CHAPTER

25

Managing the Development of Secure Systems

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

One of the most important problems we face today, as techniques and systems become more and more pervasive, is the risk of missing that fine, human point that may well make the difference between success and failure, fair and unfair, right and wrong . . . no IBM computer has an education in the humanities.

— Tom Watson

Management is that for which there is no algorithm. Where there is an algorithm, it's administration.

— Roger Needham

25.1 Introduction

So far we've discussed a great variety of security applications, techniques and concerns. If you're a working IT manager or consultant, paid to build a secure system, you will by now be looking for a systematic way to select protection aims and mechanisms. This brings us to the topics of system engineering, risk analysis and, finally, the secret sauce: how you manage a team to write secure code.

Business schools reckon that management training should be conducted largely through case histories, stiffened with focussed courses on basic topics such as law, economics and accounting. I have broadly followed their model in this book. We went over the fundamentals, such as protocols, access control

and crypto, and then looked at a lot of different applications with a lot of case histories.

Now we have to pull the threads together and discuss how to go about solving a general security engineering problem. Organizational issues matter here as well as technical ones. It's important to understand the capabilities of the staff who'll operate your control systems, such as guards and auditors, to take account of the managerial and work-group pressures on them, and get feedback from them as the system evolves. You also have to instil suitable ways of thinking and working into your development team. Success is about attitudes and work practices as well as skills. There are tensions: how do you get people to think like criminals, yet work enthusiastically for the good of the product?

25.2 Managing a Security Project

The hardest part of the project manager's job is usually figuring out what to protect and how. Threat modelling and requirements engineering are what separate out the star teams from the also-rans.

The first killer problem is understanding the tradeoff between risk and reward. Security people naturally focus too much on the former and neglect the latter. If the client has a turnover of $10 m, profits of $1 m and theft losses of $150,000, the security consultant may make a loss-reduction pitch about 'how to increase your profits by 15%'; but it could well be in the shareholders' interests to double the turnover to $20 m, even if this triples the losses to $450,000. Assuming the margins stay the same, the profit is now $1.85 m, up 85%.

So if you're the owner of the company, don't fall into the trap of believing that the only possible response to a vulnerability is to fix it, and distrust the sort of consultant 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. But the security team — whether internal developers or external consultants — usually has an incentive to play up the threats, and as it has more expertise on the subject it's hard to gainsay. The same mechanisms that drive national overreaction to terrorist incidents are at work in the corporation too.

25.2.1 A Tale of Three Supermarkets

My thumbnail case history to illustrate this point concerns three supermarkets. Among the large operational costs of running a retail chain are the salaries of the checkout and security staff, and the stock shrinkage due to theft. Checkout queues aggravate your customers, so cutting staff isn't always an option, and working them harder might mean more shrinkage. So what might technology do to help?

One supermarket in South Africa decided to automate completely. All produce would carry an RFID tag, so that an entire trolley-load could be scanned automatically. If this had worked, it could have killed both birds with one stone; the same RFID tags could have cut staff numbers and made theft harder. There was a pilot, but the idea couldn't compete with barcodes. Customers had to use a special trolley, which was large and ugly — and the RF tags also cost money. There has been a lot of investment in RFID, but there's still a problem: tags fixed to goods that conduct electricity, such as canned drinks, are hard to read reliably.

Another supermarket in a European country believed that much of their losses were due to a hard core of professional thieves, and wanted to use RFID to cut this. When they eventually realized this wouldn't work, they then talked of building a face-recognition system to alert the guards whenever a known villain came into a store. But current technology can't do that with low enough error rates. In the end, the chosen route was civil recovery. When a shoplifter is caught, then even after the local magistrates have fined her a few bucks, the supermarket sues her in the civil courts for wasted time, lost earnings, attorneys' fees and everything else they can think of; and then armed with a judgement for a few thousand bucks they go round to her house and seize all the furniture. So far so good. But their management spent their time and energy getting vengeance on petty thieves rather than increasing sales. Soon they started losing market share and saw their stock price slide. Diverting effort from marketing to security was probably a symptom of their decline rather than a cause, but may have contributed to it.

The supermarket that seemed to be doing best when I wrote the first edition in 2001 was Waitrose in England, which had just introduced self-service scanning. When you go into their store you swipe your store card in a machine that dispenses a portable barcode scanner. You scan the goods as you pick them off the shelves and drop them into your shopping bag. At the exit you check in the scanner, get a printed list of everything you've bought, swipe your credit card and head for the car park. This might seem rather risky — but then so did the self-service supermarket back in the days when traditional grocers' shops had all the goods behind the counter. In fact, there are several subtle control mechanisms at work. Limiting the service to store card holders not only lets you exclude known shoplifters, but also helps market the store card. By having one you acquire a trusted status visible to any neighbors you meet while shopping — so losing your card (whether by getting caught stealing, or more likely falling behind on your payments) could be embarrassing. And trusting people removes much of the motive for cheating as there's no kudos in beating the system. Of course, should the guard at the video screen see someone lingering suspiciously near the racks of hundred-dollar wines, it can always be arranged for the system to 'break' as the suspect gets to the checkout, which gives the staff a non-confrontational way to recheck the bag's contents.

Since then, the other supermarkets in the UK have adopted self-service, but the quality of the implementation varies hugely. The most defensive stores — where the security folks have had too much say in the design — force you to scan at specially-designed self-service checkout lanes, and weigh each item as you place it on the packing stand after scanning, in an attempt to detect anyone packing an item without scanning it first. These systems are flaky and frequently interrupt the shopper with complaints, which communicate distrust. Waitrose seems to be going from strength to strength, while the most defensive store (with the most offensive systems) is facing a takeover bid as I write.

25.2.2 Risk Management

Security policies tend to come from a company's risk management mechanisms. Risk management is one of the world's largest industries: it includes not just security engineers but also fire and casualty services, insurers, the road safety industry and much of the legal profession. Yet it is startling how little is really known about the subject. Engineers, economists, actuaries and lawyers all come at the problem from different directions, use different language and arrive at quite incompatible conclusions. There are also strong cultural factors at work. For example, if we distinguish risk as being where the odds are known but the outcome isn't, from uncertainty where even the odds are unknown, then most people are more uncertainty-averse than risk-averse. Where the odds are directly perceptible, a risk is often dealt with intuitively; but even there, our reactions are colored by the various cognitive biases discussed in Chapter 2. Where the science is unknown or inconclusive, people are free to project all sorts of fears and prejudices. But risk management is not just a matter of actuarial science colored by psychology. Organisations matter too, whether governments or businesses.

The purpose of business is profit, and profit is the reward for risk. Security mechanisms can often make a real difference to the risk/reward equation but ultimately it's the duty of a company's board of directors to get the balance right. In this *risk management* task, they may draw on all sorts of advice — lawyers, actuaries, security engineers — as well as listening to their marketing, operations and financial teams. A sound corporate risk management strategy involves much more than attacks on information systems; there are non-IT operational risks such as fires and floods as well as legal risks, exchange rate risks, political risks, and many more. Company bosses need the big picture view to take sensible decisions, and a difficult part of their task is to ensure that advisers from different disciplines work together closely enough, but without succumbing to groupthink.

In the culture that's grown up since Enron and Sarbanes-Oxley, risk management is supposed to drive internal control. The theory and practice of

this are somewhat divergent. In theory, internal controls are about mitigating and managing the tensions between employees' duty to maximise shareholder utility, and their natural tendency to maximise their own personal utility instead. At the criminal end of things this encompasses theft and fraud from the company; I discussed in Chapter 10 how to control that. However, internal controls are also about softer conflicts of interest. Managers build empires; researchers tackle interesting problems rather than profitable ones; programmers choose tools and platforms that will look good on their CVs, rather than those best suited to the company's tasks. A large body of organizational theory applies microeconomic analysis to behaviour in firms in an attempt to get a handle on this. One of its effects is the growing use of stock options and bonus schemes to try to align employees' interests with shareholders'.

The practice of risk management has been largely determined by the rules evolved by the Big Four audit firms in response to Sarbanes-Oxley. A typical firm will show that it's discharging its responsibilities by keeping a risk register that identifies the main risks to its financial performance and ranks them in some kind of order. Risks then get 'owners', senior managers who are responsible for monitoring them and deciding on any specific countermeasures. Thus the finance director might be allocated exchange-rate and interest-rate risks, some of which he'll hedge; operational risks like fires and floods will be managed by insurance; and the IT director might end up with the system risks. The actual controls tend to evolve over time, and I'll discuss the process in more detail in section 25.4.1.2 later.

25.2.3 Organizational Issues

It goes without saying that advisers should understand each others' roles and work together rather than trying to undermine each other. But, human nature being what it is, the advisers may cosy up with each other and entrench a consensus view that steadily drifts away from reality. So the CEO, or other responsible manager, has to ask hard questions and stir the cauldron a bit. It's also important to have a variety of experts, and to constantly bring in new people. One of the most important changes post-Enron is the expectation that companies should change their auditors from time to time; and one of the most valuable tasks the security engineer gets called on to perform is when you're brought in, as an independent outsider, to challenge groupthink. On perhaps a third of the consulting assignments I've done, there's at least one person at the client company who knows exactly what the problem is and how to fix it — they just need a credible mercenary to beat up on the majority of their colleagues who've got stuck in a rut. (This is one reason why famous consulting firms that exude an air of quality and certainty may have a competitive advantage over specialists, but a generalist consultant may have

difficulty telling which of the ten different dissenting views from insiders is the one that must be listened to.)

