing their files to become inaccessible.

Anyway, there are several possible approaches to the provision of secure time.

- You could furnish every computer with a radio clock, but that can be expensive, and radio clocks — even GPS — can be jammed if the opponent is serious.
- There are clock synchronization protocols described in the research literature in which a number of clocks vote in a way that should make clock failures and network delays apparent. Even though these are designed to withstand random (rather than malicious) failure, they can no doubt be hardened by having the messages digitally signed.
- You can abandon absolute time and instead use *Lamport time* in which all you care about is whether event A happened before event B, rather than what date it is [766]. Using challenge-response rather than timestamps in security protocols is an example of this; another is given by timestamping services that continually hash all documents presented to them into a running total that's published, and can thus provide proof that a certain document existed by a certain date [572].

However, in most applications, you are likely to end up using the *network time protocol* (NTP). This has a moderate amount of protection, with clock voting and authentication of time servers. It is dependable enough for many purposes.

6.3 Fault Tolerance and Failure Recovery

Failure recovery is often the most important aspect of security engineering, yet it is one of the most neglected. For many year, most of the research papers on computer security have dealt with confidentiality, and most of the rest with authenticity and integrity; availability has been neglected. Yet the actual expenditures of a typical bank are the other way round. Perhaps a third of all IT costs go on availability and recovery mechanisms, such as hot standby processing sites and multiply redundant networks; a few percent more get invested in integrity mechanisms such as internal audit; and an almost insignificant amount goes on confidentiality mechanisms such as encryption boxes. As you read through this book, you'll see that many other applications, from burglar alarms through electronic warfare to protecting a company from Internet-based service denial attacks, are fundamentally about availability. Fault tolerance and failure recovery are a huge part of the security engineer's job.