CHAPTER

26

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.
— Attorney General Janet Reno [1068]

Open source is good for security because it prevents you from even trying to violate Kerckhoffs's Law.
— Eric Raymond

26.1 Introduction

I've covered a lot of material in this book, some of it quite difficult. But I've left the hardest parts to the last. These are the questions of *assurance* — whether the system will work — and *evaluation* — how you convince other people of this. How do you make a decision to ship the product, and how do you sell the safety case to your insurers?

Assurance fundamentally 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 How do you deal with people who protect the wrong thing, because their model of the requirements is out-of-date or plain wrong?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? There are many systems which 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 often used to make it more transparent.

26.2 Assurance

A working definition of *assurance* could be 'our estimate of the likelihood that a system will fail in a 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.

26.2.1 Perverse Economic Incentives

A good starting point is to look at the various principals' motives, and as a preliminary let's consider the things of which we may need assurance. Right at the start of this book, in Figure 1.1, I presented a framework for thinking rationally about security engineering based on incentives, policy, mechanism and assurance.

- *Incentives* are critical, as we've seen time and again. If people don't actually want to protect a system it's hard to make them. They fall somewhat outside the formal assurance process, but are the most critical part of the environment within which the security policy has to be defined.
- *Policy* is often neglected, as we've seen: people often end up protecting the wrong things, or protecting the right things in the wrong way. Recall from Chapter 9, for example, how the use of the Bell-LaPadula model in the healthcare environment caused more problems than it solved. We spent much of Part II of the book exploring security policies for different applications, and much of the previous chapter discussing how you'd go about developing a policy for a new application.
- *Mechanisms* have been in the news repeatedly. U.S. export controls on crypto led to products like DVD being shipped with 40-bit keys that were intrinsically vulnerable. Strength of mechanisms is independent of policy, but can interact with it. For example, 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.

- Assurance traditionally focussed on *implementation*, and was about 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 overflows, race conditions and the like. Finding and fixing them absorbs most of the effort of the assurance community.

In the last few years, since the world moved from the Orange Book to the Common Criteria, policy has come within the overall framework of assurance, through the mechanism of protection profiles. Firms that do evaluations can be asked to assess a new protection profile, just as they can be asked to verify that a particular system meets an existing one. The mechanisms aren't quite satisfactory, though, for reasons I'll discuss later.

I should mention here that the big missing factor in the traditional approach to evaluation is usability. Most system-level (as opposed to purely technical) failures have a significant human component. Usability is a cross-cutting issue in the above framework: if done properly, it has a subtle effect on policy, a large effect on choice of mechanisms, and a huge effect on how systems are tested. However, designers often see assurance simply as an absence of obvious bugs, and tie up the technical protection mechanisms without stopping to consider human frailty. (There are some honourable exceptions: bookkeeping systems are designed from the start to cope with both error and sin, while security printing technologies are often optimized to make it easier for lay people to spot forgeries.) Usability is not purely a matter for end-users, but concerns developers too. We've seen how 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.

The above factors are interdependent, and interact in many ways with the functionality of a product. The incentives are also often adrift in that the customers and the vendors want different things.

A rational PC user, for example, might want high usability, medium assurance (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 (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 (each feature benefits a more concentrated group of users than pay its costs, rapid product versioning prevents the market being commoditized, and complementary vendors who grab too much market share can be undermined), low strength of mechanisms (except for cryptography where the escrow debate led them 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 7, we described why this won't change any time soon. Companies racing for dominance in platform markets start out by shipping too little security, as it gets in the way of complementers to whom they must appeal; and such security as they do ship is often of the wrong kind, as it's designed to dump costs on users even when these could be better borne by complementers. Once a firm has achieved dominance in a platform market, it will add security, but again of the wrong kind, as its incentive is now to lock its customers in. We've seen this not just with the PC platform but with mainframes, mobile phones and even with telephone switchgear. So the vendors provide less security than a rational customer would want; and, in turn, the customer wants less than would be socially optimal, as many of the costs of attaching an insecure machine to the Internet fall on others.

Government agencies' ideals are also frustrated by economics. They would like to be able to buy commercial off-the-shelf products, replace a small number of components (such as by plugging in crypto cards to replace the standard crypto with a classified version) and just be able to use the result on defense networks. So they want multilevel, or role-based, functionality, and high implementation assurance. There is little concern with usability as the agencies (wrongly) assume that their workforce is trainable and well-disciplined. This wish list is unrealistic given not just the cost of high assurance (which I'll discuss below), 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, a million government computer users can't expect to impose their will on 500 million users of Windows and Office.

It's in this treacherous landscape with its many intractable political conflicts that the assurance game is played.

26.2.2 Project Assurance

Assurance as a process is very much like the development of code or of 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 something that's done as a one-off project, or the subject of continuous evolution. An example of the latter is given by the huge databases of known malware which anti-virus software vendors accumulate over the years to do regression testing of their products. It can lie between the two, as with the spiral model where a fixed number of prototyping stages are used in a project to help generate a whole lot of test data which weren't clear at the start.

It can also be 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 and then at the evolutionary issues.