Although the goals and management structures in government may be slightly different, exactly the same principles apply. Risk management is often harder because people are more used to compliance with standards rather than case-by-case requirements engineering. Empire-building is a particular problem in the public sector. James Coyne and Normal Kluksdahl present in [331] a classic case study of information security run amok at NASA. There, the end of military involvement in Space Shuttle operations led to a security team being set up at the Mission Control Center in Houston to fill the vacuum left by the DoD's departure. This team was given an ambitious charter; it became independent of both development and operations; its impositions became increasingly unrelated to budget and operational constraints; and its relations with the rest of the organization became increasingly adversarial. In the end, it had to be overthrown or nothing would have got done.

The main point is that it's not enough, when doing a security requirements analysis, to understand the education, training and capabilities of the guards (and the auditors, and the checkout staff, and everyone else within the trust perimeter). Motivation is critical, and many systems fail because their designers make unrealistic assumptions about it. Organizational structures matter. There are also risk dynamics that can introduce instability. For example, an initially low rate of fraud can make people complacent and careless, until suddenly things explode. Also, an externally induced change in the organization — such as a merger, political uncertainty — can undermine morale and thus control. (Part of my younger life as a security consultant was spent travelling to places where local traumas were causing bank fraud to rocket, such as Hong Kong in 1989.)

So you have to make allowance in your designs for the ways in which human frailties express themselves through the way people behave in organizations.

25.2.3.1 The Complacency Cycle and the Risk Thermostat

Phone fraud in the USA has a seven year cycle: in any one year, one of the 'Baby Bells' is usually getting badly hurt. They hire experts, clean things up and get everything under control — at which point another of them becomes the favored target. Over the next six years, things gradually slacken off, then it's back to square one. This is a classic example of organizational complacency. How does it come about?

Some interesting and relevant work has been done on how people manage their exposure to risk. John Adams studied mandatory seat belt laws, and established that they don't actually save lives: they just transfer casualties from vehicle occupants to pedestrians and cyclists. Seat belts make drivers feel safer, so they drive faster in order to bring their perceived risk back up to

its previous level. He calls this a *risk thermostat* and the model is borne out in other applications too [10, 11]. The complacency cycle can be thought of as the risk thermostat's corporate manifestation. Firms where managers move every two years and the business gets reorganised every five just can't maintain a long corporate memory of anything that wasn't widespread knowledge among the whole management; and problems that have been 'solved' tend to be forgotten. But risk management is an interactive business that involves all sorts of feedback and compensating behavior. The resulting system may be stable, as with road traffic fatalities; or it may oscillate, as with the Baby Bells.

Feedback mechanisms can also limit the performance of risk reduction systems. The incidence of attacks, or accidents, or whatever the organization is trying to prevent, will be reduced to the point at which there are not enough of them — as with the alarm systems described in Chapter 10 or the intrusion detection systems described in section 21.4.4. Then the sentries fall asleep, or real alarms are swamped by false ones, or organizational budgets are eroded to (and past) the point of danger. I mentioned in Chapter 12 how for 50 years the U.S. Air Force never lost a nuclear weapon. Then the five people who were supposed to check independently whether a cruise missile carried a live warhead or a blank failed to do so — each relied on the others. Six warheads were duly lost for 36 hours. Colonels will be court-martialled, and bombs will be counted carefully for a while. But eventually the courts martial will be forgotten. (How would you organize it differently?)

25.2.3.2 Interaction with Reliability

Poor internal control often results from systems where lots of transactions are always going wrong and have to be corrected manually. Just as in electronic warfare, noise degrades the receiver operating characteristic. A high tolerance of chaos undermines control, as it creates a high false alarm rate for many of the protection mechanisms at once. It also tempts staff: when they see that errors aren't spotted they conclude that theft won't be either.

The correlation between quality and security is a recurring theme in the literature. For example, it has been shown that investment in software quality will reduce the incidence of computer security problems, regardless of whether security was a target of the quality program or not; and that the most effective quality measure from the security point of view is the code walk-through [470]. The knowledge that one's output will be read and criticized has a salutary effect on many programmers.

Reliability can be one of your biggest selling points when trying to get a client's board of directors to agree on protective measures. Mistakes cost money; no-one really understands what software does; if mistakes are found then the frauds should be much more obvious; and all this can be communicated to top management without embarrassment on either side.

25.2.3.3 Solving the Wrong Problem

Faced with an intractable problem, it is common for people to furiously attack a related but easier one; we saw the effects of this in the public policy context earlier in section 24.3.10. Displacement activity is also common in the private sector, where an example comes from the smartcard industry. As discussed in section 16.7.4, the difficulty of protecting smartcards against probing and power-analysis attacks led the industry to concentrate on securing the chip mask instead. Technical manuals are available only under NDA; plant visitors have to sign an NDA at reception; much technical material isn't available at all; and vendor facilities have almost nuclear-grade physical security. Physical security overkill may impress naive customers — but almost all of the real attacks on fielded smartcard systems used technical attacks that didn't depend on design information.

One organizational driver for this is an inability to deal with uncertainty. Managers prefer approaches that they can implement by box-ticking their way down a checklist. So if an organization needs to deal with an actual risk, then some way needs to be found to keep it as a process, and stop it turning into a due-diligence checklist item. But there is constant pressure to replace processes with checklists, as they are less demanding of management attention and effort. The quality bureaucracy gets in the way here; firms wanting quality-assurance certification are prodded to document their business processes and make them repeatable. I noted in section 8.7 that bureaucratic guidelines had a strong tendency to displace critical thought; instead of thinking through a system's protection requirements, designers just reached for their checklists.

Another organizational issue is that when exposures are politically sensitive, some camouflage may be used. The classic example is the question of whether attacks come from insiders or outsiders. We've seen in system after system that the insiders are the main problem, whether because some of them are malicious or because most of them are careless. But it's often hard to enforce controls too overtly against line managers and IT staff, as this will alienate them, and it's also hard to get them to manage such controls themselves. It's not easy to sell a typical company's board of directors on the need for proper defences against insider attack, as this impugns the integrity and reliability of the staff who report to them. Most company boards are (quite rightly) full of entrepreneurial, positive, people with confidence in their staff and big plans for the future, rather than dyspeptic bean-counters who'd like to control everyone even more closely. So the complaint of information security managers down the years — that the board doesn't care — may not actually be a bug, but a feature.

Often a security manager will ask for, and get, money to defend against nonexistent 'evil hackers' so that he can spend most of it on controls to manage

the real threat, namely dishonest or careless staff. I would be cautious about this strategy because protection mechanisms without clear justifications are likely to be eroded under operational pressure — especially if they are seen as bureaucratic impositions. Often it will take a certain amount of subtlety and negotiating skill, and controls will have to be marketed as a way of reducing errors and protecting staff. Bank managers love dual-control safe locks because they understand that it reduces the risk of their families being taken hostage; and requiring two signatures on transactions over a certain limit means extra shoulders to take the burden when something goes wrong. But such consensus on the need for protective measures is often lacking elsewhere.

25.2.3.4 Incompetent and Inexperienced Security Managers

Things are bad enough when even a competent IT security manager has to use guile to raise money for an activity that many of his management colleagues regard as a pure cost. In real life, things are even worse. In many traditional companies, promotions to top management jobs are a matter of seniority and contacts; so if you want to get to be the CEO you'll have to spend maybe 20 or 30 years in the company without offending too many people. Being a security manager is absolutely the last thing you want to do, as it will mean saying no to people all the time. It's hardly surprising that the average tenure of computer security managers at U.S. government agencies is only seven months [605].

Matters are complicated by reorganizations in which central computer security departments may be created and destroyed every few years, while the IT audit function oscillates between the IT department, an internal audit department and outside auditors or consultants. The security function is even less likely than other business processes to receive sustained attention and analytic thought, and more likely to succumb to a box-ticking due diligence mentality. Also, the loss of institutional memory is often a serious problem.

25.2.3.5 Moral Hazard

Companies often design systems so that the risk gets dumped on third parties. This can easily create a *moral hazard* by removing the incentives for people to take care, and for the company to invest in risk management techniques. I mentioned in Chapter 10 how banks in some countries claimed that their ATMs could not possibly make mistakes, so that any disputes must be the customer's fault. This led to a rise in fraud as staff got lazy and even crooked. So, quite in addition to the public policy aspects, risk denial can often make the problem worse: a company can leave itself open to staff who defraud it knowing that a prosecution would be too embarrassing.

Another kind of moral hazard is created when people who take system design decisions are unlikely to be held accountable for their actions. This can happen for many reasons. IT staff turnover could be high, with much reliance placed on contract staff; a rising management star with whom nobody wishes to argue can be involved as a user in the design team; imminent business process re-engineering may turn loyal staff into surreptitious job-seekers. In any case, when you are involved in designing a secure system, it's a good idea to look round your colleagues and ask yourself which of them will shoulder the blame three years later when things go wrong.

Another common incentive failure occurs when one part of an organization takes the credit for the profit generated by some activity, while another part picks up the bills when things go wrong. Very often the marketing department gets the praise for increased sales, while the finance department is left with the bad debts. A rational firm would strike a balance between risk and reward, but internal politics can make firms behave irrationally. The case of the three supermarkets, mentioned above, is just one example. Companies may swing wildly over a period of years from being risk takers to being excessively risk averse, and (less often) back again. John Adams found that risk taking and risk aversion are strongly associated with different personality types: the former tend to be individualists, a company's entrepreneurs, while the latter tend to be hierarchists. As the latter usually come to dominate bureaucracies, it is not surprising that stable, established organizations tend to be much more risk averse than rational economics would dictate.

So what tools and concepts can help us cut through the fog of bureaucratic infighting and determine a system's protection requirements from first principles?

The rest of this chapter will be organized as follows. The next section will look at basic methodological issues such as top-down versus iterative development. After that, I'll discuss how these apply to the specific problem of security requirements engineering. Having set the scene, I'll then return to risk management and look at technical tools. Then I'll come back and discuss how you manage people. That's really critical. How do you get people to care about vulnerabilities and bugs? This is partly incentives and partly culture; the two reinforce each other, and many companies get it wrong.

