CHAPTER

27

Secure Systems Development

My own experience is that developers with a clean, expressive set of specific security requirements can build a very tight machine. They don't have to be security gurus, but they have to understand what they're trying to build and how it should work.

– RICK SMITH

When it comes to being slaves to fashion, American managers make adolescent girls look like rugged individualists.

– GEOFF NUNBERG

The fox knows many things; the hedgehog one big thing.

– ARCHILOCHUS

27.1 Introduction

So far we've discussed a great variety of security applications, technologies and concerns. If you're a working engineer, manager or consultant, paid to build or maintain a system with some security assurance requirements, you will by now be looking for a systematic way to go about it. This brings us to such topics as risk analysis, system engineering methodology, and, finally, the secret sauce: how you manage a team to write secure code.

The secret is that there isn't actually a secret, whether sauce or anything else. Lots of people claim there is one and get religious fervour for the passion of the moment, from the Orange Book in the 1980s to Agile Development now. But the first take offered on this was the right one. In the 1960s Fred Brooks led the team on the world's first really large software project, the operating system for the IBM S/360 mainframe. In his classic book "The Mythical Man-Month" he describes all the problems they struggled with, and his conclusion is that "there is no silver bullet" [329]. There's no magic formula that makes an intrinsically hard job easy. There's also the famous line from Archilochus at the head of this chapter: the fox knows many things, while the hedgehog

knows one big thing. Managing secure development is fox knowledge rather than hedgehog knowledge. An experienced security engineering manager has to know thousands of little things; that's why this book is so fat! And the security engineering manager's job is getting harder all the time as software gets everywhere and starts to interact with safety.

In 2017, I changed the way I teach undergraduates at Cambridge. Up till then we'd taught security courses separately from software engineering, with the latter focusing on safety. But most real-world systems require both, and they're entangled in complex ways. Both safety and security are emergent properties that really have to be baked in from the beginning. Both involve systematic thinking about what can go wrong, whether by accident or as a result of malice. Accidents can expose systems to attacks, and attacks can degrade systems so they become dangerous. The course was developed further by my colleague Alastair Beresford while I was on sabbatical in 2019, and the 2020 course on software and security engineering is now online as ten video lectures, thanks to the pandemic [90]. That course is designed to give our first-year undergraduates a solid foundation for later work in security, cryptography and software engineering. Like this book, it introduces the basics, from definitions through the basics of protocols and crypto, then the importance of human and organizational issues as well as technical ones, illustrated with case histories. It discusses how you set goals for safety and security, how you manage them as a system evolves, and how you instil suitable ways of thinking and working into your team. Success is about attitudes and work practices as well as skills.

The two questions you have to ask are, "Are we building the right system?" and "Are we building it right?" In the rest of this chapter I'm going to start with how we assess and manage risks – to both safety and security; and then go on to discuss how we build systems, once we've got a specification to work to. I'll then discuss some of the hazards that arise as a result of organisational behaviour – a real but often ignored kind of insider threat.

27.2 Risk management

At the heart of both safety engineering and security engineering lie decisions about priorities: how much to spend on protection against what. Risk management must be done within a broader framework of managing all the risks to an enterprise or indeed to a nation. That is often done badly. The coronavirus crisis should have made it obvious to everyone that although pandemics were at the top of the risk register of many countries, including the UK, most governments spent much more of their resilience budget on terrorism, which was several places down the list. Countries with recent experience of SARS or MERS, such as Taiwan and South Korea, did better: they were ready to test residents and trace contacts at scale, and responded quickly. Britain wasted two

months before realising the disease was serious, at a cost of tens of thousands of lives.

So what actually is a *risk register*? A common methodology, as used by the governing body of my university, is to draw up a list of things that could go wrong, giving them scores of 1 to 5 for seriousness and for probability of occurrence, and multiplying these together to get a number between 1 and 25. For example, a university might rate 'loss of research contract income due to economic downturn' at 5/5 for seriousness if 20% of its income is from that source, and rate 'probability' at 4/5 as downturns happen frequently but not every year, giving a raw product of 20. You then write down the measures you take to mitigate each of these risks, and have an argument in a risk committee about how well each risk is mitigated. For example, you control the risk of variable research contract income by making a rule that it can be used to hire only contract staff, not tenured faculty; you might then agree that this rule cuts that risk from 20 to 16. You then rank all the risks in order and assign one senior officer to be the owner of each of them.

National risk assessments are somewhat similar: you rate each possible bad event (pandemic, earthquake, forest fire, terrorist attack, …) by how many people it might kill (millions? thousands? dozens?) and then rate it for probability by how many you expect each century. The UK national risk register, for example, put pandemic influenza at the top, with a 5 for severity (could kill up to 750,000) and a 4 for likelihood, saying in 2017: "one or more major hazards can be expected to materialise in the UK in every five-year period. The most serious are pandemic influenza, national blackout and severe flooding" [363]. You then work out what's reasonably practical by way of mitigation, be it quarantine plans and PPE stockpiles for a pandemic, or building codes and zoning to limit the damage from floods and earthquakes. You do the cost-benefit analysis and turn priorities into policy. You can get things wrong in various ways. The UK largely ignored pandemics because the National Security Council had been captured by the security and intelligence agencies; they prioritised terrorism, and the health secretary was not a regular attendee [1852]. I already discussed terrorism in section 26.3; here I'll just add that another aspect of the failure was policy overshoot. When 9/11 taught the world that terrorist attacks can kill thousands rather than just dozens, and the agencies got a lot more of the resilience budget, it made them greedy: they started talking up the risk of terrorists getting hold of a nuke so they'd have an even scarier threat on the register to justify their budgets.

In business too you can find that both political behaviour and organisational behaviour get in the way of rational risk management. But you often have real data on the more common losses, so you can attempt a more quantitative approach. The standard method is to calculate the *annual loss expectancy* (ALE) for each possible loss scenario, as the expected loss multiplied by the number of incidents expected in an average year. A typical ALE analysis for a bank's

IT systems might have several hundred entries, including items such as we see in Figure 27.1.

Loss type	Amount	Incidence	ALE
SWIFT fraud	$50,000,000	.005	$250,000
ATM fraud (large)	$250,000	.2	$100,000
ATM fraud (small)	$20,000	.5	$10,000
Teller takes cash	$3,240	200	$648,000

Figure 27.1: Items of annualized loss expectancy (ALE)

Note that while accurate figures are likely to be available for common losses (such as 'teller takes cash'), the incidence of low-probability high-risk losses such as a large money-transfer fraud is largely guesswork – though you can sometimes get a rough sanity check by asking for insurance quotes.

ALEs have long been standardized by NIST as the technique to use in US government procurements. The UK government uses a tool called CRAMM for systematic analysis of information security risks, and the modern audit culture is spreading such tools everywhere. But the process of producing such a table for low-probability threats tends to be just iterative guesswork. The consultants list all the threats they can think of, attach notional probabilities, work out the ALEs, add them up, and find that the bank's ALE exceeds its income. They then tweak the total down to whatever will justify the largest security budget that their client the CISO has said is politically possible. I'm sorry if this sounds a bit cynical; but it's what often seems to happen. The point is, ALEs may be of some value, but you need to understand what parts are based on data, what parts on guesswork and what parts on office politics.

Product risks are different. Different industries do things differently because of the way they evolved and the history of regulation. The rules for each sector, whether cars or aircraft or medical devices or railway signals, have evolved in response to accidents and industry lobbying. Increasingly, the European Union is becoming the world's safety regulator as it's the biggest market, as Washington cares less about safety than Brussels does, and as it's simpler for OEMs to engineer to EU safety specifications than to have multiple products. I'll discuss safety and security certification in more detail in the next chapter. For present purposes, software for cars, planes and medical devices must be developed according to approved procedures, subjected to analyses we'll discuss later, and tested in specific ways.

Insurance can be of some help in managing large but unlikely risks. But the insurance business is not completely scientific either. Your insurance premiums used to give some signal of the risk your business was running, especially if you bought cover for losses of eight figures or above. But insurance is a cyclical

industry, and since about 2017 a host of new companies have started offering insurance against cybercrime, squeezing the profits out of the market. As a result, customers will no longer put up with intrusive questionnaires, let alone site visits from assessors. So most insurers' ability to assess risk is now limited; I will discuss the mechanics of what they do further in section 28.2.9. They are also wary of correlated risks that give rise to many claims at once, as that would force them to hold greater reserves; as some cyber risks are correlated, policies tend to either exclude them or be relatively expensive [276]. (The coronavirus crisis is teaching firms about correlated risk as some insurers refuse to pay up on business-interruption risk policies – even those that explicitly mention the risk of staff not being able to get to the office because of epidemics; businesses are asking insurers in turn what the point of insurance is.)

Actuarial risks aside, a very important reason for large companies to take out insurance cover – and for much other corporate behaviour – is to protect executives, rather than shareholders. The risks that are being tackled may seem on the surface to be operational but are actually legal, regulatory and PR risks. Directors demand liability insurance, and under UK and US law, professional negligence occurs when a professional fails to perform their responsibilities to the level required of a reasonably competent person in their profession. So negligence claims are assessed by the current standards of the industry or profession, giving a strong incentive to follow the herd. This is one reason why management is such a fashion-driven business (as per the quote at the head of this chapter). This spills over into the discourse used to justify security budgets. During the mid 1980s, everyone talked about hackers (even if their numbers were tiny). From the late 80s, viruses took over the corporate imagination, and people got rich selling antivirus software. In the mid-1990s, the firewall became the star product. The late 1990s saw a frenzy over PKI. By 2017 it was blockchains. Amidst all this hoopla, the security professional must keep a level head and strive to understand what the real threats are.

We will return to organisational behaviour in a later section. First, let's see what we can learn from safety engineering.

27.3 Lessons from safety-critical systems

Critical computer systems are those in which a certain class of failure is to be avoided if at all possible. Depending on the class of failure, they may be safety-critical, business-critical, security-critical, or critical to the environment. Obvious examples of the safety-critical variety include flight controls and automatic braking systems. There's a large literature on this subject, and a lot of methodologies have been developed to help manage risk intelligently.

27.3.1 Safety engineering methodologies

Safety engineering methodologies, like classical security engineering, tend to work systematically from a safety analysis to a specification through to a product, and assume you're building safety in from the start rather than trying to retrofit it. The usual procedure is to identify hazards and assess risks; decide on a strategy to cope with them (avoidance, constraint, redundancy …); trace the hazards to hardware and software components which are thereby identified as critical; identify the operator procedures which are also critical and study the various applied psychology and operations research issues; set out the safety functional requirements which specify what the safety mechanisms must do, and safety integrity requirements that specify the likelihood of a safety function being performed satisfactorily; and finally decide on a test plan. The outcome of testing is not just a system you're confident to run live, but an integrated part of a *safety case* to justify running it. The basic framework is set out in standards such as ISO 61508, a basic safety framework for relatively simple programmable electronics such as the control systems for chemical plants. This has been extended with more specialised standards for particular industries, such as ISO 26262 for road vehicles.

This safety case will provide the evidence, if something does go wrong, that you exercised due care. It will typically consist of the hazard analysis, the safety functional and integrity requirements, and the results of tests (both at component level and system level), which show that the required failure rates have been achieved. The testing may have to be done by an accredited third party; motor vehicles firms get away with the safety case being done by a different department in the same company, with independent management. Vehicles are a more complex case because of their supply chains. At the top is the brand, whose badge you see on the front of the car. Then there's the *original equipment manufacturer* (OEM), which in the case of cars is usually the same company, but not always; in other industries the brand and the OEM are quite separate. A modern car will have components from dozens of manufacturers, of which the Tier 1 suppliers who deal directly with the brand do much of the research and development work but get components from other firms in turn. In the car industry, the brand puts the car through type approval and carries the primary liability, but demands indemnities from component suppliers in case things go wrong (the law in most countries does not allow you to disclaim liability for death and injury). The brand relies on the supply chain for significant parts of the safety functionality and integrity and thus for the safety case. There are also tensions: as we already noted, safety certification can prevent the timely application of security patches. Let's now look at common safety engineering methods and what they can teach us.

27.3.2 Hazard analysis

In an ideal case, we might be able to design hazards out of a system completely. As an example, consider the motor reversing circuits in Figure 27.2. In the design on the left, a double-pole double-throw switch reverses the current passing from the battery through the motor. However, this has a potential problem: if only one of the two poles of the switch moves, the battery will be shorted, and a fire may result. The solution is to exchange the battery and the motor, as in the modified circuit on the right. Here, a switch failure will only short out the motor, not the battery. Safety engineering is not just about correct operation, but about correct failure too.

Hazard elimination is useful in security engineering too. We saw an example in the early design of SWIFT in section 12.3.2: there, the keys used to authenticate transactions between one bank and another were exchanged between the banks directly, so SWIFT did not have the means to forge a valid transaction, and its staff and systems had to be trusted less. In general, minimizing the trusted computing base is an exercise in hazard elimination. The same applies in privacy engineering too. For example, if you're designing a contact tracing app to monitor who might have infected whom in an epidemic, one approach is to have a central database of everyone's mobile phone location history. However, that has obvious privacy hazards, which can be reduced by keeping a Bluetooth contact history on everyone's mobile phone instead, and uploading the contact history of anyone who calls in sick. You then have a policy decision to take between better privacy and better tracing.