26.2.2.1 Security Testing

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

- First look for any architectural flaws. Does the system use guessable or too-persistent session identifiers? Is there any way you can inject code, for example by sneaking SQL through a webserver into a back-end database? Is the security policy coherent, or are there gaps in between the assumptions? Do these lead to gaps in between the mechanisms where you can do wicked things?
- Then look for implementation flaws, such as stack overflows and integer overflows. This will usually involve not just looking at the code, but using specialist tools such as fuzzers.
- Then work down a list of less common flaws, such as those described in the various chapters of this book. If the product uses crypto, look for weak keys (or keys set to constant values) and poor random-number generators; if it has components with different trust assumptions, try to manipulate the APIs between them, looking for race conditions and other combinations of transactions that have evil effects.

This is an extremely telegraphic summary of one of the most fascinating and challenging jobs in the world. Many of the things the security tester needs to look for have been described at various places in this book; he should also be familiar with David Litchfield's *'Database Hacker's Handbook'*, Greg Hoglund and Gary McGraw's *'Exploiting Software'*, and Gary McGraw's '*Software Security*', at a very minimum.

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; and in some industries, such as bank inspection, there are more or less established checklists (in banking, for example, there's [114]).

26.2.2.2 Formal Methods

In Chapter 3, I presented 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 designing crypto protocols — where intuition is often inadequate, and formal verification can be helpful. Military purchasers go further, and require their use 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 [1117]. The quote by Lars 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 — as I discussed in the context of key management protocols in section 3.7.1.

26.2.2.3 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 succinctly 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*, in which a number of errors are deliberately introduced into the code at random. If there are a hundred such errors, and the tester finds seventy of them plus a further seventy that weren't deliberately introduced, then once the thirty remaining deliberate errors are removed you might expect that there are thirty bugs left that you don't know about. (This assumes that the errors you don't know about are distributed the same as the ones you do; reality will almost always be worse than this [219].)

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 8 of the first edition of this book reviewed by a fairly large number of people, as I took a draft to a conference on the topic. Given the bugs they found, and the number of people who reviewed the other chapters, I estimated that there are maybe three dozen errors of substance left in the book. Seven years and 25,000 copies later, readers had reported 99 errors (and I'd found a handful more myself while writing the second edition). Most were typos: the sixteen errors of substance discovered so far include 4 inaccurate definitions, 3 errors in formulae, one error in arithmetic, three things wrongly described, three acronyms incorrectly expanded, and two cryptographers' names wrong (Bruce Schneier, who wrote the foreword, miraculously became Prince Schneier). As the second edition adds about a third as much new material again, we might estimate that about fifteen bugs remain from the first edition and a further score have been introduced: a dozen in the new material and the rest in the course of rewriting existing chapters. (In the absence of automated regression tests, software engineers reckon that rewriting code will bring back about 20% of the old bugs.) So the second edition probably also has about three dozen bugs, but given that it's a couple of hundred pages longer I can call it a quality improvement.

So we get feedback from the rate at which instances of known bugs are discovered in products once they're fielded. 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 vulnerabilities and exploits end up in research papers that bring fame and ultimately promotion. The incentives in government agencies and corporate labs are slightly different but the overall effect is the same: a large group of capable motivated people looking for new exploits. Academics usually publish, government scientists usually don't, and corporate researchers sometimes do.

This all provides valuable input for reliability growth models. Once you have a mature product, you should have a good idea of the rate at which bugs will be discovered, the rate at which they're reported, the speed with which they're fixed, the proportion of your users who patch their systems quickly, and thus the likely vulnerability of your systems to attackers who either develop zero-day exploits of their own, or who reverse your patches looking for the holes you were trying to block.

26.2.3 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 who's done system development knows, some programmers produce code with an order of magnitude fewer bugs than others. There are also some organizations that produce much better quality code than others. This is the subject of much attention in the industry.

As I remarked in the previous chapter, some of the differences between high-quality and low-quality developers are amenable to direct management intervention. A really important factor 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). They communicated with each other only via 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, so 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 to clear up the mess. Over time, both quality and productivity sagged. Industry analysts ascribed IBM's near-death experience in the early 1990s, which cost over $100 billion in asset value, to this [273]. 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 through how you train them and the work habits you inculcate. (See Maguire for an extended discussion of the Microsoft policy [829].)

For some years, internal auditors have included process issues in 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. So recent work has striven for a more holistic assessment of a team's capability; a lead contender is the *Capability Maturity Model* (CMM) from the Software Engineering Institute at Carnegie-Mellon University.

CMM is based on the idea that competence is a function of teams rather than just individual developers. There's more to a band than just throwing together half-a-dozen competent musicians, and the same holds for software. Developers start off with different coding styles, different conventions for commenting and formatting code, different ways of managing APIs, and even different workflow rhythms. As I described in the last chapter, a capable project manager will bring them together as a team by securing agreement on conventions and ways of working, developing an understanding of different people's strengths, and matching them better to tasks. The Carnegie-Mellon research showed that newly-formed teams tended to underestimate the amount of work in a project, and also had a high variance in the amount of time they took; the teams that worked best together were much better able to predict how long they'd take, in terms of the mean development time, but reduced the variance as well.