25.3 Methodology

Software projects usually take longer than planned, cost more than budgeted for and have more bugs than expected. (This is sometimes known as 'Cheops' law' after the builder of the Great Pyramid.) By the 1960s, this had become known as the *software crisis*, although the word 'crisis' is hardly appropriate

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.*

This encompassed the hope that the problem could be solved in the same way that one builds ships and aircraft, with a proven scientific foundation and a set of design rules [954]. Since then much progress has been made. However, the results of the progress 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. Now, we still see about 30% of large projects failing — but the failures are much bigger. The tools get us farther up the complexity mountain before we fall off, but the rate of failure appears to be exogenous, set by such factors as company managers' appetite for risk[1].

Anyway, 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 complex problems. A bank's administrative systems, for example, may involve tens of millions of lines of code and be too complex for any one person to understand.

Incidental complexity is largely dealt with using technical tools. The most important of these 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. There are also formal methods that enable particularly error-prone design and programming tasks to be checked. The obvious security engineering example is provided by the BAN logic for verifying cryptographic protocols, described in section 3.8.

Intrinsic complexity usually requires methodological tools that help divide up the problem into manageable subproblems and restrict the extent to which these subproblems can interact. There are many tools on the market to help you do this, and which you use may well be a matter of your client's policy. But there are basically two approaches — top-down and iterative.

[1] A related, and serious, problem is that while 30% of large projects fail in industry, perhaps only 30% of large projects in the public sector succeed; for a topical case history, see the National Academies' report on the collapse of the FBI's attempt to modernise its case file system [860].

25.3.1 Top-Down Design

The classical model of system development is the *waterfall model* developed by Win Royce in the 1960s for the U.S. Air Force [1090]. 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 25.1). Until recently, this was how all systems for the U.S. Department of Defense had to be developed.

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 critical thing about 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. Therein lie the waterfall model's strengths, and also its weaknesses.

The strengths of the waterfall model are that it compels early clarification of system goals, architecture, and interfaces; it makes the project manager's

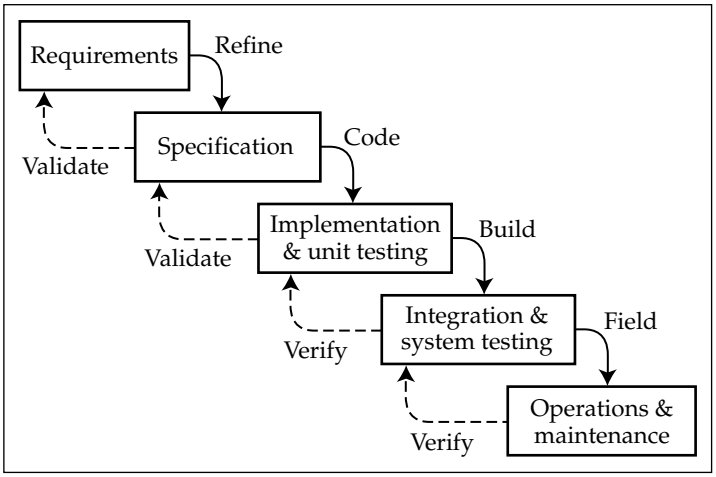

Figure 25.1: The waterfall model

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.

But very often the detailed requirements aren't known in advance and then an iterative approach is necessary. There are quite a few possible reasons for this. Perhaps the requirements aren't understood yet by the customer, and a prototype is necessary to clarify them rather than more discussion; the technology may be changing; the environment could be changing; or a critical part of the project may involve the design of a human-computer interface, which will probably involve several prototypes. In fact, very often the designer's most important task is to help the customer decide what he wants, and although this can sometimes be done by discussion, there will often be a need for some prototyping[2].

The most common reason of all for using an iterative development is that we're starting from an existing product which 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. Nowadays, as software becomes ever more packaged and the packages become ever more complex, the reality in many software firms is that 'the maintenance is the product'. The only way to write something as complex as Office is to start off from an existing version and enhance it. That does not mean that the waterfall model is obsolete; on the contrary, it may be used to manage a project to develop a major new feature. However, we also need to think of the overall management of the product, and that's likely to be based on iteration.

25.3.2 Iterative Design

So many development projects need iteration, whether to firm up the specification by prototyping, or to manage the complexity of enhancing an already large system.

[2]The Waterfall Model had a precursor in a methodology developed by Gerhard Pahl and Wolfgang Beitz in Germany just after World War 2 for the design and construction of mechanical equipment such as machine tools [1001]; 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 a totally prescriptive system it developed into.

In the first case, a common approach is Barry Boehm's *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. It's called the spiral model because the process is often depicted as in Figure 25.2.

In the second case, the standard model is *evolutionary development*. An early advocate for this approach was Harlan Mills, who taught that one 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 works: software products nowadays quickly become so complex that they could not be economically developed (or redeveloped) from scratch. Indeed, Microsoft has tried more than once to rewrite Word, but gave up each time. Perhaps the best book on the evolutionary development model is by Maguire, a Microsoft author [829]. In this view of the world, products aren't the result of a project but of a process that involves continually modifying previous versions.

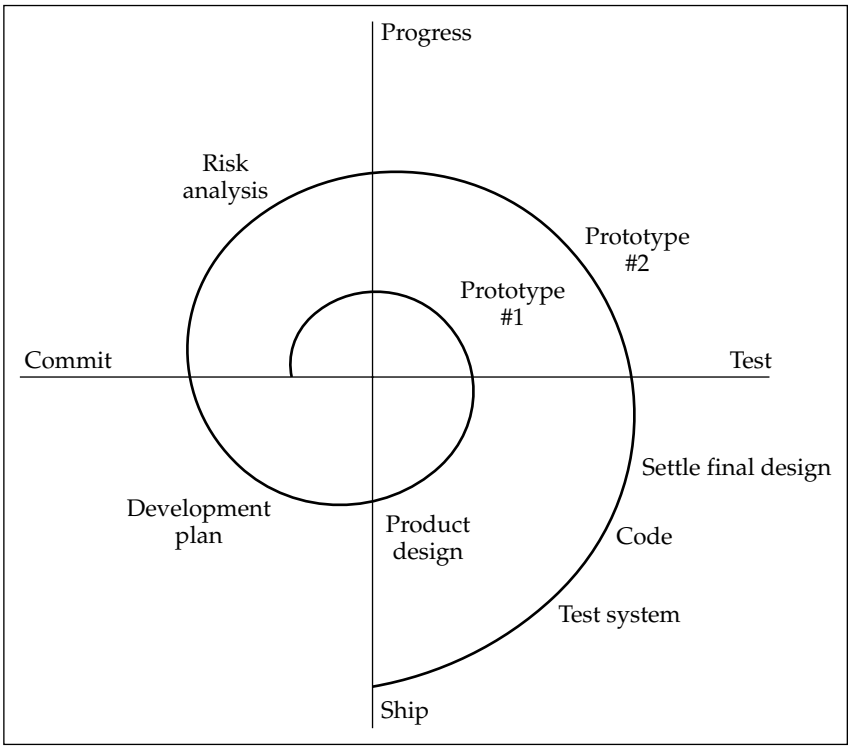

Figure 25.2: The spiral model

Unfortunately, evolutionary development tends to be neglected in academic courses and books on software engineering, and it can cause some particular problems for the security engineer.

The critical thing 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. Of course, it's always possible that a build just doesn't work at all, and there may be quite long disruptions as a major change is implemented. So we consider the current 'generation' of the product to be the last build that worked. One way or another, we always have viable code that we can ship out for beta testing or whatever our next stage is.

The technology of testing is probably the biggest practical improvement in software engineering during the 1990s. Before automated regression tests were widely used, engineers used to reckon that 15% of bug fixes either introduced new bugs or reintroduced old ones [9]. But automated testing is less useful for the security engineer for a number of reasons. Security properties are more diverse, and security engineers are fewer in number, so we haven't had as much investment in tools and the available tools are much more fragmentary and primitive than those available to the general software engineering community. Many of the flaws that we want to find and fix — such as stack overflow attacks — tend to appear in new features rather than to reappear in old ones. Specific types of attack are also often easier to fix using specific remedies — such as the canary in the case of stack overflow. 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 regression testing is still really important. It finds functionality that has been affected by a change but not fully understood.

Much the same applies to safety critical systems, which are similar in many respects to secure systems. Some useful lessons can be drawn from them.

25.3.3 Lessons from Safety-Critical Systems

Critical computer systems can be defined as 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, critical to the environment or whatever. Obvious examples of the safety-critical variety include flight controls and automatic braking systems. There is a large literature

on this subject, and a lot of methodologies have been developed to help manage risk intelligently.

Overall, these methodologies tend to follow the waterfall view of the universe. The usual procedure is to identify hazards and assess risks; decide on a strategy to cope with them (avoidance, constraint, redundancy . . .); to trace the hazards down to hardware and software components which are thereby identified as critical; to identify the operator procedures which are also critical and study the various applied psychology and operations research issues; and finally to decide on a test plan and get on with the task of testing. The outcome of the testing is not just a system you're confident to run live, but a *safety case* to justify running it.

The 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 documentation linking this to component reliability and human factor issues, and the results of tests (both at component level and system level) which show that the required failure rates have been achieved.

The ideal system design avoids hazards entirely. A good illustration comes from the motor reversing circuits in Figure 25.3. In the first 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 short circuited 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.

Hazard elimination is useful in security engineering too. We saw an example in the early design of SWIFT in section 10.3.1: there, the keys used to authenticate transactions between one bank and another were exchanged between the banks directly. In this way, SWIFT personnel and systems did not have the means to forge a valid transaction and had to be trusted much less. In general, minimizing the trusted computing base is to a large extent an exercise in hazard elimination.

