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.

Classical fault tolerance is usually based on mechanisms such as logs and locking, and is greatly complicated when it must withstand malicious attacks on these mechanisms. Fault tolerance interacts with security in a number of ways: the failure model, the nature of resilience, the location of redundancy used to provide it, and defense against service denial attacks. I'll use the following definitions: a *fault* may cause an *error*, which is an incorrect state; this may lead to a *failure* which is a deviation from the system's specified behavior. The resilience which we build into a system to tolerate faults and recover from failures will have a number of components, such as fault detection, error recovery and if necessary failure recovery. The meaning of *mean-time-before-failure* (MTBF) and *mean-time-to-repair* (MTTR) should be obvious.

6.3.1 Failure Models

In order to decide what sort of resilience we need, we must know what sort of attacks are expected. Much of this will come from an analysis of threats specific to our system's operating environment, but there are some general issues that bear mentioning.

6.3.1.1 Byzantine Failure

First, the failures with which we are concerned may be normal or *Byzantine*. The Byzantine fault model is inspired by the idea that there are n generals defending Byzantium, t of whom have been bribed by the Turks to cause as much confusion as possible in the command structure. The generals can pass oral messages by courier, and the couriers are trustworthy, so each general can exchange confidential and authentic communications with each other general (we could imagine them encrypting and computing a MAC on each message). What is the maximum number t of traitors which can be tolerated?

The key observation is that if we have only three generals, say Anthony, Basil and Charalampos, and Anthony is the traitor, then he can tell Basil 'let's attack' and Charalampos 'let's retreat'. Basil can now say to Charalampos 'Anthony says let's attack', but this doesn't let Charalampos conclude that Anthony's the traitor. It could just as easily have been Basil; Anthony could have said 'let's retreat' to both of them, but Basil lied when he said 'Anthony says let's attack'.

This beautiful insight is due to Leslie Lamport, Robert Shostak and Marshall Pease, who proved that the problem has a solution if and only if $n \geq 3t + 1$ [767]. Of course, if the generals are able to sign their messages, then no general dare say different things to two different colleagues. This illustrates the power of digital signatures in particular and of end-to-end security mechanisms in general. Relying on third parties to introduce principals to each

other or to process transactions between them can give great savings, but if the third parties ever become untrustworthy then it can impose significant costs.

Another lesson is that if a component that fails (or can be induced to fail by an opponent) gives the wrong answer rather than just no answer, then it's much harder to build a resilient system using it. This has recently become a problem in avionics, leading to an emergency Airworthiness Directive in April 2005 that mandated a software upgrade for the Boeing 777, after one of these planes suffered a 'flight control outage' [762].

6.3.1.2 Interaction with Fault Tolerance

We can constrain the failure rate in a number of ways. The two most obvious are by using *redundancy* and *fail-stop processors*. The latter process error-correction information along with data, and stop when an inconsistency is detected; for example, bank transaction processing will typically stop if an out-of-balance condition is detected after a processing task. The two may be combined; IBM's System/88 minicomputer had two disks, two buses and even two CPUs, each of which would stop if it detected errors; the fail-stop CPUs were built by having two CPUs on the same card and comparing their outputs. If they disagreed the output went open-circuit, thus avoiding the Byzantine failure problem.

In general distributed systems, either redundancy or fail-stop processing can make a system more *resilient*, but their side effects are rather different. While both mechanisms may help protect the integrity of data, a fail-stop processor may be more vulnerable to service denial attacks, whereas redundancy can make confidentiality harder to achieve. If I have multiple sites with backup data, then confidentiality could be broken if any of them gets compromised; and if I have some data that I have a duty to destroy, perhaps in response to a court order, then purging it from multiple backup tapes can be a headache.

It is only a slight simplification to say that while replication provides integrity and availability, tamper resistance provides confidentiality too. I'll return to this theme later. Indeed, the prevalence of replication in commercial systems, and of tamper-resistance in military systems, echoes their differing protection priorities.

However, there are traps for the unwary. In one case in which I was called on as an expert, my client was arrested while using a credit card in a store, accused of having a forged card, and beaten up by the police. He was adamant that the card was genuine. Much later, we got the card examined by VISA who confirmed that it was indeed genuine. What happened, as well as we can reconstruct it, was this. Credit cards have two types of redundancy on the magnetic strip — a simple checksum obtained by combining together all the bytes on the track using exclusive-or, and a cryptographic checksum which we'll describe in detail later in section 10.5.2. The former is there to

detect errors, and the latter to detect forgery. It appears that in this particular case, the merchant's card reader was out of alignment in such a way as to cause an even number of bit errors which cancelled each other out by chance in the simple checksum, while causing the crypto checksum to fail. The result was a false alarm, and a major disruption in my client's life.

Redundancy is hard enough to deal with in mechanical systems. For example, training pilots to handle multi-engine aircraft involves drilling them on engine failure procedures, first in the simulator and then in real aircraft with an instructor. Novice pilots are in fact more likely to be killed by an engine failure in a multi-engine plane than in a single; landing in the nearest field is less hazardous for them than coping with suddenly asymmetric thrust. The same goes for instrument failures; it doesn't help to have three artificial horizons in the cockpit if, under stress, you rely on the one that's broken. Aircraft are much simpler than many modern information systems — yet there are still regular air crashes when pilots fail to manage the redundancy that's supposed to keep them safe. All too often, system designers put in multiple protection mechanisms and hope that things will be 'all right on the night'. This might be compared to strapping a 40-hour rookie pilot into a Learjet and telling him to go play. It really isn't good enough. Please bear the aircraft analogy in mind if you have to design systems combining redundancy and security!