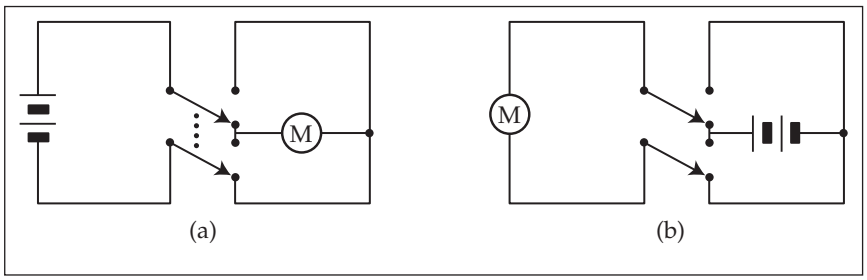

Figure 27.2: Hazard elimination in motor reversing circuit

27.3.3 Fault trees and threat trees

Once you have eliminated as many hazards as possible, the next step is to identify failures that could cause accidents. A common top-down way of identifying the things that can go wrong is *fault tree analysis* where a tree is

constructed whose root is the undesired behavior and whose successive nodes are its possible causes. This top-down approach is natural where you have a complex system with a small number of well-known bad outcomes that you have to avoid. It carries over in a natural way to security engineering. Figure 27.3 shows an example of a fault tree (or *threat tree*, as it's often called in security engineering) for fraud from automatic teller machines.

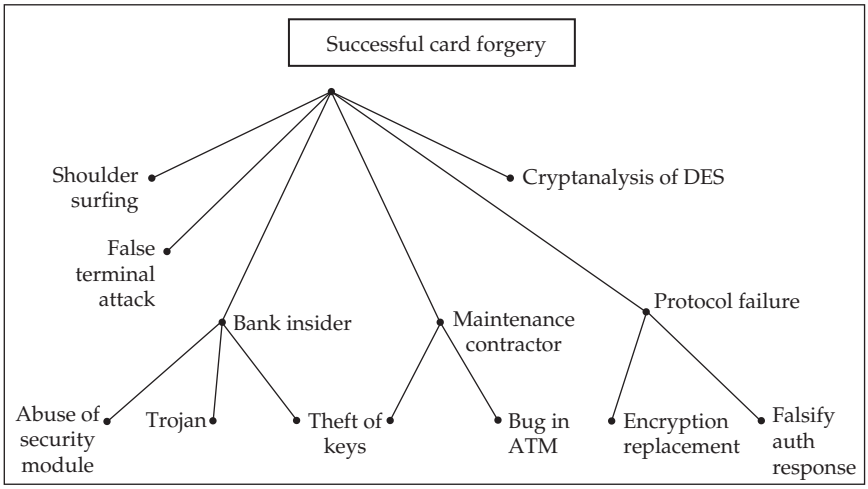

Figure 27.3: A threat tree

Threat trees are used in the US Department of Defense. You start out from each undesirable outcome, and work backwards by writing down each possible immediate cause. You then recurse by adding each precursor condition. By working round the tree's leaves you should be able to see each combination of technical attack, operational blunder, physical penetration and so on that could break your security policy. The other nice thing you get from this is a visualisation of commonality between attack paths, which makes it easier to reason about how to disrupt the most attacks with the least effort. In some variants, attack branches have countermeasure sub-branches, which may have counter-countermeasure attack branches, and so on, in different colours for emphasis. A threat tree can amount to an attack manual for the system, so it may be highly classified, but it's a DoD requirement – and if the system evaluators or accreditors can find significant extra attacks, they may fail the product.

27.3.4 Failure modes and effects analysis

Returning to the safety-critical world, another way of doing hazard analysis is *failure modes and effects analysis* (FMEA), pioneered by NASA, which is

bottom-up rather than top-down[1]. This involves tracing the consequences of a failure of each of the system's components all the way up to the effect on the mission. This is the natural approach in systems with a small number of well-understood critical components or subsystems, such as aircraft. For example, if you're going to fly a plane over an ocean or mountains where you can't glide to an airport in the case of engine failure, then engine power is critical. You therefore study the mean time to failure of your powerplant and its failure modes, from a broken connecting rod to running out of fuel. You insist that single-engine aircraft use reliable engines and you regulate the maintenance schedules; planes have more than one fuel tank. When carrying a lot of passengers, you insist on multi-engine aircraft and drill the crews to deal with engine failure.

An aerospace example of people missing a failure mode that turned out to be critical is the 1986 loss of the space shuttle Challenger. The O-rings in the booster rockets were known to be a risk by the NASA project manager, and damage had been found on previous flights; meanwhile the contractor knew that low temperatures increased the risk; but the concerns did not come together or get through to NASA's top management. An O-ring, made brittle by the cold, failed – causing the loss of the shuttle and seven crew. On the resulting board of inquiry, the physicist Richard Feynman famously demonstrated this on TV by putting a sample of O-ring in a clamp, freezing it in iced water and then showing that when he released it, it remained dented and did not spring back [1615]. This illustrates that failures are often not just technical but also involve how people behave in organisations: when protection mechanisms cross institutional boundaries, as for example with cars, you need to think of the law and economics as well as just the engineering. Such problems will become much more complex as we move towards autonomous vehicles, which will rely on all sorts of third-party services and infrastructure.

27.3.5 Threat modelling

Both fault trees and FMEA depend on the analyst understanding the system really well; they are hard to automate, not fully repeatable and can be up-ended by a subtle change to a subsystem. So a thorough analysis of failure modes will often combine top-down and bottom-up approaches with some methods specific to the application that people have learned over time. Many industries now have to rethink their traditional safety analysis methods to incorporate security.

[1]FMEA is bottom-up in the technical sense that the analysis works up from individual components, but its actual management often has a top-down flavour as you start work on the safety case once you have an outline design and refine it progressively as the design is evolved into a product.

In car safety, complex supply chains mean we have to do multiple interlocking analyses of vehicles and their subsystems. A traditional subsystem analysis might work through the failure modes of headlamps, since losing them while driving at night can lead to an accident. As well as mitigating the risk of a lamp failure by having two or more lamps, you worry about switch failure, and when the switch becomes electronic you build a fault tree of possible hardware and software faults. When we extend this from safety to security, we think about whether an attacker might take over the entertainment system in a car, and use it to send a malicious 'lamp off' message on the CAN bus once the car is moving quickly enough for this to be dangerous. This analysis may lead to a design decision to have a firewall between the cabin CAN bus and the powertrain CAN bus. (This is the worked example in the new draft ISO 21434 standard for cybersecurity in road vehicles [964].)

More generally, the shift from safety to security means having to think systematically about insiders. Just as double-entry bookkeeping was designed to be resilient against a single dishonest clerk and has been re-engineered against the similar threat of a clerk with malware on their PC, so modern large-scale systems are typically designed to limit the damage if a single component is compromised. So how can you incorporate malicious insiders into a threat model? If you're using FMEA, you can just add an opponent at various locations, as with our malicious 'lamp off' message. As for more complex systems, the methodology adopted by Microsoft following its big push in 2003 to make Windows and Office more secure is described by Frank Swiderski and Window Snyder [1855]. Rather than being purely top-down or bottom-up, this is a meet-in-the-middle approach. The basic idea is that you list not just the assets you're trying to protect (ability to do transactions, access to confidential data, whatever) but also the assets available to an attacker (perhaps the ability to subscribe to your system, or to manipulate inputs to the smartcard you supply him, or to get a job at your call center). You then trace the attack paths through the system, from one module to another. You try to figure out what the trust levels might be; where the barriers are; and what techniques, such as spoofing, tampering, repudiation, information disclosure, service denial and elevation of privilege, might be used to overcome particular barriers. The threat model can be used for various purposes at different points in the security development lifecycle, from architecture reviews through targeting code reviews and penetration tests.

There are various ways to manage the resulting mass of data. An elementary approach is to construct a matrix of hazards against safety mechanisms, and if the safety policy is that each serious hazard must be constrained by at least two independent mechanisms, then you can check for two entries in each of the relevant columns. So you can demonstrate graphically that in the presence of the hazard in question, at least two failures will be required to cause an accident. An alternative approach, *system theoretic process analysis* (STPA),

starts off with the hazards and then designs controls in a top-down process, leading to an architectural design for the system; this can be helpful in teasing apart interacting control loops [1152]. Such methodologies go across to security engineering [1559]. One way or another, in order to make the complexity manageable, you may have to organise a hierarchy of safety and security goals. The security policies discussed in Part 2 of this book may give you the beginnings of an answer for the applications we discussed there, and some inspiration for others. This hierarchy can then drive a risk matrix or risk treatment plan depending on the terminology in use in your industry.

27.3.6 Quantifying risks

The safety-critical systems community has a number of techniques for dealing with failure and error rates. Component failure rates can be measured statistically; the number of bugs in software can be tracked by techniques I'll discuss in the next chapter; and there is a lot of experience with the probability of operator error at different types of activity. The bible for human-factors engineering in safety-critical systems is James Reason's book *'Human Error'*; I discussed in Chapter 3 the rising tide of research in security usability through the 2010s as the lessons from the safety world have started to percolate into our field.

The error rate in a task depends on its familiarity and complexity, the amount of pressure and the number of cues to success. Where a task is simple, performed often and there are strong cues to success, the error rate might be 1 in 100,000 operations. However, when a task is performed for the first time in a confusing environment where logical thought is required and the operator is under pressure, then the odds can be against successful completion. Three Mile Island and Chernobyl taught nuclear engineers that no matter how many design walkthroughs you do, it's when the red lights go on for real that the worst mistakes get made. The same lesson has come out of one air accident investigation after another. When dozens of alarms go off at once, there's a fair chance that someone will push the wrong button. One guiding principle is to default to a safe state: to damp down a nuclear reaction, to return an aircraft to straight and level flight, or to bring an autonomous vehicle to a stop at the side of the road. No principle is foolproof, and a safe state may be hard to measure. A vehicle can find it hard to tell where the side of the road is if there's a grass verge; and in the Boeing 737Max crashes (which I describe in detail in section 28.2.4) the flight control computer tried to keep the plane level but was confused by a faulty angle-of-attack sensor and dived the plane into the ground instead.

Another principle of safety usability in an emergency is to keep the information given to operators, and the controls available for them to use, both simple and intuitive. In the old days, each feed went to a single gauge or dial and

there was only so much space for them. The temptation nowadays is to give the operator everything, because you can. In the old days, designers knew that an emergency would give the pilots tunnel vision so they put the six instruments they really needed right in the middle. Nowadays there can be fifty alarms rather than two and pilots struggle to work out which screen on which menu of the electronic flight information system to look at. It is much broader than aviation. A naval example is the 2017 collision of the USS McCain in the Straits of Singapore, where UI confusion was a major factor. Steering control was shifted to the wrong helm station and an engine was not throttled back in time, resulting in an uncommanded turn to port across a busy shipping lane, impact with a chemical tanker, and the death of ten sailors [1933].

So systems that are not fully autonomous must remain controllable, and for that the likely human errors need to be understood. Quite a lot is known about the cognitive biases and other psychological factors that make particular types of error more common; we discussed them in Chapter 3, and a prudent engineer will study how they work out in their field. Errors are rare in frequently-performed tasks at which the operator has developed some skill, and are more likely when operators are stressed and surprised. This starts to get us out of the territory of risk, where the odds are known, and into that of uncertainty, where they're not.

In security systems, too, the most egregious blunders can be expected in important but rarely performed tasks. Security usability isn't just about presenting a nice intuitive interface to the end-user. It should present the risks in a way that accords with common mental models of threat and protection, and the likely user reactions to stress should lead to safe outcomes.

It is important to be realistic about the skill level of the people who will perform each critical task and any known estimates of the likelihood of error. An airplane designer can rely on a predictable skill level from anyone with a commercial pilot's license, and a shipbuilder knows the strengths and weaknesses of an officer in the merchant marine. Cars can and do get operated by drivers who are old and frail, young and inexperienced, distracted by passengers, or under the influence of alcohol. At the professional end of things, usability testing can be profitably integrated with staff training: when pilots go for their refresher courses in the simulator, instructors throw all sorts of combinations of equipment failure, bad weather, cabin crisis and air-traffic-control confusion at them. They observe what combinations of stress result in fatal accidents, and how these differ across cockpit types. Such data are valuable feedback to cockpit designers. In aviation, the incentives for safe operation are sufficiently strong and well aligned, and the scale is large enough, to support a learning system. Even so, there are expensive disasters, such as the Boeing 737Max flight control software. This not only had at least one serious bug, but escaped a proper failure modes and effects analysis because the engineers responsible – under pressure from their managers to

complete the project on time – wrongly assumed that pilots would be able to cope with any failure [90]. As a result, the software relied on a single angle-of-attack sensor rather than using the two sensors with which the aircraft was fitted, and sensor failure led to fatal accidents[2].