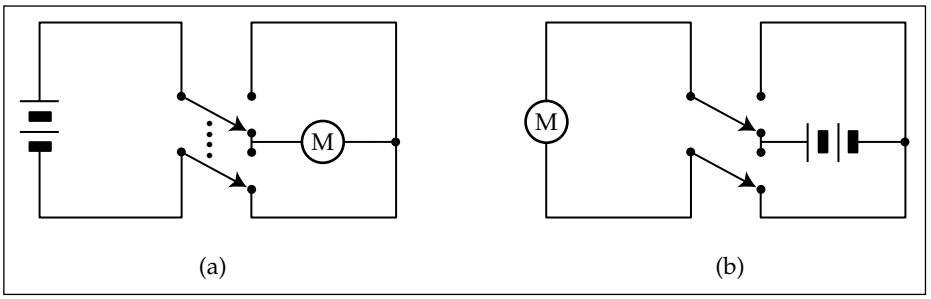

Figure 25.3: Hazard elimination in motor reversing circuit

Once as many hazards have been eliminated 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* as a tree is constructed whose root is the undesired behavior and whose successive nodes are its possible causes. This carries over in a fairly obvious way to security engineering, and here's an example of a fault tree (or *threat tree*, as it's often called in security engineering) for fraud from automatic teller machines (see Figure 25.4).

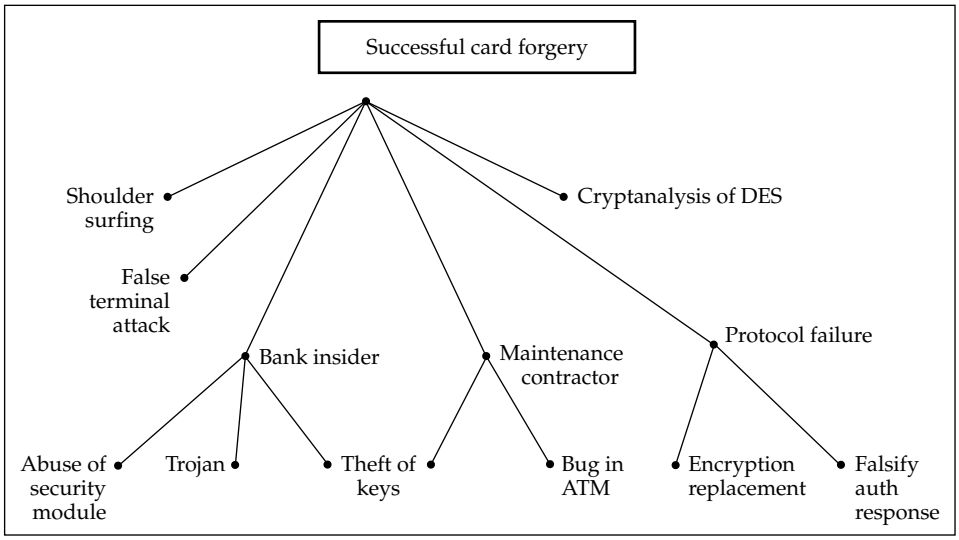

Figure 25.4: A threat tree

Threat trees are used in the U.S. Department of Defense. You start out from each undesirable outcome, and work backwards by writing down each possible immediate cause. You then work backwards by adding each precursor condition, and recurse. Then 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 which would break security. This can amount to an attack manual for the system, and so it may be highly classified. Nonetheless, it must exist, and if the system evaluators or accreditors can find any significant extra attacks, then they may fail the product.

Returning to the safety-critical world, another way of doing the hazard analysis is *failure modes and effects analysis* (FMEA), pioneered by NASA, which is bottom-up rather than top-down. 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 often useful in security engineering; it's a good idea to understand the consequences of a failure of any one of your protection mechanisms.

A really thorough analysis of failure modes may combine top-down and bottom-up approaches. There are various ways to manage the resulting mass of data. For example, one can 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 we can check that there are two entries in each of the relevant columns. In this way, we can demonstrate graphically that in the presence of the hazard in question, at least two failures will be required to cause an accident. This methodology goes across unchanged to security engineering, as I'll discuss below.

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 various techniques which 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 would probably consider anyone who was building human interfaces to security-critical systems and who hadn't read this book to be negligent of something went wrong.

The telegraphic summary is that the error rate depends on the familiarity and complexity of the task, 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 of the task. Quite a lot is known about the cognitive biases and other psychological factors that make particular types of error more common, and a prudent engineer will understand and avoid these. Nonetheless, designers of things like nuclear reactors are well aware (at least since Three Mile Island) that no matter how many design walkthroughs you do, it's when the red lights go on for the first time that the worst mistakes get made.

Similarly, in security systems, it tends to be important but rarely performed tasks such as getting senior managers to set up master crypto keys where the most egregious blunders can be expected. A classic example from [34] was when a bank wished to create a set of three master keys to link their cash machine network to VISA and needed a terminal to drive the security module. A contractor obligingly lent them a laptop PC, together with software which emulated the desired type of terminal. With this the senior managers duly created the required keys and posted them off to VISA. None of them realized that most PC terminal emulation software packages can be set to log all the transactions passing through, and this is precisely what the contractor did. He captured the clear zone key as it was created, and later used it to decrypt the

bank's master PIN key. The lesson to take from this is that security usability isn't just about presenting a nice intuitive interface to the end-user that accords with common mental models of threat and protection in the application area, as discussed in Chapter 2. It's pervasive, and extends all the way through the system's operations, at the back end as well as the front.

So when doing security requirements engineering, special care has to be paid to the skill level of the staff who will perform each critical task and estimates made of the likelihood of error. Be cautious here: an airplane designer can rely on a fairly predictable skill level from anyone with a commercial pilot's licence, and even a shipbuilder knows the strengths and weaknesses of a sailor in the Navy. Usability testing can (and should) be integrated with staff training: when pilots go for their six-monthly 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. This in turn provides valuable feedback to the cockpit designers.

The security engineer usually has no such luck. Many security failures remind me of a remark made by a ranger at Yosemite about the devices provided to keep bears from getting at campers' food supplies: that it's an impossible engineering problem because the brighter bears are smarter than the dumber campers.

As well as the problem of testing usability, there are also technical testability issues. A common problem with redundant systems is *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. Several air crashes have resulted from flying an airliner with one of the flight control systems dysfunctional; although pilots may be intellectually aware that one of the cockpit displays is unreliable, their training may lead them to rely on it under pressure rather than checking with other instruments. So a further failure can be catastrophic. In such cases, it's better to arrange things so that displays give no reading at all rather than an inaccurate one. A security example is the ATM problem mentioned in section 10.4.2 where a bank issued all its customers with the same PIN. In such cases, the problem often isn't detected until much later. The fault gets masked by the handling precautions applied to PINs, which ensure that even the bank's security and audit staff only get hold of the PIN mailer for their own personal account. So some thought is needed about how faults can remain visible and testable even when their immediate effects are masked.

Our final lesson from safety critical systems is that although there will be a safety requirements specification and safety test criteria as part of the safety case for the lawyers or regulators, it is good practice to integrate this with the general requirements and test documentation. If the safety case is a separate

set of documents, then it's easy to sideline it after approval is obtained and thus fail to maintain it properly. If, on the other hand, it's an integral part of the product's management, then not only will it likely get upgraded as the product is, but it is also much more likely to be taken heed of by experts from other domains who might be designing features with possible interactions.

As a general rule, safety must be built in as a system is developed, not retrofitted; the same goes for security. The main difference is in the failure model. Rather than the effects of random failure, we're dealing with 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. In effect, our task is to program a computer which gives answers which 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 [74]. This provides an insight into one of the reasons security engineering is hard: Satan's computer is hard to test [1126].

25.4 Security Requirements Engineering

In Chapter 8, I defined a *security policy model* to be a concise statement of the protection properties that a system, or generic type of system, must have. This was driven by the *threat model*, which sets out the attacks and failures with which the system must be able to cope. The security policy model is further refined into a *security target*, which is a more detailed description of the protection mechanisms a specific implementation provides, and how they relate to the control objectives. The security target forms the basis for testing and evaluation of a product. The policy model and the target together may be referred to loosely as the *security policy*, and the process of developing a security policy and obtaining agreement on it from the system owner is the process of *requirements engineering*.

Security requirements engineering is often the most critical task of managing secure system development, and can also be the hardest. It's where 'the rubber hits the road'. It's at the intersection of the most difficult technical issues, the most acute bureaucratic power struggles, and the most determined efforts at blame avoidance.

The available methodologies have consistently lagged behind those available to the rest of the system engineering world [123]. In my view, the critical insight is that the process of generating a security policy and a security target is not essentially different from the process of producing code. Depending on the application, you can use a top-down, waterfall approach; a limited iterative approach such as the spiral model; or a continuing iterative process such as the evolutionary model. In each case, we need to build in the means to manage

risk, and have the risk assessment drive the development or evolution of the security policy.

Risk management must also continue once the system is deployed. It's rather hard to tell what a new invention will be useful for, and this applies to the dark side too: novel attacks are just as difficult to predict as anything else about the future. Phone companies spent the 1970s figuring out ways to stop phone phreaks getting free calls, but once premium-rate numbers appeared the real problem became stopping fraud. We worried about crooks hacking bank smartcards, and put in lots of back-end protection for the early electronic purses; the attacks came on pay-TV smartcards instead, while the bank fraud folks concentrated on mag-stripe fallback and on phishing. People worried about the security of credit card numbers used in transactions on the net, but it turned out that the main threat to online businesses was refunds and disputes. As they say, 'The street finds its own uses for things.' So you can't expect to get the protection requirements completely right at the first attempt. We've also seen many cases where the policy and mechanisms were set when a system was first built, and then undermined as the environment (and the product) evolved, but the protection did not.