The proper way to do things is to consider all the possible use cases and abuse cases of a system, think through all the failure modes that can happen by chance (or be maliciously induced), and work out how all the combinations of alarms will be dealt with — and how, and by whom. Then write up your safety and security case and have it evaluated by someone who knows what they're doing. I'll have more to say on this later in the chapter on 'System Evaluation and Assurance'.

Even so, large-scale system failures very often show up dependencies that the planners didn't think of. For example, Britain suffered a fuel tanker drivers' strike in 2001, and some hospitals had to close because of staff shortages. The government allocated petrol rations to doctors and nurses, but not to schoolteachers. So schools closed, and nurses had to stay home to look after their kids, and this closed hospitals too. We are becoming increasingly dependent on each other, and this makes contingency planning harder.

6.3.2 What Is Resilience For?

When introducing redundancy or other resilience mechanisms into a system, we need to be very clear about what they're for. An important consideration is whether the resilience is contained within a single organization.

In the first case, replication can be an internal feature of the server to make it more trustworthy. AT&T built a system called *Rampart* in which a number

of geographically distinct servers can perform a computation separately and combine their results using threshold decryption and signature [1065]; the idea is to use it for tasks like key management [1066]. IBM developed a variant on this idea called *Proactive Security*, where keys are regularly flushed through the system, regardless of whether an attack has been reported [597]. The idea is to recover even from attackers who break into a server and then simply bide their time until enough other servers have also been compromised. The trick of building a secure 'virtual server' on top of a number of cheap off-the-shelf machines has turned out to be attractive to people designing certification authority services because it's possible to have very robust evidence of attacks on, or mistakes made by, one of the component servers [337]. It also appeals to a number of navies, as critical resources can be spread around a ship in multiple PCs and survive most kinds of damage that don't sink it [489].

But often things are much more complicated. A server may have to protect itself against malicious clients. A prudent bank, for example, will assume that some of its customers would cheat it given the chance. Sometimes the problem is the other way round, in that we have to rely on a number of services, none of which is completely trustworthy. Since 9/11, for example, international money-laundering controls have been tightened so that people opening bank accounts are supposed to provide two different items that give evidence of their name and address — such as a gas bill and a pay slip. (This causes serious problems in Africa, where the poor also need banking services as part of their path out of poverty, but may live in huts that don't even have addresses, let alone utilities [55].)

The direction of mistrust has an effect on protocol design. A server faced with multiple untrustworthy clients, and a client relying on multiple servers that may be incompetent, unavailable or malicious, will both wish to control the flow of messages in a protocol in order to contain the effects of service denial. So a client facing several unreliable servers may wish to use an authentication protocol such as the Needham-Schroeder protocol I discussed in section 3.7.2; then the fact that the client can use old server tickets is no longer a bug but a feature. This idea can be applied to protocol design in general [1043]. It provides us with another insight into why protocols may fail if the principal responsible for the design, and the principal who carries the cost of fraud, are different; and why designing systems for the real world in which everyone (clients and servers) are unreliable and mutually suspicious, is hard.

At a still higher level, the emphasis might be on *security renewability*. Pay-TV is a good example: secret keys and other subscriber management tools are typically kept in a cheap smartcard rather than in an expensive set-top box, so that even if all the secret keys are compromised, the operator can recover by mailing new cards out to his subscribers. I'll discuss in more detail in Chapter 22, 'Copyright and Privacy Protection'.

6.3.3 At What Level Is the Redundancy?

Systems may be made resilient against errors, attacks and equipment failures at a number of levels. As with access control systems, these become progressively more complex and less reliable as we go up to higher layers in the system.

Some computers have been built with redundancy at the hardware level, such as the IBM System/88 I mentioned earlier. From the late 1980's, these machines were widely used in transaction processing tasks (eventually ordinary hardware became reliable enough that banks would not pay the premium in capital cost and development effort to use non-standard hardware). Some more modern systems achieve the same goal with standard hardware either at the component level, using *redundant arrays of inexpensive disks* ('RAID' disks) or at the system level by massively parallel server farms. But none of these techniques provides a defense against faulty or malicious software, or against an intruder who exploits such software.

At the next level up, there is *process group redundancy*. Here, we may run multiple copies of a system on multiple servers in different locations, and compare their outputs. This can stop the kind of attack in which the opponent gets physical access to a machine and subverts it, whether by mechanical destruction or by inserting unauthorized software, and destroys or alters data. It can't defend against attacks by authorized users or damage by bad authorized software, which could simply order the deletion of a critical file.

The next level is *backup*. Here, we typically take a copy of the system (also known as a *checkpoint*) at regular intervals. The backup copies are usually kept on media that can't be overwritten such as write-protected tapes or DVDs. We may also keep *journals* of all the transactions applied between checkpoints. In general, systems are kept recoverable by a transaction processing strategy of logging the incoming data, trying to do the transaction, logging it again, and then checking to see whether it worked. Whatever the detail, backup and recovery mechanisms not only enable us to recover from physical asset destruction; they also ensure that if we do get an attack at the logical level — such as a time bomb in our software which deletes our customer database on a specific date — we have some hope of recovering. They are not infallible though. The closest that any bank I know of came to a catastrophic computer failure that would have closed its business was when its mainframe software got progressively more tangled as time progressed, and it just wasn't feasible to roll back processing several weeks and try again.

