CHAPTER

23

System Evaluation and Assurance

If it's provably secure, it probably isn't.

—LARS KNUDSEN

I think any time you expose vulnerabilities it's a good thing

—U.S. ATTORNEY GENERAL JANET RENO [642]

23.1 Introduction

I've covered a lot of material in this book, some of it quite difficult. But I've left the hardest topics to the last. These are the questions of *assurance*, whether the system will work, and *evaluation*, how you convince other people of this.

Fundamentally, assurance comes down to the question of whether capable, motivated people have beat up on the system enough. But how do you define enough? And how do you define the system? How do you deal with people who protect the wrong thing, because their model of the requirements is out-of-date or plain wrong? And how do you allow for human failures? Many systems can be operated just fine by alert experienced professionals, but are unfit for purpose because they're too tricky for ordinary folk to use or are intolerant of error.

But if assurance is hard, evaluation is even harder. It's about how you convince your boss, your clients—and, in extremis, a jury—that the system is indeed fit for purpose; that it does indeed work (or that it did work at some particular time in the past). The reason that evaluation is both necessary and hard is that, often, one principal carries the cost of protection while another carries the risk of failure. This creates an obvious tension, and third-party evaluation schemes such as the Common Criteria are marketed as a means of making it more transparent.

23.2 Assurance

A working definition of *assurance* could be "our estimate of the likelihood that a system will not fail in some particular way." This estimate can be based on a number of factors, such as the process used to develop the system; the identity of the person or team who developed it; particular technical assessments, such as the use of formal methods or the deliberate introduction of a number of bugs to see how many of them are caught by the testing team; and experience—which ultimately depends on having a model of how reliability grows (or decays) over time as a system is subjected to testing, use, and maintenance.

23.2.1 Perverse Economic Incentives

A good starting point for the discussion of assurance is to look at the various principals' motives. As a preliminary let's consider the things for which we may need assurance:

- *Functionality* is important and often neglected. It's all too common to end up protecting the wrong things or protecting the right things in the wrong way. Recall from Chapter 8, for example, how the use of the Bell-LaPadula model in the healthcare environment caused more problems than it solved.
- *Strength of mechanisms* has been much in the news, thanks to U.S. export controls on crypto. Many products, such as DVD, were shipped with 40-bit keys and were thus intrinsically vulnerable. Strength of mechanisms is independent of functionality, but can interact with it. For example, in Chapter 14, I remarked how the difficulty of preventing probing attacks on smartcards led the industry to protect other, relatively unimportant things such as the secrecy of chip masks.
- *Implementation* is the traditional focus of assurance. This involves whether, given the agreed functionality and strength of mechanisms, the product has been implemented correctly. As we've seen, most real-life technical security failures are due to programming bugs—stack overflow vulnerabilities, race conditions, and the like. Finding and fixing them absorbs most of the effort of the assurance community.
- *Usability* is the missing factor—one might even say the spectre at the feast. Perhaps the majority of system-level (as opposed to purely technical) failures have a large human interface component. It is very common for secure system designers to tie up the technical aspects of protection tightly, without stopping to consider human frailty. There are some notable exceptions. The bookkeeping systems described in Chapter 9 are designed to cope with user error; and the security printing technologies discussed in Chapter 12 are often optimized to make it easier for untrained and careless people to spot forgeries. But usability concerns developers as well as users. A developer usability issue, mentioned in Chapter 4 is that the access controls provided with commodity operating systems often aren't used, as it's so much simpler to make code run with administrator privilege.

These four factors are largely independent, and the system builder has to choose an appropriate combination of them to aim at. A personal computer user, for example, might want high usability, medium assurance (because high would be expensive, and we can live with the odd virus), high strength of mechanisms (they don't cost much more), and simple functionality (as usability is more important). But the market doesn't deliver this, and a moment's thought will indicate why.

Commercial platform vendors go for rich functionality (rapid product versioning prevents the market being commoditized, and complementary vendors that grab too much market share can be undermined), low strength of mechanisms (except for cryptography where the escrow debate has led vendors to regard strong crypto as an essential marketing feature), low implementation assurance (so the military-grade crypto is easily defeated by Trojan horses), and low usability (application programmers matter much more than customers, as they enhance network externalities).

In Chapter 22, I described why this won't change any time soon. The strategy of "ship it Tuesday and get it right by version 3" isn't a personal moral defect of Bill Gates, as some of his critics allege, but is dictated by the huge first-mover advantages inherent in the economics of networks. And mechanisms that compelled application developers to use operating system access controls would alienate them, raising the risk that they might write their code for competitors' platforms. Thus, the current insecurity of commercial systems is perfectly rational from the economists' viewpoint, however undesirable from the users'.

Government agencies' ideals are also frustrated by economics. Their dream is to be able to buy commercial off-the-shelf products, replace a small number of components (such as by removing commercial crypto and plugging in Fortezza cards in its place), and end up with something they can use with existing defense networks. That is, they want Bell-LaPadula functionality (never mind that it fails to support mechanisms some of the vendors' other customers need) and high implementation assurance. There is little concern with usability, as a trainable and disciplined workforce is assumed (however wrongly), and low strength of crypto is preferred so as to limit the benefits that potential enemies can gain from otherwise high-assurance systems being on the market. This wish list is unrealistic given not just the cost of high assurance (which I'll discuss shortly), but also the primacy of time-to-market, the requirement to appease the developer community, and the need for frequent product versioning to prevent the commoditization of markets. Also, larger networks usually swamp smaller ones; so a million government computer users can't expect to impose their will on 100 million users of Microsoft Office.

The dialogue between user advocates, platform vendors, and government is probably condemned to remain a dialogue of the deaf. But that doesn't mean there's nothing more of interest to say on assurance.

23.2.2 Project Assurance

Assurance is a process very much like the development of code or documents. Just as you will have bugs in your code and in your specification, you will also have bugs in your test procedures. So assurance can be done as a one-off project or be the subject of continuous evolution. An example of the latter is given by the huge databases of known computer viruses that anti-virus software vendors accumulate over the years to

do regression-testing of their products. Assurance can also involve a combination, as when a step in an evolutionary development is managed using project techniques and is tested as a feature before being integrated and subjected to system-level regression tests. Here, you also have to find ways of building feature tests into your regression test suite.

Nonetheless, it's helpful to look first at the project issues, then at the evolutionary issues.

23.2.2.1 Security Testing

In practice, security testing usually comes down to reading the product documentation, reviewing the code, then performing a number of tests. (This is known as *white-box testing*, as opposed to *black-box testing*, for which the tester has the product but not the design documents or source code). The process is:

1. First look for any obvious flaws, the definition of which will depend on the tester's experience.
2. Then look for common flaws, such as stack-overwriting vulnerabilities.
3. Then work down a list of less common flaws, such as those described in the various chapters of this book.