When testing the usability of redundant systems, you need to pay attention to *fault masking*: if the output is determined by majority voting between three processors, and one of them fails, then the system will continue to work fine – but its safety margin will have been eroded, perhaps in ways the operators won't understand properly. Several air crashes have resulted from flying an airliner with one of the cockpit systems out of action; although pilots may be intellectually aware that one of the data feeds to the cockpit displays is unreliable, they may rely on it under pressure by reflex rather than checking with other instruments. So you have to think hard about how faults can remain visible and testable even when their immediate effects are mitigated.

Another lesson from safety-critical systems is that although a safety requirements specification and test criteria will be needed as part of the safety case for the lawyers and regulators, it is good practice to integrate both of them with the mainstream product documentation. If the safety case is separate, then it's easy to sideline it after approval and fail to maintain it properly. (This was a factor in the Boeing 737Max disaster as the usability assumptions underlying the safety case for the flight control software were not updated from the previous model of 737.) The move from project-based software management to agile methodologies, and via DevOps to DevSecOps, is finally starting to embed security management into the way products evolve. We will discuss this in the next section.

Finally, safety is like security in that it really has to be built in as a system is developed, rather than retrofitted. The main difference between the two is in the failure model. Safety deals with the effects of random failure, while in security we assume a hostile opponent who can cause some of the components of our system to fail at the least convenient time and in the most damaging way possible. People are naturally more risk-averse in the presence of an adversary; I will discuss this in section 28.4. A safety engineer will certify a critical flight-control system with an MTBF of 10^9 hours; a security engineer has to worry whether an adversary can force the preconditions for that one-in-a-billion failure and crash the plane on demand.

In effect, our task is to program a computer that gives answers that are subtly and maliciously wrong at the most inconvenient moment possible. I've described this as 'programming Satan's computer' to distinguish it from the more common problem of programming Murphy's [114]. This is one of the reasons security engineering is hard: Satan's computer is harder to test [1671].

[2] Aviation safety standards such as DO178 and DO254 generally require diversity in measurement type, physics, processing characteristics in addition to redundancy to mitigate common-mode failures.

27.4 Prioritising protection goals

If you've a project to create an entirely new product, or to radically change an existing one, it's an idea to spend some time thinking through the protection priorities from first principles. A careful safety analysis or threat modelling exercise can provide some numbers to inform this. When developing a safety case or a security policy in detail, it's essential to understand the context, and much of this book has been about the threat models relevant to a wide range of applications. You should try to refine numerical estimates of risk from the environment or context as well.

In the case of a business system, analysis will hinge on the tradeoff between risk and reward. Security people often focus too much on the former. If your firm has a turnover of $10m, gross profits of $1m and theft losses of $150,000, you might make a loss-reduction pitch about 'how to increase profits by 15% by stopping theft'; but if you could double the turnover to $20m, then the shareholders would prefer that even if it triples the losses to $450,000. Profit is now $1.55m, up 85%, rather than 15%. This is borne out by the experience of online fraud engines. When discussing fraud management strategies with a number of retailers, I noticed that the firms who got the best results were those where the fraud management team reported to sales rather than finance. A typical bricks-and-clicks retailer in the UK might decline something like 4% of offered shopping baskets because the fraud engine alerts at the combination of goods, delivery address and payment details. So if you can improve the fraud engine and reject only 3%, that's 1% more sales – a prospect to light up your Chief Marketing Officer's eyes. But if the fraud team reports instead to the Chief Financial Officer, they're likely to be seen as a cost rather than as an opportunity.

Similarly, the site reliability engineers of online services have learned not to make a system too reliable. If local Internet availability is only 99%, then a service that's up 99.9% of the time will be fine; there's no point spending millions more to hit 99.99% if none of your users will notice the difference. You're better off deliberately setting an 0.1% *error budget*, which you can use productively – such as by causing occasional deliberate failures to exercise your resilience mechanisms [237]. This brings me to one of the open debates in security management: should one aim at having no CVEs open in any of the software on which one relies? The tick-box approach is to say 'Of course there must be no open CVEs', but that may impose a rather high compliance cost. If you're Google, and wrote all your own infrastructure, maybe you can aim at that; many firms can't and have to prioritise. I'll discuss CVEs in more detail in section 27.5.7.1 later.

So don't trust people who can only talk about 'tightening security'. Often it's too tight already, and what you really need to do is just focus it slightly

differently. In the first edition of this book, I presented a case study of self-service checkout at supermarkets. Twenty years ago, a number of supermarkets started to introduce self-checkout lanes. Some started to obsess about losses, and let security get in the way of usability by aggressively challenging customers about product weight. One of the stores that got an advantage started with a more forgiving approach that they tuned up gradually in the light of experience. Eventually the industry figured out how to operate self-checkout lanes, but the quality of the implementation still varies significantly. By early 2020, the pioneers are small convenience stores like Lifvs in Sweden that have no staff; you open the store's door with an app, scan your purchases and pay online. Amazon was also experimenting with fully self-service food stores. We saw the next 20 years of innovation crammed into the few months of the 2020 coronavirus lockdown; by June, other supermarkets have been urging us to download their scanning app, scan our purchases as we pick them, charge them to a card, and just go.

Many modern business models were once considered too risky, starting with the self-service supermarket itself back in the days when grocers kept all the goods behind the counter. Everyone thought Richard Sears would go bust when he adopted the slogan 'Satisfaction guaranteed or your money back' in the 1880s, yet he invented the modern mail-order business. In business, profit is the reward for risk. But entrepreneurs who succeed may have to improve security quickly. One recent example is the videoconferencing platform Zoom – which grew from 20 million users to 200 million in March 2020, and changed in the process from an enterprise platform into something more like a public utility – forcing them into a major security engineering effort [1767].

Trade-offs in safety are harder. Logically, the value of a human life in a developed country might be a few million dollars, that being an average person's lifetime earnings. However, our actual valuation of a human life as revealed by safety behaviour varies from about $50,000 for improvements to road junctions, up to over $500m for train protection systems – and that's just in the context of transport policy. The variance in health policy is even greater, with costs per life saved ranging from a few hundred dollars for flu jabs and some cancer screening to billions for the least effective interventions [1872]. In other safety contexts, domestic smoke alarms cost a few hundred dollars per life saved while the number for the "war on terror" is in the billions [1352]. The reasons for this irrationality are fairly well understood – I discussed the psychology in section 3.2.5 and the policy aspects in 26.3.3. Safety preferences can be changed very sharply by the threat of hostile action; people may shrug off a 1-in-10,000 risk of being killed by poorly-designed medical devices until there's a possibility that the devices might be hacked, at which point even a 1-in-10,000,000 risk becomes scary. I discuss this phenomenon in section 28.4.

27.5 Methodology

Software projects usually take longer than planned, cost more than budgeted and have more bugs than expected[3]. By the 1960s, this had become known as the *software crisis*, although the word 'crisis' may be inappropriate for a state of affairs that has now lasted, like computer insecurity, for two generations. Anyway, the term *software engineering* was proposed by Brian Randall in 1968 and defined to be:

> *Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

The pioneers hoped that the problem could be solved in the same way we build ships and aircraft, with a foundation in basic science and a framework of design rules [1422]. Since then there's been a lot of progress, but the results have been unexpected. Back in the late 1960s, people hoped that we'd cut the number of large software projects failing from the 30% or so that was observed at the time. But we still see about 30% of large projects failing – the difference is that the failures are much bigger. Modern tools get us farther up the complexity mountain before we fall off, but the rate of failure is set by company managers' appetite for risk. We'll discuss this further in section 27.5.8 at the end of this chapter.

Software engineering is about managing complexity, of which there are two kinds. There is the *incidental complexity* involved in programming using inappropriate tools, such as the assembly languages which were all that some early machines supported; programming a modern application with a graphical user interface in such a language would be impossibly tedious and error-prone. There is also the *intrinsic complexity* of dealing with large and complicated problems. A bank's core systems, for example, may involve tens of millions of lines of code that implement hundreds of different products sold through several different delivery channels, and are just too much for any one person to understand.

Incidental complexity is largely dealt with using technical tools. The most important are high-level languages that hide much of the drudgery of dealing with machine-specific detail and enable the programmer to develop code at an appropriate level of abstraction. They bring their own costs; many vulnerabilities are the result of the properties of the C language, and if we were rerunning history we'd surely use something like Rust instead. There are also formal methods such as static analysis tools, that enable particularly error-prone design and programming tasks to be checked.

[3]This is sometimes known as "Cheops' law" after the builder of the Great Pyramid.

Intrinsic complexity requires something subtly different: methodologies that help us divide up a problem into manageable subproblems and restrict the extent to which these subproblems can interact. These in turn are supported by their own sets of tools. There are basically two approaches – top-down and iterative.

27.5.1 Top-down design

The classical model of system development is the *waterfall model* formalised by Win Royce in the 1960s for the US Air Force [1631]. The idea is that you start from a concise statement of the system's requirements; elaborate this into a specification; implement and test the system's components; then integrate them together and test them as a system; then roll out the system for live operation (see Figure 27.4). From the 1970s until the mid-2000s, this was how all systems for the US Department of Defense were supposed to be developed, and their lead was followed by many governments worldwide, including not just in defence but in administration and healthcare. When I worked in banking in the 1980s, it was the approved process there too, promoted assiduously by IBM, by governments and by the big accountancy firms.

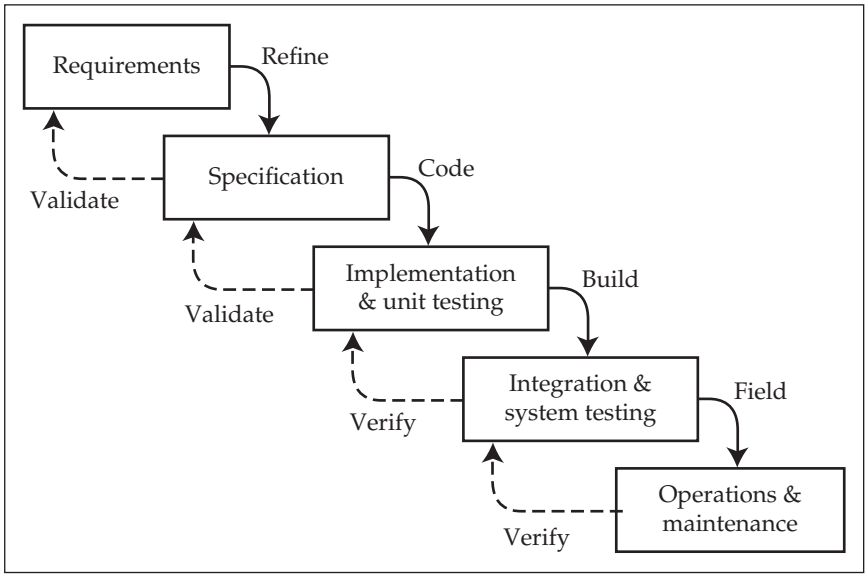

Figure 27.4: The waterfall model

The idea is that the requirements are written in the user language, the specification is written in technical language, the unit testing checks the units against the specification and the system testing checks whether the requirements are met. At the first two steps in this chain there is feedback on whether we're

building the right system (*validation*) and at the next two on whether we're building it right (*verification*). There may be more than four steps: a common elaboration is to have a sequence of *refinement* steps as the requirements are developed into ever more detailed specifications. But that's by the way.

The defining feature of the waterfall model is that development flows inexorably downwards from the first statement of the requirements to the deployment of the system in the field. Although there is feedback from each stage to its predecessor, there is no system-level feedback from (say) system testing to the requirements.

There is a version used in safety-critical systems development called the V model, where the system flows down to implementation, then climbs back up a hill of verification and validation on the other side, where it's tested successively against the implementation, the specification and the requirements. This is a German government standard, and also used in the aerospace industry worldwide; it's found in the ISO 26262 standard for car software safety. But although it's written from left to right rather than top-down, it's still a one-way process where the requirements drive the system and the acceptance test ensures that the requirements were met, rather than a mechanism for evolving the requirements in the light of experience. It's more a different diagram than a different animal.

The waterfall model had a precursor in a methodology developed by Gerhard Pahl and Wolfgang Beitz in Germany just after World War II for the design and construction of mechanical equipment such as machine tools [1492]; apparently one of Pahl's students later recounted that it was originally designed as a means of getting the engineering student started, rather than as an accurate description of what experienced designers actually do. Win Royce also saw his model as a means of starting to get order out of chaos, rather than as the prescriptive system it developed into.

The strengths of the waterfall model are that it compels early clarification of system goals, architecture, and interfaces; it makes the project manager's task easier by providing definite milestones to aim at; it may increase cost transparency by enabling separate charges to be made for each step, and for any late specification changes; and it's compatible with a wide range of tools. Where it can be made to work, it's often the best approach. The critical question is whether the requirements are known in detail in advance of any development or prototyping work. Sometimes this is the case, such as when writing a compiler or (in the security world) designing a cryptographic processor to implement a known transaction set and pass a certain level of evaluation. Sometimes a top-down approach is necessary for external reasons, as with an interplanetary space probe where you'll only get one shot at it.