Backup is not the same as *fallback*. A fallback system is typically a less capable system to which processing reverts when the main system is unavailable. An example is the use of manual imprinting machines to capture credit card transactions from the card embossing when electronic terminals fail.

Fallback systems are an example of redundancy in the application layer — the highest layer we can put it. We might require that a transaction above a certain limit be authorized by two members of staff, that an audit trail be kept of all transactions, and a number of other things. We'll discuss such arrangements at greater length in the chapter on banking and bookkeeping.

It is important to realise that these are different mechanisms, which do different things. Redundant disks won't protect against a malicious programmer who deletes all your account files, and backups won't stop him if rather than just deleting files he writes code that slowly inserts more and more errors. Neither will give much protection against attacks on data confidentiality. On the other hand, the best encryption in the world won't help you if your data processing center burns down. Real world recovery plans and mechanisms can get fiendishly complex and involve a mixture of all of the above.

The remarks that I made earlier about the difficulty of redundancy, and the absolute need to plan and train for it properly, apply in spades to system backup. When I was working in banking we reckoned that we could probably get our backup system working within an hour or so of our main processing centre being destroyed, but the tests we did were limited by the fact that we didn't want to risk processing during business hours. The most impressive preparations I've ever seen were at a UK supermarket, which as a matter of policy pulls the plug on its main processing centre once a year without warning the operators. This is the only way they can be sure that the backup arrangements actually work, and that the secondary processing centre really cuts in within a minute or so. Bank tellers can keep serving customers for a few hours with the systems down; but retailers with dead checkout lanes can't do that.

6.3.4 Service-Denial Attacks

One of the reasons we want security services to be fault-tolerant is to make service-denial attacks less attractive, more difficult, or both. These attacks are often used as part of a larger attack plan. For example, one might swamp a host to take it temporarily offline, and then get another machine on the same LAN (which had already been subverted) to assume its identity for a while. Another possible attack is to take down a security server to force other servers to use cached copies of credentials.

A powerful defense against service denial is to prevent the opponent mounting a selective attack. If principals are anonymous — or at least there is no name service which will tell the opponent where to attack — then he may be ineffective. I'll discuss this further in the context of burglar alarms and electronic warfare.

Where this isn't possible, and the opponent knows where to attack, then there are some types of service-denial attacks which can be stopped by redundancy and resilience mechanisms, and others which can't. For example, the TCP/IP protocol has few effective mechanisms for hosts to protect themselves against various network flooding attacks. An opponent can send a large number of connection requests and prevent anyone else establishing a connection. Defense against this kind of attack tends to involve moving your site to a beefier hosting service with specialist packet-washing hardware — or tracing and arresting the perpetrator.

Distributed denial-of-service (DDoS) attacks had been known to the research community as a possibility for some years. They came to public notice when they were used to bring down Panix, a New York ISP, for several days in 1996. During the late 1990s they were occasionally used by script kiddies to take over chat servers. In 2000, colleagues and I suggested dealing with the problem by server replication [1366], and in 2001 I mentioned them in passing in the first edition of this book. Over the following three years, small-time extortionists started using DDoS attacks for blackmail. The modus operandi was to assemble a *botnet*, a network of compromised PCs used as attack robots, which would flood a target webserver with packet traffic until its owner paid them to desist. Typical targets were online bookmakers, and amounts of $10,000–$50,000 were typically demanded to leave them alone. The typical bookie paid up the first time this happened, but when the attacks persisted the first solution was replication: operators moved their websites to hosting services such as Akamai whose servers are so numerous (and so close to customers) that they can shrug off anything that the average botnet could throw at them. In the end, the blackmail problem was solved when the bookmakers met and agreed not to pay any more blackmail money, and the Russian police were prodded into arresting the gang responsible.

Finally, where a more vulnerable fallback system exists, a common technique is to force its use by a service denial attack. The classic example is the use of smartcards for bank payments in countries in Europe. Smartcards are generally harder to forge than magnetic strip cards, but perhaps 1% of them fail every year, thanks to static electricity and worn contacts. Also, foreign tourists still use magnetic strip cards. So card payment systems have a fallback mode that uses the magnetic strip. A typical attack nowadays is to use a false terminal, or a bug inserted into the cable between a genuine terminal and a branch server, to capture card details, and then write these details to the magnetic stripe of a card whose chip has been destroyed (connecting 20V across the contacts does the job nicely). In the same way, burglar alarms that rely on network connections for the primary response and fall back to alarm bells may be very vulnerable if the network can be interrupted by an attacker: now that online alarms are the norm, few people pay attention any more to alarm bells.

6.4 Naming

Naming is a minor if troublesome aspect of ordinary distributed systems, but it becomes surprisingly hard in security engineering. The topical example in the 1990s was the problem of what names to put on public key certificates. A certificate that says simply 'the person named Ross Anderson is allowed to administer machine X' is little use. Before the arrival of Internet search engines, I was the only Ross Anderson I knew of; now I know of dozens of us. I am also known by different names to dozens of different systems. Names exist in contexts, and naming the principals in secure systems is becoming ever more important and difficult.