The process is usually structured by the requirements of a particular evaluation environment. For example, it might be necessary to show that each of a list of control objectives was assured by at least one protection mechanism; in some industries, such as bank inspection, there are more or less established checklists (see, for example, [72]).

23.2.2.2 Formal Methods

In Chapter 2, I gave an example of a formal method: the BAN logic that can be used to verify certain properties of cryptographic protocols. The working engineer's take on formal methods may be that they're widely taught in universities, but not used anywhere in the real world. This isn't quite true in the security business. There are problems—such as in designing crypto protocols—where intuition is often inadequate and where formal verification can be helpful. Military purchasers go further, and require the use of formal methods as a condition of higher levels of evaluation under the Orange Book and the Common Criteria. I'll discuss this further below. For now, it's enough to say that this restricts high evaluation levels to relatively small and simple products, such as line encryption devices and operating systems for primitive computers such as smartcards. Even so, formal methods aren't infallible. Proofs can have errors, too; and often the wrong thing gets proved [673]. The quote by Knudsen at the head of this chapter refers to the large number of breaks of cryptographic algorithms or protocols that had previously been proven secure. These breaks generally occur because one of the proof's assumptions is unrealistic, or has become so over time.

23.2.2.3 Quis Custodiet?

Just as mistakes can be made by theorem provers and by testers, so they can also be made by people who draw up checklists of things for the testers to test (and by the security textbook writers from whose works the checklist writers draw). This is the old problem of *quis custodiet ipsos custodes*, as the Romans more succintly put it: who shall watch the watchmen?

There are a number of things one can do, few of which are likely to appeal to the organization whose goal is a declaration that a product is free of faults. The obvious one is *fault injection*, whereby a number of errors are deliberately introduced into the code at random. If there are 100 such errors, and the tester finds 70 of them, plus a further 70 that weren't deliberately introduced, then once the 30 remaining deliberate errors are removed, you can expect that there are 30 bugs left that you don't know about. (This assumes that the unknown errors are distributed the same as the known ones; reality will almost always be worse than this [133].)

Even in the absence of deliberate bug insertion, a rough estimate can be obtained by looking at which bugs are found by which testers. For example, I had Chapter 7 of this book reviewed by a fairly large number of people, as I took a draft of it to a conference on the topic. Given the bugs they found, and the number of people who reviewed the other chapters, I'd estimate that there are maybe three dozen errors of substance left in the book. The sample sizes aren't large enough in this case to justify more than a guess, but where they are large enough, we can use statistical techniques, which I'll describe shortly.

Another factor is the rate at which new attacks are discovered. In the university system, we train graduate students by letting them attack stuff; new vulnerabilites and exploits end up in research papers, which bring fame and, ultimately, promotion. The mechanics in government agencies and corporate labs are slightly different, but the overall effect is the same: a large group of capable, motivated people look for new exploits. Academics usually publish, government scientists usually don't, and corporate researchers sometimes do. So you need some means of adding new procedures to your test suite as fresh ideas come along, and to bear in mind that it will never be complete.

Finally, we get feedback from the rate at which instances of known bugs are discovered in products once they're fielded. This also provides valuable input for reliability growth models.

23.2.3 Process Assurance

In recent years, less emphasis has come to be placed on assurance measures focused on the product, such as testing, and more on process measures, such as who developed the system. As anyone with experience of system development knows, some programmers produce code with an order of magnitude fewer bugs than others. Also, some organizations produce much better quality code than others. This is the subject of much attention in the industry.

Some of the differences between high-quality and low-quality development teams are amenable to direct management intervention. Perhaps the most notable is whether people are responsible for correcting their own bugs. In the 1980s, many organizations

interpreted the waterfall model of system development to mean that one team wrote the specification, another wrote the code, yet another did the testing (including some bug fixing), while yet another did the maintenance (including the rest of the bug fixing). The teams communicated with each other only by means of the project documentation. This was justified on the grounds that it is more efficient for people to concentrate on a single task at a time; interrupting a programmer to ask him to fix a bug in code he wrote six months ago and had forgotten about could cost a day's productivity, while getting a maintenance programmer to do it might cost only an hour.

But the effect was that the coders produced megabytes of buggy code, and left it to the poor testers and maintenance people to clear up after them. Over time, both quality and productivity sagged. Industry analysts have ascribed IBM's near-death experience in the early 1990s, which cost over $100 billion in asset value, to this [169]. For its part, Microsoft considers that one of its most crucial lessons learned as it struggled with the problems of writing ever larger programs was to have a firm policy that "if you wrote it, you fix it." Bugs should be fixed as soon as possible; and even though they're as inevitable as death and taxes, programmers should never give up trying to write clean code.

Many other controllable aspects of the organization can have a significant effect on output quality, ranging from how bright your hires are to how you train them and the work habits you inculcate. (See Maguire for an extended discussion of Microsoft policy [521].)

For some years, internal auditors have included process issues while evaluating the quality of security code. This is harder to do than you might think, because a large part of an organization's quality culture is intangible. While some rules (such as "fix your own bugs") seem to be fairly universal, imposing a large number of specific rules would induce a bureaucratic box-ticking culture, rather than a dynamic competitive one. Consequently, recent work has aimed for a more holistic assessment of a team's capability; the lead contender is the *Capability Maturity Model* (CMM) from the Software Engineering Institute at Carnegie-Mellon University.

CMM is based on the idea that, as a team acquires experience, it can progress through a series of levels. The model has five levels—initial, repeatable, defined, managed, and optimizing—with a list of new things to be added as you go up the hierarchy. Thus, for example, project planning must be introduced to move up from initial to repeatable, and peer reviews to make the transition from repeatable to defined. There is a fuller description and bibliography in [767]; several attempts have been made to adapt CMM to security work, and a significant number of vendors already use it [545, 822].

An even more common process assurance approach is the ISO 9001 standard. The essence of this standard is that a company must document its processes for design, development, testing, documentation, audit, and management control, generally. For more detail, see [767]; there is now a whole industry of consultants helping companies get ISO 9001 certification. At its best, it can provide a framework for incremental process improvement; companies can monitor what goes wrong, trace it back to its source, fix it, and prevent it happening again. At its worst, it can be an exercise in box-ticking which merely replaces chaos with more bureaucratic chaos.

Many writers have remarked that organizations have a natural cycle of life, just as people do. Joseph Schumpeter argued that economic depressions perform a valuable societal function of clearing out companies that are past it or just generally unfit, in much the same way that fires rejuvenate forests. Successful companies become com-

placent and bureaucratic, so that some insiders opt for the good life while others leave (it used to be commonly said that the only people who ever left IBM were the good ones). Too rapid growth also brings problems: Microsoft insiders blame many of the current problems on the influx of tens of thousands of new hires in the late 1990s, many of whom were motivated more by the prospect of making millions from stock options than by the mission to write good code and get it running on every computer in the known universe.