If you're running a company, it's futile to spend a lot of money on trying to think up new attacks; that's research, and best left to university folks like me. What you do need is twofold: a mechanism for stopping your developers building systems that are vulnerable to known bugs like stack overflows and weak cryptography; and a mechanism for monitoring, and acting on, changing protection requirements.

Unlike in the previous section, we'll look at the case of evolving protection requirements first, as it is more common.

25.4.1 Managing Requirements Evolution

Most of the time, security requirements have to be tweaked for one of four reasons. First, we might need to fix a bug. Second, we may want to improve the system; as we get more experience of the kind of attacks that happen, we will want to tune some aspect of the controls. Third, we may want to deal with an evolving environment. For example, if an online ordering system that was previously limited to a handful of major suppliers is to be extended to all a firm's suppliers then the controls are likely to need review. Finally, there may be a change in the organization. Firms are continually undergoing mergers, management buyouts, business process re-engineering, you name it.

Of course, any of these could result in such a radical change that we would consider it to be a redevelopment rather than an evolution. The dividing line between the two is inevitably vague, but many evolutionary ideas carry over into one-off projects, and many systems' evolution contains occasional large leaps that are engineered in specific projects.

25.4.1.1 Bug Fixing

Most security enhancements fall into the category of bug fixes or product tuning. Fortunately, they are usually the easiest to cope with provided you have the right mechanisms for getting information about bugs, testing fixes and shipping upgrades.

If you sell software that's at all security critical — and most anything that can communicate with the outside world is potentially so — then the day will come when you hear of a vulnerability or even 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). That doesn't work any more: public expectations are higher now. With mass market products you can expect press publicity; even with more specialized products there is a risk of it. Expectations are backed by laws. By 2007, most U.S. states had security breach notification laws, obliging firms to notify attacks to all individuals whose privacy could have thereby been compromised, and the European Union had such a law in the pipeline too. Now it's not inevitable that a vulnerability report will trigger such a law — if you're lucky the alarm won't be raised because of an exploit, but from one of your customers' technical staff noticing a problem and reporting it to stop it becoming an exploit. But, either way, you need a plan to deal with it. This will have four components: monitoring, repair, distribution and reassurance.

First, you need to be sure that you learn of vulnerabilities as soon as you can — and preferably no later than the press (or the bad guys) do. 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. The idea of vulnerability markets was first suggested by Jean Camp and Catherine Wolfram in 2000 [256]; two firms, iDefense and Tipping Point, are now openly buying vulnerabilities, so the market actually exists. Unfortunately, the prices are not published and they only trade in bugs in the major platforms; but this shouldn't stop you setting up a reward scheme. Then, however you get the bug reports in, you then need to make someone responsible for monitoring them, and also for reading relevant mailing lists, such as bugtraq [239].

Second, you need to be able to respond appropriately. In organizations such as banks with time-critical processing requirements, it's normal for one member of each product team to be 'on call' with a pager in case something goes wrong at three in the morning and needs fixing at once. This might be excessive for a small software company, but you should still know the home phone numbers of everyone who might be needed urgently; see to it that there's more than one person with each critical skill; and have supporting procedures. For example, emergency bug fixes must be run through the full testing process

as soon as possible, and the documentation's got to be upgraded too. This is critical for evolutionary security improvement, but too often ignored: where the bug fix changes the requirements, you need to fix their documentation too (and perhaps your threat model, and even top level risk management paperwork).

Third, you need to be able to distribute the patch to your customers rapidly. So it needs to be planned in advance. The details will vary depending on your product: if you only have a few dozen customers running your code on servers at data centers that are staffed 24 x 7, then it may be very easy, but if it involves patching millions of copies of consumer software then a lot of care is needed. It may seem simple enough to get your customers to visit your website once a day and check for upgrades, but this just doesn't work. There is a serious tension between the desire to patch quickly to forestall attacks, and the desire to delay so as to test the patch properly [135]: pioneers who apply patches quickly end up discovering problems that break their systems, but laggards are more vulnerable to attack. There are also two quite different tensions between the vendor and the customer. First, the vendor would usually like to patch more quickly than the customer at the operational level, and second, the customer would probably want the threat of eventual breach disclosure, because without it the vendor would be less likely to issue patches at all [89].

Considerations like these led Microsoft to 'patch Tuesday', the policy of releasing a whole set of patches on the first Tuesday of every month. A monthly cycle seems a reasonable compromise between security, dependability and manageability. Individual customers usually patch automatically, while firms know to schedule testing of their enterprise systems for then so they can patch as quickly as possible thereafter. Most recent malware exploits have targeted vulnerabilities that were already patched — the bad guys reverse-engineer the patches to find the vulns and then get the machines that were patched late or not at all. So once a patch cycle is set up, it can become a treadmill. There are also quite a lot of details you have to get right to set up such a scheme — coping with traffic volumes, giving customers adequate legal notification of the effects of changes, and securing the mechanism against abuse. But as time goes on, I expect that more and more firms will have to introduce patch cycle management, as software gets more complex (and thus buggy), and as it spreads into more and more devices.

Finally, 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 switchboard operator as you struggle madly to fix the bug. Have a set of press releases for incidents of varying severity ready to roll, so that you only have to pick the right one and fill in the details. The release can then ship as soon as the first (or perhaps the second) journalist calls.

25.4.1.2 Control Tuning and Corporate Governance

The main process by which organizations like banks develop their bookkeeping systems and their other internal controls is by tuning them in the light of experience. A bank with 25,000 staff might be sacking about one staff member a day for petty theft or embezzlement, and it's traditionally the internal audit department that will review the loss reports and recommend system changes to reduce the incidence of the most common scams. I gave some examples in section 10.2.3.

It is important for the security engineer to have some knowledge of internal controls. There is a shortage of books on this subject: audit is largely learned on the job, but know-how is also transmitted by courses and through accounting standards documents. There is a survey of internal audit standards by Janet Colbert and Paul Bowen [312]; the most influential is the Risk Management Framework from the *Committee of Sponsoring Organizations* (COSO), a group of U.S. accounting and auditing bodies [318]. This is the yardstick by which your system will be judged if it's used in the U.S. public sector or by companies quoted on U.S. equity markets. The standard reference on COSO is a book by Steven Root [1082], who also tells the story of how U.S. accounting standards evolved in the years prior to Enron.

The COSO model is targeted not just on internal control but on the reliability of financial reporting and compliance with laws and regulations. Its basic process is an evolutionary cycle: in a given environment, you assess the risks, design controls, monitor their performance, and then go round the loop again. COSO emphasizes soft aspects of corporate culture more than hard system design issues so may be seen as a guide to managing and documenting the process by which your system evolves. However, its core consists of the internal control procedures whereby senior management check that their control policies are being implemented and achieving their objectives, and modify them if not.

It is also worthwhile for the security engineer to learn about the more specialized information systems audit function. The IS auditor should not have line responsibility for security but should monitor how it's done, look into things that are substandard or appear suspicious, and suggest improvements. Much of the technical material is common with security engineering; if you have read and understood this book so far, you should be able to get well over 50% on the Certified Information Systems Auditor (CISA) exam (details are at [639]). The Information Systems Audit and Control Association, which administers CISA, has a refinement of COSO known as the *Control Objectives for Information and related Technology* (CobiT) which is more attuned to IT needs, more international and more accessible than COSO (it can be downloaded from [638]). It covers much more than engineering requirements, as issues such as personnel management, change control and project management are

also the internal auditor's staples. (The working security engineer needs some familiarity with this material too.)

These general standards are necessarily rather vague. They provide the engineer with a context and a top-level checklist, but rarely with any clear guidance on specific measures. For example, CobiT 5.19 states: 'Regarding malicious software, such as computer viruses or trojan horses, management should establish a framework of adequate preventative, detective and corrective control measures'. More concrete standards are often developed to apply such general principles to specific application areas. For example, when I was working in banking security in the 1980's, I relied on guidelines from the Bank for International Settlements [113]. Where such standards exist, they are often the ultimate fulcrum of security evolutionary activity.

Further important drivers will be your auditors' interpretation of specific accounting laws, such as Sarbanes-Oxley for U.S. publicly-listed companies, Gramm-Leach-Bliley for U.S. financial-sector firms, and HIPAA for U.S. healthcare providers. Sarbanes-Oxley, for example, requires information security measures to be documented, which fits well enough with CobiT. In Europe some comparable pressure comes from privacy laws. Recently the spread of security-breach disclosure laws has created much greater sensitivity about the protection of personal information about customers (and in particular about their credit card numbers, dates of birth or any other data that could be used in financial fraud). There are now breach disclosure laws in most U.S. states, and they're in the pipeline for Europe. If you have to disclose that your systems have been hacked and millions of customer credit card numbers compromised, you can expect to be sued and to see your stock fall by several percent. If it happens to you more than once, you can expect to lose customers: customer churn might only be 2% after one notified breach, but 30% after two and near 100% after three [1356]. The silver lining in this cloud is that, for the first time ever, information security has become a CEO issue; this means that you'll occasionally have access to the boss to make your case for investment.

It's also a good idea to have good channels of communication to your internal audit department. But it's not a good idea to rely on them completely for feedback. Usually the people who know most about how to break the system are the ordinary staff who actually use it. Ask them.