Engineers observed then that using more names than you need to causes unnecessary complexity. For example, A certificate that simply says 'the bearer of this certificate is allowed to administer machine X' is a straightforward bearer token, which we know how to deal with; but once my name is involved, then presumably I have to present some kind of ID to prove who I am, and the system acquires a further dependency. Worse, if my ID is compromised the consequences could be extremely far-reaching.

Since 9/11 the terms of this debate have shifted somewhat, as many governments have rushed to issue their citizens with ID cards. In order to justify the expense and hassle, pressure is often put on commercial system operators to place some reliance on government-issue ID where this was previously not thought necessary. In the UK, for example, you can no longer board a domestic flight using just the credit card with which you bought the ticket, and you have to produce a passport or driving license to cash a check or order a bank transfer for more than £1000. Such measures cause inconvenience and introduce new failure modes into all sorts of systems.

No doubt this identity fixation will abate in time, as governments find other things to scare their citizens with. However there is a second reason that the world is moving towards larger, flatter name spaces: the move from barcodes (which code a particular product) to RFID tags (which contain a 128-bit unique identifier that code a particular item.) This has ramifications well beyond naming — into issues such as the interaction between product security, supply-chain security and competition policy.

For now, it's useful to go through what a generation of computer science researchers have learned about naming in distributed systems.

6.4.1 The Distributed Systems View of Naming

During the last quarter of the twentieth century, the distributed systems research community ran up against many naming problems. The basic algorithm used to bind names to addresses is known as *rendezvous*: the principal

exporting a name advertises it somewhere, and the principal seeking to import and use it searches for it. Obvious examples include phone books, and directories in file systems.

However, the distributed systems community soon realised that naming can get fiendishly complex, and the lessons learned are set out in a classic article by Needham [958]. I'll summarize the main points, and look at which of them apply to secure systems.

1. *The function of names is to facilitate sharing*. This continues to hold: my bank account number exists in order to provide a convenient way of sharing the information that I deposited money last week with the teller from whom I am trying to withdraw money this week. In general, names are needed when the data to be shared is changeable. If I only ever wished to withdraw exactly the same sum as I'd deposited, a bearer deposit certificate would be fine. Conversely, names need not be shared — or linked — where data will not be; there is no need to link my bank account number to my telephone number unless I am going to pay my phone bill from the account.
2. *The naming information may not all be in one place, and so resolving names brings all the general problems of a distributed system*. This holds with a vengeance. A link between a bank account and a phone number assumes both of them will remain stable. So each system relies on the other, and an attack on one can affect the other. In the days when electronic banking was dial-up rather than web based, a bank which identified its customers using calling line ID was vulnerable to attacks on telephone systems (such as tapping into the distribution frame in an apartment block, hacking a phone company computer, or bribing a phone company employee). Nowadays some banks are using two-channel authorization to combat phishing — if you order a payment online you get a text message on your mobile phone saying 'if you want to pay $X to account Y, please enter the following four digit code into your browser'. This is a bit tougher, as mobile phone traffic is encrypted — but one weak link to watch for is the binding between the customer and his phone number. If you let just anyone notify you of a customer phone number change, you'll be in trouble.
3. *It is bad to assume that only so many names will be needed*. The shortage of IP addresses, which motivated the development of IP version 6 (IPv6), is well enough discussed. What is less well known is that the most expensive upgrade which the credit card industry ever had to make was not Y2K remediation, but the move from thirteen digit credit card numbers to sixteen. Issuers originally assumed that 13 digits would be enough, but the system ended up with tens of thousands of banks (many with dozens of products) so a six digit *bank identification number*

(BIN number) was needed. Some card issuers have millions of customers, so a nine digit account number is the norm. And there's also a *check digit* (a one-digit linear combination of the other digits which is appended to detect errors).

4. *Global names buy you less than you think.* For example, the 128-bit in IPv6 can in theory enable every object in the universe to have a unique name. However, for us to do business, a local name at my end must be resolved into this unique name and back into a local name at your end. Invoking a unique name in the middle may not buy us anything; it may even get in the way if the unique naming service takes time, costs money, or occasionally fails (as it surely will). In fact, the name service itself will usually have to be a distributed system, of the same scale (and security level) as the system we're trying to protect. So we can expect no silver bullets from this quarter. One reason the banking industry was wary of initiatives such as SET that would have given each customer a public key certificate on a key with which they could sign payment instructions was that banks already have perfectly good names for their customers (account numbers). Adding an extra name has the potential to add extra costs and failure modes.
5. *Names imply commitments, so keep the scheme flexible enough to cope with organizational changes.* This sound principle was ignored in the design of a UK government's key management system for secure email [76]. There, principals' private keys are generated by encrypting their names under departmental master keys. So the frequent reorganizations meant that the security infrastructure would have to be rebuilt each time — and that money would have had to be spent solving many secondary problems such as how people would access old material.
6. *Names may double as access tickets, or capabilities*. We have already seen a number of examples of this in Chapters 2 and 3. In general, it's a bad idea to assume that today's name won't be tomorrow's password or capability — remember the Utrecht fraud we discussed in section 3.5. (This is one of the arguments for making all names public keys — 'keys speak in cyberspace' in Carl Ellison's phrase — but we've already noted the difficulties of linking keys with names.)

I've given a number of examples of how things go wrong when a name starts being used as a password. But sometimes the roles of name and password are ambiguous. In order to get entry to a car park I used to use at the university, I had to speak my surname and parking badge number into a microphone at the barrier. So if I say, 'Anderson, 123', which of these is the password? In fact it was 'Anderson', as anyone can walk through the car park and note down valid badge

numbers from the parking permits on the car windscreens. Another example, from medical informatics, is a large database of medical records kept by the UK government for research, where the name of the patient has been replaced by their postcode and date of birth. Yet access to many medical services requires the patient to give just these two items to the receptionist to prove who they are. I will have more to say on this later.

