CHAPTER
22

Management Issues

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

22.1 Introduction

To this point, I've outlined a variety of security applications, techniques, and concerns. If you're a working IT manager, 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 threat assessment.

The experience of the business schools is that management training should be conducted largely through the study of case histories, stiffened with focused courses on basic topics such as law, economics, and accounting. I have followed this 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. Now we have to pull the threads together and discuss how a security engineering problem should be tackled. 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.

22.2 Managing a Security Project

The core of the security project manager's job is usually requirements engineering—figuring out what to protect and how. When doing this, it is critical to understand the trade-off between risk and reward. Security people have a distinct tendency to focus too much on the former and neglect the latter. If the client has a turnover of $10 million, profits of $1 million and theft losses of $150,000, the security consultant may make a pitch about "how to increase your profits by 15%" when often what's really in the shareholders' interests is to double the turnover to $20 million, even if this triples the losses to $450,000. Assuming the margins remain the same, the profit is now $1.85 million, an increase of 85%. The point is, 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 talk only about "tightening security." Often, it's too tight already.

22.2.1 A Tale of Three Supermarkets

My thumbnail case history to illustrate this point concerns three supermarkets. Among the large operational costs of running a supermarket are the salaries of the checkout and security staff, and the stock shrinkage due to theft. Checkout delays are also a significant source of aggravation: just cutting the number of staff isn't an option, and working them harder might mean more shrinkage. What might technology do to help?

One supermarket in South Africa decided to automate completely. All produce would carry an RF tag, so that an entire shopping cart could be scanned automatically. If this had worked, it would have killed both birds with one stone: the same RF tags could have been used to make theft very much harder. Though there was a pilot, the idea couldn't compete with barcodes. Customers had to use a special cart, which was large and ugly, and the RF tags also cost money.

Another supermarket in a European country believed that much of their losses were due to a hard core of professional thieves, and thought of building a face recognition system to alert the guards whenever one of these habitual villains came into a store. But current technology can't do that with low enough error rates to be useful. In the end, the chosen route was civil recovery. When a shoplifter is caught, then even after the local magistrates have fined him about the price of a lunch, the supermarket goes after him in the civil courts for wasted time, lost earnings, attorneys' fees and everything else they can think of; and then armed with a judgment for about the price of a car they go round to his house and seize all his furniture. So far so good. But their management got too focused on cutting losses rather than increasing sales. In the end, they started losing market share and saw their stock price slide. Diverting effort into looking for a security-based solution was probably a symptom of their decline rather than a cause, but may well have contributed to it.

The supermarket that appears to be doing best is Waitrose in England which has introduced self-service scanning. When you go into the 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 put them into your shopping bag. At the checkout, you hand back the scanner, get a printed list of everything you bought, swipe your credit card, and head for the parking lot. This might seem rather risky—but then so did the self-service supermarket back in the days when traditional grocers' shops stocked all the goods behind the counter, in fact, there are a number of subtle control mechanisms at work. Limiting the service to store cardholders not only enables the managers to exclude known shoplifters, but also helps market the store card. By having a card, you acquire a trusted status visible to any neighbors you meet while shopping; conversely, 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 a customer lingering suspiciously near the racks of hundred-pound 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.

22.2.2 Balancing Risk and Reward

The purpose of business is profit, and profit is the reward for risk. Security mechanisms can often make a significant 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 listen to their marketing, operations, and financial teams. A sound corporate risk management strategy involves much more than the operational risks from 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 make sensible decisions, and a difficult part of their task is to see to it that advisers from different disciplines work together just closely enough, but no more.

Advisers need to understand each others' roles, and work together rather than try to undermine each other; but if the company boss doesn't ask hard questions and stir the cauldron a bit, then the advisers may cosy up with each other and entrench a consensus view that steadily drifts away from reality. One of the most valuable tasks the security engineer is called on to perform (and the one needing the most diplomatic skill) is when you're brought in to contribute, as an independent outsider, to challenging this sort of groupthink. In fact, 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 colleagues who're averse to change. (This is one reason why famous consulting firms that exude an air of quality and certainty often have a competitive advantage over specialists; however, in the cases where specialists are needed, but the work is given to "suits," some fairly spectacular things can go wrong.)

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 an approach based on compliance with a set of standards (such as the Orange Book) rather than case-by-case requirements engineering. James Coyne and Norman Kluksdahl present in [208] 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 the development and operations teams; 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.

22.2.3 Organizational Issues

Although this chapter is about management, I'm not so much concerned with how you train and grade the guards as with how you build a usable system. However, you need to understand the guards (and the auditors, and the checkout staff, and ...) or you won't be able to do even a halfway passable job. Many systems fail because their designers make unrealistic assumptions about the ability, motivation, and discipline of the people who will operate it. This isn't just a matter of one-off analysis. For example, an initially low rate of fraud can cause people to get complacent and careless, until suddenly things explode. Also, an externally induced change in the organization—such as a merger or acquisition—can undermine control.