But very often the detailed requirements aren't known in advance and an iterative approach is necessary. The technology may be changing; the environment could be changing; or a critical part of the project may be the design of

a human-computer interface, which will probably involve testing several prototypes. Very often the designer's most important task is to help the customer decide what they want, and although this can sometimes be done by discussion, there will often be a need for some prototyping.

Sometimes a formal project is just too slow. Reginald Jones attributes much of the UK's relative success in electronic warfare in World War II to the fact that British scientists hacked stuff together quickly, while the Germans used a rigid top-down development methodology, getting beautifully engineered equipment but always six months too late [993].

But the most common reason for using iterative development is that we're starting from an existing product that we want to improve. Even in the early days of computing, most programmer effort was always expended on maintaining and enhancing existing programs rather than developing new ones; surveys suggest that 70–80% of the total cost of ownership of a successful IT product is incurred after it first goes into service, even when a waterfall methodology was used [2063]. Nowadays, as software becomes a matter of embedded code, apps and cloud services–which all become ever more complex–the reality in many firms is that 'the maintenance is the product'.

Even in the late 1990s, when the most complex human artefacts were software packages such as Microsoft Office, the only way to write such a thing was to start off from the existing version and enhance it. That does not make the waterfall model obsolete; on the contrary, it is often used to manage a project to develop a major new feature, or to refactor existing code. However, the overall management of a major product nowadays is likely to be based on iteration.

27.5.2 Iterative design: from spiral to agile

There are different flavours of iterative development, ranging from a rapid prototyping exercise to firm up the specification of a new product, through to a managed process for fixing or enhancing an existing system.

In the first case, one approach is the *spiral model* in which development proceeds through a pre-agreed number of iterations in which a prototype is built and tested, with managers being able to evaluate the risk at each stage so they can decide whether to proceed with the next iteration or to cut their losses. Devised by Barry Boehm, it's called the spiral model because the process is often depicted as in Figure 27.5. There are many applications where an initial prototype is the key first step; from a startup aiming to produce a demo to show to investors, through a company building a mockup of a new product to show a focus group, to DARPA seedling projects that aim to establish that some proposed technology isn't completely impossible. Prototype applications for the security engineer range from security usability testbeds to proof-of-concept attack code. The key is to solve the worst problem you're facing, so as to reduce the project risk as much as possible.

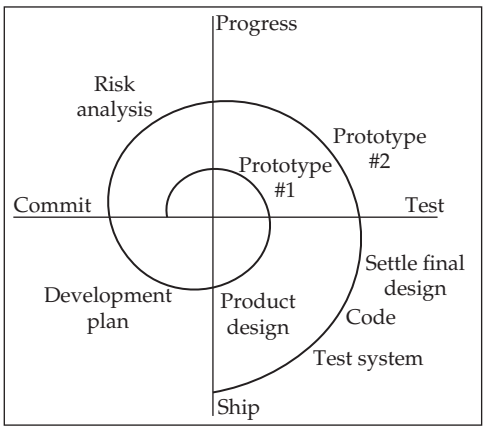

Figure 27.5: The spiral model

The second case we now describe as *agile development*, which may be summed up in the slogan: "Solve your worst problem. Repeat".

An early advocate for an evolutionary approach was Harlan Mills, who taught that you should build the smallest system that works, try it out on real users, and then add functionality in small increments. This is how the packaged software industry had learned to work by the 1990s: as PCs became more capable, software products became so complex that they could not be economically developed (or redeveloped) from scratch. Indeed, Microsoft tried more than once to rewrite Word, but gave up each time. A landmark early book on evolutionary development was 'Debugging the Development Process' by Steve Maguire of Microsoft in 1994 [1211]. In this view of the world, products aren't the result of a project but of a process that involves continually modifying previous versions. Microsoft contrasted its approach with that of IBM, then still the largest IT company; in the IBM ecosystem, the waterfall approach was dominant. (IBMers for their part decried Microsoft as a bunch of undisciplined hackers who produced buggy, unreliable code; but IBM's near-death experience after Microsoft stole their main business markets has been ascribed to the rigidity of the IBM approach to development [392].) Professional practice has evolved in the quarter century since then, and evolutionary development is now known as 'agile', but it is recognisably the same beast.

A key insight about evolutionary development is that just as each generation of a biological species has to be viable for the species to continue, so each generation of an evolving software product must be viable. The core technology is *regression testing*. At regular intervals – typically once a day – all the teams working on different features of a product check in their code, which gets compiled to a *build* that is then tested automatically against a large set of inputs. The regression test checks whether things that used to work still work, and that old

bugs haven't found their way back. It's always possible that someone's code broke the build, so we consider the current 'generation' to be the last build that worked. Things are slightly more complex when systems have to work together, as when an app has to talk to a cloud service, or when several electronic components in a vehicle have to work together, or where a single vehicle component has to be customised to work in several different vehicles. You can end up with a hierarchy of builds and test regimes. But one way or another, we always have viable code that we can ship out for beta testing, or whatever the next stage of our process might be.

The technology of testing was probably the biggest practical improvement in software engineering during the 1990s and early 2000s. Before automated regression tests were widely used, IBM engineers used to reckon that 15% of bug fixes either introduced new bugs or reintroduced old ones [18]. The move to evolutionary development was associated with a number of other changes. For example, IBM had separated the roles of system analyst, programmer and tester; the analyst spoke to the customer and produced a design, which the programmer coded, and then the tester looked for bugs in the code. The incentives weren't quite right, as the programmer could throw lots of buggy code over the fence and hope that someone else would fix it. This was slow and led to bloated code. Microsoft abolished the distinction between analysts, programmers and testers; it had only developers, who spoke to the customer and were also responsible for fixing their own bugs. This held up the bad programmers who wrote lots of bugs, so that more of the code was produced by the more skilful and careful developers. According to Steve Maguire, this is what enabled Microsoft to win the battle to rule the world of 32-bit operating systems; their better development methodology let them take a $100bn business-software market from IBM [1211].

27.5.3 The secure development lifecycle

By the early 2000s, Microsoft had overtaken IBM as the leading tech company, but it was facing ever more criticism for security vulnerabilities in Windows and Office that led to more and more malware. Servers were moving to Linux and individual users were starting to buy Macs. Eventually in January 2002 Bill Gates sent all staff a 'trustworthy computing' memo ordering them to prioritise security over features, and stopping all development while engineers got security training. Their internal training materials became books and papers that helped drive change in the broader ecosystem. I already discussed their threat modelling in section 27.3.5; their first take on secure development appeared in 2002 in Michael Howard and David LeBlanc's '*Writing Secure Code*' [929], which sets out the early Microsoft approach to managing the security lifecycle, and which I discussed in the second edition of this book. More appeared

over time and their *security development lifecycle* (SDL) appeared in 2008, being adopted widely by Windows developers.

The widely used 2010 'simplified implementation' of SDL is essentially a waterfall process [1310]. It 'aims to reduce the number and severity of vulnerabilities in software' and 'introduces security and privacy throughout all phases of the development process'. The 'pre-SDL' component is security training; it's assumed that all the developers get a basic course, the contents of which will depend on whether they're building operating systems, web services or whatever. There are then five SDL components.

1. Requirements: this involves a risk assessment and the establishment of quality gates or 'bug bars' that will prevent code getting to the next stage if it contains certain types of flaw. The requirements themselves are reviewed regularly; at Microsoft, the reviews are never more than six months apart.
2. Design: this stage requires threat modelling and establishment of the attack surface, to feed into the detailed design of the product.
3. Implementation: here, developers have to use approved tools, avoid or deprecate unsafe functions, and perform static analysis on the code to check this has been done.
4. Verification: this step involves dynamic analysis, fuzz testing, and a review of the attack surface.
5. Release: this is predicated on an incident response plan and a final security review.

As well as providing some basic security training to all developers, there are some further organisational aspects. First, security needs a subject-matter expert (SME) from outside the dev team, and a security or privacy champion within the team itself to check that everything gets done.

Second, there is a maturity model. Starting in 1989, Watts Humphrey developed the *Capability Maturity Model* (CMM) at the Software Engineering Institute at Carnegie-Mellon University (CMU), based on the idea that competence is a function of teams rather than just individual developers. There's more to a band than just throwing together half-a-dozen competent musicians, and the same holds for software. Developers start off with different coding styles, different conventions for commenting and formatting code, different ways of managing APIs, and even different workflow rhythms. The CMU research showed that newly-formed teams tended to underestimate the amount of work in a project, and also had a high variance in the amount of time they took; the teams that worked best together were much better able to predict how long they'd take, in terms of the mean development time, but reduced the variance as well [1941]. This requires the self-discipline to sacrifice some efficiency in resource allocation in order to provide continuity for

individual engineers and to maintain the team's collective expertise. Microsoft adapted this and defines four levels of security maturity for developer teams.

27.5.4 Gated development

It's telling that the biggest firm pushing evolutionary development reverted to a waterfall approach for security. Many of the security engineering approaches of the time were tied up with waterfall assumptions, and automated testing on its own is less useful for the security engineer for a number of reasons. Security properties are both emergent and diverse, we security engineers are fewer in number, and there hasn't been as much investment in tools. Specific attack types often need specific remedies, and many security flaws cross a system's levels of abstraction, such as when specification errors interact with user interface features – the sort of problem for which it's difficult to devise automated tests. But although regression testing is not sufficient, it is necessary, as it finds functionality that's been affected by a change. It's particularly important when development sprints add lots of features that can interact with each other. For this reason, security patches to Windows are an example of *gated development*: at regular intervals, a pre-release version of the product is pushed through a whole series of additional tests and reviews and prepared for release. This is fairly common across systems with safety or security requirements. The preparation may involve testing with a wide variety of peripherals and applications in the case of Windows, or recertification in the case of software for a regulated product.

An issue many neglect is that security requirements evolve, and also have to be maintained and upgraded. They can be driven by changing environments, evolving threats, new dependencies on platforms old and new, and a bundle of other things. Some changes are implicit; for example, when you upgrade your static analysis tools you may find hundreds of 'new' bugs in your existing codebase, which you have to triage. Once more Microsoft was a pioneer here. When a vulnerability was found in Windows, it's not enough to just patch it; whoever wrote it might have written a dozen similar ones that are now scattered throughout the codebase, and once you publish a patch, the bad guys study it and understand it. So rather than just fixing a single bug, you update your toolchain so you find and eliminate all similar bugs across your products. In order to manage the costs, both for Microsoft and its customers, the company started bundling patches together into a monthly update, the now famous 'patch Tuesday', in 2003. From then until 2015, all customers – from enterprises to the users of home PCs and tablets – had their software updated on the second Tuesday every month. And such patching creates further dependencies. Modern quality tools can help you check that no code has a CVE open, so all your customers should have to patch too, if they

live by such tools. But many don't: as many as 70% of apps on both phones and desktops have vulnerabilities in the open-source libraries they use, and which could usually be fixed by a simple update [1698]. Since 2015, Windows home users receive continuous updates[4].

Much the same considerations apply to safety-critical systems, which are similar in many respects to secure systems. Safety, like security, is an emergent property of whole systems, and it doesn't compose. Safety used to depend, in most applications, on extensive pre-market testing. But it's hard for a connected device to have safety without security, and now that devices such as cars are connected to the Internet, they are acquiring patch cycles too. Yet ensuring that the latest version of a safety-critical system satisfies the safety case may require extensive and expensive testing. For example, a car may contain dozens of *electronic control units* (ECUs) from different component suppliers, and in addition to testing the individual ECUs you have to test how they work together. Firms in the car industry are mutually suspicious and won't share source code with each other, even under NDA, so testing can be complex. The main test rig may be a 'lab car' containing all the electronics from a particular model of car, plus extra test systems that let you simulate various maneuvers and even accidents. These cost real money, and you also need to keep real vehicles for road testing. The cost of maintaining fleets of lab cars and real test cars is one of the reasons car companies dragged their heels when the EU decided to require them to patch car software for ten years after the last vehicle left the showroom.

This is one respect in which Tesla has a significant advantage; as a tech company with software at the core of its business, Tesla can test and ship changes in weeks that take the legacy car firms years, as they leave most of the software development to the component suppliers [406]. Traditionally, automotive software contracts involved ten years' support; now you need to support a product for three years' development, seven years in the showroom and a further ten after that. I'll discuss the sustainability aspects of this in the next chapter. Meanwhile, Tesla is forcing the legacy industry to raise its game, with VW announcing they've spent $8bn to create a proper software division, just as their main electric car project runs late [1689].