7. *Things are made much simpler if an incorrect name is obvious*. In standard distributed systems, this enables us to take a liberal attitude to cacheing. In payment systems, credit card numbers may be accepted while a terminal is offline so long as the credit card number appears valid (i.e., the last digit is a proper check digit of the first fifteen) and it is not on the hot card list. Certificates provide a higher-quality implementation of the same basic concept.

   It's important where the name is checked. The credit card check digit algorithm is deployed at the point of sale, so it is public. A further check — the *card verification value* on the magnetic strip — is computed with secret keys but can be checked at the issuing bank, the acquiring bank or even at a network switch (if one trusts these third parties with the keys). This is more expensive, and still vulnerable to network outages.

8. *Consistency is hard, and is often fudged. If directories are replicated, then you may find yourself unable to read, or to write, depending on whether too many or too few directories are available*. Naming consistency causes problems for e-commerce in a number of ways, of which perhaps the most notorious is the bar code system. Although this is simple enough in theory — with a unique numerical code for each product — in practice it can be a nightmare, as different manufacturers, distributors and retailers attach quite different descriptions to the bar codes in their databases. Thus a search for products by 'Kellogg's' will throw up quite different results depending on whether or not an apostrophe is inserted, and this can cause great confusion in the supply chain. Proposals to fix this problem can be surprisingly complicated [618]. There are also the issues of convergence discussed above; data might not be consistent across a system, even in theory. There are also the problems of timeliness, such as whether a product has been recalled.

   Now, many firms propose moving to RFID chips that contain a globally unique number: an item code rather than a product code. This may move name resolution upstream; rather than the shop's computer recognising that the customer has presented a packet of vitamin C at the checkout, it may go to the manufacturer to find this

out. Manufacturers push for this on safety grounds; they can then be sure that the product hasn't passed its sell-by date and has not been recalled. But this also increases their power over the supply chain; they can detect and stop gray-market trading that would otherwise undermine their ability to charge different prices in different towns.

9. *Don't get too smart. Phone numbers are much more robust than computer addresses*. Amen to that — but it's too often ignored by secure system designers. Bank account numbers are much easier to deal with than the public-key certificates which were once believed both necessary and sufficient to secure credit card payments online. I'll discuss specific problems of public key infrastructures in section 21.4.5.7.
10. *Some names are bound early, others not; and in general it is a bad thing to bind early if you can avoid it*. A prudent programmer will normally avoid coding absolute addresses or filenames as that would make it hard to upgrade or replace a machine. He will prefer to leave this to a configuration file or an external service such as DNS. (This is another reason not to put addresses in names.) It is true that secure systems often want stable and accountable names as any third-party service used for last minute resolution could be a point of attack. However, designers should read the story of Netgear, who got their home routers to find out the time using the Network Time Protocol from a server at the University of Wisconsin-Madison. Their product was successful; the university was swamped with hundreds of thousands of packets a second. Netgear ended up paying them $375,000 to maintain the time service for three years. Shortly afterwards, D-Link repeated the same mistake [304].

So Needham's ten principles for distributed naming apply fairly directly to distributed secure systems.

6.4.2 What Else Goes Wrong

Needham's principles, although very useful, are not sufficient. They were designed for a world in which naming systems could be designed and imposed at the system owner's convenience. When we move from distributed systems in the abstract to the reality of modern web-based (and interlinked) service industries, there is still more to say.

6.4.2.1 Naming and Identity

The most obvious difference is that the principals in security protocols may be known by many different kinds of name — a bank account number, a company registration number, a personal name plus a date of birth or a postal address,

a telephone number, a passport number, a health service patient number, or a userid on a computer system.

As I mentioned in the introductory chapter, a common mistake is to confuse naming with identity. *Identity* is when two different names (or instances of the same name) correspond to the same principal (this is known in the distributed systems literature as an *indirect name* or *symbolic link*). The classic example comes from the registration of title to real estate. It is very common that someone who wishes to sell a house uses a different name than they did at the time it was purchased: they might have changed their name on marriage, or after a criminal conviction. Changes in name usage are also common. For example, the DE Bell of the Bell-LaPadula system[2] wrote his name 'D. Elliot Bell' in 1973 on that paper; but he was always known as David, which is how he now writes his name too. A land-registration system must cope with a lot of identity issues like this.

The classic example of identity failure leading to compromise is check fraud. Suppose I steal a high-value check made out to Mr Elliott Bell. I then open an account in that name and cash it; banking law in both the USA and the UK absolves the bank of liability so long as it pays the check into an account of the same name. The modern procedure of asking people who open bank accounts for two proofs of address, such as utility bills, probably makes the bad guys' job easier; there are hundreds of utility companies, many of which provide electronic copies of bills that are easy to alter. The pre-9/11 system, of taking up personal references, may well have been better.