A surprising number of human frailties express themselves through the way people behave in organizations, and for which you have to make allowance in your designs.

22.2.3.1 The Complacency Cycle and the Risk Thermostat

The effects of organizational complacency are well illustrated by phone fraud in the United States. There is a seven-year cycle: in any one year there will be one of the "Baby Bells" that is getting badly hurt. This causes its managers to 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.

Some interesting and relevant work has been done on how people manage their exposure to risk. Adams studied the effect of mandatory seat belt laws, and established that these laws 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 to bring their perceived risk back up to its previous level. Adams calls this a *risk thermostat* and the model is borne out in other applications too [8,9]. The complacency cycle can be thought of as the risk thermostat's corporate manifestation. No matter how these phenomena are described, risk management remains an interactive business that involves the operation of 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.

The feedback mechanisms may provide a systemic limit on the performance of some 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 attacks"—as with the alarm systems described in Chapter 10 and the intrusion detection systems discussed in Section 18.5.3. Perhaps systems will always reach an equilibrium at which the sentries fall asleep, or real alarms are swamped by false ones, or organizational budgets are eroded to (and past) the point of danger. It is not at all obvious how to use technology to shift this equilibrium point.

Risk management may be 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 appear to be more uncertainty-averse than risk-averse. Where the odds are directly perceptible, a risk is often dealt with intuitively; but where the science is unknown or inconclusive, people are liberated to project all sorts of fears and prejudices. So perhaps the best medicine is education. Nonetheless, there are some specific things that the security engineer should either do, or avoid.

22.2.3.2 Interaction with Reliability

A significant cause of poor internal control in organizations is that the systems are insufficiently reliable, so lots of transactions are always going wrong and have to be corrected manually. A high tolerance of chaos undermines control, as it creates a high false alarm rate for many of the protection mechanisms. It also tempts staff: when they see that errors aren't spotted, they conclude that theft won't be either.

A recurring theme is the correlation between quality and security. 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 [292]. It seems that 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 business a lot of money; no one really understands what software does; if mistakes are found, the frauds should be much more obvious; and all this can be communicated to top management without embarrassment on either side.

22.2.3.3 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 in 21.2.5.3. Displacement activity is also common in the private sector. An example comes from the smartcard industry. As discussed in Section 14.7.2, the difficulty of protecting smartcards against microprobing attacks has led the industry to concentrate on securing other things instead. Even programming manuals are available only under *nondisclosure agreements* (NDA) even 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 probing attacks rather than any kind of inside information.

One organizational driver for this is an inability to deal with uncertainty. Managers prefer approaches that can be implemented by box-ticking their way down a checklist, and if an organization needs to deal with an ongoing risk, then some way must be found to keep it as a process and to stop it turning into a due-diligence checklist item. But there will be constant pressure to replace processes with checklists, as they demand less management attention and effort. I noted in Section 7.6.6 that bureaucratic guidelines for military systems had a strong tendency to displace critical thought; instead of thinking through a system's security requirements, designers just reached for their checklists. Commercial systems are not much different.

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 imprudent to enforce controls too overtly against line managers and IT staff, as this will alienate them and it's often hard to get them to manage the controls themselves. It's also hard to sell a typical company's board of directors on the need for proper defenses against insider attack, as this means impugning the integrity and reliability of the staff who report to them.

Thus, a security manager will often ask for, and get, lots of money to defend against nonexistent "evil hackers" so that she 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 that their families will be taken hostage; and requiring two signatures on transactions over a certain limit means that there are extra shoulders to take the burden when something goes wrong. But such consensus on the need for protective measures is usually lacking elsewhere.

22.2.3.4 Incompetent or Inexperienced Security Managers

The situation is bad enough even with a competent IT security manager, who has to use all sorts of guile to raise money for an activity that many of her management colleagues will tend to regard as a pure cost. In real life, the situation is 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 [384].

Things 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, the 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.

22.2.3.5 Moral Hazard

Companies often design systems so that the risk gets dumped on third parties. I mentioned in Chapter 21 that one of the attractions of digital signatures is that they can allow the risk associated with a forged signature to be transferred from the relying party to the alleged signer; thus, for example, transferring much of the risk associated with online banking from the bank to the customer. I also discussed in Chapter 9, how banks in some countries claimed that their automatic teller machines could not possibly make mistakes, so that any disputes must be the customer's fault.

In addition to the public policy aspects, and macroeconomic effects which I'll come to in Section 22.6, this has effects on the dumping company internally. It creates a *moral hazard*, by removing the incentives for people to take care, and for the company to invest in appropriate risk management techniques. Worse, a company whose policy is to deny vigorously that some particular type of fraud is possible leaves itself open to staff who defraud it knowing that a prosecution would be too embarrassing.