27.5.5 Software as a Service

Since the early 2010s, more and more software has been hosted on central servers, accessed by thin clients and paid for on a subscription basis, rather

[4]This also breaks things: we were once about to demonstrate an experiment using a body motion-capture suit to a TV crew when the Windows laptop we used to drive it updated itself, and suddenly the capture software wouldn't work any more. There followed frantic phone calls to the software developer in the Netherlands and thankfully we got their update a few hours later, just in time for the show.

than being sold and distributed to users. The typical customer has many costs for running software beyond the license fee, including not just the cost of servers and operators but of deploying it, upgrading it regularly and managing it. If the vendor can take over these tasks from all their customers, many duplicated costs are removed, and they can manage things better because of their specialised knowledge. Software can be instrumented so that developers can monitor all aspects of its performance on a dashboard.

The key technical innovations behind *Software as a Service* (SaaS) are *continuous integration* and *continuous deployment*. Rather than having thousands of customers managing dozens of different versions of the software, the vendor can migrate a few customers to a new version to test it, and then migrate the rest. Upgrades become much more controllable, as they can be tested in a dry run against a snapshot of the real customer data, called a *staging environment*. Some companies now deploy several times a day, as their experience is that frequent small changes can be safer and have less risk of breaking something than a larger deployment, such as Microsoft's Patch Tuesday.

Deployment itself is tentative. A SaaS company will typically run its software on a number of service instances running on VMs behind a load balancer, which provides a point of indirection for managing running services. The separate instances also provide separate *failure domains* to improve robustness. To do a *rolling deployment* we configure a load balancer to send say 1% of the traffic to an instance with the new version, often called the 'canary' after the caged bird used by miners to detect carbon monoxide leaks. If the canary survives, deployment can be rolled forward progressively to new service instances. If the logging system detects any problems, developers are alerted. Some care needs to be taken that things don't go wrong if users flap between old and new versions of a design between transactions. If you make a change that breaks backwards compatibility, you typically build an intermediate stage that will work with both old and new systems (we were doing this in the world of bank mainframes back in the 1980s anyway).

The ability to manage risks through phased release and rolling deployment changes the economics of testing. The fact that you can fix bugs extremely quickly mean that you can achieve a target quality level with much less testing. You can also see everything the users do, so for the first time you can really understand how usability fails from the point of view of security, safety – and revenue. Of course it's revenue that usually drives the exploitation of this. Analytics collectors write all behavioural events to a log, which is fed into a data pipeline for metrics, analytics and queries. This in turn supports experiment frameworks that can do extensive A/B testing of possible features. Ad-driven services can optimise by engagement metrics such as active users, time per user session and use of specific features. Controlled experiments are used to improve security too; for example, Google has tuned its browser warnings by measuring how millions of users react to different warnings of

expired certificates. Such improvements are usually fairly small by themselves, so you really need controlled experiments to measure them; but when you do lots of them, they add up. The investment in building such frameworks into the phased deployment mechanisms gives an increasing return to scale; the more users you have, the faster you can achieve statistical significance. So large firms can optimise their products more quickly than their smaller competitors; SaaS, like a lot of other digital technology, not only cuts costs in the short term, but increases lock-in in the long term. Each time you access a service from a large SaaS firm, you may be an unwitting participant in tens or even hundreds of experiments. There are lots of fiddly details about running multiple concurrent experiments while also deploying system enhancements.

Things can get more complex still when you have services put together from multiple microservices. This brings us to the world of *infrastructure as code*, also known as cloud native development or DevOps, where everything is developed in containers, VMs etc., so all the infrastructure is based on code and can be replicated quickly. You can also use containers to simplify things, packaging as many security dependencies with the code as possible. New code can be deployed to a test infrastructure rapidly and tested realistically. You could if you wanted manage rolling deployment manually, but this is not scalable and prone to error. The solution is to write *deployment code*, as part of the application development process, that uses the cloud platform APIs to allow applications to deploy themselves and the associated infrastructure, and to hook into the monitoring mechanisms. In the last few years, some toolkits have become available that allow engineers to do this in a more declarative fashion.

The best guide to this I know is Google's 2013 book *'Site Reliability Engineering'*; SRE is their term for DevOps [237]. Google led the industry in the art of building large dependable systems out of large fleets of low-cost PCs, building the necessary engineering for load balancing, replication, sharding and redundancy. As they operated at a larger scale than anybody else through the 2000s and early 2010s, they had to automate more tasks and became good at it. The goals of SRE are availability, latency, performance, efficiency, change management, monitoring, emergency response, and capacity planning. The core strategy is to apply software engineering techniques to automate system administration tasks so as to balance rapid innovation with availability.

As we already noted, there's no point striving for 99.9999% availability if ISPs only let users get to your servers 99% or 99.9% of the time. If you set a realistic error budget, say 0.1% or 0.01% unavailability, you can use that to achieve a number of things. First, most outages are due to live system changes, so you monitor latency, traffic, errors and saturation well and roll back quickly whenever anything goes wrong. You use the rest of the error budget to support your experimental framework, and doing controlled outages to flush dependencies. (This was pioneered by Netflix whose 'chaos monkey' would occasionally take

down routers, servers, load balancers and other components, to check that the resilience mechanisms worked as intended; such 'fire drills' are now an industry standard and involve taking down whole data centres.)

In section 12.2.6.2, we mentioned *technical debt*. This concept, due to Ward Cunningham, encapsulates the observation that development shortcuts are like debt. Whenever we skimp on documentation, fix a problem with a quick-and-dirty kludge, don't test a fix thoroughly, fail to build in security controls, or fail to work through the consequences of errors, we're storing up problems that may have to be repaid with interest in the future [42]. Technical debt may make sense for a startup, or a system nearing the end of its life, but it's more often a product of poor management or poorly-aligned incentives. Over time, systems can fall so deeply into debt that they become too hard to maintain or to use; they have to be refactored or replaced. For a bank to have to replace its core banking systems is hugely expensive and disruptive. So managing technical debt is really important; this is one of the changes in system management thinking since the second edition of this book. One important aspect of the philosophy of DevOps is to run debt-free.

27.5.6 From DevOps to DevSecOps

As I write, in 2020, the cutting edge is applying agile ideas and methodology not just to development and operations, but to security too. In theory this can mean a strategy of 'everything as code'; in practice it means not just maintaining an existing security rating (and safety case if relevant) but responding to new threats, environmental changes, and surprising vulnerabilities. Bringing the two together involves real work, and sometimes things need to be reinvented. I mentioned for example in section 12.2.2 that DevOps undermines the separation between development and production on which banks have relied for years; where separation of duties is necessary, we have to reimagine it.

We see several different approaches in the companies with which we work. In what follows I will give two examples, which we might roughly call the Microsoft world and the Google world. There are of course many others.

27.5.6.1 The Azure ecosystem

Most of the world's largest commercial firms from banks and insurers through retail to shipping and mining have built their enterprise systems on Windows over the past 25 years and are now migrating them to Azure, often using systems integration and facilities management firms to do the actual work. The typical client has a mixture of on-premises and cloud systems with new developments mostly migrating from the former to the latter. Here policy is largely set by the Big Four auditors who, in addition to their standard set of internal

control features, follow Microsoft in requiring a secure development lifecycle. The several dozen tools used to do threat modelling, static analysis, dynamic analysis, fuzz testing, app and network monitoring, security orchestration and incident response impose a significant overhead with dozens of people copying data from one tool to another. The DevSecOps task here is to progressively integrate the tools by automating these administrative tasks.

To support this ecosystem, Microsoft has extended its SDL with further steps: defining metrics and compliance reporting; threat modelling; cryptography standards; managing the security risks of third-party components; penetration testing; and a standardised incident response. The firm now claims that 10% of its engineering investment is in cybersecurity. The capable system integration and facilities management firms have worked out ways of building these steps into their workflows; much of the actual work involves integrating the third-party security products that they or their customers have bought. Appropriate automation is vital for the security team to continue raising their game, extending their scope and increasing effectiveness; without it, they fall further and further behind, and burn out [1850].

The organising principles for DevSecOps in such a company will be to 'shift left', which can cover a number of things: the unifying theme is moving security, like software and infrastructure, into the codebase. One strategy is to cause things to 'fail fast' including engaging security experts early enough in the development process to avoid delays later: doing pre-commit static analysis of each developer's code to minimise failed builds; buying or building specialist tools to detect errors such as incorrect authentication, mistakes in using crypto functions, and injection opportunities; both automated and manual security testing of new versions; and automated testing of configuration and deployment including scanning of the staging network and checks on credentials, encryption keys and so on. And while, back in 2010, Microsoft considered operational security to be separate from software security, a modern Azure shop will close the loop by following up deployment with continuous monitoring, manual penetration tests and finally bug bounties for third parties who spot something wrong. We will discuss these in more detail later.

27.5.6.2 The Google ecosystem

A second view comes from engineers working on infrastructure, and the best reference I know is a 2020 book by six Google engineers, *'Building Secure and Reliable Systems'* [23]. Amazon's DevSecOps strategy is somewhat similar, but optimised for their product offerings; it is described by their CTO Werner Vogels at [1970]. However, the Google experience is described in much more detail. This section draws on their book, and on colleagues who have worked recently at the major service firms.

When building infrastructure systems on which hundreds of millions of people will rely, it is critical to automate support functions quickly, and to have really robust processes for threat identification, incident response, damage limitation and service recovery. So while a facilities-management firm might work at integrating support functions to save money and reduce errors, the emphasis at major service firms is reliability. I already mentioned the Google approach to site reliability engineering: set a realistic target, of say 99.9% availability, and then use the residual error budget of 0.1% downtime by apportioning it between failure recovery, upgrades and experiments.

This in turn drives further principles such as design for recoverability, design for understandability, and a desire to stop humans touching production systems wherever possible. It's not enough to have automation for the incremental deployment of new binaries; you also want to stop sysadmins having to type complicated command lines into routers to configure networks; this is where most of the network outages come from, as we noted in section 21.2.1. You manage such risks by building suitable tool proxies. This can involve quite a lot of work to align the update of binary and config files and work out how to allocate support and recovery effort between SRE and security engineering teams. Further complexity arises with secure testing. How do you build test infrastructures to exercise least privilege? How do you test systems that contain large amounts of personal information? How do you test the break-glass mechanisms that give SRE teams emergency human access to live systems? Most of these are questions we already had to deal with in the mainframe world of the 1980s, but they arose only occasionally and were dealt with by human ingenuity and by trusting some key staff. Scaling everything up from thousands of users to billions means that a lot more has to be automated.

There are still tensions. In site reliability engineering, alarms should be as simple, predictable and reliable as possible; but in security, some randomisation is often a good idea.

At the application level, systems are increasingly compartmentalised into microservice components with defensible security boundaries and tamper-resistant security contexts, so that if Alice compromises a shopping system's catalogue, she still can't spend money as Bob as the payment service is separate. Each component will typically be implemented as a number of parallel copies or shards, giving still smaller failure domains. Such domains enable you to limit the blast radius of any compromise; ideally, you want to be able to deal with an intrusion without taking your whole system offline. Compartmentalised systems can be engineered for resilience too, but this is not straightforward. When a failure domain fails, when do you just spin up a new one, and when do you do something different? What are the dependencies? Which components should fail open, and which should fail secure? What sort of degraded performance is acceptable under congestion, or under attack?

What's the role of load shedding and throttling? And what sort of pain can you rationally inflict on users, and on business models? Do you ditch some of the ads, require extra CAPTCHAs for logons, or both? And how do you test and validate all these resilience mechanisms?

Large firms invest a lot of engineering time in building application frameworks for such services. There are also standard frameworks for web pages, which should not only prevent SQL injection and cross-site scripting attacks in the first place, but also provide support for dozens of different languages. Having a single front end to terminate all http(s) and TLS traffic means that if you have to update your certificate management mechanisms or ciphersuites you only need to do it once, not in all your different services. A single front end can also provide a single location for load balancing and DDoS protection, as well as for many other functions such as supporting dozens of different languages.

Using type encapsulation to enforce properties of URLs, SQL and so on can reduce the amount of code you need to verify. If you have secure-by-construction APIs that are also understandable, that's best. Google has a crypto API called Tink that forces more correct use. It requires use of a key management service, whether in the Google cloud, AWS or the Android keystore. This fits into an overall framework for managing crypto termination, code provenance, integrity verification and workload isolation, called BeyondProd [1000].

27.5.6.3 Creating a learning system

Whether you follow the Microsoft approach, the Google approach or your own, to tune such a process you need metrics, and suitable candidates include the numbers of security tickets opened to dev teams, the number of security-failed builds, and the time it takes for a new application to achieve compliance under the relevant regulation (whether SOX, GDPR or HIPAA). As Dev, Sec and Ops converge, the metrics and management processes converge with the network defence mechanisms discussed in section 21.4, from network monitoring to security incident and event management. But all this needs to be managed intelligently. A well-run firm can make the security process more visible to all the dev/ops staff via the sprints that you do to work up a privacy impact assessment, improve access controls, extend logging or whatever. A badly-run firm will manage to the metrics, which will create tensions: their security staff can end up with conflicting goals of keeping the bad guys out, and also of 'feeding the beast' by hitting all the metrics used to justify the team's own existence [1850]. It's important to understand where conflicts naturally arise as a function of the organisation's management structure, and somehow keep them constructive.