Moving to verifying government-issue photo-ID on account opening adds to the mix statements such as 'The Elliott Bell who owns bank account number 12345678 is the Elliott James Bell with passport number 98765432 and date of birth 3/4/56'. This may be seen as a symbolic link between two separate systems — the bank's and the passport office's. Note that the latter part of this 'identity' encapsulates a further statement, which might be something like 'The US passport office's file number 98765432 corresponds to the entry in birth register for 3/4/56 of one Elliott James Bell'. In general, names may involve several steps of recursion, and this gives attackers a choice of targets. For example, a lot of passport fraud is *pre-issue fraud*: the bad guys apply for passports in the names of genuine citizens who haven't applied for a passport already and for whom copies of birth certificates are easy enough to obtain. Postmortem applications are also common. Linden Labs, the operators of Second Life, introduced in late 2007 a scheme whereby you prove you're over 18 by providing the driver's license number or social security number of someone who is. Now a web search quickly pulls up such data for many people, such as the rapper Tupac Amaru Shakur; and yes, Linden Labs did accept Mr Shakur's license number — even through the license is expired, and

[2] I'll discuss this in Chapter 8, 'Multilevel Secure Systems'.

he's dead. Indeed, someone else managed to verify their age using Mohammed Atta's driver's license [389].

6.4.2.2 Cultural Assumptions

The assumptions that underlie names often change from one country to another. In the English-speaking world, people may generally use as many names as they please; a name is simply what you are known by. But some countries forbid the use of aliases, and others require them to be registered. This can lead to some interesting scams: in at least one case, a British citizen evaded pursuit by foreign tax authorities by changing his name. On a less exalted plane, women who pursue academic careers and change their name on marriage may wish to retain their former name for professional use, which means that the name on their scientific papers is different from their name on the payroll. This caused a row at my university which introduced a unified ID card system, keyed to payroll names, without support for aliases.

In general, many of the really intractable problems arise when an attempt is made to unify two local naming systems which turn out to have incompatible assumptions. As electronics invades everyday life more and more, and systems become linked up, conflicts can propagate and have unexpected remote effects. For example, one of the lady professors in the dispute over our university card was also a trustee of the British Library, which issues its own admission tickets on the basis of the name on the holder's home university library card.

Even human naming conventions are not uniform. Russians are known by a forename, a patronymic and a surname; Icelanders have no surname but are known instead by a given name followed by a patronymic if they are male and a matronymic if they are female. This causes problems when they travel. When US immigration comes across 'Maria Trosttadóttir' and learns that 'Trosttadóttir' isn't a surname or even a patronymic, their standard practice was to compel her to adopt as a surname a patronymic (say, 'Carlsson' if her father was called Carl). This causes unnecessary offence.

The biggest cultural divide is often thought to be that between the English speaking countries (where identity cards were long considered to be unacceptable on privacy grounds[3]) and the countries conquered by Napoleon or by the Soviets, where identity cards are the norm.

There are further subtleties. I know Germans who have refused to believe that a country could function at all without a proper system of population registration and ID cards, yet admit they are asked for their ID card only rarely (for example, to open a bank account or get married). Their card number can't be used as a name, because it is a document number and changes every time a new card is issued. A Swiss hotelier may be happy to register a German guest on

[3] unless they're called drivers' licences or health service cards!

sight of an ID card rather than a credit card, but if he discovers some damage after a German guest has checked out, he may be out of luck. And the British passport office will issue a citizen with more than one passport at the same time, if he says he needs them to make business trips to (say) Cuba and the USA; so our Swiss hotelier, finding that a British guest has just left without paying, can't rely on the passport number to have him stopped at the airport.

There are many other hidden assumptions about the relationship between governments and people's names, and they vary from one country to another in ways which can cause unexpected security failures.

6.4.2.3 Semantic Content of Names

Another hazard arises on changing from one type of name to another without adequate background research. A bank got sued after they moved from storing customer data by account number to storing it by name and address. They wanted to target junk mail more accurately, so they wrote a program to link up all the accounts operated by each of their customers. The effect for one poor customer was that the bank statement for the account he maintained for his mistress got sent to his wife, who divorced him.

Sometimes naming is simple, but sometimes it merely appears to be. For example, when I got a monthly ticket for the local swimming baths, the cashier simply took the top card off a pile, swiped it through a reader to tell the system it was now live, and gave it to me. I had been assigned a random name — the serial number on the card. Many US road toll systems work in much the same way. Sometimes a random, anonymous name can add commercial value. In Hong Kong, toll tokens for the Aberdeen tunnel could be bought for cash, or at a discount in the form of a refillable card. In the run-up to the transfer of power from Britain to Beijing, many people preferred to pay extra for the less traceable version as they were worried about surveillance by the new government.

Semantics of names can change. I once got a hardware store loyalty card with a random account number (and no credit checks). I was offered the chance to change this into a bank card after the store was taken over by the supermarket and the supermarket started a bank. (This appears to have ignored money laundering regulations that all new bank customers must be subjected to due diligence.)

Assigning bank account numbers to customers might have seemed unproblematic — but as the above examples show, systems may start to construct assumptions about relationships between names that are misleading and dangerous.

6.4.2.4 Uniqueness of Names

Human names evolved when we lived in small communities. We started off with just forenames, but by the late Middle Ages the growth of travel led

governments to bully people into adopting surnames. That process took a century or so, and was linked with the introduction of paper into Europe as a lower-cost and more tamper-resistant replacement for parchment; paper enabled the badges, seals and other bearer tokens, which people had previously used for road tolls and the like, to be replaced with letters that mentioned their names.