Now one problem is that firms are forever reorganising for various reasons, and this disrupts established and capable teams. How can one push back on this, so as to maintain capability? Well, CMM offers a certification process whereby established teams may get themselves considered to be assets that are not to be lightly squandered. The details of the model are perhaps less important than this institutional role. But, in any case, it has five levels — initial, repeatable, defined, managed and optimizing — with a list of new things to be added as you go up 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'. For a fuller description and bibliography, see Hans van Vliet [1281]; there have been several attempts to adapt it to security work and a significant number of vendors have adopted it over the years [873, 1378].

An even more common process assurance approach is the ISO 9001 standard. The essence is that a company must document its processes for design, development, testing, documentation, audit and management control generally. For more detail, see [1281]; there is now a whole industry of consultants helping companies get ISO 9001 certification. At its best, this 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. But very often ISO 9001 is an exercise in ass-covering and box-ticking that merely replaces chaos by more bureaucratic chaos.

Many writers have remarked that organizations have a natural life cycle, 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 fires rejuvenate forests. It's certainly true that successful companies become complacent and bureaucratic. Many insiders opt for an easy life, while other more ambitious ones leave: it used to be 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 security and other problems of Windows products on the influx of tens of thousands of new hires in the 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 Moore's law, network externalities and a freewheeling entrepreneurial culture. The telecomms industry suffered severe trauma as the two industries merged and the phone companies' fifteen year product cycles collided with the fifteen month cycles of Cisco, Microsoft and others. The information security industry is feeling the same pressures. Teams that worked steadily for decades on cost-plus contracts to develop encryptors or MLS systems for the NSA were suddenly exposed to ferocious technological and market forces, and told to build quite different things. Some succeeded: the MLS supplier TIS reinvented itself as a firewall and antivirus vendor. Others failed and disappeared. So management may question the value of a team of MLS greybeards. And expert teams may depend on one or two key gurus; when they go off to do a startup, the team's capability can evaporate overnight.

A frequently overlooked point is that assurance schemes, like crypto protocols, should support revocation. CMM would be more powerful if teams that had lost their stars, their sparkle or their relevance also lost their ranking. Perhaps we should rather have the sort of ranking system used in food guides, where you can't declare a new establishment to be 'the best Asian restaurant in San Francisco' unless you dislodge the incumbent. 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. But by and large the restaurant guide system works, and academic peer review works somewhat along the same lines.

26.2.4 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 nowadays is packaged rather than bespoke, and is developed by continual evolutionary enhancement rather than in a one-off project. So what 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 bug-introduction rate depends on the kind of development controls already described, measuring the bug-removal rate requires different tools — models of how the reliability of software improves under testing.

There's quite a lot 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 [805]. So where the reliability of a system is dominated by a single bug — as when we're looking for the first bug in a system, or the last one — 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 documented in the bug history of IBM mainframe operating systems [9], and has been confirmed in many other studies [819]. 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' [247]. 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 defence.

The reason for the k/t behaviour emerged in [174]; colleagues and I then proved it under much more general assumptions in [219]. The model gives a number of other interesting results. Under assumptions which 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 which 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 which survive a selection process is maximised.

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 maximised, 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, and 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 then they will be selected for alertness rather than speed. The variability in speed will remain, so if foxes arrive in the neighbourhood 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 Sir Ronald Fisher proved this in 1930 [475], he was also proving that complex software will exhibit the maximum possible number of bugs when you migrate it to a new environment.

The evolutionary model also points to fundamental limits on the reliability gains to be had from re-usable 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 safety-critical systems community's observation that test results are often a poor performance indicator [805]: 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 I mentioned in section 25.4.3, it is often more economic for different testers to work on a program in parallel rather than in series.

So complex systems only become reliable following prolonged testing. The wide use of mass market software enables thorough debugging in principle, but in practice the constant new versions dictated by network economics place severe limits on what may reasonably be expected.

26.2.5 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 simplified example. Suppose a complex product such as Windows Vista has 1,000,000 bugs each with an MTBF of 1,000,000,000 hours. Suppose that Ahmed works in Bin Laden's cave where his job is to break into the U.S. Army's network to get the list of informers in Baghdad, while Brian is the army assurance guy whose job is to stop Ahmed. So he must learn of the bugs before Ahmed does.

Ahmed has to move around to avoid the Pakistani army, so he can only do 1000 hours of testing a year. Brian has full Vista 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 also runs the government's scheme to send round consultants to critical industries such as power and telecomms to find out how to hack them (pardon me, to advise them how to protect their systems). So Brian does 10,000,000 hours a year of testing.

After a year, Ahmed finds a bug, while Brian has found 10,000. But the probability that Brian has found Ahmed's bug is only 1%. Even if Brian drafts 50,000 computer science graduates to Fort Meade and sets them trawling through the Windows source code, he'll still only get 100,000,000 hours of testing done each year. After ten years he will find Ahmed's bug. But by then Ahmed 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 stopped fixing them.

In other words, Ahmed has thermodynamics on his side. Even a very moderately resourced attacker can break anything that's large and complex. There is nothing that can be done to stop this, so long as there are enough different security vulnerabilities to do statistics. In real life, vulnerabilities are correlated rather than independent; if 90% of your vulnerabilities are stack overflows, and you introduce compiler technology to trap them, then for modelling purposes there was only a single vulnerability. However, it's taken many years to sort-of-not-quite fix that particular vulnerability, and new ones come along all the time. So if you are actually responsible for Army security, you can't just rely on a large complex commercial off-the-shelf product. You have to have mandatory access controls, implemented in something like a mail guard that's simple enough to verify. Simplicity is the key to escaping the statistical trap.