The cycle of corporate birth, death, and reincarnation turns much more quickly in the computer industry than elsewhere, thanks to the combination of technological progress and multiple network externalities. The telecoms industry is suffering severe trauma as the computer and communications industries merge and the phone companies' 15-year product cycles have to shorten to 15 months to keep up with Microsoft. The security industry is starting to feel the same pressures. Teams that worked steadily for decades on cost-plus contracts to develop encryptors or MLS systems for the military have suddenly been exposed to ferocious technological and market forces, and have been told to build completely different things. Some have succeeded, as with the MLS supplier TIS, which reinvented itself as a firewall vendor; others have failed and disappeared. Thus, the value of a team of MLS "graybeards" is questionable. In any case, expert teams usually depend on one or two key gurus, and when they go off to do a startup, the team's capability can evaporate overnight.

Schemes such as ISO 9001 and CMM would be more convincing if there were some effective means of taking certification away from teams that had lost their stars, their sparkle, or their relevance. It is tempting to think that a solution might lie in the sort of ranking system used in food guides, where declaring a new establishment to be "the best Asian restaurant in San Francisco" entails dislodging the previous holder of this title. Of course, if certification were a more perishable asset, it would have to confer greater market advantage for companies to invest the same amount of effort in getting it. This may be feasible: the restaurant guide system works, and academic peer review works somewhat along the same lines.

23.2.4 Assurance Growth

Another aspect of process-based assurance is that most customers are not so much interested in the development team as in its product. But most software today is packaged rather than bespoke, and is developed by a process of continual evolutionary enhancement rather than in a one-off project. What, then, can usefully be said about the assurance level of evolving products?

The quality of such a product can reach equilibrium if the rate at which new bugs are introduced by product enhancements equals the rate at which old bugs are found and removed. But there's no guarantee that this will happen. (There are second-order effects, such as *senescence*, when repeated enhancement makes code so complex that its underlying reliability and maintainability drop off, but I'll ignore them for the sake of simplicity.)

While controlling the rate at which bugs are introduced will depend on the kind of development controls I've already described, measuring the rate at which they are re-

moved requires different tools—models of how the reliability of software (and systems in general) improves under testing.

A lot is known about reliability growth, as it's of interest to many more people than just software engineers.

Where the tester is trying to find a single bug in a system, a reasonable model is the Poisson distribution: the probability, p, that the bug remains undetected after t statistically random tests is given by $p = e^{-Et}$, where E depends on the proportion of possible inputs that it affects [506]. Where the reliability of a system is dominated by a single bug—as when we're looking for the first, or last, bug in a system—reliability growth can be exponential.

But extensive empirical investigations have shown that in large and complex systems, the likelihood that the t-th test fails is not proportional to e^{-Et} but to k/t for some constant k, so the system's reliability grows very much more slowly. This phenomenon was first noticed and documented in the bug history of IBM mainframe operating systems [7], and has been confirmed in many other studies [514]. As a failure probability of k/t means a *mean time between failure* (MTBF) of about t/k, reliability grows linearly with testing time. This result is often stated by the safety-critical systems community as, 'If you want a mean time between failure of a million hours, then you have to test for (at least) a million hours' [150]. This has been one of the main arguments against the development of complex, critical systems that can't be fully tested before use, such as ballistic missile defense.

The reason for the k/t behavior emerged in [105], and was proved under much more general assumptions in [133]. The latter uses techniques of statistical thermodynamics, and its core idea is that where a population of bugs with individual survival probabilities $p_i = e^{-E_i t}$ is large enough for certain statistical assumptions to hold, and they are eliminated over a long period of time, then the $e^{-E_i t}$ statistics of the individual bugs sum to k/t for the whole system. If they were eliminated any more slowly than this, the software would never work at all; and if they were eliminated any more quickly, the product would rapidly become bug-free—which, as we know, it usually doesn't.

This model gives a number of other interesting results. Under assumptions that are often reasonable, it is the best possible: the rule that you need a million hours of testing to get a million hours MTBF is inescapable, up to some constant multiple that depends on the initial quality of the code and the scope of the testing. This amounts to a proof of a version of Murphy's Law, that the number of defects that survive a selection process is maximized.

The model is similar to mathematical models of the evolution of a biological species under selective pressure. The role of bugs is played, roughly, by genes that reduce fitness. But some of the implications are markedly different. Murphy's Law, that the number of defects that survive a selection process is maximized, may be bad news for the engineer, but it's good news for biological species. While software testing removes the minimum possible number of bugs, consistent with the tests applied, biological evolution enables a species to adapt to a changed environment at a minimum cost in early deaths, meanwhile preserving as much diversity as possible. This diversity helps the species survive future environmental shocks. For example, if a population of rabbits is preyed on by snakes, the rabbits will be selected for alertness rather than speed. The variability in speed will remain, so if foxes arrive in the neighborhood, the rabbit population's average running speed will rise sharply under selective predation. More formally, the *fundamental theorem of natural selection* says that a species with a high

genic variance can adapt to a changing environment more quickly. But when Fisher proved this in 1930 [297], he was also proving that complex software will exhibit the maximum possible number of bugs when it is migrated to a new environment.

The evolutionary model also points to fundamental limits on the reliability gains to be had from reusable software components such as objects or libraries; well-tested libraries simply mean that overall failure rates will be dominated by new code. It also explains the observation of the safety-critical systems community that test results are often a poor performance indicator [506]: the failure time measured by a tester depends only on the initial quality of the program, the scope of the testing and the number of tests, so it gives virtually no further information about the program's likely performance in another environment. There are also some results that are unexpected, but obvious in retrospect. For example, each bug's contribution to the overall failure rate is independent of whether the code containing it is executed frequently or rarely—intuitively, code that is executed less is also tested less. Finally, as mentioned in Section 22.4.3, it is often more economic for different testers to work on a program in parallel rather than in series.

In short, complex systems become reliable only following prolonged testing. Thus, this book may be pretty reliable once thousands of people have read it and sent me bug reports; but if there's a second edition with a lot of new material, I can expect new bugs to creep in too. As for mass-market software, its wide use enables rapid debugging in principle; but, in practice, the constant new versions dictated by network economics place severe limits on what may reasonably be expected.

There appears to be no reason why these results don't go across in their entirety if a bug is defined to be a defect that causes a security vulnerability, rather than just any old defect—just as long as the number of bugs is large enough to do statistics.

23.2.5 Evolution and Security Assurance

Evolutionary growth of reliability may be much worse for the software engineer than for a biological species, but for the security engineer it's worse still.

Rather than going into the detailed mathematics, let's take a slightly simplified example. Suppose a large and complex product such as Win2K has a million bugs, each with an MTBF of a billion hours. Also suppose that Paddy works for the Irish Republican Army, and his job is to break into the British Army's computer to get the list of informers in Belfast, while Brian is the army assurance guy whose job is to stop Paddy. So he must learn of the bugs before Paddy does.