The mass movement of people, business and administration to the Internet in the decade after 1995 has been too fast to allow any such social adaptation. There are now many more people (and systems) online than we are used to dealing with. As I remarked at the beginning of this section, I used to be the only Ross Anderson I knew of, but thanks to search engines, I now know dozens of us. Some of us work in fields I've also worked in, such as software engineering and electric power distribution; the fact that I'm `www.ross-anderson.com` and `ross.anderson@iee.org` is down to luck — I got there first. (Even so, `rjanderson@iee.org` is somebody else.) So even the combination of a relatively rare name and a specialized profession is still ambiguous. Another way of putting this is that 'traditional usernames, although old-school-geeky, don't scale well to the modern Internet' [21].

Sometimes system designers are tempted to solve the uniqueness problem by just giving everything and everyone a number. This is very common in transaction processing, but it can lead to interesting failures if you don't put the uniqueness in the right place. A UK bank wanted to send £20m overseas, but the operator typed in £10m by mistake. To correct the error, a second payment of £10m was ordered. However, the sending bank's system took the transaction sequence number from the paperwork used to authorise it. Two payments were sent to SWIFT with the same date, payee, amount and sequence number — so the second was discarded as a duplicate [218].

6.4.2.5 Stability of Names and Addresses

Many names include some kind of address, yet addresses change. About a quarter of Cambridge phone book addresses change every year; with email, the turnover is probably higher. A project to develop a directory of people who use encrypted email, together with their keys, found that the main cause of changed entries was changes of email address [67]. (Some people had assumed it would be the loss or theft of keys; the contribution from this source was precisely zero.)

A serious problem could arise with IPv6. The security community assumes that v6 IP addresses will be stable, so that public key certificates can bind principals of various kinds to them. All sorts of mechanisms have been proposed to map real world names, addresses and even document content indelibly and eternally on to 128 bit strings (see, for example, [573]). The data communications community, on the other hand, assumes that IPv6 addresses

will change regularly. The more significant bits will change to accommodate more efficient routing algorithms, while the less significant bits will be used to manage local networks. These assumptions can't both be right.

Distributed systems pioneers considered it a bad thing to put addresses in names [912]. But in general, there can be multiple layers of abstraction with some of the address information at each layer forming part of the name at the layer above. Also, whether a namespace is better flat depends on the application. Often people end up with different names at the departmental and organizational level (such as `rja14@cam.ac.uk` and `ross.anderson@cl.cam.ac.uk` in my own case). So a clean demarcation between names and addresses is not always possible.

Authorizations have many (but not all) of the properties of addresses. Kent's Law tells designers that if a credential contains a list of what it may be used for, then the more things are on this list the shorter its period of usefulness. A similar problem besets systems where names are composite. For example, some online businesses recognize me by the combination of email address and credit card number. This is clearly bad practice. Quite apart from the fact that I have several email addresses, I have several credit cards. The one I use will depend on which of them is currently offering the best service or the biggest bribes.

There are many good reasons to use pseudonyms. It's certainly sensible for children and young people to use online names that aren't easily linkable to their real names. This is often advocated as a child-protection measure, although the number of children abducted and murdered by strangers in developed countries remains happily low and stable at about 1 per 10,000,000 population per year. A more serious reason is that when you go for your first job on leaving college aged 22, or for a CEO's job at 45, you don't want Google to turn up all your teenage rants. Many people also change email addresses from time to time to escape spam; I give a different email address to every website where I shop. Of course, there are police and other agencies that would prefer people not to use pseudonyms, and this takes us into the whole question of traceability online, which I'll discuss in Part II.

6.4.2.6 Adding Social Context to Naming

The rapid growth recently of social network sites such as Facebook points to a more human and scaleable way of managing naming. Facebook does not give me a visible username: I use my own name, and build my context by having links to a few dozen friends. (Although each profile does have a unique number, this does not appear in the page itself, just in URLs.) This fixes the uniqueness problem — Facebook can have as many Ross Andersons as care to turn up — and the stability problem (though at the cost of locking me into Facebook if I try to use it for everything).

Distributed systems folks had argued for some time that no naming system can be simultaneously globally unique, decentralized, and human-meaningful. It can only have two of those attributes (Zooko's triangle) [21]. In the past, engineers tanded to look for naming systems that were unique and meaningful, like URLs, or unique and decentralised, as with public-key certificates[4]. The innovation from sites like Facebook is to show on a really large scale that naming doesn't have to be unique at all. We can use social context to build systems that are both decentralised and meaningful — which is just what our brains evolved to cope with.

6.4.2.7 Restrictions on the Use of Names

The interaction between naming and society brings us to a further problem: some names may be used only in restricted circumstances. This may be laid down by law, as with the US *Social Security Number* (SSN) and its equivalents in many European countries. Sometimes it is a matter of marketing. I would rather not give out my residential address (or my home phone number) when shopping online, and I avoid businesses that demand them.

Restricted naming systems interact in unexpected ways. For example, it's fairly common for hospitals to use a patient number as an index to medical record databases, as this may allow researchers to use pseudonymous records for some limited purposes without much further processing. This causes problems when a merger of health maintenance organizations, or a new policy directive in a national health service, forces the hospital to introduce uniform names. In the UK, for example, the merger of two records databases — one of which used patient names while the other was pseudonymous — has raised the prospect of legal challenges to processing on privacy grounds.