26.3 Evaluation

A working definition of *evaluation* is 'the process of assembling evidence that a system meets, or fails to meet, a prescribed assurance target'. (It overlaps with testing and is sometimes confused with it.) As I mentioned above, this evidence might only be needed to convince your boss that you've completed the job. But often it is needed to reassure principals who will rely on the system. 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 fairly manageable, as when you design a burglar alarm to standards set by insurance underwriters and have it certified by inspectors at their laboratories. Sometimes it's still visible but more complex, as when designing to government security standards which 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 can be an emergent property that arises from managers following their own personal and departmental incentives.

For example, managers often buy products and services that they know to be suboptimal or even defective, but which are from big name suppliers — just to minimize the likelihood of getting fired when things go wrong. (It used to be said 20 years ago that 'no-one ever got fired for buying IBM'.) Corporate lawyers don't condemn this as fraud, but praise it as due diligence. The end result may be that someone who relies on a system — in this case, the bank's customer — has no say, 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 narrow technical aspects of the system, and ignore system issues like usability — not to mention organizational issues such as appropriate internal control and good corporate governance. Company directors also want assurance: that the directed 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 a matter for a system administrator setting up configuration files.

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 critical 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.

26.3.1 Evaluations by the Relying Party

In Chapter 11, I discussed the concerns insurers have with physical security systems, and how they go about approving equipment for use with certain sizes of risk. The approval process itself is 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 (for example) 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.

The main failure mode of this process is that it doesn't always respond well enough to progress in attack technology. In the case of high-security locks, a lab may demand ten minutes' resistance to picking and say nothing about bumping. Yet bumping tools have improved enough to be a major threat, and picks have got better too. A product that got its certificate ten years ago might now be easy to bump (so the standard's out of date) and also be vulnerable to picking within 2–3 minutes (so the test result is too). Insurance labs in some countries, such as Germany, have been prepared to withdraw certifications as attacks got better; in other countries, like the USA, they're reluctant to do so, perhaps for fear of being sued.

In section 8.4, I mentioned another model of evaluation — that are done from 1985–2000 at the U.S. National Computer Security Center on computer security products proposed for government use. These evaluations were conducted according to the *Orange Book* — the Trusted Computer Systems Evaluation Criteria [375]. 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, and to Windows NT. (Both of these were conditional on a particular 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. Labelling 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, it must be tamper-resistant, and it must 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 determined what spread of information could be processed on it. The example I gave in section 8.6.2 was that a system evaluated to B3 may process information at Unclassified, Confidential and

Secret, or at Confidential, Secret and Top Secret. The complete rule set can be found in [379].

When the Orange Book was written, the Department of Defense thought the real problem was that markets for defense computing equipment were too small, leading to high prices. The solution was to expand the market for high-assurance computing. So the goal of was to develop protection measures that would be standard in all major operating systems, rather than an expensive add-on for captive government markets.

However, the business model of Orange Book evaluations followed traditional government work practices. A government user would want some product evaluated; the NSA would allocate people to do it; given traditional civil service caution and delay, this could take two or three years; the product, if successful, would join the evaluated products list; and the bill was 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. This meant that evaluated products were just not acceptable in commercial markets. The defense computing market stayed small, and prices stayed high.

The Orange Book wasn't the only evaluation scheme running in America. I mentioned in section 16.4 the FIPS 140-1 scheme for assessing the tamper-resistance of cryptographic processors; this uses a number of independent laboratories as contractors. 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; they would no longer have to have separate certification in Britain, France, 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 — that the evaluation was not paid for by the government but by the vendor seeking an evaluation on its product. This was an attempt to kill several birds with one stone: saving public money while promoting a more competitive market. However, the incentive issues were not properly thought through.

This change in the rules motivated the vendor to shop around for the evaluation contractor who would give his product the easiest ride, whether by asking fewer questions, charging less money, taking the least time, or all of the above[1]. 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 supposed to offset the commercial pressures to cut corners.

26.3.2 The Common Criteria

This sets the stage for the Common Criteria. The Defense Department began to realise that the Orange Book wasn't making procurement any easier, and contractors detested having to obtain separate evaluations for their products in the USA, Canada and Europe. Following the collapse of the Soviet Union in 1989, budgets started being cut, and it wasn't clear where the capable motivated opponents of the future would come from. The mood was in favour of radical cost-cutting. Eventually agreement was reached to scrap the national evaluation schemes and replace them with a single standard; the Common Criteria for Information Technology Security Evaluation [935].

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 at all but the highest levels are done by CLEFs and are supposed to be recognised 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*. There are protection profiles for operating systems, access control systems, boundary control devices, intrusion detection systems, smartcards, key management systems, VPN clients, and even waste-bin identification systems — for transponders that identify when a domestic bin was last emptied. The tent is certainly a lot broader than with the Orange Book. However, anyone can propose a protection profile and have it evaluated by the lab of his choice, and so there are a lot of profiles. It's not that Department of Defense has abandoned multilevel security, so much as tried to get commercial IT vendors to use the system for other purposes too, and thus defeat the perverse incentives described above. The aspiration was to create a

[1]The same may happen with FIPS 140-1 now that commercial companies are starting to rely on it for third-party evaluations.

bandwagon effect that would result in the commercial world adapting itself somewhat to the government way of doing things.