Paddy also has a day job, so he can only do 1,000 hours of testing a year. Brian, on the other hand, has full Windows source code, dozens of PhDs, control of the commercial evaluation labs, an inside track on CERT, an information-sharing deal with other UKUSA member states, and he runs the government's scheme to send consultants to critical industries such as power and telecoms to find out how to hack them (pardon me, to advise them how to protect their systems). Brian does ten million hours a year of testing.

After a year, Paddy finds a bug, while Brian has found 10,000. But the probability that Brian has found Paddy's bug is only 1%. Even if Brian declares martial law, drafts all Britain's 50,000 computer science graduates to a concentration camp in Gloucestershire, and sets them trawling through the Windows source code, he'll still only get 100 million hours of testing done each year. After ten years, he will find Paddy's bug. But by then Paddy will have found nine more, and it's unlikely that Brian will know of all

of them. Worse, Brian's bug reports will have become such a firehose that Bill will have killfiled them.

In other words, Paddy has thermodynamics on his side. Even a very moderately resourced attacker can break anything that's at all large and complex. There is nothing that can be done to stop this, as long as there are enough different security vulnerabilities to do statistics. The ray of hope is that, if all your vulnerabilities are, say, stack overflows, and you start using a new compiler that traps them, then for modelling purposes, there was only a single vulnerability, and you escape the statistical trap.

23.3 Evaluation

A working definition of *evaluation* is "the process of assembling evidence that a system meets, or fails to meet, a prescribed assurance target." (Evaluation often overlaps with testing, and is sometimes confused with it.) As I mentioned, this evidence might be needed only to convince your boss that you've completed the job. But, often, it is needed to reassure principals who will rely on the system that the principal who developed it, or who operates it, has done a workmanlike job. The fundamental problem is the tension that arises when the party who implements the protection and the party who relies on it are different.

Sometimes the tension is simple and visible, as when you design a burglar alarm to standards set by insurance underwriters, and have it certified by inspectors at the insurers' laboratories. Sometimes it's still visible but more complex, as when designing to government security standards that try to reconcile dozens of conflicting institutional interests, or when hiring your company's auditors to review a system and tell your boss that it's fit for purpose. It is harder when multiple principals are involved; for example, when a smartcard vendor wants an evaluation certificate from a government agency (which is trying to encourage the use of some feature such as key escrow that is in no one else's interest), in order to sell the card to a bank, which in turn wants to use it to dump the liability for fraud on to its customers. That may seem all rather crooked; but there may be no clearly criminal conduct by any of the people involved. The crookedness may be an emergent property that arises from managers following their own personal and departmental imperatives.

For example, managers often buy products and services that they know to be suboptimal or even defective, but which are from big-name suppliers. This is known to minimize the likelihood of getting fired when things go wrong. Corporate lawyers don't condemn this as fraud, but praise it as due diligence. The end result may be that the relying party, the customer, has no say whatsoever, and will find it hard to get redress against the bank, the vendor, the evaluator, or the government when things go wrong.

Another serious and pervasive problem is that the words "assurance" and "evaluation" are often interpreted to apply only to the technical aspects of the system, and ignore usability (not to mention the even wider issues of appropriate internal controls and good corporate governance). Company directors also want assurance—that the di-

rected procedures are followed, that there are no material errors in the accounts, that applicable laws are being complied with, and dozens of other things. But many evaluation schemes (especially the Common Criteria) studiously ignore the human and organizational elements in the system. If any thought is paid to them at all, the evaluation of these elements is considered to be a matter for the client's IT auditors, or even for a system administrator setting up configuration files. All that said, I'll focus on technical evaluation in what follows.

It is convenient to break evaluation into two cases. The first is where the evaluation is performed by the relying party; this includes insurance assessments, the independent verification and validation done by NASA on mission-critial code, and the previous generation of military evaluation criteria, such as the Orange Book. The second is where the evaluation is done by someone other than the relying party. Nowadays, this often means the Common Criteria evaluation process.

23.3.1 Evaluations by the Relying Party

In Chapter 10, I discussed many of the concerns that insurers have with burglar alarm systems, and the considerations that go into approving equipment for use with certain sizes of risk. The approval process itself if simple enough; the insurance industry operates laboratories where tests are conducted. These might involve a fixed budget of effort (perhaps one person for two weeks, or a cost of $15,000). The evaluator starts off with a fairly clear idea of what a burglar alarm should and should not do, spends the budgeted amount of effort looking for flaws, and writes a report. The laboratory then either approves the device, turns it down, or demands some changes.

In Section 7.4, I described another model of evaluation, that done from 1985–2000 at the NSA's National Computer Security Center on computer security products proposed for U.S. government use. These evaluations were conducted according to the Orange Book, the Trusted Computer Systems Evaluation Criteria [240]. The Orange Book and its supporting documents set out a number of evaluation classes:

C1: Discretionary access control by groups of users. In effect, this is considered to be equal to no protection.

C2: Discretionary access control by single users; object reuse; audit. C2 corresponds to carefully configured commercial systems; for example, C2 evaluations were given to IBM mainframe operating systems with RACF, and to Windows NT. (Both of these were conditional on a particular version and configuration; in NT's case, for example, it was restricted to diskless workstations).

B1: Mandatory access control. All objects carry security labels, and the security policy (which means Bell-LaPadula or a variant) is enforced independently of user actions. Labeling is enforced for all input information.

B2: Structured protection. As B1, but there must also be a formal model of the security policy that has been proved consistent with security axioms. Tools must be provided for system administration and configuration management. The TCB must be properly structured and its interface clearly defined. Covert channel analysis must be performed. A trusted path must be provided from the user to the TCB. Severe testing, including penetration testing, must be carried out.

B3: Security domains. As B2, but the TCB must be minimal; it must mediate all access requests, be tamper-resistant, and be able to withstand formal analysis and testing. There must be real-time monitoring and alerting mechanisms, and structured techniques must be used in implementation.

A1: Verification design. As B3, but formal techniques must be used to prove the equivalence between the TCB specification and the security policy model.

The evaluation class of a system determines what spread of information may be processed on it. The example I gave in Section 7.5.2 was that a system evaluated to B3 may in general process information at Unclassified, Confidential, and Secret, or at Confidential, Secret, and Top Secret. (The complete rule set can be found in [244].) Although these ratings will cease to be valid after the end of 2001, they have had a decisive effect on the industry.

The business model of Orange Book evaluations followed traditional government service work practices. A government user would want some product evaluated; the NSA would allocate people to do it; they would do the work (which, given traditional civil service caution and delay, could take two or three years); the product, if successful, would join the evaluated products list; and the bill would be picked up by the taxpayer. The process was driven and controlled by the government—the party that was going to rely on the results of the evaluation—while the vendor was the supplicant at the gate. Because of the time the process took, evaluated products were usually one or two generations behind current commercial products, and often an order of magnitude more expensive.