Finally, when we come to law and policy, the definition of a name turns out to be unexpectedly tricky. Regulations that allow police to collect communications data — that is, a record of who called whom and when — are often very much more lax than the regulations governing phone tapping; in many countries, police can get this data just by asking the phone company. There was an acrimonious public debate in the UK about whether this enables them to harvest the URLs which people use to fetch web pages. URLs often have embedded in them data such as the parameters passed to search engines. Clearly there are policemen who would like a list of everyone who hit a URL such as http://www.google.com/search?q=cannabis+cultivation+UK; just as clearly, many people would consider such large-scale trawling to be an unacceptable invasion of privacy. The police argued that if they were limited to

[4]Carl Ellison, Butler Lampson and Ron Rivest went so far as to propose the SPKI/SDSI certificate system in which naming would be relative, rather than fixed with respect to central authority. The PGP web of trust worked informally in the same way.

monitoring IP addresses, they could have difficulties tracing criminals who use transient IP addresses. In the end, Parliament resolved the debate when it passed the Regulation of Investigatory Powers Act in 2000: the police just get the identity of the machine under the laxer regime for communications data.

6.4.3 Types of Name

The complexity of naming appears at all levels — organisational, technical and political. I noted in the introduction that names can refer not just to persons (and machines acting on their behalf), but also to organizations, roles ('the officer of the watch'), groups, and compound constructions: *principal in role* — Alice as manager; *delegation* — Alice for Bob; *conjunction* — Alice and Bob. Conjunction often expresses implicit access rules: 'Alice acting as branch manager plus Bob as a member of the group of branch accountants'.

That's only the beginning. Names also apply to services (such as NFS, or a public key infrastructure) and channels (which might mean wires, ports, or crypto keys). The same name might refer to different roles: 'Alice as a computer game player' ought to have less privilege than 'Alice the system administrator'. The usual abstraction used in the security literature is to treat them as different principals. This all means that there's no easy mapping between names and principals.

Finally, there are functional tensions which come from the underlying business processes rather than from system design. Businesses mainly want to get paid, while governments want to identify people uniquely. In effect, business wants a credit card number while government wants a passport number. Building systems which try to be both — as many governments are trying to encourage — is a tar-pit. There are many semantic differences. You can show your passport to a million people, if you wish, but you had better not try that with a credit card. Banks want to open accounts for anyone who turns up with some money; governments want them to verify people's identity carefully in order to discourage money laundering. The list is a long one.

6.5 Summary

Many secure distributed systems have incurred huge costs, or developed serious vulnerabilities, because their designers ignored the basic lessons of how to build (and how not to build) distributed systems. Most of these lessons are still valid, and there are more to add.

A large number of security breaches are concurrency failures of one kind or another; systems use old data, make updates inconsistently or in the wrong order, or assume that data are consistent when they aren't and can't be. Knowing the right time is harder than it seems.

Fault tolerance and failure recovery are critical. Providing the ability to recover from security failures, and random physical disasters, is the main purpose of the protection budget for many organisations. At a more technical level, there are significant interactions between protection and resilience mechanisms. Byzantine failure — where defective processes conspire, rather than failing randomly — is an issue, and interacts with our choice of cryptographic tools. There are many different flavors of redundancy, and we have to use the right combination. We need to protect not just against failures and attempted manipulation, but also against deliberate attempts to deny service which may often be part of larger attack plans.

Many problems also arise from trying to make a name do too much, or making assumptions about it which don't hold outside of one particular system, or culture, or jurisdiction. For example, it should be possible to revoke a user's access to a system by cancelling their user name without getting sued on account of other functions being revoked. The simplest solution is often to assign each principal a unique identifier used for no other purpose, such as a bank account number or a system logon name. But many problems arise when merging two systems that use naming schemes that are incompatible for some reason. Sometimes this merging can even happen by accident — an example being when two systems use a common combination such as 'name plus date of birth' to track individuals, but in different ways.

Research Problems

In the research community, secure distributed systems tend to have been discussed as a side issue by experts on communications protocols and operating systems, rather than as a discipline in its own right. So it is a relatively open field, and one still holds much promise.

There are many technical issues which I've touched on in this chapter, such as how we design secure time protocols and the complexities of naming. But perhaps the most important research problem is to work out how to design systems that are resilient in the face of malice, that degrade gracefully, and whose security can be recovered simply once the attack is past. This may mean revisiting the definition of convergent applications. Under what conditions can one recover neatly from corrupt security state?

What lessons do we need to learn from the onset of phishing and keylogging attacks on electronic banking, which mean that at any given time a small (but nonzero) proportion of customer accounts will be under criminal control? Do we have to rework recovery (which in its classic form explores how to rebuild databases from backup tapes) into resilience, and if so how do we handle the tensions with the classic notions of atomicity, consistency,

isolation and durability as the keys to convergence in distributed systems? What interactions can there be between resilience mechanisms and the various protection technologies? In what respects should the protection and resilience mechanisms be aligned, and in what respects should they be separated? What other pieces are missing from the jigsaw?

Further Reading

There are many books on distributed systems; I've found Mullender [912] to be helpful and thought-provoking for graduate students, while the textbook we recommend to our undergraduates by Bacon [104] is also worth reading. Geraint Price has a survey of the literature on the interaction between fault tolerance and security [1043]. The research literature on concurrency, such as the SIGMOD conferences, has occasional gems. There is also a 2003 report from the U.S. National Research Council, *'Who Goes There? Authentication Through the Lens of Privacy'* which discusses the tradeoffs between authentication and privacy, and how they tend to scale poorly [710].