A slightly different kind of moral hazard is created when people who make system design decisions are unlikely to be held accountable for their actions. There are many possible causes. 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; or imminent business process re-engineering may turn loyal staff into surreptitious job seekers. In any case, when you design a secure system, it's a good idea to look at 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 process, 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. One might think that they would between them strike a balance between risk and reward, but this is very often not so. The case of the three supermarkets, mentioned above, is just one example of many. Companies may swing wildly over a period of years from being risk takers to being excessively risk averse, and (less often) back again. Adams documents in [9] 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.

Which tools and concepts can help 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 explain 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 talk about some of the economic issues, and finally discuss the things that go wrong.

22.3 Methodology

Large 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, people had started talking about the *software crisis*, although the word crisis is hardly appropriate for a starte of affairs that has now lasted (like computer insecurity) for two generations. Anyway, the term *software engineering* was proposed by Brian Randell 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 [583]. Since then, much progress has been made, though never as much as one would like.

Software engineering is about managing complexity, of which there are two kinds. One is the *incidental complexity* involved in programming using inappropriate tools, such as the assembly languages that 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. The other is 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, which I described in Section 2.7.

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

22.3.1 Top-Down Design

The classical model of system development is the *waterfall model* developed by Winston Royce in 1970 for the U.S. Air Force [653]. 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, integrate and test them as a system, then roll out the system for live operation (see Figure 22.1).

The idea is that the requirements are written in the user language, and the specification 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 you're building the right system (*validation*) and at the next two on whether you'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 downward 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 task easier by providing definite milestones to aim at; it increases 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 usually 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 tamper-resistant cryptographic processor to implement a known transaction set and pass a certain level of FIPS evaluation.

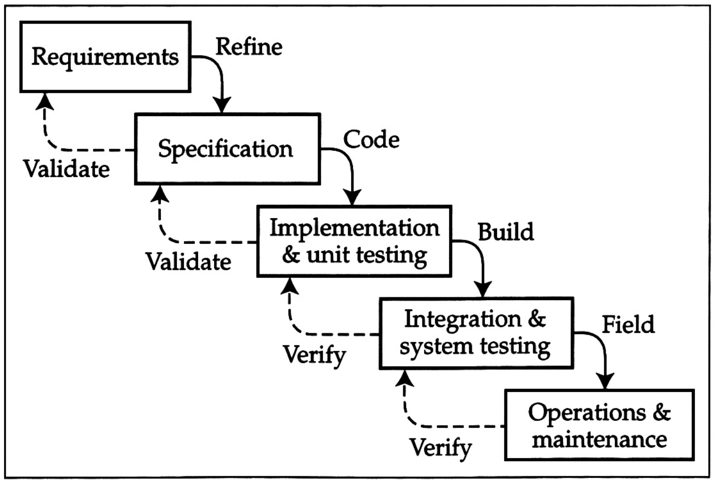

Figure 22.1 The waterfall model.

But, often, the detailed requirements aren't known in advance, so 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 feature, such as a human-computer interface, which we know from experience will involve several prototypes. (No matter how well engineered the internals of a protection system, user interface problems are to be expected, and a pilot is advisable if the business model allows it.)

22.3.2 Iterative Design

Many development projects just need an iterative approach to development, but the iteration might never terminate satisfactorily. You could build a prototype for the client who would play with it, then say, "No, I want it this way instead." Then you would build another one, come up against another objection, and never get anything fielded at all.

There are two common ways to deal with this. The first is Barry Boehm's *spiral model* in which development proceeds through a pre-agreed number of iterations. In each of these, 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 shown in Figure 22.2.

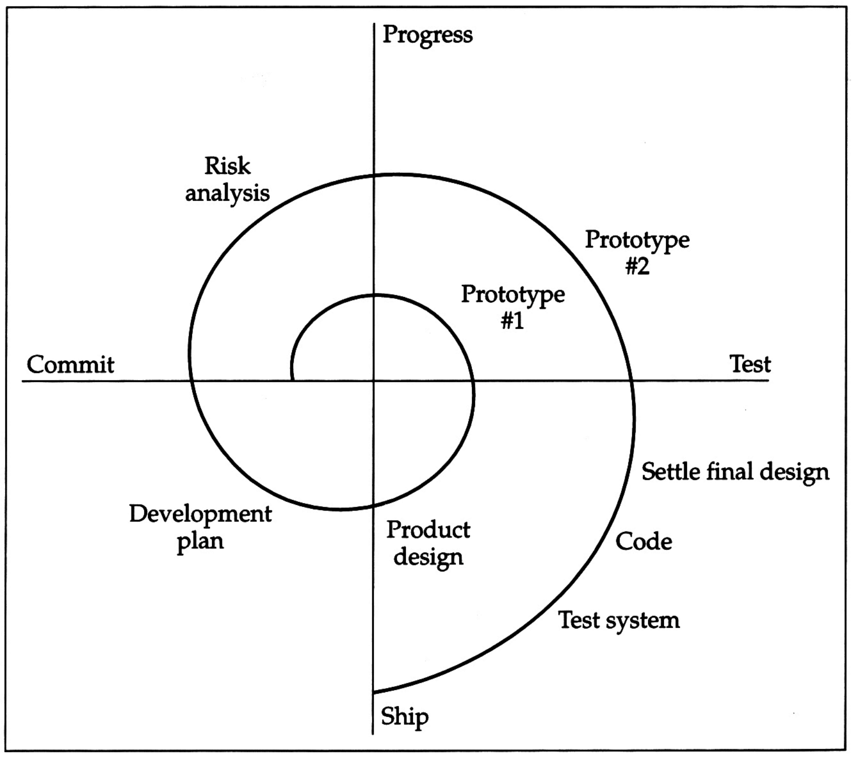

Figure 22.2 The spiral model.