The Orange Book wasn't the only evaluation scheme running in America. I mentioned in Section 14.4 the FIPS 140-1 scheme for assessing the tamper-resistance of cryptographic processors; this uses a number of independent laboratories as contractors. Independent contractors are also used for *Independent Verification and Validation* (IV&V), a scheme set up by the Department of Energy for systems to be used in nuclear weapons, and later adopted by NASA for manned space flight, which has many similar components (at least at the rocketry end of things). In IV&V, there is a simple evaluation target: zero defects. The process is still driven and controlled by the relying party—the government. The IV&V contractor is a competitor of the company that built the system, and its payments are tied to the number of bugs found.

Other governments had similar schemes. The Canadians had the *Canadian Trusted Products Evaluation Criteria* (CTPEC), while a number of European countries developed the *Information Technology Security Evaluation Criteria* (ITSEC). The idea was that a shared evaluation scheme would help European defense contractors compete against U.S. suppliers, with their larger economies of scale; Europeans would no longer be required to have separate certification in Britain, France, and Germany. ITSEC combined ideas from the Orange Book and IV&V processes, in that there were a number of different evaluation levels; and for all but the highest of these levels, the work was contracted out. However, ITSEC introduced a pernicious innovation: the evaluation was not paid for by the government but by the vendor seeking an evaluation on its product.

This was the usual civil service idea of killing several birds with one stone: saving public money and at the same time promoting a more competitive market. As usual, the stone appears to have done more damage to the too-clever hunter than to either of the birds.

This change in the rules provided the critical perverse incentive. It motivated the vendor to shop around for the evaluation contractor who would give its product the easiest ride, whether by asking fewer questions, charging less money, taking the least time, or all of these. (The same may happen with FIPS 140-1 now that commercial companies are starting to rely on it for third-party evaluations.) To be fair, the potential for this was realized, and schemes were set up whereby contractors could obtain approval as a *commercial licensed evaluation facility* (CLEF). The threat that a CLEF might have its license withdrawn was intended to offset the commercial pressures to cut corners.

23.3.2 The Common Criteria

This sets the stage for the Common Criteria. The original goal of the Orange Book was to develop protection measures that would be standard in all major operating systems, not an expensive add-on for captive government markets (as Orange Book evaluated products became). The problem was diagnosed as too-small markets, and the solution was to expand them. Because defense contractors detested having to obtain separate evaluations for their products in the United States, Canada, and Europe, agreement was reached to scrap the national evaluation schemes and replace them with a single standard. The work was substantially done in 1994–1995, and the European model won out over the U.S. and Canadian alternatives. As with ITSEC, evaluations under the Common Criteria, at all but the highest levels are done by CLEFs, and are supposed to be recognized in all participating countries (though any country can refuse to honor an evaluation if it says its national security is at stake); and vendors pay for the evaluations.

There are some differences. Most crucially, the Common Criteria have much more flexibility than the Orange Book. Rather than expecting all systems to conform to Bell-LaPadula, a product is evaluated against a *protection profile*, which, at least in theory, can be devised by the customer. This doesn't signify that the Department of Defense has abandoned multilevel security as much as an attempt to broaden the tent, get lots of commercial IT vendors to use the Common Criteria scheme, and thus defeat the perverse economic incentives described in Section 23.2.1 above. The aspiration was to create a bandwagon effect, which would result in the commercial world adapting itself somewhat to the government way of doing things.

23.3.2.1 Common Criteria Terminology

To discuss the Common Criteria in detail, I need to introduce some more jargon. The product under test is known as the *target of evaluation* (TOE). The rigor with which the examination is carried out is the *evaluation assurance level* (EAL); it can range from EAL1, for which functional testing is sufficient, all the way up to EAL7, for which thorough testing is required as well as a formally verified design. The highest evaluation level commonly obtained for commercial products is EAL4, although there

is one smartcard operating system with an EAL6 evaluation (obtained, however, under ITSEC rather than under CC).

A *protection profile* is a set of security requirements, their rationale, and an EAL. The profile is supposed to be expressed in an implementation-independent way to enable comparable evaluations across products and versions. A *security target* (ST) is a refinement of a protection profile for a given target of evaluation. In addition to evaluating a product, one can evaluate a protection profile (the idea is to ensure that it's complete, consistent, and technically sound) and a security target (to check that it properly refines a given protection profile). When devising something from scratch, the idea is to first create a protection profile, and evaluate it (if a suitable one doesn't exist already), then do the same for the security target, then finally evaluate the actual product. The result of all this is supposed to be a registry of protection profiles and a catalogue of evaluated products.

A protection profile should describe the environmental assumptions, the objectives, and the protection requirements (in terms of both function and assurance), and break them down into components. There is a stylized way of doing this. For example, `FCO_NRO` is a functionality component (hence `F`) relating to communications (`CO`), and it refers to nonrepudiation of origin (`NRO`). Other classes include `FAU` (audit), `FCS` (crypto support), and `FDP`, which means data protection (this isn't data protection as in European law, but refers to access control, Bell-LaPadula information flow controls, and related properties). The component catalogue is heavily biased toward supporting MLS systems.

There are also catalogues of:

- Threats, such as `T.Load_Mal`—"Data loading malfunction: an attacker may maliciously generate errors in set-up data to compromise the security functions of the TOE."
- Assumptions, such as `A.Role_Man`—"Role management: management of roles for the TOE is performed in a secure manner" (in other words, the developers, operators and so on behave themselves).
- Organizational policies, such as `P.Crypt_Std`—"Cryptographic standards: cryptographic entities, data authentication, and approval functions must be in accordance with ISO and associated industry or organizational standards."
- Objectives, such as `O.Flt_Ins`—"Fault insertion: the TOE must be resistant to repeated probing through insertion of erroneous data."
- Assurance requirements, such as `ADO_DEL.2`—"Detection of modification: the developer shall document procedures for delivery of the TOE or parts of it to the user."

I mentioned that a protection profile will contain a *rationale.* This typically consists of tables showing how each threat is controlled by one or more objectives, and, in the reverse direction, how each objective is necessitated by some combination of threats or environmental assumptions, plus supporting explanations. It will also justify the selection of an assurance level and requirements for strength of function.

The fastest way to get the hang of this is to read a few of the existing profiles, such as that for smart cards [579]. As with many protection profiles, this provides a long list of things that can go wrong and things that a developer can do to control them, and so is a useful checklist. The really important aspects of card protection, though, are found in `O.Phys_Prot`, "Physical protection: the TOE must be resistant to physical attack or be

able to create difficulties in understanding the information derived from such an attack" [p. 24]. An inexperienced reader might not realize that this objective is the whole heart of the matter; and as explained in Chapter 14, it's extremely hard to satisfy. (There's an admission on pp. 100–101 that competent attackers will still get through, but this is couched in terms likely to be opaque to the lay reader.) In general, the Criteria and the documents generated using them are unreadable, and this undermines the value they were intended to bring to the nonspecialist engineer.