25.4.1.3 Evolving Environments and the Tragedy of the Commons

I've discussed a lot of systems that broke after their environment changed and appropriate changes to the protection mechanisms were skimped, avoided or forgotten. Card-and-PIN technology that worked fine with ATMs became vulnerable to false-terminal attacks when used with retail point-of-sale terminals; smartcards that were perfectly good for managing credit card numbers and PINs were inadequate to keep out the pay-TV pirates; and even very

basic mechanisms such as authentication protocols had to be redesigned once they started to be used by systems where the main threat was internal rather than external. Military environments evolve particularly rapidly in wartime, as attack and defence coevolve; R. V. Jones attributes much of the UK's relative success in electronic warfare in World War 2 to the fact that the Germans used a rigid top-down development methodology, which resulted in beautifully engineered equipment that was always six months too late [670].

Changes in the application aren't the only problem. An operating system upgrade may introduce a whole new set of bugs into the underlying platform. Changes of scale as businesses go online can alter the cost-benefit equation, as can the fact that many system users may be in foreign jurisdictions with ineffective computer crime laws (or none at all). Also, attacks that were known by experts for many years to be possible, but which were ignored because they didn't happen in practice, can suddenly start to happen — good examples being phishing and the distributed denial-of-service attack.

Where you own the system, things are merely difficult. You manage risk by ensuring that someone in the organization has responsibility for maintaining its security rating; this may involve an annual review driven by your internal audit bureaucracy, or be an aspect of change control. Maintaining organizational memory is hard, thanks to the high turnover of both IT and security staff which we discussed in section 25.2.3.4 above. Keeping developer teams interested and up-to-date in security can be a hard problem, and I'll return to it towards the end of this chapter.

That's tough enough, but where many of the really intractable problems arise is where no-one owns the system at all. The responsibility for established standards, such as how ATMs check PINs, is diffuse. In that case, the company which developed most of the standards (IBM) lost its leading industry role and its successor, Microsoft, is not interested in the ATM market. Cryptographic equipment is sold by many specialist firms; although VISA used to certify equipment, they stopped in about 1990 and Mastercard never got into that business. The EMV consortium got going later, but for a while there was no one person or company in charge, and responsibility for standards outside the world of EMV smartcards is not always clear. So each player — equipment maker or bank — had a motive to push the boundaries just a little bit further, in the hope that when eventually something did go wrong, it would happen to somebody else.

This problem is familiar to economists, who call it the *tragedy of the commons* [806]. If a hundred peasants are allowed to graze their sheep on the village common, where the grass is finite, then whenever another sheep is added its owner gets almost the full benefit, while the other ninety-nine suffer only a very small disadvantage from the decline in the quality of the grazing. So they aren't motivated to object, but rather to add another sheep of their own and get

as much of the declining resource as they can. The result is a dustbowl. In the world of agriculture, this problem is tackled by community mechanisms, such as getting the parish council set up a grazing control committee. One of the challenges facing the world of computer security is to devise the appropriate mix of technical and organizational mechanisms to achieve the same sort of result that was already achieved by a tenth-century Saxon village, only on the much larger and more diffuse scale of the Internet.

25.4.1.4 Organizational Change

Organizational issues are not just a contributory factor in security failure, as with the loss of organizational memory and the lack of community mechanisms for monitoring changing threat environments. They can often be a primary cause.

The early 1990s saw a management fashion for *business process re-engineering* which often meant using changes in business computer systems to compel changes in the way people worked. There have been some well-documented cases in which poorly designed systems interacted with resentful staff to cause a disaster.

Perhaps the best known case is that of the London Ambulance Service. They had a manual system in which incoming emergency calls were written on forms and sent by conveyer belt to three controllers who allocated vehicles and passed the form to a radio dispatcher. Industrial relations were poor, and there was pressure to cut costs: managers got the idea of solving all these problems by automating. Lots of things went wrong, and as the system was phased in it became clear that it couldn't cope with established working practices, such as crew taking the 'wrong' ambulance (staff had favorite vehicles with long-serving staff getting the better ones). Managers didn't want to know and forced the system into use on the 26th October 1992 by reorganizing the room so that controllers and dispatchers had to use terminals rather than paper.

The result was meltdown. A number of positive feedback loops became established which caused the system progressively to lose track of vehicles. Exception messages built up, scrolled off screen and were lost; incidents were held as allocators searched for vehicles; as the response time stretched, callbacks from patients increased (the average ring time for emergency callers went over ten minutes); as congestion increased, the ambulance crews got frustrated, pressed the wrong buttons on their new data terminals, couldn't get a result, tried calling on the voice channel, and increased the congestion; as more and more crews fell back on the methods they understood, they took the wrong vehicles even more; many vehicles were sent to an emergency, or none; and finally the whole service collapsed. It's reckoned that perhaps twenty people died as a direct result of not getting paramedic assistance in time. By

the afternoon it was the major news item, the government intervened, and on the following day the system was switched back to semi-manual operation.

This is only one of many such disasters, but it's particularly valuable to the engineer as it was extremely well documented by the resulting public enquiry [1205]. In my own professional experience I've seen cases where similar attempts to force through changes in corporate culture by replacing computer systems have so undermined morale that honesty became a worry. (Much of my consulting work has had to do with environments placed under stress by corporate reorganization or even national political crises.)

In extreme cases, a step change in the environment brought on by a savage corporate restructuring will be more like a one-off project than an evolutionary change. There will often be some useful base to fall back on, such as an understanding of external threats; but the internal threat environment may become radically different. This is particularly clear in banking. Fifteen years ago, bank branches were run by avuncular managers and staffed by respectable middle-aged ladies who expected to spend their entire working lives there. Nowadays the managers have been replaced by product sales specialists and the teller staff are youngsters on near-minimum wages who turn over every year or so. It's simply not the same business.

25.4.2 Managing Project Requirements

This brings us to the much harder problem of how to do security requirements engineering for a one-off project. A common example in the 1990s was building an e-commerce application from scratch, whether for a start-up or for an established business desperate to create new online distribution channels. A common example in the next ten years might be an established application going online as critical components acquire the ability to communicate: examples I've discussed in Part II include postage meters, burglar alarms and door locks. There will be many more.

Building things from scratch is an accident-prone business; many large development projects have crashed and burned. The problems appear to be very 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 the general software engineering literature, and indeed the broader engineering literature. For example, according to Herb Simon's classic model of engineering design, you start off from a goal, desiderata, a utility function and budget constraints; then work through a design tree of decisions until you find a design that's 'good enough'; then iterate the search until you find the best design or run out of time [1153].

At least as important as guidance on 'how to do it' are warnings about how not to. The classic study of large software project disasters was written by Bill Curtis, Herb Krasner, and Neil Iscoe [339]: 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. They suggested that the solution was to find an 'exceptional designer' with a deep understanding of the problem who would assume overall responsibility.

The millennium bug gives another useful data point, which many writers on software engineering still have to digest. If one accepts that many large commercial and government systems needed extensive repair work, and the conventional wisdom that a significant proportion of large development projects are late or never delivered at all, then the prediction of widespread chaos at the end of 1999 was inescapable. It didn't happen. Certainly, the risks to the systems used by small and medium sized firms were overstated [50]; nevertheless, the systems of some large firms whose operations are critical to the economy, such as banks and utilities, did need substantial fixing. But despite the conventional wisdom, there have been no reports of significant organizations going belly-up. 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'.

So the requirements engineer needs to acquire a deep knowledge of the application as well as of the people who might attack it and the kind of tools they might use. If domain experts are available, well and good. When interviewing them try to distinguish things that are done for a purpose from those which are just 'how things are done around here'. Probe constantly for the reasons why things are done, and be sensitive to after-the-fact rationalizations. Focus particularly on the things that are going to change. For example, if dealing with customer complaints depends on whether the customer is presentable or not, and your job is to take this business online, then ask the experts what alternative controls might work in a world where it's much harder to tell a customer's age, sex and social class. (This should probably have been done round about the time of the civil rights movement in the 1960's, but better late than never.)

When tackling a new application, dig into its history. I've tried to do that throughout this book, and bring out the way in which problems repeat. To find out what electronic banking will be like in the twenty-first century, it's a good idea to know what it was like in the nineteenth; human nature doesn't change much. Historical parallels will also make it much easier for you to sell your proposal to your client's board of directors.

An influential recent publication is a book on threat modelling by Frank Swiderski and Window Snyder [1236]. This describes the methodology adopted by Microsoft following its big security push. The basic idea is that you list the assets you're trying to protect (ability to do transactions, access to classified data, whatever) and 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 through the system, from one module to another. You try to figure out what the trust levels are and where the attack paths 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. What you're likely to find is that in order to make the complexity manageable, you have to impose a security policy — as an abstraction, or even just a rule of thumb that enables you to focus on the exceptions. Hopefully the security policies discussed in Part II of this book will give you some guidance and inspiration.

You will likely find that a security requirements specification for a new project requires iteration, so it's more likely to be spiral model than waterfall model. In the first pass, you'll describe the new application and how it differs from any existing applications for which loss histories are available, set out a model of the risks as you perceive them, and draft a security policy (I'll have more to say on risk analysis and management in the next section). In the second pass, you might get comments from your client's middle management and internal auditors, while meantime you scour the literature for useful checklist items and ideas you can recycle. The outcome of this will be a revised, more quantitative threat model; a security policy; and a security target which sketches how the policy will be implemented in real life. It will also set out how a system can be evaluated against these criteria. In the third pass, the documentation will circulate to a wider group of people including your client's senior management, external auditors, insurers and perhaps an external evaluator. The Microsoft model does indeed support iteration, and its authors advise that it be used even at the level of features when a product is undergoing evolution.