The other common model is *evolutionary development*. This has become increasingly important, because it's how the packaged software industry works, and has recently been popularized under the name of "extreme programming." Unfortunately, it tends to be neglected in academic courses and books on software engineering.

As the world moves from bespoke software developed in formal projects to packages whose owners put in more and more features to appeal to ever wider markets, software products become so complex that they cannot 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 Steve Maguire, a Microsoft manager [521].) In this view of the world, products aren't the result of a project but of a process, which involves continually modifying previous versions.

The critical point 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 for this is *regression testing*. At regular intervals—perhaps once a day—all the teams working on different features of a product upgrade check in their code, and it gets compiled to a *build*, which is then tested automatically against a large set of inputs. This step checks whether things that used to work still work, and that old bugs that had been removed haven't found their way back in. 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. Thus, 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 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 reckoned that 15% of bug fixes either introduced new bugs or reintroduced old ones [7]. 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; moreover, 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 mentioned in Section 4.4.5 in the case of stack overflow. And many security flaws result from subtle bugs that 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 important. It finds functionality that has been affected by a change but that is 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.

22.3.3 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...); trace the hazards down to hardware and software components which are thereby identified as critical; identify the operator procedures which are also critical and study the various applied psychology and operations research issues; and, finally, 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 and system levels), 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 shown in Figure 22.3. 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. There, a switch failure will short out only the motor, not the battery.

Hazard elimination is useful in security engineering, too. Recall the example in the early design of SWIFT in Section 9.3.1: 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.

Once as many hazards as possible have been eliminated, 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 to conduct a *fault tree analysis:* 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; Figure 22.4 shows an example of a fault tree (or *threat tree*, as it's often called in security engineering) for fraud from automatic teller machines. Threat trees are standard practice in the U.S. Department of Defense.

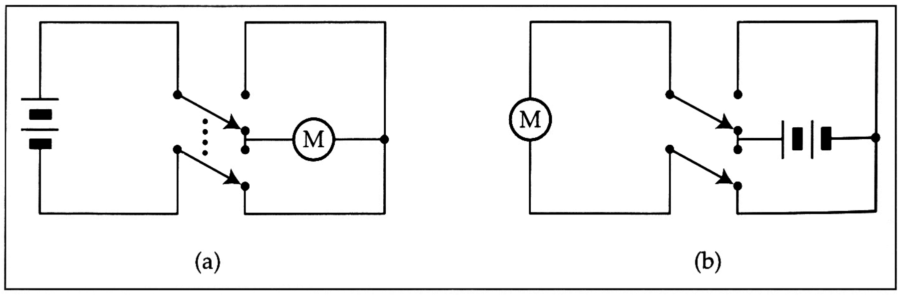

Figure 22.3 Hazard elimination in motor-reversing circuit.

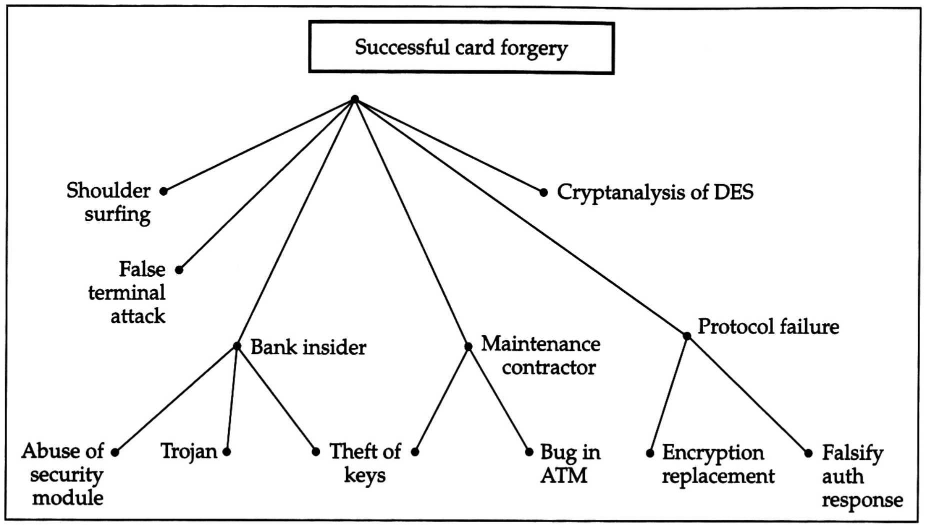

Figure 22.4 A threat tree.