Still, the Common Criteria can be useful to the security engineer in that they provide such extensive lists of things to check. They can also provide a management tool for keeping track of all the various threats and ensuring that they're all dealt with (otherwise, it's very easy for one to be forgotten in the mass of detail). But if the client insists on an evaluation—especially at higher levels—then these lists are apt to turn from a help into a millstone. Before accepting all the costs and delays this will cause, it's important to understand what the Common Criteria don't do.

23.3.2.2 What the Common Criteria Don't Do

The documents admit that the Common Criteria don't deal with administrative security measures, nor "technical-physical" aspects such as Emsec, nor crypto algorithms, nor the evaluation methodology, nor how the standards are to be used. The documents claim not to assume any specific development methodology (but then go on to assume a waterfall approach). There is a nod in the direction of evolving the policy in response to experience, but reevaluation of products is declared to be outside the scope. Oh, and there is no requirement for evidence that a protection profile corresponds to the real world; and I've seen a few that studiously ignore published work on relevant vulnerabilities. In other words, the Criteria avoid all the hard and interesting bits of security engineering, and can easily become a cherry pickers' charter.

The most common specific criticism (apart from cost and bureaucracy) is that the Criteria are too focused on the technical aspects of design. For example, in `ADO_DEL.2` (and elsewhere) we find that procedures are seen as secondary to technical protection (the philosophy is to appeal to procedures where a technical fix isn't available). But, as explained in Section 12.6, when evaluating a real system, you have to assess the capability and motivation of the personnel at every stage in the process. This is fundamental, not something that can be added on afterward.

Even more fundamental, is that business processes should not be driven by the limits of the available technology (and especially not by the limitations of the available expensive, out-of-date military technology). System design should be driven by business requirements; and technical mechanisms should be used only where they're justified, not just because they exist. In particular, technical mechanisms shouldn't be used where the exposure is less than the cost of controlling it, or where procedural controls are cheaper. Remember why Samuel Morse beat the dozens of other people who raced to build electric telegraphs in the early nineteenth century. They tried to build modems, so they could deliver text from one end to the other; Morse realized that, given the technology then available, it was cheaper to train people to be modems.

So much for the theory of what's wrong with the Criteria. As always, the practical vulnerabilities are different, and at least as interesting.

23.3.3 What Goes Wrong

In none of the half-dozen or so affected cases I've been involved in has the Common Criteria approach proved satisfactory. (Perhaps that's because I am called in only when things go wrong—but my experience still indicates a lack of robustness in the process.)

One of the first points that must be made is that the CLEFs that do the evaluations are beholden for their registration to the local intelligence agency, and their staff must all have clearances. This leaves open a rather wide path toward what one might call *institutional corruption.*

Corruption doesn't have to involve money changing hands or even an explicit exchange of favors. For example, when the Labor party won the 1997 election in Britain, I soon received a phone call from an official at the Department of Trade and Industry. He wanted to know whether I knew any computer scientists at the University of Leeds, so that the department could award my group some money to do collaborative research with them. It transpired that the incoming science minister represented a constituency in Leeds. This does not imply that the minister told his officials to find money for his local university; almost certainly it was an attempt by the officials to schmooze him.

23.3.3.1 Corruption, Manipulation, and Inertia

This preemptive cringe, as one might call it, appears to play a large part in the conduct of the evaluation labs. The most egregious example in my experience occurred in the British National Health Service. The service had agreed, under pressure from the doctors, to encrypt traffic on the health service network; GCHQ made no secret of its wish that key escrow products be used. Trials were arranged; one of them used commercial encryption software from a Danish supplier that had no key escrow, and cost £3,000, while the other used software from a U.K. defense contractor that had key escrow, and cost £100,000. To GCHQ's embarrassment, the Danish software worked, but the British supplier produced nothing that was usable. The situation was quickly salvaged by having a company with a CLEF license evaluate the trials. In its report, it claimed the exact reverse: that the escrow software worked fine, while the foreign product had all sorts of problems. Perhaps the CLEF was simply told what to write; it's just as likely that the staff wrote what they knew GCHQ wanted to read.

Sometimes, an eagerness to please the customer becomes apparent. In the context of the Icelandic health database (Section 8.3.4.1 above), its promoters wanted to defuse criticism from doctors about its privacy problems, so they engaged a British CLEF to write a protection profile for them. This simply repeated, in Criteria jargon, the promoters' original design and claims; it studiously avoided noticing flaws in this design, which had already been documented and even discussed on Icelandic TV [38].

Sometimes the protection profiles might be sound, but the way they're mapped to the application isn't. For example, European governments and IT vendors are currently working on regulations for the "advanced electronic signatures," which, as mentioned in Section 21.2.4.4, will shortly have to be recognized as the equivalent of handwritten signatures in all EU member states. The present proposal is that the signature creation device should be a smartcard evaluated to above level EAL4. (The profile [579] is for EAL4 augmented, which, as mentioned, is sufficient to keep out all attackers but the competent ones.) But no requirements are proposed for the PC that displays to you the material that you think you are signing. The end result will be a "secure" (in the sense

of non-repudiable) signature on whatever the virus or Trojan in your PC sent to your smartcard.

Of course, insiders figure out even more sophisticated ways to manipulate the system. A nice example comes from how the French circumvented British and German opposition to the smartcard-based electronic tachograph described in Section 10.4. They wrote a relaxed protection profile and sent it to a British CLEF to be evaluated. The CLEF was an army software company; whatever their knowledge of MLS, they knew nothing about smartcards. But this didn't lead them to turn down the business. They also didn't know that the U.K. government was opposed to approval of the protection profile. Thus, Britain was left with a choice between accepting defective road safety standards as a fait accompli, and undermining confidence in the Common Criteria.

Given all the corruption, greed, incompetence, and manipulation, it's like a breath of fresh air to find some good, old-fashioned bureaucratic inertia. An example is the healthcare protection profile under development for the U.S. government. Despite all the problems with using the MLS protection philosophy in healthcare, which I discussed in Chapter 9, that's what the profile ended up using [34]. It assumed that no users would be hostile (despite the fact that almost all attacks on health systems are from insiders), and insisted that multiple levels be supported, even though, as described in Chapter 9, levels don't work in that context. It also provided no rules as to how classifications or compartments should be managed, but left access control policy decisions to the catch-all phrase "need to know."

23.3.3.2 Underlying Problems

In general, the structure of the Common Criteria is strongly oriented toward MLS systems and to devices that support them, such as government firewalls and encryption boxes. This is unsurprising given the missions of the agencies that developed them. They assume trained obedient users, small systems that can be formally verified, uniform MLS-type security policies, and an absence of higher-level attacks, such as legal challenges. This makes them essentially useless for most of the applications one finds in the real world.