One of the big drivers in either case, though, will be the vulnerability lifecycle. The processes whereby bugs become exploits and then attacks, and

these attacks are noticed leading to vulnerability reports, interim defences using devices such as firewalls, then definitive patches that are rolled out not just to direct users but along complex supply chains, is ever more central to security management.

27.5.7 The vulnerability cycle

Back in the 1970s and 1980s, people sometimes described the evolutionary procedure of finding security bugs in systems and then fixing them dismissively as *penetrate-and-patch*. It was hoped that some combination of an architecture that limited the attack surface and the application of formal methods would enable us to escape. As we've seen, that didn't really work, except in a few edge cases such as cryptographic equipment. By the early 2000s, we had come to the conclusion that we just had to manage the patch cycle better, and the modern approach of security breach disclosure laws, CERTs and responsible disclosure bedded down during this period. I discussed the security economics of this in section 8.6.2; let's now look at the technical details.

The vulnerability cycle consists of the process whereby someone, the *researcher*, discovers a vulnerability in a system that is maintained by a *vendor*. The researcher may be a *customer*, an academic, a contractor for a national intelligence agency or even a criminal. They may sell it in a market. The idea of vulnerability markets was first suggested by Jean Camp and Catherine Wolfram in 2000 [373]; firms were set up to buy vulnerabilities, and over time several markets emerged. Most of the big software and service firms now offer bug bounties, which can range from thousands to hundreds of thousands of dollars; at the other extreme are operators who buy up exploits for sale to *exploiters* such as cyber-arms manufacturers (who sell to military and intelligence agencies) and forensic firms (who sell to law enforcement). Such operators now offer millions of dollars for persistent remote exploits of Android and iOS.

The researcher may also disclose the bug to the vendor directly – nowadays many vendors have a *bug bounty program* that pays rewards for disclosed vulnerabilities that attempt to match market prices, at least in order of magnitude. As market prices for zero-day exploits against popular platforms have headed into six and even seven figures, so have bug bounties. Apple, for example, offers $1M for anyone who can hack the iOS kernel without requiring any clicks by the user. In 2019, it emerged that at least six hackers have now earned over $1M through the bug bounty platform HackerOne alone [2033]. A downside of large bug bounties is that while bugs used to occur naturally, we now see them being introduced deliberately, for example by contributors to open-source projects whose code ends up in significant platforms. Such *supply-chain attacks* used to be the preserve of nation states; now they're opening up [892].

If an exploit is used in the wild before the vendor issues a patch, it is called a *zero day*, and is typically used for targeted attacks. If it's used enough, then eventually someone will notice; the attack gets reported, and then the vendor issues a patch, which may be reverse engineered so that many other actors now have exploit code. Customers who fail to patch their systems are now vulnerable to multiple exploits that can be deployed at scale by crime gangs.

Getting the patching cycle right is a problem in the economics of information security as much as anything else, because the interests of the various stakeholders can diverge quite radically.

1. The vendor would prefer that bugs weren't found at all, to spare the expense of patching. They'll patch if they have to but want to minimise the cost, which may include a lot of testing if their code appears in lots of product versions. Indeed, if their code is used in customer devices that now need patching (like cars) they may have to pay an indemnity to cover their customer's costs; so in such industries there's an even more acute incentive for foot-dragging and denial.
2. The average customer might prefer that bugs weren't found, to avoid the hassle of patching. Lazy customers may fail to patch, and get infected as a result. (If all the infected machines do is send a bit of spam, their owners may not notice or care.)
3. The typical security researcher wants some reward for their discoveries, whether fame, cash or getting a fix for a system they rely on.
4. The intelligence agencies want to learn of vulnerabilities quickly, so they can be used in zero-day exploits before a patch is shipped.
5. The security software firms benefit from unpatched vulnerabilities as their firewalls and AV software can look for their indicators of compromise to block attacks that exploit them.
6. Large companies don't like patches, and neither do government departments, as the process of testing a new patch against the enterprise's critical systems and rolling it out is expensive. The better ones have built automation to deal with regular events like Microsoft's Patch Tuesday, but updating or risk-assessing the zillions of IoT devices in their offices and factories will be a headache for years to come. Most firms just don't have a good enough asset inventory system to cope.

During the 1990s, the debate was driven by people who were frustrated at software vendors for leaving products unpatched for months or even years. The bugtraq mailing list was set up to provide a way for people to disclose bugs anonymously; but this meant that a product might be completely vulnerable for a month or two until a patch was written, tested and shipped, and until

customer firms had tested it and installed it. This led to a debate on 'responsible disclosure' with various proposals about how long a breathing space the researcher should give the vendor [1575].

As we discussed in section 8.6.2, the consensus that emerged was responsible disclosure: that researchers should disclose vulnerabilities to a computer emergency response team (CERT)[5] and the global network of CERTs would inform the vendor, with a delay for a patch to be issued before the vulnerability was published. The threat of eventual disclosure got vendors off their butts; the delay gave them enough time to test a fix properly before releasing it; researchers got credit to put on their CVs; customers got bug fixes at the same time as bug reports; and the big companies organised regular updates for which their corporate customers can plan. Oh, and the agencies had a hot line into their local CERT, so they learned of naturally occurring exploits in advance and could exploit them. This was part of the deal described in section 26.2.7.3 that ended Crypto War 1 back in 2000.

27.5.7.1 The CVE system

An industrial aspect is the *Common Vulnerabilities and Exposures* (CVE) system, launched in 1999, which assigns numbers to reported vulnerabilities in publicly released software packages. This is maintained by Mitre, but it delegates the assignment of CVEs to large vendors. CVE IDs are commonly included in security advisories, enabling you to search for details of the reporting date, affected products, available remedies and other relevant information. There is a Common Vulnerability Scoring System (CVSS) that provides a numerical representation of the severity of a vulnerability. The method for calculating this has become steadily more complex over time and now depends on whether the attack requires local access, its complexity, the effort required, its effects, the availability of exploit code and of patches, the number of targets and the potential for damage.

NIST's *National Vulnerability Database* (NVD), described as a "comprehensive cybersecurity vulnerability database that integrates all publicly available US Government vulnerability resources and provides references to industry resources", is based on the CVE List. These resources are critical for automating the tracking of vulnerabilities and updates. There are now so many thousands of vulnerabilities reported, and so many hundreds of patches shipped, that automation is essential.

As the system was bedding down, it became a subject of study by security economists. Traditionalists argued that since bugs are many and uncorrelated, and since most exploits use vulnerabilities reverse-engineered from existing patches, there should be minimal disclosure. Pragmatists argued that,

[5]The EU is renaming these CSIRTs – computer security incident response teams.

from both theoretical and empirical perspectives, the threat of disclosure was needed to get vendors to patch. I discussed this argument in section 8.6.2. Since then we have seen the introduction of automatic upgrades for mass-market users, the establishment of firms that make markets in vulnerabilities, and empirical research on the extent to which bugs are correlated. Modulo some tuning, the current computer industry way of doing things has been stable for over a decade.

27.5.7.2 Coordinated disclosure

Yet some industries are lagging well behind. In section 4.3.1 I described how Volkswagen sued academics at Birmingham and Nijmegen universities after they discovered, and responsibly disclosed, vulnerabilities in Volkswagen's remote key entry system that were already being exploited in car-theft tools that were available online. This was Volkswagen's mistake; it drew attention to the vulnerability, and they also lost in court. Companies like Microsoft and Google have had twenty years to learn that running bug bounty programs and monthly patching works better than threatening to sue people, but a lot of firms in legacy industries still haven't worked this out even though their products contain more and more software.

One of the problems in the Volkswagen case was that the researchers initially disclosed the vulnerability to the supplier of its key entry system, which in turn told Volkswagen only at the last minute. As a result of supply chain problems like this, responsible disclosure has given way to *coordinated disclosure*. Few firms build all their own tools any more, and even a child's toy may have multiple software dependencies. If it does speech and gesture recognition, it probably contains an Arm chip running some flavour of Linux or FreeBSD, communicates with a cloud service running another flavour of Linux, and can be controlled by an app that may run on Android or iOS. The safety of the toy will depend on secure communications; for example, it was discovered in February 2019 that the communications between Enox's 'Safe-KID-One' toy watch and its back-end server were unencrypted, so that hackers could in theory track and call kids. The response was an immediate EU-wide safety recall [653]. Getting this sort of thing wrong can be sudden death for your product, and your company.

Now what happens when someone discovers an exploitable bug in a platform used in dozens of embedded products? This can be traumatic, as with the Shellshock bug in Linux and the Heartbleed bug in OpenSSL (which also affected Linux). If Linux gets an emergency patch, coordinating the disclosure is a nightmare: the Linux maintainers may be able to work in private with the main Linux distributions, and with derivatives like Android whose developers keep in close contact with them. But there are the thousands

of products that incorporate Linux, from alarm clocks to TVs and from kids' toys to land mines. You may suddenly find that the CCTV cameras in your building security system have all become hackable, and the vendor can't fix them quickly or at all. Coordinating disclosure on platforms is one of the seriously hard problems. There is no silver bullet, but there are still many things you can do, ranging from documenting your upstream and downstream dependencies, through aggressive testing of software you depend on so you get to exercise and understand the bug reporting mechanisms, to becoming part of its developer community.

Dealing with such shocks is just one aspect of a process that in the late 2010s became a speciality of its own, namely security incident and event management.

27.5.7.3 Security incident and event management

You need an incident response plan for what you'll do when you learn of a vulnerability or an attack. In the old days, vendors could take months to respond with a new version of the product, and would often do nothing at all but issue a warning (or even a denial). Nowadays, breach-notification laws in both the USA and Europe oblige firms to disclose attacks where individuals' privacy could have been compromised, and people expect that problems will be fixed quickly. Your plan needs four components: monitoring, repair, distribution and reassurance.

First, make sure you learn of vulnerabilities as soon as you can – and preferably no later than the bad guys (or the press) do. This means building a threat intelligence team. In some applications you can just acquire threat intelligence data from specialist firms, while if you're an IoT vendor, it may be prudent to operate your own honeypots so you get immediate warning of people attacking your products. Listening to customers is important: you need an efficient way for them to report bugs. It may be an idea to provide some incentive, such as points towards their next upgrade, lottery tickets or even cash. You absolutely need to engage with the larger technical ecosystem of bug bounties, vulnerability markets, CERTs and CVEs described in section 27.5.7.

Second, you need to be able to repair the problem. Twenty years ago, that meant having one member of each product team 'on call' with a pager in case something needed fixing at three in the morning. Nowadays it means preparing an orchestrated response to anything from a vulnerability report to a major breach. This will extend from the intrusion-detection and network monitoring functions we discussed in section 21.4.2.3 and the threat intelligence team through to identifying the dev teams responsible and notifying both your suppliers upstream and your customers downstream. Responder teams may also need alternative means of communication. Did you ever stop to think whether you need satellite phones?

Third, you need to be able to deploy the patch rapidly: if all the software runs on your own servers, then it may be easy, but if it involves patching code in millions of consumer devices, then advance planning is needed. It may seem easy to get your customers to visit your website once a day and check for upgrades, but if their own systems depend on your devices and they need to test any dependencies, there's a tension [196]: pioneers who apply patches quickly can discover problems that break their systems, while people who take time to test will be more vulnerable to attack. The longer the supply chains get, the harder the conflicts of interest are to manage. Operations matter hugely: an emergency patch process that isn't tested may do more harm than good, and experience teaches that in an emergency you just run your normal patch process as fast as possible [23].

Finally, you need to educate your CEO and main board directors in advance about the need to deal quickly and honestly with a security breach in order to keep confidence and limit damage, by giving them compelling examples of firms that did well and others that did badly. You need to have a mechanism to get through to your CEO and brief them immediately so they can show the thing's under control and reassure your key customers. So you need to know the mobile and home phone numbers of everyone who might be needed urgently. And you need a plan to deal with the press. The last thing you need is for dozens of journalists to phone up and be stonewalled by your PR person or even your switchboard operator as you struggle madly to fix the bug. Have a set of press releases ready for incidents of varying severity, so that your CEO only has to pick the right one and fill in the details. This can then ship as soon as the first (or perhaps the second) journalist calls.

Remind your CEO that both the USA and Europe have security-breach disclosure laws, so if your systems are hacked and millions of customer card numbers compromised, you have to notify all current and former customers, which costs real money. As we discussed in section 26.6.2, you can expect to lose customers and take a hit to your stock price if you have a large breach or more than one small one; and if it's really bad your CEO can get fired. Information security is a CEO issue.