Here's how a threat tree works. You start out from each undesirable outcome, and work backward by writing down each possible immediate cause. You work backward from there by adding each precursor condition, and recurse. Then, working around 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. Note that 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 other attacks, 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 have a clear picture of 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, and there are various ways to manage the resulting mass of data. For example, you 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 you can check that there are two entries in each of the relevant columns. In this way, you 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 explain 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 describe in the next chapter; and there is a lot of experience with the probability of operator error at different types of activity. 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. Designers of systems such as nuclear reactors are well aware (at least since Three Mile Island) that 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 the 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 was when a bank wanted to create a set of three master keys to link its cash machine network to VISA, and needed a terminal to drive the security module [20]. A contractor obligingly lent the bank a laptop PC, together with software that emulated the desired type of terminal. With this, the senior managers duly created the required keys and sent 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.

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 must be 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 a shipbuilder knows the strengths and weaknesses of a sailor in the Navy. The security engineer usually has no such luck. Many security failures remind me of a remark made by a ranger at Yosemite National Park about the devices provided to prevent bears from getting at campers' food supplies: that it's an impossible engineering problem because the brighter bears are smarter than the dumber campers.

There are also 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 airplane crashes have resulted from flying a craft with one of the navigation or flight control systems dysfunctional; although pilots may be intellectually aware that their display is unreliable, their reaction under pressure will be to rely on it rather than to check it against other instruments. A further failure can then be catastrophic. A security example is the ATM problem mentioned in Section 9.4.2 where a bank issued all its customers with the same PIN. In that cases, the fault got masked by the handling precautions applied to PINs, which ensured that even the bank's security and audit staff get hold of only the PIN mailer for their own personal account. Clearly, some thought is needed about how faults can remain visible and testable even when their immediate effects are masked.

The 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 the safety case 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, 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 that are subtly and maliciously wrong at the most inconvenient moment possible. This has been referred to as "programming Satan's computer," to distinguish it from the more common problem of programming Murphy's [48]. This provides an insight into one of the reasons security engineering is hard: Satan's computer is hard to test [682].

22.4 Security Requirements Engineering

In Chapter 7, 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 I introduced in Chapter 3 and sets out the attacks and failures with which the system should 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 that 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*.

Requirements engineering is the most critical task of managing secure system development, and is also 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 [77].

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 policy development or evolution.

Risk management must also continue once the system has been deployed. It's notoriously hard to tell what a new invention will be useful for; attacks are just as difficult to predict. Phone companies spent the 1970s figuring out ways to stop phone phreaks getting free calls; as it turned out, the real problem was crooks abusing the system to make calls that would be hard for the police to trace. Some people worried about crooks hacking bank smartcards, and put in lots of back-end protection for the early electronic purses; but the attacks came on pay-TV smartcards instead. Other people worried about the security of credit card numbers used in transactions on the Net, only to learn that the real threat to online businesses was not hackers but refunds and disputes. As they say, “The street finds its own uses for things.” The point is, don’t expect to get the protection requirements completely right at the first attempt. In many cases, the policy and mechanisms were set when a system was first built, then undermined as the environment (and the product) evolved, but the protection did not. There must be a mechanism for monitoring, and acting on, changing protection requirements.

In this section, unlike in the previous one, I’ll describe the case of evolving protection requirements first, as it is both more common and easier to manage.

22.4.1 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 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 of 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 as I’ll explain, many evolutionary ideas carry over into one-off projects.

22.4.1.1 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 that the right structures are in place.

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 get a report of a vulnerability or even an attack. In the old days, vendors could take months to respond with a new version of the product, or would do nothing at all but issue a warning (or even a denial). Public expectations are higher nowadays. With mass-market products, you can expect press publicity; even with more specialized products there is a risk of press coverage. In short, you had better have a plan to deal with it. This will have four components: monitoring, repair, distribution, and reassurance.

First, be sure to 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; provide an efficient way for them to report bugs. Consider offering an incentive, such as points toward their next upgrade, lottery tickets, or even cash. Then make someone responsible for monitoring these reports, and for reading relevant mailing lists, such as bugtraq [144].

Second, 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 via a pager in case something goes wrong at three in the morning and needs fixing immediately. This might be excessive for a small software company, but you should still know the home phone numbers of people whose skills 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 must be upgraded, too; this is critical for evolutionary security improvement, but too often ignored. When 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, be able to distribute the patch or other repair to your customers rapidly. This must be planned in advance. The details will vary depending on your product: if you have only a few dozen customers running your code on servers at data centers that are staffed 24/7, then it may be very easy, but if it involves patching millions of copies of consumer software a lot of care will be needed. It may seem simple enough to get your customers to visit your Web site once a day and check for upgrades, but to do this safely there are a surprising number of details you have to get right. Will the server be able to cope with the increased traffic? Have you given your customers adequate legal notification that their software might be changed under their feet? Could an opponent—such as a disgruntled former employee—hijack the mechanism and trash your entire customer base?