As for the organizational aspects, I mentioned in 23.2.3 that process-based assurance systems fail if accredited teams don't lose their accreditation when they lose their sparkle. This clearly applies to CLEFs. Even if CLEFs were licensed by a body independent of the intelligence community, many would deteriorate as key staff leave or as skills don't keep up with technology; and as clients shop around for easier evaluations, there will inevitably be grade inflation. Yet, at present, I can see no usable mechanism whereby a practitioner with very solid evidence of incompetence (or even dishonesty) can challenge a CLEF and have it removed from the list. In the absence of sanctions for misbehavior, institutional corruption will remain a serious risk.

When presented with a new security product, the engineer must always consider whether the sales rep is lying or mistaken, and how. The Common Criteria were supposed to fix this problem, but they don't. When presented with a product from the evaluated list, you have to ask how the protection profile was manipulated and by whom; whether the CLEF was dishonest or incompetent; what pressure from which government was applied behind the scenes; and how your rights are eroded by the certificate.

For example, if you use an unevaluated product to generate digital signatures, then a forged signature turns up and someone tries to use it against you, you might reasonably expect to challenge the evidence by persuading a court to order the release of full documentation to your expert witnesses. A Common Criteria certificate might make a court very much less ready to order disclosure, and thus could severely prejudice your rights. In fact, agency insiders admit after a few beers that the main issue is "confidence"—that is, getting people to accept systems as secure even when they aren't.

A cynic might suggest that this is precisely why, in the commercial world, it's the vendors of products that rely on obscurity and that are designed to transfer liability (such as smartcards), to satisfy due-diligence requirements (such as firewalls) or to impress naive users (such as PC access control products), who are most enthusiastic about the Common Criteria. A really hard-bitten cynic might point out that since the collapse of the Soviet Union, the agencies justify their existence by economic espionage, and the Common Criteria signatory countries provide most of the interesting targets. A false U.S. evaluation of a product that is sold worldwide may compromise 250 million Americans; but as it will also compromise 400 million Europeans and 100 million Japanese, the balance of advantage lies in deception. The balance is even stronger with small countries such as Britain, which has fewer citizens to protect and more foreigners to attack. In addition, agencies get brownie points (and budgets) for foreign secrets they steal, not for local secrets that foreigners didn't manage to steal.

An economist, then, is unlikely to trust a Common Criteria evaluation. Perhaps I'm one of the cynics, but I tend to view them as being somewhat like a rubber crutch. Such a device has all sorts of uses, from winning a judge's sympathy through wheedling money out of a gullible government to smacking people round the head. (Just don't try to put serious weight on it!)

Fortunately, the economics discussed in Section 23.2.1 should limit the uptake of the Criteria to sectors where an official certification, however irrelevant, erroneous, or mendacious, offers some competitive advantage.

23.4 Ways Forward

In his classic book, *The Mythical Man-Month*, Brooks argues compellingly that there is no "silver bullet" to solve the problems of software projects that run late and over budget [140]. The easy parts of the problem, such as developing high-level languages in which programmers can work much better than in assembly language, have been done. The removal of much of the accidental complexity of programming means that the intrinsic complexity of the application is what's left. I discussed this in Chapter 22, in the general context of system development methodology; the above discussion should convince you that exactly the same applies to the problem of assurance and, especially, to evaluation.

A more realistic approach to evaluation and assurance would look not just at the technical features of the product but at how it behaves in real applications. Usability is ignored by the Common Criteria, but is in reality all-important; a U.K. government

email system that required users to reboot their PC whenever they changed compartments frustrated users so much that they made informal agreements to put everything in common compartments—in effect wasting a nine-figure investment. (Official secrecy will no doubt continue to protect the guilty parties from punishment.) The kind of features I described in the context of bookkeeping systems in Chapter 9, which are designed to limit the effects of human frailty, are also critical. In most applications, we must assume that people are always careless, usually incompetent, and occasionally dishonest.

It's also necessary to confront the fact of large, feature-rich programs that are updated frequently. Network economics cannot be wished away. Evaluation and assurance schemes, such as the Common Criteria, ISO9001, and even CMM, try to squeeze a very volatile and competitive industry into a bureaucratic straightjacket, to provide purchasers with the illusion of stability. If such stability did exist, the whole industry would flock to it; but the best people can do is flock to brands, such as IBM in the 1970s and 1980s, and Microsoft now. The establishment and maintenance of these brands involves huge market forces; security plays a small role.

I've probably given you enough hints by now about how to cheat the system and pass off a lousy product as a secure one—at least long enough for the problem to become someone else's. In the remainder of this book, I'll assume that you're making an honest effort to protect a system and that you want risk reduction, rather than due diligence or some other kind of liability dumping. In many cases, it's the system owner who loses when the security fails; I've cited a number of examples earlier (nuclear command and control, pay-TV, prepayment utility meters, etc.) and they provide some of the more interesting engineering examples.

When you really want a protection property to hold it is vital that the design be subjected to hostile review. It will be eventually, and it's better if it's done before the system is fielded. As discussed in one case history after another, the motivation of the attacker is almost all-important; friendly reviews, by people who want the system to pass, are essentially useless, compared with contributions by people who are seriously trying to break it.

23.4.1 Semi-Open Design

One way of doing this is, to hire multiple experts from different consultancy firms or universities. Another is to use multiple different accreditation bodies: I mentioned in 21.6.4 how voting systems in the United States are vetted independently in each state; and in the days before standards were imposed by organizations such as VISA and SWIFT, banks would build local payment networks, with each of them having the design checked by its own auditors. Neither approach is infallible, though; there are some really awful legacy voting and banking systems.

Another, very well established, technique is what I call *semi-open design*. Here, the architectural-level design is published, even though the implementation details may not be. Examples that I've given include the smartcard banking protocol discussed in Section 2.7.1, and the nuclear command and control systems mentioned in Chapter 11.

Another approach to semi-open design is to use an openly available software package, which anyone can experiment with. This can be of particular value when the main threat is a legal attack. It is unreasonable to expect a court to grant access to the source code of a spreadsheet product such as Excel, or even to accounting software sold by a medium-sized vendor; the opposing expert will just have to buy a copy, experiment with it, and see what she can find. You just have to take your chances that a relevant bug will be found in the package later, or that some other feature will turn out to undermine the evidence that it produces.

23.4.2 Open Source

Open source extends the philosophy of openness from the architecture to the implementation detail. A number of security products have publicly available source code, of which the most obvious is the PGP email encryption program. The Linux operating system and the Apache Web server are also open source, and are relied on by many people to protect information. There is also a drive to adopt open source in government.