27.5.8 Organizational mismanagement of risk

Organizational issues are not just a contributory factor in system failure, as with the loss of organizational memory and the lack of mechanisms for monitoring changing environments. They can often be a primary cause. There's a large literature on how people behave in organisations, which I touched on in section 8.6.8, and I've given a number of further examples in various chapters. However, the importance of organisational factors increases as projects get bigger. Bezos' law says you can't run a dev project with more people than can

be fed from two pizzas. A team of eight people is just about manageable, but you can't go six times as fast by having six such teams in parallel. If a project involves multiple teams the members can't talk to each other at random, or you get chaos; and they can't route all their communications through the lowest common manager as there isn't the bandwidth. As you scale up, the coordination will start to involve a proliferation of middle managers, staff departments and committees. The communications complexity of a clean military chain of command, for N people with no lateral interaction, is $\log N$; where everybody has to consult everybody else, it's N^2; and where any subset can form a committee to think about the problem, it can head towards 2^N. Business school people have written extensively about this, and their methodology is generally based on case studies.

Many large development projects have crashed and burned. The problems appear to be much the same whether the disaster is a matter of safety, of security or of the software simply never working at all; so security people can learn a lot from studying project failures documented in the general engineering literature.

A classic study of large software project disasters was written by Bill Curtis, Herb Krasner, and Neil Iscoe [504]. They found that failure to understand the requirements was mostly to blame: a thin spread of application domain knowledge typically led to fluctuating and conflicting requirements, which in turn caused a breakdown in communication. The example I give in my undergraduate lectures is the meltdown of a new dispatch system for the London Ambulance Service where a combination of an overly ambitious project, an inadequate specification and no real testing led to the city being without ambulance cover for a day. There are all too many such examples; I use the London Ambulance Service case because the subsequent inquiry documented the causes rather well [1809]. I also happened to be in London that day, so I remember it. If you haven't ever read the inquiry report, I recommend you do so. (In fact, I strongly recommend that you read lots of case studies of project failure.)

The millennium bug gives another useful data point. If one accepts that many large commercial and government systems needed extensive repair work to change two-digit dates into four-digit ones in preparation for the year 2000, and the conventional experience that a significant proportion of large development projects are late or never delivered at all, many people naturally assumed that a significant number of systems would fail at the end of 1999, and predicted widespread chaos. But this didn't happen. Certainly, the risks to the systems used by small and medium-sized firms were overstated; we did a thorough check of all our systems at the university, and found nothing much that couldn't be fixed fairly easily [70]. Nevertheless, the systems of some large firms whose operations are critical to the economy, such as banks and utilities, did need substantial repairs. Yet there were no reports of high-consequence

failures. This appears to support Curtis, Krasner, and Iscoe's thesis. The requirement for Y2K bug fixes was known completely: "I want this system to keep on working, just as it is now, through into 2000 and beyond".

This is one of the reasons I chose the quote from Rick Smith to head this chapter: "My own experience is that developers with a clean, expressive set of specific security requirements can build a very tight machine. They don't have to be security gurus, but they have to understand what they're trying to build and how it should work."

Organisations have difficulty dealing with uncertainty, as it gets in the way of setting objectives and planning to meet them. So capable teams tackle the hard problem first, to reduce uncertainty; that was DARPA's mission, and the core of the spiral model. There's a significant business-school literature on how to manage uncertainty in projects [1180]. But it's easy to get this wrong, even in a fairly well-defined project. Faced with a hard problem, it is common for people to furiously attack a related but easier one; we've seen a number of examples, such as in section 26.2.7.4.

Risk management can be even worse in security where the problem is open-ended. We really have no idea where the next shitstorm will come from. In the late 1990s, we thought we'd got secure smartcards; then along came differential power analysis. In the mid-2010s we thought we had secure enough CPUs for competitor firms to run their workloads on the same machines in Amazon data centres; then along came Spectre. We also used to think that Apple products couldn't get malware and that face recognition would never be good enough to be a real privacy threat. Even though Moore's law is slowing down, there will be more surprises.

Middle managers prefer approaches that they can implement by box-ticking their way down a checklist, but to deal with uncertainties and open-ended risks, you need a process of open learning, with people paying attention to the alerts, or the frauds, or the safety incidents, or the customer complaints – whatever you can learn from. But checklists demand less management attention and effort, and the quality bureaucracy loves them. I noted in section 9.6.6 that certified processes had a strong tendency to displace critical thought; instead of constantly reviewing a system's protection requirements, designers just reach for their checklists. The result is often perverse. By not tackling the hard problem first, you hide the uncertainty and it's worse later[6]. Also, people rapidly learn how to game checklists. There is the eternal tension between us security experts telling firms to pay smart people to anticipate what might go wrong, and boards telling managers to deliver product faster using fewer and cheaper engineers.

[6]I will discuss ISO 27001 in the next chapter. The executive summary for now is that almost every firm hit by a big data breach had ISO 27001 certification, but it failed because their auditors said something was OK that wasn't.

When the threat model is politically sensitive, things get more complicated. The classic question is whether attacks come from insiders or outsiders. Insiders are often the biggest security risk, whether because some of them are malicious or because most of them are careless. But you can't just train all your staff to be unhelpful to each other and to customers, unless perhaps you are a government department or other monopoly. You have to find the sweet spot for control, and that often means working out how to embed it in the culture. For example, bank managers know that dual-control safe locks reduce the risk of their families being taken hostage, and requiring two signatures on large transactions means extra shoulders to take the burden when something goes wrong.

Getting the risk ecosystem right in an organisation can take both subtlety and persistence. The cultural embedding of controls and other protective measures is hard work; if you come into contact with multiple firms then it's interesting to observe how they manage their rules around everything from code audits (which the tech majors insist on) to tailgating (which semiconductor firms are at pains to prevent) and whether people are expected to keep one hand on a banister as they walk up and down the stairs (a favourite of energy companies). Where do these risk cultures come from, how are they promoted, and why do they cluster by sector? Their transactional internal control structures may be heavily influenced by their auditors, as we discussed in section 12.2.6.3, but the broader security culture varies a lot – and matters.

A further factor is that good CISOs are almost as rare as hens' teeth. There are some stars at the top tech and fintech firms, but being a CISO can be a thankless job. Good engineers often don't want it, or don't have the people skills to cope, while ambitious managers tend to avoid the job. In many organisations, promotions are a matter of seniority and contacts; so if you want to be the CEO you'll have to spend 20 years climbing up the hierarchy without offending too many people on the way. Being CISO will mean saying no to people all the time, and a generalist with no tech background can't hack it anyway. The job also brings a lot of stress, and the risk of burnout; a CISO's average tenure is about two years [432]. In any case, embedding an appropriate culture around risk and security is for the CEO and the board. If they don't think it's important, the CISO has no chance. But breaches have now led to enough CEOs being fired, or losing millions on their stock, that other members of that tribe are starting to pay attention.

One way the risk ecosystem can be skewed is that if a company manages to arrange things so that some of the risks of the systems it operates get dumped on third parties. This creates a moral hazard by removing the incentives to take care. We discussed this in section 12.5.2 in the context of banks trying to shift fraud liability in payment systems to cardholders, merchants or both. Staff can get lazy or even crooked if they know that customer complaints will be brushed off. Another example is Henry Ford, who took the view that if you were injured

by one of his cars, you should sue the driver, not him; it took decades for courts and lawmakers to nail down product liability.

Companies may also swing from being risk takers to being too risk averse, and back again. The personality of key executives does matter. My own university has been gung-ho when we hired an engineer to be Vice-Chancellor, timorous when we hired a lawyer, and in the middle when we hired a medic.

Another source of problems is when system design decisions are taken by people who are unlikely to be held accountable for them. This can happen for many reasons. IT staff turnover could be high, with much reliance placed on contract staff; fear of redundancy can turn loyal staff into surreptitious job-seekers. This can be a particular problem in big public-sector IT projects: none of the ministers or civil servants involved expect to be around when the thing is delivered seven years from now. So when working on a big system project, don't forget to look round and ask yourself who'll take the blame later when things go wrong.

Yet another is that when hiring security or safety consultants to help with product design, firms have an incentive to go for a firm that is 'good enough' but will not be too demanding; a gentle review from a Big Four firm will be much more useful than a detailed review from an expert who might recommend much more expensive design changes. Indeed, if a firm was determined to get a completely secure product, then they should hire multiple experts. We described in section 14.2.3 how this helped with the design of prepayment electricity meters, and a later experiment with students confirmed that the more people you got to think about a proposed system design, the more potential hazards and vulnerabilities they could spot [69]. Of course, this rarely happens.

27.6 Managing the team

To develop secure and reliable code, you need to build a team with the right culture, the right mix of skills, and the right incentives.

Many modern systems are already so complex that few developers can cope with all aspects of them. So how do you build strong development teams with complementary skills? This has been a subject of vigorous debate for over fifty years now, with different writers reflecting their personal style or company culture. It has long been entangled with cultural issues such as diversity, although these have only got serious attention since the mid-2010s.

27.6.1 Elite engineers

Going back to the 1960s, Fred Brooks's famous book 'The Mythical Man-Month' describes the lessons learned from developing the world's first large

software product, the operating system for the IBM S/360 mainframe [329]. He describes the 'chief programmer team', a concept evolved by his colleague Harlan Mills, in which a chief programmer – a development lead, in today's language – is supported by a toolsmith, a tester and a language lawyer. The thinking was that some programmers are much more productive than others, so rather than promoting them to management and 'losing' them you create posts for them with the salary and esteem of senior managers. The same approach was found in other tech companies in the 1960s through the 1980s, and even in bank IT departments where I worked in the late 1980s.

The view taken by more modern companies such as Microsoft, Google, Facebook and Netflix is that you only want to hire the ultra-productive engineers in the first place – especially if you get a million CVs a year but plan to hire only 20,000 new engineers. One approach is to hire people as contractors for a few months to see how they do. But that's harder with fresh graduates, as even bright students from elite schools can take a few months to become productive in a commercial team. Productivity is also a matter of culture; engineers who thrive at one company may do much less well at another. A related issue is that if you have each candidate interviewed by a number of your engineers, that's not just a drain on engineer time, but can also perpetuate a culture that's not very welcoming to women engineers. Elite universities are in a similar situation to the tech majors, with dozens of applicants for each place; over the years we've developed mechanisms to monitor diversity in hiring and admissions.

The two approaches are not in conflict. Modern tech firms employ multiple tech superstars from famous designers to Turing-award winning computer scientists. The view at one such firm is that you cannot expect to write good software if you don't have a career structure for programmers. People who want to spend their lives writing software, and are good at it, have to get respect, however your organisation signals that – whether it's salary, bonuses, stock or fripperies like access to the executive dining room. Universities get this; we professors run the place. Tech companies get it too, and one or two banks have started to. But governments are generally appalling. In the UK civil service, the motto is that "scientists should be on tap but not on top." And more than one car company I know of has real problems hiring and retaining decent software engineers. In one of them, software engineers are expected to become managers after five years or remain on a junior pay grade, while in another all engineers are expected to wear business suits to work (and still get lousy money). I'll return to this in section 27.6.6.

27.6.2 Diversity

At the beginning of computing, there were plenty of women programmers – they were the majority until the late 1960s, and included pioneers such

as Grace Hopper and Dame Stephanie Shirley (who ran her company for years as 'Steve Shirley'). When I started in the early 1970s there was still a much better gender balance than today. There were minorities too; the orbital calculations for the Mercury, Gemini and Apollo missions were led by an African-American woman, Katharine Johnson. But things have become male-dominated in the USA and the UK. Since I became an academic in the 1990s, about a sixth of local computer science students have been women, despite significant efforts to recruit more women students. But in the formerly communist countries of Eastern Europe, the ratio is about a third. (We've improved our gender balance by admitting lots of students from southern and eastern Europe.) In India there's close to gender balance. So this is a cultural issue, and there's a lot of debate on how it came about. Is it a lack of role models, or is it the fault of careers advisers in schools, or are many IT shops just an unpleasant working environment for women? That has certainly been an issue: the Gamergate scandal, which I discussed in section 2.5.1, exposed deep misogyny in some gaming communities, while the `#MeToo` movement has highlighted many cases of sexism in Silicon Valley.

Even within computer science we see a lot of subcultural variation. The last time I went to a hardware conference – an Arm developer event – I saw about 500 men but only three women (all of them Indian). In the security field, we were overwhelmingly male in the 1990s when the emphasis was cryptology and operating system internals, but are much more balanced now we have embraced the importance of design, usability and psychology. Role models and history do matter. Research groups with a woman faculty member get more applications from able women[7].

More diverse teams are more effective, and the real change doesn't come with the first woman you hire, but when you have enough to change the team culture. That might mean three or more. It also means getting more enlightened managers. More subtly, if you want to attract more women and retain them, it can be an idea to manage the people rather than the work. You have to protect your staff and give them space to do what they're good at. Clearly it's a bad idea to hire misogynistic bullies, though it can be hard to spot them in advance. Bullies are often creeps too; as well as bossing the people under them they suck up to the people above them. Very often such people don't understand what's going on technically so they have no idea who's productive and have to judge people by timekeeping or by how much they ingratiate themselves. It's essential to identify such bullies and get rid of them, hard though firing can be. If this management style spreads through an organisation, smart people will go somewhere else.