25.4.3 Parallelizing the Process

Often there isn't an expert to hand, as when something is being done for the first time, or when we're building a competitor to a proprietary system whose owners won't share their loss history with us. An interesting question here is how to brainstorm a specification by just trying to think of all the things that could go wrong. The common industry practice is to hire a single consulting firm to draw up a security target; but the experience described in 12.2.3 suggested that using several experts in parallel would be better. People with backgrounds in crypto, access control, internal audit and so on will see a problem from different angles. There is also an interesting analogy with the world of software testing where it is more cost efficient to test in parallel rather than series: each tester has a different focus in the testing space

and will find some subset of flaws faster than the others. (I'll look at a more quantitative model of this in the next chapter.)

This all motivated me to carry out an experiment in 1999 to see if a high-quality requirements specification could be assembled quickly by getting a lot of different people to contribute drafts. The idea was that most of the possible attacks would be considered in at least one of them. So in one of our University exam questions, I asked what would be a suitable security policy for a company planning to bid for the licence for a public lottery.

The results are described in [49]. The model answer was that attackers, possibly in cahoots with insiders, would try to place bets once the result of the draw is known, whether by altering bet records or forging tickets; or place bets without paying for them; or operate bogus vending stations which would pay small claims but disappear if a client won a big prize. The security policy that follows logically from this is that bets should be registered online with a server which is secured prior to the draw, both against tampering and against the extraction of sufficient information to forge a winning ticket; that there should be credit limits for genuine vendors; and that there should be ways of identifying bogus vendors.

Valuable and original contributions from the students came at a number of levels, including policy goal statements, discussions of particular attacks, and arguments about the merits of particular protection mechanisms. At the policy level, there were a number of shrewd observations on the need to maintain public confidence and the threat from senior managers in the operating company. At the level of technical detail, one student discussed threats from refund mechanisms, while another looked at attacks on secure time mechanisms and observed that the use of the radio time signal in lottery terminals would be vulnerable to jamming (this turned out to be a real vulnerability in one existing lottery).

The students also came up with quite a number of routine checklist items of the kind that designers often overlook — such as 'tickets must be associated with a particular draw'. This might seem obvious, but a protocol design which used a purchase date, ticket serial number and server-supplied random challenge as input to a MAC computation might appear plausible to a superficial inspection. Experienced designers appreciate the value of such checklists[3].

The lesson to be learned from this case study is that requirements engineering, like software testing, is susceptible to a useful degree of parallelization. So if your target system is something novel, then instead of paying a single consultant to think about it for twenty days, consider getting fifteen people

[3]They did miss one lottery fraud that actually happened — when a couple won about half a million dollars, the employee to whom they presented the winning ticket claimed it himself and absconded overseas with the cash. The lottery compounded the failure by contesting the claim in court, and losing [765]. One might have thought their auditors and lawyers could have advised them better.

with diverse backgrounds to think about it for a day each. Have brainstorming sessions with a variety of staff, from your client company, its suppliers, and industry bodies.

But beware — people will naturally think of the problems that might make their own lives difficult, and will care less about things that inconvenience others. We learned this the hard way at our university where a number of administrative systems were overseen by project boards made up largely of staff from administrative departments. This was simply because the professors were too busy doing research and teaching to bother. We ended up with systems that are convenient for a small number of administrators, and inconvenient for a much larger number of professors. When choosing the membership of your brainstorming sessions, focus groups and project boards, it's worth making some effort to match their membership to the costs and risks to which the firm is exposed. If a university has five times as many professors as clerks, you should have this proportion involved in design; and if you're designing a bank system that will be the target of attack, then don't let the marketing folks drive the design. Make sure you have plenty internal audit folks, customer helpline managers and other people whose lives will be made a misery if there's suddenly a lot more fraud.

25.5 Risk Management

That brings us to our next topic. Whether our threat model and security policy evolve or are developed in a one-off project, at their heart lie business decisions about priorities: how much to spend on protection against what. This is risk management, and it must be done within a broader framework of managing non-IT risks.

Many firms sell methodologies for this. Some come in the form of do-it-yourself PC software, while others are part of a package of consultancy services. Which one you use may be determined by your client's policies; for example, if you're selling anything to the UK government you're likely to have to use a system called CRAMM. The basic purpose of such systems is to prioritise security expenditure, while at the same time providing a financial case for it to senior management.

The most common technique is to calculate the *annual loss expectancy* (ALE) for each possible loss scenario. This is the expected loss multiplied by the number of incidents expected in an average year. A typical ALE analysis for a bank's computer systems might consist of several hundred entries, including items such as we see in Figure 25.5.

Note that accurate figures are likely to be available for common losses (such as 'teller takes cash'), while for the uncommon, high-risk losses such as a large funds transfer fraud, the incidence is largely guesswork.

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 25.5: Computing annualized loss expectancy (ALE)

ALEs have long been standardized by NIST as the technique to use in U.S. government procurements [1005], and the audit culture pos-Enron is spreading them everywhere. But in real life, the process of producing such a table is all too often just iterative guesswork. The consultant lists all the threats he can think of, attaches notional probabilities, works out the ALEs, adds them all up, and gets a ludicrous result: perhaps the bank's ALE exceeds its income. He then tweaks the total down to whatever will justify the largest security budget he thinks the board of directors will stand (or which his client, the chief internal auditor, has told him is politically possible). The loss probabilities are then massaged to give the right answer. (Great invention, the spreadsheet.) I'm sorry if this sounds a bit cynical; but it's what seems to happen more often than not. The point is, ALEs may be of some value, but they should not be elevated into a religion.

Insurance can be of some help in managing large but unlikely risks. But the insurance business is not completely scientific either. For years the annual premium for bankers' blanket bond insurance, which covered both computer crime and employee disloyalty, was 0.5% of the sum insured. This represented pure profit for Lloyds, which wrote the policies; then there was a large claim, and the premium doubled to 1% per annum. Such policies may have a deductible of between $50,000,000 and $10,000,000 per incident, and so they only remove a small number of very large risks from the equation. As for nonbanks, business insurance used to cover computer risks up until about 1998, when underwriters started excluding it because of the worries about the millennium bug. When it became available again in 2001, the premiums were much higher than before. They have since come down, but insurance is historically a cyclical industry: companies compete with low premiums whenever rising stock-market values boost the value of their investment portfolios, and jack up prices when markets fall. Around 2000, the end of the dotcom boom created a downswing that coincided with the millennium bug scare. Even now that markets are returning to normal, some kinds of cover are still limited because of correlated risks. Insurers are happy to cover events of known probability and local effect, but where the risk are unknown and the effect could be global (for example, a worm that took down the Internet for several days) markets tend to fail [200].

Anyway, a very important reason for large companies to take out computer crime cover — and do many other things — is due diligence. The risks that are being tackled may seem on the surface to be operational risks but are actually legal, regulatory and PR risks. Often they are managed by 'following the herd' — being just another one of the millions of gnu on the African veld, to reuse our metaphor for Internet security. This is one reason why information security is such a fashion-driven business. During the mid 1980's, hackers were what everyone talked about (even if their numbers were tiny), and firms selling dial-back modems did a roaring business. From the late 80's, viruses took over the corporate imagination, and antivirus software made some people rich. In the mid-1990s, the firewall became the star product. The late 1990s saw a frenzy over PKI. These are the products that CFOs see on TV and in the financial press. Amidst all this hoopla, the security professional must retain a healthy scepticism and strive to understand what the real threats are.

25.6 Managing the Team

It's now time to pull all the threads together and discuss how you manage a team of developers who're trying to produce — or enhance — a product that does useful work, while at the same time not introduce vulnerabilities that turn out to be show-stoppers. For this, you need to build a team with the right culture, the right mix of skills, and above all the right incentives.

One of the hardest issues to get right is the balance between having everyone on the team responsible for securing their own code, and having a security guru on whom everyone relies.

There is a growing consensus that, in order to get high-quality software, you have to make programmers test their own code and fix their own bugs. An important case history is how Microsoft beat IBM in the PC operating systems market. During the late 1980s, Microsoft and IBM collaborated on OS/2, but the partnership broke up in 1990 after which Microsoft developed Windows into the dominant position it has today. An important reason was Microsoft's impatience at IBM's development process, which was slow and produced bloated code. IBM followed the waterfall model, being careful to specify modules properly before they were written; it also divided up its developers into analysts, programmers and testers. This created a moral hazard, especially when teams were working under pressure: programmers would write a lot of shoddy code and 'throw it over the wall' for the testers to fix up. As a result, the code base was often so broken that it wouldn't run, so it wasn't possible to use regression tests to start pulling the bugs out. Microsoft took the view that they did not have programmers or testers, only developers: each programmer was responsible for fixing his own software, and a lot of attention

was paid to ensuring that the software would build every night for testing. One of the consequences was that people who wrote buggy software ended up spending most of their time hunting and fixing bugs in their code, so more of the code base ended up being written by the more careful programmers. Another consequence was Microsoft won the battle to rule the world of 32-bit operating systems; their better development methodology let them take a $100 bn market from IBM. The story is told by Steve Maguire in [829].

When engineering systems to be secure — as opposed to merely on time and on budget — you certainly need to educate all your developers to the point that they understand the basics, such as the need to sanitise input to prevent overflows, and the need to lock variables to prevent race conditions. But there is a strong case for some extra specialised knowledge and input, especially at the testing phase. You can lecture programmers about stack overflows until you're blue in the face, and they'll dutifully stare at their code until they've convinced themselves that it doesn't have any. It's only when someone knowledgeable runs a fuzzing tool on it, and it breaks, that the message really gets through. So how do you square the need for specialists, in order to acquire and maintain know-how and tools, with the need for developers to test their own code, in order to ensure that most of your code is written by the most careful coders?

A second, and equally hard, problem is how you maintain the security of a system in the face of incremental development. You might be lucky to start off with a product that's got a clean architecture tied to a well-understood protection profile, in which case the problem may be maintaining its coherence as repeated bug fixes and feature enhancements add complexity. The case history of cryptographic processors which I described in the chapter on API security shows how you can suddenly pass a point at which feature interactions fatally undermine your security architecture. An even worse (and more likely) case is where you start off with a product that's already messy, and you simultaneously have to fix security problems and provide new features. The former require the product to be made simpler, while the latter are simultaneously making it more complex.