Its success has been mixed. The only major case I can recall of a large firm using evaluations in its general marketing was Oracle, which started a marketing campaign after 9/11 describing its products as 'Unbreakable', claiming that each of its fourteen security evaluations 'represented an extra million dollars' investment in security [989]. (This campaign ran out of steam after a number of people found ways to break their products.) But there have been quite a few cases of evaluations being used in more specialised markets; I already discussed bank card terminals (which turned out to be easy to break) in Chapter 10. The smartcard industry has been a particular fan of evaluation: in 2007, there are no less than 26 smartcard protection profiles, and a number of derivative profiles for smartcard-based devices such as TPMs and electronic signature creation devices. But perhaps it's worked too well. There's a big choice and it's not easy to understand what an evaluation certificate actually means. (We came across this problem in Chapter 16: some cryptographic processors were certified secure as hardware devices but broken easily as they were used to run inappropriate software.)

To discuss the Common Criteria in detail, we need 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) and can range from EAL 1, for which functional testing is sufficient, all the way up to EAL7 for which not only thorough testing is required but a formally verified design. The highest evaluation level commonly obtained for commercial products is EAL4, although there was one smartcard operating system at EAL6.

A protection profile consists of security requirements, their rationale, and an EAL. It's 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. As well as evaluating a specific target, one can evaluate a protection profile (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 end result of all this is 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 non-repudiation 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 means access control, Bell-LaPadula information flow controls, and related properties).

There are 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 mechanism.

The fastest way to get the hang of this is probably to read a few of the existing profiles. You will realise that the quality varies quite widely. For example, the Eurosmart protection profile for smartcards relates largely to maintaining confidentiality of the chip design by imposing NDAs on contractors, shredding waste and so on [448], while in practice most attacks on smartcards used probing or power-analysis attacks for which knowledge of the chip mask was not relevant. Another example of an unimpressive protection profile is that for automatic cash dispensers, which is written in management-speak, complete with clip art, 'has elected not to include any security policy' and misses even many of the problems that were well known and documented when it was written in 1999. Indeed it states that it relies on the developer to document vulnerabilities and demands that 'the evaluator shall determine that the TOE is resistant to penetration attacks performed by an attacker possessing a moderate attack potential' [240]. A better example is the more recent profile devised by the German government for health professional cards — the cards used by doctors and nurses to log on to hospital computer systems [241]. This goes into great detail about possible threats involving physical tampering, power analysis, functionality abuse, and so on, and ties the protection mechanisms systematically to the control objectives.

So the first question you should ask when told that some product has a Common Criteria Evaluation is: 'against what protection profile?' Is it a secret design that will resist a 'moderate attack', or has it been evaluated against something thorough that's been thought through by people who know what they're doing? The interesting thing is that all the three profiles I mentioned in the above paragraph are for evaluation to level EAL4+ — so it's not the nominal CC level that tells you anything, but the details of the PP.

The Criteria do say that 'the fact that an IT product has been evaluated has meaning only in the context of the security properties that were evaluated and the evaluation methods that were used', but it goes on to say both that 'Evaluation authorities should carefully check the products, properties and methods to determine that an evaluation will provide meaningful results' and 'purchasers of evaluated products should carefully consider this context to determine whether the evaluated product is useful and applicable to their specific situation and needs.' So who's actually liable? Well, we find that 'the CC does not address the administrative and legal framework under which the criteria may be applied by evaluation authorities' and that 'The procedures for use of evaluation results in accreditation are outside the scope of the CC'.

The Common Criteria can sometimes be useful to the security engineer in that they give you an extensive list of things to check, and the framework can also help in keeping track of all the various threats and ensuring that they're all dealt with systematically (otherwise it's very easy for one to be forgotten in the mass of detail and slip through). What people often do, though, is to find a protection profile that seems reasonably close to what they're trying to do and just use it as a checklist without further thought. If you pick a profile that's a poor match, or one of the older and vaguer ones, you're asking for trouble.

26.3.3 What the Common Criteria Don't Do

It's also important to understand the Criteria's limitations. The documents claim that they don't deal with administrative security measures, nor 'technical physical' aspects such as Emsec, nor crypto algorithms, nor the evaluation methodology (though there's a companion volume on this), nor how the standards are to be used. They 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 re-evaluation 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: things like usability are almost ignored, and the interaction of a firm's administrative procedures with the technical controls is glossed over as outside the scope.

Another common criticism is that the Criteria don't cope well with change. A lot of commercial products have an evaluation that's essentially meaningless: operating systems such as Windows and Linux have been evaluated, but in very restricted configurations (typically, a workstation with no network connection or removable media — where all the evaluation is saying is that there's a logon process and that filesystem access controls work). Products with updates, such as Windows with its monthly security patches, are outside the scope. This can lead to insecurity directly: I remarked that the first real distributed denial-of-service attack used hospital PCs, as the evaluation and accreditation process in use at the time compelled hospitals to use an obsolete and thus unsafe configuration.