[7]We have gender balance in our natural language processing group, started in the 1960s by the late Karen Spärck Jones.

27.6.3 Nurturing skills and attitudes

Modern development has a tension between the desire to keep teams together, so that they get more efficient and predictable, and moving people around to develop their skills, stop them going stale, and ensure that there's more than one person able to maintain everything that matters.

You will also need a diversity of skills. If you're writing an app, for example, you may want a couple of people to write the Android code, a couple for the Apple code and a couple for the server. Depending on the task, there may be a user advocate who leads usability testing, advocates for safety and security, an architect to keep the overall design clean and efficient, a language lawyer who worries about APIs, a test engineer who runs the regression testing machinery and a toolsmith who maintains the static and dynamic analysis tools. If you're doing continuous integration you'll have an engineer specialising in A/B testing while if you have a gated approach the test emphasis might be on compatibility with third-party products or with security certification. You'll need to give some thought to how many of these skills you try to get in each dev, and how many are subject-matter experts who work across teams or come in as consultants. And as you can't run a project with more people than you can feed from two pizzas, you want some of your people to have two or more of these skills. Good tech firms rotate engineers slowly through the company to acquire a range of skills that maximises their value to the firm (even though it also maximises their value to others, and makes it easier for them to leave) [1211].

But skills are not enough: you need to get people to work together. Here, too, working practices have evolved over the years. By about 2010, agile developers had adopted the 'scrum' where the whole dev team has a stand-up meeting for five minutes each day, at which the only people allowed to speak are the developers. They describe what they've done, what they're about to do and what the problems are. Some firms have moved teams to collaboration tools such as Jira. In our team we combined daily lunches together with a formal progress meeting once a week. (Since the coronavirus lockdown the formal meeting has become more important and we've worked to complement it with other online activities.)

It's bad practice if people who find bugs (even bugs that they coded themselves) just fix them quietly; as bugs are correlated, there are likely to be more. Bug tracking matters, and a ticketing system that enables good statistics to be kept is an important tool in improving quality. As an example of good practice, in air traffic control it's expected that controllers making an error should not only fix it but declare it at once by open outcry: "I have Speedbird 123 at flight level eight zero in the terminal control area by mistake, am instructing to descend to six zero." That way any other controller with potentially conflicting traffic can notice, shout out, and coordinate. Software is less dramatic,

but is no different: you need to get your devs comfortable with sharing their experiences, including their errors.

Another factor in team building is the adoption of a standard style. One signal of a poorly-managed team is that the codebase is in a chaotic mixture of styles, with everybody doing their own thing. When a programmer checks out some code to work on it, they may spend half an hour formatting it and tweaking it into their own style. Apart from the wasted time, reformatted code can trip up your analysis tools. You also want comments in the code, as people typically spend more time reading code than writing it. You want to know what a programmer who wrote a vulnerability thought they were doing: was it a design error, or a coding blunder? But teams can easily fight about the 'right' quantity and style of comments. So when you start a project, sit everyone down and let them spend an afternoon hammering out what your house style will be. Provided it's enough for reading the code later and understanding bugs, it doesn't matter hugely what the style is: but it does matter that there is a consistent style that people accept and that is fit for purpose. Creating this style is a better team-building activity than spending the afternoon paintballing, or whatever the latest corporate team-building fad happens to be.

27.6.4 Emergent properties

One debate is whether you make everyone responsible for securing their own code, or have a security guru on whom everyone relies. The same question applies to safety in fields such as avionics. The answer, as the leading firms have discovered, is 'both'. We already noted that Microsoft found it more effective to have developers responsible for evolving their own designs and fixing their own bugs, rather than splitting these functions between analysts, programmers and testers, as IBM did in the last century. Both Microsoft and Google now put rookie engineers through a security 'boot camp', so that everyone knows the basics, and also have subject matter experts at a number of levels. These range from working security consultants with a masters degree or the equivalent internal qualification, to people with PhDs in the intricate details of cryptography or virtualisation.

The trick lies in managing the amount of specialisation in the team, and the way in which the specialists (such as the security architect and the testing guru) interact with the other developers.

27.6.5 Evolving your workflow

You also need to think hard about the tools you'll use. Professional development teams avoid a large number of the problems described in this book by using appropriate tools. You avoid buffer overflows by using a modern

language such as Rust, or if you must use C or C++ then have strict coding conventions and enforce them using static-analysis tools such as SonarQube and Coverity. You avoid crypto problems, such as timing attacks and weak random number generators, by using well-maintained libraries. But you need to understand the limitations of your tools. In the case of Coverity, for example, its authors explain that while it's great if you use it from the start of a project, adopting it in midstream imposes real costs, as you suddenly have 20,000 more bug reports to triage, and your ship date slips by a few months [236]. Improvements in static analysis tools, say in response to a new kind of attack, can also throw up a lot of alarms in an existing codebase. In the case of crypto libraries, we discussed in Chapter 5 how they tend to offer weak modes of operation such as ECB as defaults, so you need to ensure your team uses GCM instead. (Crypto is one of the areas where you need to talk to a subject matter expert.)

You'll be constantly adding new tools, whether to avoid cross-site scripting vulnerabilities and SQL injection as you update your website, or to make sure you don't leave your client data world-readable in an S3 bucket. If you don't follow the security news you may not be aware of the latest exploits and attacks, so you may not realise when you have to either grow your own expertise or buy it in. But you can't just buy everything in; the security industry has lots of unscrupulous operators who exploit ignorant customers. You need to understand what you need to buy, and why, and then you will need to integrate it with your existing tools, or your security ops people will spend ever more of their time copying IP addresses from one tool to another. Doing some of your own automation helps empower your staff as well as saving time.

Your tools and libraries have to support your architecture. One critical thing here is that you need to be able to evolve APIs safely. A system's architecture is defined more than anything else by its interfaces, and it decays by a thousand small cuts: by a programmer needing a file handling routine that uses two more parameters than the existing one, and who therefore writes a new routine – which may be dangerous in itself, or may just add to complexity and thus contribute indirectly to an eventual failure. In an ideal world, you'd rely on your programming language to prevent API problems using type safety mechanisms.

But the cross-system fan-out of dependencies is a real hazard to safe APIs. We saw in section 20.5 how the APIs of cryptographic hardware security modules were extended to support hundreds of banks' legacy ATM systems until we suddenly realised that the resulting feature interactions made them completely insecure. There are similar tensions in many other application areas, from mobile phone baseband software used in over a hundred different models of phone, to vehicle components used in over a hundred different cars. There must be better ways of managing this; I expect that applications with high fan-out will move in the direction of a microservices architecture with a common core and pluggable proxies for different calling applications.

27.6.6 And finally …

You also need to understand how to manage people, and the HR department can't do this for you[8]. Tech management cannot be done by generalists as they're unlikely to win the trust of their staff[9]. It also cannot be done well by engineers who are too introverted to engage and motivate others. Far too many managers went for the job not because they thought they might be good at it, but because it was the only way to get a decent salary. Successful managers in tech have to love and understand tech; they also have to love and understand people.

For your star engineers, you need to create other leadership roles. They may be innovators who will be most productive in an R&D lab. They may be the custodians of your institutional memory: old-timers who know the thirty years of history behind your product and can stop people repeating the mistakes of the past. They may provide moral leadership to your engineering staff and reassurance to your customers. They can help attract bright young recruits who want to work with them. But the key, I feel, is this: that you have one or more engineering professions in your firm. What's their shape? Who leads them? How do they compare to those in your competitors? How do you grow and develop them? If you realise that all of a sudden you have to unify the safety engineering and security engineering professions in your company, who is going to do that, and how?

27.7 Summary

Managing a project to build, or enhance, a system that has to meet critical requirements for security, safety or both, is a hard problem. As more and more devices acquire CPUs and communications, we need to build things that do real work while keeping out any vulnerabilities that would make them a target for attack. In other words, you want software security – together with other functionality, and other emergent properties such as safety and real-time performance.

If you're building something entirely new, or a major functional enhancement of an existing system, then understanding the requirements is often the hardest part of the process. More gentle system evolution can involve subtler changes to requirements. Larger changes can be forced externally; systems that succeed and get popular, can expect to get attacked.

[8]The main job of HR is damage limitation – stopping leavers from suing you.

[9]As a math geek I always tended to see the MBA types and other corporate politicians much as the Earl of Rochester saw King Charles II: "Here lies our sovereign lord the king, Whose word no man relies on; He never says a foolish thing, Nor ever does a wise one."

Writing secure code is hard because of this dynamic context: the first problem is to figure out what you're trying to do. However, even given a tight specification, or constant feedback from people hacking your product, you're not home and dry. There are a number of challenges in hiring the right people, giving them the right tools, helping them develop the right ways of working, backing them up with expertise in the right way, and above all creating an environment in which they work to improve their security capability.

Research problems

The issues discussed in this chapter are among the hardest and the most important of any in our field. However, they receive little attention because they lie at the boundaries with software engineering, applied psychology, economics and management. Each of these interfaces could be a productive area of research. Security economics and security psychology have made great strides in the last few years, and we now know we need to do a lot more work on making security tools easier for developers to use. One logical next step is integrating what we know with safety economics and safe usability.

Yet many failures are due to organisational behaviour. Every experienced developer or security consultant has their share of horror stories about firms with perverse incentives, toxic cultures, high staff turnover, incompetent management and all the rest of the things we see in the Dilbert cartoons. It could be useful if someone were to collect a library of case histories of security failures caused by unsatisfactory incentives in organisations, such as [878]. What might follow given a decent empirical foundation?

The late Jack Hirshleifer took the view that we should try to design organizations in which managers were forced to learn from their mistakes: how could we do that? How might you set up institutional structures to monitor changes in the threat environment and feed them through into not just systems development but into supporting activities such as internal control? Maybe we need something like Management as Code? How can you design an organization that is 'safety-incentive-compatible' in the sense that staff behave with an appropriate level of care? And what might the cultural anthropology of organisations have to say? We saw in the last chapter how the response of governments to the apparently novel threats posed by Al-Qaida was maladaptive in many ways: far too much of our social resilience budget was spent on anti-terror theatre, at the expense of preparedness for other societal risks such as pandemics. Similarly, far too much of the typical firm's resilience budget has been captured by compliance, safety theatre and security theatre. As a result, too much of the security development effort is aimed at compliance rather than managing security and safety risks properly. How can we design feedback mechanisms that will enable us to put the right amount of effort in the

right place? Or do we need broader structural change, such as the breakup of the Big Four accountancy firms?

Further reading

Managing the development of information systems has a large, diffuse and multidisciplinary literature. There are classics everyone should read, such as Fred Brooks's '*Mythical Man Month*' [329] and Nancy Leveson's '*Safeware*' [1151]. An influential modern classic is Reed Hastings' culture slide deck, describing his management policy when building Netflix [872]. The economics of the software life cycle are discussed by Brooks and by Barry Boehm [273]. The modern books everyone should read, as of 2020, are probably the Google books on '*Site Reliability Engineering*' [237] and on '*Building Secure and Reliable Systems*' [23]. The Microsoft approach to the security development lifecycle has many online resources; their doctrine on threat modelling is discussed by Frank Swiderski and Window Snyder [1855]; and their security VP Mike Nash describes the background to the big security push and the adoption of the security development lifecycle at [1387]. The most general set of standards on safety functional and integrity requirements, and the associated engineering processes, is IEC 61508; there are further sets of industry-specific standards. For example, there's IEC 61511 for process plant control systems, IEC 62061 for safety of machinery, and the EN 5012x series for railways. In aviation it's RTCA DO-254 for electronic hardware and RTCA DO-178C for software, while in the motor industry it's ISO 26262 for safety and ISO 21434 for security – though at the time of writing this is still just a draft. Standards for the Internet of Things are also a work in progress, and the current draft is ETSI EN 303 645 V2.1.

We can learn a lot from other engineering disciplines. Henry Petroski discusses the history of bridge building, why bridges fall down, and how civil engineers learned to learn from the collapses: what tends to happen is that an established design paradigm is stretched and stretched until it suddenly fails for some unforeseen reason [1520]. IT project failures are another necessary subject of study; there's a casebook on how to manage uncertainty in projects by Christoph Loch, Arnoud DeMeyer and Michael Pich [1180]. For security failures, it's important to follow the leading security blogs such as Schneier on Security, Krebs on Security and SANS, as well as the trade press.

Organizational aspects are discussed at length in the business school literature, but this can be bewildering to the outsider. Many business academics praise business, which is fine for selling airport books, but what we need is a more critical understanding of how organisations fail. If you're only going

to read one book, make it Lewis Pinault's '*Consulting Demons*' – the confessions of a former insider about how the big consulting firms rip off their customers [1530]. Organisational theorists such as Charles Handy talk of firms having cultures based on power, roles, tasks or people, or some combination. It's not just who has access to whom, but who's prepared to listen to whom and who will just ignore orders from whom. Perhaps such insights might help us design more effective tools and workflows that support how people actually work best.