Finally, have a plan for dealing with the press. The last thing you need is for dozens of journalists to call and be stonewalled by your switchboard operator as you struggle madly to fix the bug. Have a set of press release templates for incidents of varying severity on file in your word processor, so that all you have to do is 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.

22.4.1.2 Control Tuning and Corporate Governance

The main process by which organizations such as banks develop their bookkeeping systems and their other internal controls is by tuning them in the light of experience. A bank with 25,000 employees might be firing about one staff member a day for petty theft or embezzlement, and, traditionally, it's 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 9.2.3.

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 available via courses and through accounting standards documents. There is a survey of internal audit standards by Janet Colbert and Paul Bowen [193]; the most influential is the Risk Management Framework from the *Committee of Sponsoring Organizations* (COSO), a group of U.S. accounting and auditing bodies [196]. 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 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, and 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, or there will be a conflict of interest: she should not be asked to assess systems that she designed or for whose operation she is responsible. Rather, she should monitor how things are 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 [408]). 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 [407]). COBIT 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 to be familiar 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 offer 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 1980s, I relied on guidelines from the Bank for International Settlements [71]. Where such standards exist, they are often the ultimate fulcrum of security evolutionary activity.

It's a good idea to have high-bandwidth channels of communication to your client's 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 staff who actually use it. Ask them.

22.4.1.3 Evolving Environments and the Tragedy of the Commons

I've described a number of systems that broke after their environment changed, and where 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 in point-of-sale applications were inadequate to keep out the pay-TV pirates; and even very basic mechanisms such as authentication protocols had to be redesigned for systems where the main threat was internal rather than external. Military environments evolve particularly rapidly in wartime, as attack and defense co-evolve; R.V. Jones attributes much of the Allies' relative success in electronic warfare in World War II to the fact that the Germans used a rigid top-down development methodology, which resulted in beautifully engineered equipment, but six months too late [424].

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 business become 'e-' 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 that were ignored because they didn't happen in practice, can suddenly start to happen—a good example being the distributed denial-of-service attack.

When 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 I discussed in Section 22.2.3.4.

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 that developed most of the standards (IBM) lost its leading industry role; its successor, Microsoft, is not interested in that market. Cryptographic equipment is sold by a number of specialist firms. Although VISA used to certify equipment, it stopped in about 1990, and Mastercard never got into that business, so there was no one person or company in charge. Each player—equipment maker or bank—had a motive to push the boundaries just a little bit further, in the expectation 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* [507]. 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. Thus, they aren't motivated to object, but rather to add another sheep of their own to 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. The cowherds in tenth-century Saxon villages were already well-enough organized to do this; one of the challenges facing us is to devise some mix of technical and organizational controls that will give us a comparable result, only on the larger scale of the Internet.

22.4.1.4 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.

In the early 1990s, management fashion was 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. It had a manual system whereby 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 senior members getting the better ones). Managers didn't want to know, and forced the new system into use on October 26, 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 that 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 10 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 often; many vehicles were sent to an emergency, or none; and, finally, the whole service collapsed. It's estimated that perhaps 20 people died as a direct result of not getting paramedic assistance in time. By the afternoon on the 26th, 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 inquiry [723]. 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 concern. (Much of my consulting work has had to do with environments placed under stress by corporate reorganization or even by 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. Today, the managers have been replaced by product sales specialists, and the teller staff are youngsters earning near-minimum wages who turn over every year or so. It's simply not the same business.

22.4.2 Managing Project Requirements

This brings us to the much more difficult problem of how to do security requirements engineering for a one-off project. The most common example might be building an e-commerce application from scratch, whether for a start-up or for an established business that wants to create new distribution channels.

Building things from scratch is an accident-prone business and there are many cases in which large software projects 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.

The classic study of large software project disasters was written by Bill Curtis, Herb Krasner, and Neil Iscoe [212]. 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 thorough 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 actually 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. But it didn't happen. Certainly, the risks to the systems used by small and medium-sized firms were overstated [37]; 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."

As a requirements engineer, you need to acquire a comprehensive 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 tasks that are done for a purpose, as opposed to those that are just "how things are done around here." Probe constantly for the reasons why things are done as they are, 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 1960s, 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.

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—from internal audit guidelines to books like this one—for useful checklist items and ideas you can recycle. The outcome of this will be a revised, more quantitative risk model, a security policy, and a security target that 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.

22.4.3 Parallelizing the Process

Often, there isn't an expert to hand, as when something is being done for the first time, or when you're building a competitor to a proprietary system whose owners won't share their loss history with you. An interesting question to ask is how to brainstorm a specification just by 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 I described in Section 10.3.3 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 in series: each tester has a different focus in the testing space, and will find some subset of flaws faster than the others. (I'll introduce a more quantitative model of this in the next chapter.)