It's actually very common, when you read the small print of the PP, to find that only some very basic features of the product have been evaluated. The vendor might have evaluated the boo code, but left most of the operating system outside the scope. This applies not just to CC evaluations, but to FIPS-140 as well; in the chapter on API Security I told how the IBM 4758 was evaluated by the U.S. government to the highest standard of physical tamper-resistance, and yet broken easily. The evaluators looked only at the tamper-resistant platform, while the attackers also looked at the software that ran on it.

Another fundamental problem is that the Criteria are technology-driven, when in most applications it's the business processes that should drive protection decisions. 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.

Over the last decade, I must have involved in half-dozen disputes about whether protection mechanisms were properly designed or implemented, and in which the Common Criteria were in some way engaged. In not one of these has their approach proved satisfactory. (Perhaps that's because I only get called in when things go wrong — but my experience still indicates a lack of robustness in the process.) There are not just the technical limitations, and scope limitations; there are also perverse incentives and political failures.

26.3.3.1 Corruption, Manipulation and Inertia

Common Criteria evaluations are done by CLEFS — contractors who are 'Commercial Licensed Evaluation Facilities'. They are licensed by the local signals intelligence agency (the NSA in America, or its counterparts in the UK, Canada, France, Germany, Spain, the Netherlands, Australia, New Zealand and Japan). One consequence is that their staff must all have clearances. As a result, the CLEFs are rather beholden to the local spooks.

One egregious example in my experience occurred in the British National Health Service. The service had agreed, under pressure from doctors, to encrypt traffic on the health service network; the signals intelligence service GCHQ made clear that it wanted key escrow products 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 UK defence 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 retrieved by having a company with a CLEF license evaluate the trials. In their report, they 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; or perhaps the staff wrote what they knew GCHQ wanted to read.

A second problem is that, as it's the vendor who pays the CLEF, the vendor can shop around for a CLEF that will give it an easy ride technically, or that will follow its political line. In the context of the Icelandic health database I discussed in section 9.3.4.1 above, its promoters wished 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 [51].

Sometimes the protection profiles might be sound, but the way they're mapped to the application isn't. For example, smartcard vendors lobbied European governments to pass laws forcing business to recognise digital signatures made using smartcards; a number of protection profiles were duly written for a smartcard to function as a 'Secure Signature-Creation Device'. But the main problem in such an application is the PC that displays to the citizen the material she thinks she's signing. As that problem's too hard, it's excluded, and 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.

Electronic signatures didn't take off; no sensible person would agree to be bound by any signature that appeared to have been made by her smartcard, regardless of whether she actually made it. (By comparison, in the world of paper, a forged manuscript signature is completely null and void, and it can't bind anyone.) In this case, the greed of the vendors and the naivete of the legislators destroyed the market they hoped to profit from.

Insiders figure out even more sophisticated ways to manipulate the system. A nice example here comes from how the French circumvented British and German opposition to the smartcard based electronic tachograph described in section 12.3. The French wrote a relaxed protection profile and sent it to a British CLEF to be evaluated. The CLEF was an army software company and, 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 UK government was opposed to approval of the protection profile (if they had done, they'd have no doubt toed the line). So Britain was left with a choice between accepting defective road safety standards as a fait accompli, or undermining confidence in the Common Criteria. In the end, no-one in London had the stomach to challenge the evaluation; eight years later, the old paper tachographs started being replaced with the new, less secure, electronic ones.

As for the organizational aspects, I mentioned in section 26.2.3 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 will 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 both corruption and grade inflation. In the first edition of this book in 2001, I wrote: '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 misbehaviour, the incentive will be for CLEFs to race to the bottom and compete to give vendors an easy ride'.

I described in section 10.6.1.1 how, in late 2007, colleagues and I discovered that PIN entry devices certified as secure by both VISA and the Common Criteria were anything but secure; indeed, the protection profile against which they'd been evaluated was unmeetable. After reporting the vulnerability to the vendors, to VISA, to the bankes' trade association and to GCHQ, we pointed out that this was a failure not just of the labs that had evaluated those particular devices, but that there was a systemic failure. What, we demanded, did VISA and GCHQ propose to do about it? The answer, it turned out, was nothing. VISA refused to withdraw its certification, and the UK government told us that as the evaluation had not been registered with them, it wasn't their problem. The suppliers are free to continue describing a defective terminal as 'CC evaluated' and even 'CC Approved', so long as they don't call it 'CC certified' or 'certified under the Common Criteria Recognition Arrangement (CCRA)'. This strikes me as a major problem with the CC brand. It will continue to be stamped on lousy products for the foreseeable future.

So the best advice I can offer is this. When presented with a security product, you must always consider whether the salesman is lying or mistaken, and how. The Common Criteria were supposed to fix this problem, but they don't. When presented with an evaluated product, you have to demand what vulnerabilities have been reported or discovered since the evaluation took place. (Get it in writing.) Then look hard at the protection profile: check whether it maps to what you really need. (Usually it doesn't — it protects the vendor, not you.) Don't limit your scepticism to the purely technical aspects: ask how it was manipulated and by whom; whether the CLEF that evaluated the profile was dishonest or incompetent; and what pressure from which government might have been applied behind the scenes.

You should also consider how your rights are eroded by the certificate. For example, if you use an unevaluated product to generate digital signatures, and a forged signature turns up which someone tries to use 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 prejudice your rights.