A large part of the answer lies in how you recruit, train and manage your team of developers, and create a culture in which they get progressively better at writing secure code. Some useful insights come from the Capability Maturity Model developed by the Software Engineering Institute at Carnegie-Mellon University [873]. Although this is aimed at dependability and at delivering code on time rather than specifically at security, their research shows that capability is something that develops in groups; it's not just a purely individual thing. This is especially important if your job is to write (say) the software for a new mobile phone and you've got 20 people only one of whom has any real security background.

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. Let's think first of all the stuff you need to keep track of to manage the development of secure code.

First, there are some things that everybody should know. Everyone must understand that security software and software security aren't the same thing; and that just about any application can have vulnerabilities. Every developer has to know about the bugs that most commonly lead to vulnerabilities — stack overflows, integer overflows, race conditions and off-by-one errors. They should even understand the more exotic stuff like double frees. Personally I'd ask questions about these topics when recruiting, to filter out the clueless. If you're stuck with an existing team then you just have to train them — get them to read Gary McGraw's book 'Software Security — Building Security In' [858], and Michael Howard and David LeBlanc's 'Writing Secure Code' [627]. (Getting them to read this book too is unlikely to do them any harm, though I say it myself.) Everyone on your team should also know about the specific problems relevant to your application. if you're developing web applications, they have to know about cross-site attacks; if you're doing an accounting system, they need to know about COSO and internal controls.

Second, you need to think hard about the tool support your team will need. If you want to prevent most stack overflows being written, you'll need static analysis tools — and these had better be tools that you can maintain and extend yourself. Bugs tend to be correlated: when you find one, you'd better look for similar ones elsewhere in your code base. Indeed, one of the big improvements Microsoft's made in its development process since Bill's security blitz is that when a bug is found, they update their tools to find all similar ones. You'd better be able to do this too. You also need fuzzers so you can check code modules for vulnerability to overflows of various kinds; these are often not obvious to visual inspection, and now that the bad guys have automated means of finding them, you need the same.

Third, you need to think about the libraries you'll use. Professional development teams avoid a large number of the problems described in this book by using libraries. You avoid crypto problems — timing attacks, weak keys, zero keys — by using good libraries, which you'll probably buy in. You avoid many buffer overflows by using your own standard libraries for I/O and for those functions that your chosen programming language does badly. Your libraries should enforce a type system, so that normal code just can't process wicked input.

Fourth, your tools and libraries have to support your architecture. The really 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. You need to use, and build on, whatever structures your programming language provides so that you can spot API violations using types.

This is already more than most developers could cope with individually. So how do you manage the inevitable specialisation? One approach is inspired by Fred Brooks' famous book, 'The Mythical Man-Month', in which he describes the lessons learned from developing the world's first large software product, the operating system for the IBM S/360 mainframe [231]. He describes the 'chief programmer team', a concept evolved by his colleague Harlan Mills, in which a chief programmer — a highly productive coder — is supported by a number of other staff including a toolsmith, a tester and a language lawyer. Modern development teams don't quite fit this vision, as there will be a number of developers, but a variant of it makes sense if you're trying to develop secure code.

Assume that there will be an architect — perhaps the lead developer, perhaps a specialist — who will act, as Brooks puts it, as the agent, the approver and the advocate for the user. Assume that you'll also have a toolsmith, a tester and a language lawyer. One of these should be your security guru. If the tough issue is evolving a security policy, or even just tidying up a messy product and giving it the logical coherence needed for customers to understand it and use it safely, then the security guy should probably be the architect. If the tough issue is pulling out lots of implementation errors that have crept into a legacy product over years — as with Microsoft's security jihad on buffer overflows in Windows XP — then it should probably be the tester, at least in the beginning. If the task then changes to one of evolving the static analysis tools so that these vulnerabilities don't creep back, the security mantle will naturally pass to the toolsmith. And if the APIs are everything, and there is constant pressure to add extra features that will break them — as in the crypto processor case history I discussed in the chapter on API security — then the language lawyer will have to pick up the burden of ensuring that the APIs retain whatever type-safety and other properties are required to prevent attacks.

So far so good. However, it's not enough to have the skills: you need to get people to work together. There are many ways to do this. One possibility is the 'bug lunch' where developers are encouraged to discuss the latest subtle errors that they managed to find and remove. Whatever format you use, the critical factor is to create a culture in which people are open about stuff that broke and got fixed. 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. An example of good practice is in air traffic control, where 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 developers comfortable with sharing their experiences with each other, including their errors.

Another very useful team-building exercise is the adoption of a standard style. One of the chronic problems with poorly-managed teams is that the codebase is in a chaotic mixture of styles, with everybody doing his own thing. The result is that when a programmer checks out some code to work on it, he may well spend half an hour formatting it and tweaking it into his own style. For efficiency reasons alone, you want to stop this. However, if your goal is to write secure code, there's another reason. When you find a bug, you want to know whether it was a design error or an implementation error. If you have no idea what the programmer who wrote it was thinking about, this can be hard. So it's important to have comments in the code which tell what the programmer thought he was doing. But teams can easily fight about the 'right' quantity and style of comments: in the OS/2 saga, IBM used a lot more than Microsoft did, so the IBM folks saw the guys from Redmond as a bunch of hackers, and they responded by disdaining the men from Armonk as a bunch of bureaucrats. So for goodness' sake sit everyone down and let them spend an afternoon hammering out what your house style will be. Provided there's enough for 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 far better team-building activity than spending the afternoon paintballing.

25.7 Summary

Managing a project to build, or enhance, a system that has to be secure is a hard problem. This used to be thought of as 'security software' — producing a product such as an antivirus monitor or encryption program using expert help. The reality nowadays is often that you're writing a system that has to do real work — a web application, for example, or a gadget that listens to network traffic — and you want to keep out any vulnerabilities that would make it a target for attack. In other words, you want software security — and that isn't the same as security software.

Understanding the requirements is often the hardest part of the whole process. Like developing the system itself, security requirements engineering can involve a one-off project; it can be a limited iterative process; or it can be a matter of continuous evolution. Evolution is becoming the commonest as systems get larger and longer-lived, whether as packaged software, online

services or gadgets. Security requirements are complicated by changes of scale, of business structures and — above all — of the environment, where the changes might be in the platform you use, the legal environment you work in, or the threats you face. Systems are fielded, get popular, and then get attacked.

Writing secure code has to be seen in this context: the big problem is to know 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, keeping them up to date with attacks, backing them up with expertise in the right places which they'll actually use, reinforcing this with the right tools and language conventions, 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 on our field. However, they tend to receive little attention because they lie at the boundaries with software engineering, applied psychology, economics and management. Each of these interfaces appears to be a potentially productive area of research. Security economics in particular has made great strides in the last few years, and people are starting to work on psychology. There is a thriving research community in decision science, where behavioural economists, psychologists and marketing folks look at why people really take the decisions they do; this field is ripe for mining by security researchers.

Yet we have all too little work on how these disciplines can be applied to organisations. For a start, it would be useful if someone were to collect a library of case histories of security failures caused by unsatisfactory incentives in organisations, such as [587, 662]. What might follow given a decent empirical foundation? For example, if organisational theory is where microeconomic analysis is applied to organisations, with a little psychology thrown in, then what would be the shape of an organisational theory applied to security? 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? Even more basically, how can you design an organization that is '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: how can you do corporate governance so that the firm doesn't fall prey to similar problems?

Further Reading

Managing the development of information systems has a large, diffuse and multidisciplinary literature. There are classics which everyone should read, such as Fred Brooks' 'Mythical Man Month' [231] and Nancy Leveson's 'Safeware' [786]. Standard textbooks on software engineering such as Roger Pressman [1041] and Hans van Vliet [1281] cover the basics of project management and requirements engineering. The economics of the software life cycle are discussed by Brooks and by Barry Boehm [199]. The Microsoft approach to managing software evolution is described by Steve McGuire [829], while their doctrine on threat modelling is discussed in a book by Frank Swiderski and Window Snyder [1236].

There are useful parallels with other engineering disciplines. An interesting book by 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 [1021]. For a survey of risk management methods and tools, see Richard Baskerville [123] or Donn Parker [1005]. Computer system failures are another necessary subject of study; a must-read fortnightly source is the `comp.risks` newsgroup of which a selection has been collated and published in print by Peter Neumann [962].

Organizational aspects are discussed at length in the business school literature, but this can be bewildering to the outsider. 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 [1028]. John Micklethwait and Adrian Wooldridge provide a critical guide to the more academic literature and draw out a number of highly relevant tensions, such as the illogicality of management gurus who tell managers to make their organizations more flexible by sacking people, while at the same time preaching the virtues of trust [882]. As for the theory of internal control, such as it is, the best book is by Steven Root, who discusses its design and evolution [1082]. The best introductory book I know to the underlying microeconomics is by Carl Shapiro and Hal Varian [1159]: the new institutional school has written much more on the theory of the firm.

Finally, the business of managing secure software development is getting more attention (at last). Microsoft's security VP Mike Nash describes the background to the big security push and the adoption of the security development lifecycle at [929]. The standard books I'd get everyone to read are Michael

Howard and David LeBlanc's 'Writing Secure Code' [627], which sets out the Microsoft approach to managing the security lifecycle, and Gary McGraw's book 'Software Security — Building Security In' [858], which is a first-class resource on what goes wrong. As I wrote, Microsoft, Symantec, EMC, Juniper Networks and SAP have just announced the establishment of an industry body, SAFEcode. to develop best practices. And about time too!