The preceding 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. Thus, in one of our university exam questions, I asked what would be a suitable security policy for a company planning to bid for the license for a public lottery.

The results are described in [36]. The model answer was that attackers, possibly in cahoots with insiders, would try to place bets once the result of the draw was known, whether by altering bet records or forging tickets; or would place bets without paying for them; or would operate bogus vending stations that 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 that 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 that 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.

The lesson to be learned from this case study is that requirements engineering, like software testing, is susceptible to a useful degree of parallelization. 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 with diverse backgrounds to think about it for a day each, then have a consultant spend a week hammering their ideas into a single coherent document.

22.5 Risk Management

Whether a 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 should be done within the broader framework of managing non-IT risks.

A number of 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 U.K. government, you're likely to have to use a system called CRAMM. The basic purpose of such systems is to prioritize security expenditure, while at the same time provide 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 those listed in Figure 22.5. 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 22.5 Example of Annual Loss Expectancies.

ALEs have been standardized by NIST as the technique to use in U.S. government procurements [602]. But in real life, the process of producing such a table is all too often just iterative guesswork. The consultant lists all the threats she can think of, attaches notional probabilities, works out the ALEs, adds them all up, and gets a ludicrous result, such as that the bank's ALE is greater than all its non-interest income. She then tweaks the total down to the amount that will justify the largest security budget she thinks the board of directors will stand for (or which her client, the chief internal auditor, has told her 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 happens more often than not. The point is, ALEs may be of some value, but they shouldn't 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' bond insurance, which covered both computer crime and employee disloyalty, was 0.5% of the sum insured. This represented pure profit for Lloyds of London, the firm that 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 and $10,000,000 per incident, so they remove only a small number of very large risks from the equation. There is a substantial benefit in having an experienced insurance assessor check out the computer system and suggest security enhancements; but this can be arranged for much less than the six-figure sum that a typical bank might pay for coverage.

The main reason that large companies take out computer crime coverage—and do many other things—is due diligence. The risks being tackled may seem on the surface to be operational, but are often actually legal, regulatory, and PR risks. Usually, they are managed by "following the herd"—being just another one of the millions of gnu on the African veld, to reuse my metaphor for Internet security. This is one reason that computer security is such a fashion-driven business. During the mid-1980s, hackers were the main concern, and firms selling dial-back modems did a booming business. From the late 1980s, viruses took over the corporate imagination, and antivirus software made some people rich. Recently, with all the fanfare about e-business, the firewall has become the new star product. These are the threats, and the products, that are seen by corporate CEOs on TV and in the financial press. Amidst all this noise, the security professional must retain a healthy scepticism and strive to understand what the real threats are.

Ultimately, knowing what computer and communications security is appropriate in a particular application comes down to judgment. Sooner or later, the client's CEO must choose one of the options, and the best you can do is to give a competent and honest assessment of the pros and cons.

22.6 Economic Issues

Many of the problems that confront the security engineer have their origin in economics. Consultants often explain that the reason a design, for which they were responsible failed was that "the client didn't want a secure system, but just the most security I could fit on the product in one week on a budget of $10,000." It's important to realize that this isn't just management stupidity.

I first discussed network effects in Section 19.6. Networks with more users are more valuable to each user, leading to strong positive feedback and, very often, a huge first-mover advantage. This is the origin of the philosophy of, "We'll ship it on Tuesday and get it right by version 3." Although often attributed by cynics to Microsoft, this is often perfectly rational economic behavior in markets where network economics apply.

Network economics has many other effects on the security management process. Rather than using a standard, well-analyzed, and tested solution, companies often prefer a proprietary, obscure one to increase customer lock-in and to increase the problems for competitors who try to create compatible products. Where possible, they will use patented algorithms (even if these are not much good) as a means of imposing licensing conditions on manufacturers—recall from Section 20.2.5 how the DVD Content Scrambling System was used as a means of requiring manufacturers of compatible equipment to agree to a whole list of copyright protection measures (and how this appears to have failed because it would have prevented the Linux operating system from running on next-generation PCs). Network owners and builders will appeal to the developers of the next generation of applications by arranging for the bulk of the support costs to fall on users rather than developers—even if this makes effective security administration impractical. Security engineers need to study network economics texts, such as Shapiro and Varian [696], to understand how the various plays that companies make to entrench monopolies, or to overturn them, interact with protection mechanisms.

There are also local economic issues. Security is about power, and a design will usually serve the perceived interests of whoever pays for the design work to be done. I described, in Chapter 8, how medical payment systems that are designed by insurers rather then by healthcare providers fail to protect patient privacy whenever this conflicts with the insurer's wish to maximize information about its clients. Chapter 9 described how banks in many countries managed for years to get their customers to bear the risk and cost of fraud; and Chapter 21 explained how some digital signature laws transfer the risk of forged signatures from the person who relies on the signature to the person alleged to have made it. Section 22.4.1.3 in this chapter explained the tragedy of the commons, where many players can dump their risks into a common pool, so that each gets a large benefit from taking a shortcut but suffers only a small share of the loss when something goes wrong; the result is that standards can decline rapidly.