A cynic might suggest that this is precisely why, in the commercial world, it's the vendors of products that are designed to transfer liability (such as smartcards), or to satisfy due diligence requirements (such as firewalls) 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 which is sold worldwide may compromise 250 million Americans, but as it will also compromise 400 million Europeans the balance of advantage lies in deception. The balance is even stronger with small countries such as Britain and the Netherlands, who have fewer citizens to protect and more foreigners to attack. In addition, agencies get brownie points (and budget) for foreign secrets they steal, not for local secrets that foreigners didn't manage to steal.

So an economist is unlikely to trust a Common Criteria evaluation. Perhaps I'm just a cynic, 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 whacking people round the head. (Just don't try to put serious weight on it!)

Fortunately, the economics discussed in section 26.2.1 should also limit the uptake of the Criteria to sectors where an official certification, however irrelevant, erroneous or mendacious, offers some competitive advantage.

26.4 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 [231]. The easy parts of the problem, such as developing high-level languages in which programmers are more productive, have been done. That removes much of the accidental complexity of programming, leaving the intrinsic complexity of the application. I discussed this in the previous chapter in the general context of system development methodology; the above discussion should convince the reader that exactly the same applies to the problem of assurance and, especially, 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 use. Usability is ignored by the Common Criteria, but is in reality all important; a UK 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 we described in the context of bookkeeping systems in Chapter 10, which are designed to limit the effects of human frailty, are also critical. In most applications, one 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. 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, in order to provide purchasers with the illusion of stability. But given the way the industry works, the best people can do is flock to brands, such as IBM in the 70s and 80s, and Microsoft now. The establishment and maintenance of these brands involves huge market forces, and security plays little role.

I've probably given you enough hints by now about how to cheat the system and pass off a lousy system 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 want risk reduction, rather than due diligence or some other kind of liability dumping. There are still many systems where the system owner loses if the security fails; we've seen a number of them above (nuclear command and control, pay-TV, prepayment utility meters, . . .) and they provide many interesting engineering examples.

26.4.1 Hostile Review

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 we've seen 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.

The classic ways of doing hostile review are contractual and conflictual. An example of the contractual approach was the Independent Validation and Verification (IV&V) program used by NASA for manned space flight; contractors were hired to trawl through the code and paid a bonus for every bug they found. An example of the conflictual approach was in the evaluation of nuclear command and control, where Sandia National Laboratories and the NSA vied to find bugs in each others' designs.

One way of combining the two is simply to hire multiple experts from different consultancy firms or universities, and give the repeat business to whoever most noticeably finds bugs and improves the design. Another is to have multiple different accreditation bodies: I mentioned in section 23.5 how voting systems in the USA 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.

26.4.2 Free and Open-Source Software

The free and open-source software movement extends the philosophy of openness from the architecture to the implementation detail. Many security products have publicly available source code, of which the first was probably 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 has recently been swinging back, and IBM is one of the stalwarts of open source.

There are a number of strong arguments in favour of open software, and a few against. First, if everyone in the world can inspect and play with the software, then bugs are likely to be found and fixed; in Raymond's famous phrase, 'To many eyes, all bugs are shallow' [1058]. 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.

A standard defense-contractor argument against open source is that once software becomes large and complex, there may be few or no capable motivated people studying it, and major vulnerabilities may take years to be found. For example, a programming bug in PGP versions 5 and 6 allowed an attacker to add an extra escrow key without the key holder's knowledge [1143]; this was around for years before it was spotted. There have also been back door 'maintenance passwords' in products such as `sendmail` that persisted for years before they were removed.

The worry 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. First, there many not be enough reviewers for many open products, as the typical volunteer finds developing code more rewarding than finding exploits. A lot of open-source development is done by students who find it helps them get good jobs later if they can point to some component of an employer's systems which they helped develop; perhaps it wouldn't be so helpful if all they could point to was a collection of bug reports that forced security upgrades. Second, as I noted in section 26.2.4, different testers find different bugs as 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 invested a mere 1000 hours might find a new vulnerability. Given the cited reliability growth models, the probabilities are easy enough to work out.

Other arguments 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 such projects can fail to achieve 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 [1059] 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.

Some people have argued that while openness helps the defenders find bugs so they can fix them, it will also help the attackers find bugs so they can exploit them. Will the attackers or the defenders be helped more? In 2002 I proved that, under the standard model of reliability growth, openness helps attack and defence equally [54]. Thus whether an open or proprietary approach works best in a given application will depend on whether and how that application departs from the standard assumptions, for example, of independent vulnerabilities. As an example, a study of security bugs found in the OpenBSD operating system revealed that these bugs were significantly correlated, which suggests that openness there was a good thing [998].

In fact there's a long history of security engineers in different disciplines being converted to openness. The long-standing wisdom of Auguste Kerckhoffs was that cryptographic systems should be designed in such a way that they are not compromised if the opponent learns the technique being used [713]. The debate about whether locksmiths should discuss vulnerabilities in locks started in Victorian times, as I discussed in the Chapter 11. The law-and-economics scholar Peter Swire has explained why governments are intrinsically less likely to embrace disclosure: although competitive forces drive even Microsoft to open up a lot of its software for interoperability and trust reasons, government agencies play different games (such as expanding their budgets and avoiding embarrassment) [1238]. Yet the security arguments have started to prevail in some quarters: from tentative beginnings in about 1999, the U.S. Department of Defense has started to embrace open source, notably through the SELinux project I discussed in Chapter 8.