Open source software is not entirely a recent invention; in the early days of computing, most system software vendors published their source code. This openness started to recede in the early 1980s when pressure of litigation led IBM to adopt an "object-code-only" policy for its mainframe software, despite bitter criticism from its user community. The pendulum now seems to be swinging back.

There are a number of strong arguments in favor of open source, and a few against. The strongest argument is that if everyone in the world can inspect and play with the software, then bugs are likely to be found and fixed; in Eric Raymond's famous phrase, "To many eyes, all bugs are shallow" [634]. This is especially so if the software is maintained in a cooperative effort, as Linux and Apache are. It may also be more difficult to insert backdoors into such a product.

Arguments against open source center on the fact that once software becomes large and complex, there may be few or no capable motivated people studying it, hence major vulnerabilities may take years to be discovered. A recent example was a programming bug in PGP versions 5 and 6, which allowed an attacker to add an extra escrow key without the keyholder's knowledge [690], and which was fielded for several years before it was spotted. (The problem may be that PGP is being developed faster than people can read the code; that many of the features with which it's getting crammed are uninteresting to the potential readers; or just that now that it's a commercial product, people are not so motivated to do verification work on it for free.)

There have also been backdoor "maintenance passwords" in products such as `sendmail` that persisted for years before they were removed. The concern is that there may be attackers who are sufficiently motivated to spend more time finding bugs or exploitable features in the published code than the community of reviewers are. In fact, it can be worse than this; as noted in Section 23.2.4, different testers find different bugs, because their test focus is different, so it's quite possible that even once a product had withstood 10,000 hours of community scrutiny, a foreign intelligence agency that in-

vested a mere 100 hours might find a new exploitable vulnerability. Given the cited reliability growth models, the probabilities are easy enough to work out.

Other arguments against open source include the observation that active open source projects add functionality and features at dizzying speed compared to closed software, which can open up nasty feature interactions; that there had better be consensus about what the security is trying to achieve; and that there are special cases, such as when protecting smartcards against various attacks, where a proprietary encryption algorithm embedded in the chip hardware can force the attacker to spend significantly more effort in reverse engineering.

So where is the balance of benefit? Eric Raymond's influential analysis of the economics of open source software [635] suggests that there are five criteria for whether a product would be likely to benefit from an open source approach: where it is based on common engineering knowledge, rather than proprietary techniques; where it is sensitive to failure; where it needs peer review for verification; where it is sufficiently business-critical that users will cooperate in finding and removing bugs; and where its economics include strong network effects. Security passes all these tests, and indeed the long-standing wisdom of Kerckhoffs is that cryptographic systems should be designed in such a way that they are not compromised if the opponent learns the technique being used [454]. There is increasing interest in open source from organizations such as the U.S. Air Force [688, 689].

It's reasonable to conclude that while an open source design is neither necessary nor sufficient, it is often going to be helpful. The important questions are how much effort was expended by capable people in checking and testing the code—and whether they tell you everything they find.

23.4.3 Penetrate-and-Patch, CERTs, and bugtraq

Penetrate-and-patch is the name given dismissively in the 1970s and 1980s to the evolutionary procedure of finding security bugs in systems and then fixing them; it was widely seen at that time as inadequate, as more bugs were always found. At that time, people hoped that formal methods would enable bug-free systems to be constructed. With the realization that such systems are too small and limited for most applications, iterative approaches to assurance are coming back into vogue, along with the question of how to manage them.

Naturally, there's a competitive element to this. The U.S. government's wish is that vulnerabilities in common products such as operating systems and communications software should be reported first to authority, so that they can be exploited for law enforcement or intelligence purposes if need be, and that vendors should ship patches only after unauthorized persons start exploiting the hole. Companies such as Microsoft share source code and vulnerability data with intelligence agency departments engaged in the development of hacking tools, and the computer emergency response teams (CERTs) in many countries are funded by defense agencies. In addition, many feel that the response of CERTs is somewhat slow. The alternative approach is open reporting of bugs as they're found—as happens on a number of mailing lists, notably bugtraq.

Neither approach is fully satisfactory. In the first case, you never know who saw the vulnerability report before you did; and in the second case, you know that anyone in

the world could see it and use it against you before a patch is shipped. Perhaps a more sensible solution was proposed in [631], under which a researcher who discovers a vulnerability should first email the software maintainer. The maintainer will have 48 hours to acknowledge receipt, failing which the vulnerability can be published; the maintainer will have a further five days to actually work on the problem, with a possibility of extension by mutual negotiation. The resulting bug fix should carry a credit to the researcher. Just before this book went to press, CERT agreed to start using this procedure, only with a delay of 45 days for the vendor to design and test a fix [174].

This way, software companies have a strong incentive to maintain an attentive and continually staffed bug-reporting facility, and in return will get enough time to test a fix properly before releasing it; researchers will get credits to put on their CVs; users will get bug fixes at the same time as bug reports; and the system will be very much harder for the agencies to subvert.

23.4.4 Education

Perhaps as an academic, I'm biased, but I feel that the problems and technologies of system protection need to be much more widely understood. I have seen case after case in which the wrong mechanisms were used, or the right mechanisms were used in the wrong way. It has been the norm for protection to be got right only at the fifth or sixth attempt. With a slightly more informed approach, it might have been the second or third. Security professionals unfortunately tend to be either too specialized and focused on some tiny aspect of the technology, or else generalists who've never been exposed to many of the deeper technical issues. But blaming the problem on the training we currently give to students—whether of computer science, business administration, or law—is too easy; the hard part is figuring out what to do about it. This book isn't the first step, and certainly won't be the last word—but I hope it will be useful.

23.5 Summary

Sometimes the hardest part of a security engineering project is knowing when you're done. A number of evaluation and assurance methodologies are available to help. In moderation they can be very useful, especially to the start-up firm whose development culture is still fluid and is seeking to establish good work habits and build a reputation. But the assistance they can give has its limits, and overuse of bureaucratic quality control tools can do grave harm. I think of them as like salt: a few shakes on your fries can be a good thing, but a few ounces definitely are not.

But although the picture is gloomy, it doesn't justify despondency. As people gradually acquire experience of what works, what gets attacked and how, and as protection requirements and mechanisms become more part of the working engineer's skill set, things gradually get better. Security may be got right only at the fourth pass, but that's better than never—which was typical 15 years ago.

Life is chaotic. Success means coping with it. Complaining too much about it is the path to failure.

Research Problems

We could do with some new ideas on how to manage evaluation. Perhaps it's possible to apply some of the tools that economists use to deal with imperfect information, from risk-pricing models to the theory of the firm. It would also be helpful if we had better statistical tools to measure and predict failure.

Further Reading

An entire industry is devoted to promoting the assurance and evaluation biz, supported by mountains of your tax dollars. Its enthusiasm can even have the flavor of religion. Unfortunately, there are nowhere near enough people writing heresy.