A particularly topical case of the tragedy of the commons comes from the recent spate of distributed denial-of-service attacks whose technical aspects I discussed in Section 18.2.2.3. In these attacks, vandals hack a number of PCs and install attack software that bombards the target with more message traffic than it can handle. The probability of becoming a victim of such an attack is so low that most normal users quite rationally ignore it, so they don't bother to protect their PCs properly. Then, just as a common pasture gets overgrazed, so the Internet becomes increasingly insecure—and with more and more people installing high-bandwidth, always-on Internet connections, the insecurity will get worse. Jean Camp and Catherine Wolfram have drawn an interesting parallel between Internet insecurity and environmental pollution in [156].

The best way to manage such situations would be for the risks to fall on the parties most able to manage them. This is an established general principle in tort law, but enough industries and applications manage to escape it one way or another. In the case of distributed denial-of-service attacks, there would be little point in victims suing whichever random users had been hacked, as most home PC users are clueless about security; and, in any case, the risk of being the unlucky individual who got hacked and then sued would be low. Hal Varian has suggested that the hacked users' Internet service provider should carry much of the risk [771]. This would create the needed incentive for firewalls to police not just incoming traffic, but outgoing traffic as well. This is the thinking behind a strategy, which I described in passing in Section 6.2.4, of responding to a service denial attack on a Web site by replicating the site to a more capable, distributed server. As the use of such services can be rented, the necessary economic incentives can be implemented in a more-or-less transparent way. (For further details, see [816].)

In practice the driving forces behind security design usually have nothing to do with an altruistic desire to protect the end-users' privacy and to reduce the risk that they will be defrauded. The motives are much more likely to be the desire to grab a monopoly, to charge different prices to different users for essentially the same service, and to dump risk. Often, this is perfectly rational. Sometimes it isn't; British banks that dumped the risk of ATM fraud on their customers installed many security mechanisms so that in case of dispute they could argue in court that they had exercised due diligence; they ended up spending more on ATM security than U.S. banks, which had always borne the liability and for which security was a rational matter of risk management [19].

In an ideal world, the removal of perverse economic incentives to create insecure systems would depoliticize most issues. Security engineering would then be a matter of rational risk management rather than risk dumping. But don't hold your breath.

22.7 Summary

Developing a security requirements specification is often the most difficult part of the entire engineering process. Like developing the system itself, it can involve a one-off project, be a limited iterative process, or be a matter of continuous evolution. Evolution is easiest to manage, though it is complicated by changes of scale, environment, and business structures. Doing it from scratch for a completely new system is hardest and most error-prone, but there are still some useful techniques and lessons that can be borrowed from elsewhere.

In the absence of anything better, I suggest to the project manager engaged in building an application with some nontrivial protection requirements that you make a best effort to understand precisely what these properties are, build them into the specification, and then use whatever methodology you would use normally to follow them through implementation, testing, and deployment. But assume that you won't get it right first time. Make sure that you have some institutional means of capturing feedback on what goes wrong and how the environment is changing, so that you can feed this back into the process of enhancing and maintaining the system. Security must be an integral part of how you manage the system lifecycle.

Research Problems

The issues discussed in this chapter are among the most important—and most difficult—of any in our field. Ironically, 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 very productive area of research—if you have the necessary background. When building systems to be robust in the face of malice, you must also build them so that they remain robust in the face of normal human behavior, and hopefully are able to tell the difference between the two often enough to do something useful.

Further Reading

Literature on managing the development of information systems is large, diffuse, and multidisciplinary. There are classics that everyone should read, such as Fred Brooks' *Mythical Man-Month* [140] and Nancy Leveson's *Safeware* [498]. Standard textbooks on software engineering, such as those by Roger Pressman [622] and Hans van Vliet [767] cover the basics of project management and requirements engineering. The economics of the software lifecycle are discussed by Fred Brooks and Barry Boehm [123]. The Microsoft approach to managing software evolution is described by Steve McGuire [521]. There are useful parallels to 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 [612]. For a survey of risk management methods and tools, see Richard Baskerville [77] or Donn Parker [602]; there are some interesting case histories at IFCI [402]. Computer system failures are another necessary subject of study; the best source is the `comp.risks` newsgroup, of which a selection has been collated and published in print by Peter Neumann [590].

Organizational aspects are discussed at length in the business school literature, but this can be bewildering to the outsider. A critical guide to the literature is provided by John Micklethwait and Adrion Wooldridge, who 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 firing people, while at the same time preaching the virtues of trust [550]. Familiarity with this material is useful for predicting the protection consequences of your client's latest reorganizational fashion. Finally, the best books I know for material on the underlying economics are a popular synopsis by Carl Shapiro and Hal Varian [696], and a standard textbook by Hal Varian [770].