So while an open 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 what you built — and whether they tell you everything they find.

26.4.3 Semi-Open Design

Where a fully open design isn't possible, you can often still get benefits by opting for a partly-open one. For example, the architectural design could be published even although some of the implementation details are not. Examples that we've seen include the smartcard banking protocol from section 3.8.1, the nuclear command and control systems mentioned in Chapter 13 and the SPDC mechanism in Blu-Ray.

Another approach to semi-open design is to use an open platform and build proprietary components on top. The best-known example here may be Apple's OS/X which combines the OpenBSD operating system with proprietary multimedia components. In other applications, a proprietary but widely-used product such as Windows or Oracle may be 'open enough'. Suppose, for example, you're worried about a legal attack. If there's an argument in court about whether the system was secure at the time of a disputed transaction, then rather than having opposing experts trawling through code, you can rely on the history of disclosed vulnerabilities, patches and attacks. Thus we find that more and more ATMs are using Windows as a platform (although the version they use does have a lot of the unnecessary stuff stripped out).

26.4.4 Penetrate-and-Patch, CERTs, and Bugtraq

Penetrate-and-patch was 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. As I discussed in Chapter 8, the hope at that time was that formal methods would enable bug-free systems to be constructed. It's now well known that formal verification only works for systems that are too small and limited for most applications. Like it or not, most software development is iterative, based on the build cycle discussed in Chapter 22. Iterative approaches to assurance are thus necessary, and the question is how to manage them.

The interests of the various stakeholders in a system can diverge quite radically at this point.

1. The vendor would prefer that bugs weren't found, to spare the expense of patching.
2. The average customer might prefer the same; lazy customers often don't patch, and get infected as a result. (So long as their ISP doesn't cut them off for sending spam, they may not notice or care.)
3. The typical security researcher wants a responsible means of disclosing his discoveries, so he can give the vendors a reasonable period of time to ship a patch before he ships his conference paper; so he will typically send a report to a local *computer emergency response team* (CERT) which in turn will notify the vendor and publish the vulnerability after 45 days.
4. The intelligence agencies want to learn of vulnerabilities quickly, so that they can be exploited until a patch is shipped. (Many CERTs are funded by the agencies and have cleared personnel.)
5. Some hackers disclose vulnerabilities on mailing lists such as `bugtraq` which don't impose a delay; this can force software vendors to ship emergency patches out of the usual cycle.
6. The security software companies benefit from the existence of unpatched vulnerabilities in that they can use their firewalls to filter for attacks using them, and the anti-virus software on their customers' PCs can often try to intercept such attacks too. (Symantec hosts the bugtraq list.)

7. Large companies don't like emergency patches, and neither do most government departments, as the process of testing a new patch against the enterprise's critical systems and rolling it out is expensive.

During the 1990s, the debate was driven by people who were frustrated at the software vendors for leaving their products unpatched for months or even years. This was one of the reasons the bugtraq list was set up; it then led to a debate on 'responsible disclosure' with various proposals about how long a breathing space the researcher should give the vendor [1055]. The CERT system of a 45-day delay emerged from this. It gives vendors a strong incentive to have an attentive bug reporting facility; in return they get enough time to test a fix properly before releasing it; researchers get credits to put on their CVs; and users get bug fixes at the same time as bug reports; and the big companies have regular updates or service packs for which their corporate customers can plan.

Is this system the best we could get? Recently, the patch cycle has become a subject of study by security economists. There was a debate at the 2004 Workshop on the Economics of Information Security between Eric Rescorla, who argued that since bugs are many and uncorrelated, and since most exploits use vulnerabilities reverse-engineered from existing patches, there should be minimal disclosure. Ashish Arora argued that, from both theoretical and empirical perspectives, the threat of disclosure was needed to get vendors to patch. I discussed this argument in section 7.5.2. There has been some innovation, notably the introduction of essentially automatic upgrades for mass-market users, and the establishment of firms that make markets in vulnerabilities; and we have some more research data, notably the fact that bugs in OpenBSD and some other systems are correlated. By and large, the current way of doing things seems reasonable and stable.

26.4.5 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 described 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, when 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.

26.5 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 which 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 aren't.

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 only be got right at the fourth pass, but that's better than never — which was typical fifteen years ago.

Life is complex. 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. At present, we use vastly different — and quite incompatible — tools to measure the level of protection that can be expected from cryptographic algorithms; from technical physical protection mechanisms; from complex software; from the internal controls in an organisation; and, via usability studies, from people. Is there any way we can join these up, so that stuff doesn't fall down between the gaps? Can we get better mechanisms than the Common Criteria, which vendors privately regard as pointless bureaucracy? 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 even be helpful if we had better statistical tools to measure and predict failure. We should also tackle some taboo subjects. Why did the Millennium Bug not bite?

Further Reading

There is a whole industry devoted to promoting the assurance and evaluation biz, supported by mountains of your tax dollars. Their enthusiasm can even have the flavour of religion. Unfortunately, there are nowhere near enough people writing heresy.