may indicate that an exception has occurred so that a calling component is aware of the problem.

3. Pass control to a run-time support system that handles the exception. This is often the default when faults occur in a program (e.g., when a numeric value overflows). The usual action of the run-time system is to halt processing. You should only use this approach when it is possible to move the system to a safe and quiescent state, before handing control to the run-time system.

Handling exceptions within a program makes it possible to detect and recover from some input errors and unexpected external events. As such, it provides a degree of fault tolerance—the program detects faults and can take action to recover from these. As most input errors and unexpected external events are usually transient, it is often possible to continue normal operation after the exception has been processed.

Guideline 4: Minimize the use of error-prone constructs

Faults in programs, and therefore many program failures, are usually a consequence of human error. Programmers make mistakes because they lose track of the numerous relationships between the state variables. They write program statements that result in unexpected behavior and system state changes. People will always make mistakes, but in the late 1960s it became clear that some approaches to programming were more likely to introduce errors into a program than others.

Some programming language constructs and programming techniques are inherently error prone and so should be avoided or, at least, used as little as possible. Potentially error-prone constructs include:

1. *Unconditional branch (go-to) statements* The dangers of go-to statements were recognized as long ago as 1968 (Dijkstra, 1968) and, as a consequence, these have been excluded from modern programming languages. However, they are still allowed in languages such as C. The use of go-to statements leads to ‘spaghetti code’ that is tangled and difficult to understand and debug.

2. *Floating-point numbers* The representation of floating-point numbers in a fixed-length memory word is inherently imprecise. This is a particular problem when numbers are compared because representation imprecision may lead to invalid comparisons. For example, 3.00000000 may sometimes be represented as 2.99999999 and sometimes as 3.00000001. A comparison would show these to be unequal. Fixed-point numbers, where a number is represented to a given number of decimal places, are generally safer because exact comparisons are possible.

3. *Pointers* Programming languages such as C and C++ support low-level constructs called pointers, which hold addresses that refer directly to areas of the machine memory (they point to a memory location). Errors in the use of pointers can be devastating if they are set incorrectly and therefore point to the wrong

area of memory. They also make bound checking of arrays and other structures harder to implement.

4. *Dynamic memory allocation* Program memory may be allocated at run-time rather than at compile-time. The danger with this is that the memory may not be properly deallocated, so eventually the system runs out of available memory. This can be a very difficult error to detect because the system may run successfully for a long time before the problem occurs.

5. *Parallelism* When processes are executing concurrently, there can be subtle timing dependencies between them. Timing problems cannot usually be detected by program inspection, and the peculiar combination of circumstances that cause a timing problem may not occur during system testing. Parallelism may be unavoidable, but its use should be carefully controlled to minimize interprocess dependencies.

6. *Recursion* When a procedure or method calls itself or calls another procedure, which then calls the original calling procedure, this is 'recursion'. The use of recursion can result in concise programs; however it can be difficult to follow the logic of recursive programs. Programming errors are therefore more difficult to detect. Recursion errors may result in the allocation of all the system's memory as temporary stack variables are created.

7. *Interrupts* These are a means of forcing control to transfer to a section of code irrespective of the code currently executing. The dangers of this are obvious; the interrupt may cause a critical operation to be terminated.

8. *Inheritance* The problem with inheritance in object-oriented programming is that the code associated with an object is not all in one place. This makes it more difficult to understand the behavior of the object. Hence, it is more likely that programming errors will be missed. Furthermore, inheritance, when combined with dynamic binding, can cause timing problems at run-time. Different instances of a method may be bound to a call, depending on the parameter types. Consequently, different amounts of time will be spent searching for the correct method instance.

9. *Aliasing* This occurs when more than one name is used to refer to the same entity in a program; for example, if two pointers with different names point to the same memory location. It is easy for program readers to miss statements that change the entity when they have several names to consider.

10. *Unbounded arrays* In languages like C, arrays are simply ways of accessing memory and you can make assignments beyond the end of an array. The run-time system does not check that assignments actually refer to elements in the array. Buffer overflow, where an attacker deliberately constructs a program to write memory beyond the end of a buffer that is implemented as an array, is a known security vulnerability.

11. *Default input processing* Some systems provide a default for input processing, irrespective of the input that is presented to the system. This is a security loophole

that an attacker may exploit by presenting the program with unexpected inputs that are not rejected by the system.

Some standards for safety-critical systems development completely prohibit the use of these constructs. However, such an extreme position is not normally practical. All of these constructs and techniques are useful, though they must be used with care. Wherever possible, their potentially dangerous effects should be controlled by using them within abstract data types or objects. These act as natural 'firewalls' limiting the damage caused if errors occur.

Guideline 5: Provide restart capabilities

Many organizational information systems are based around short transactions where processing user inputs takes a relatively short time. These systems are designed so that changes to the system's database are only finalized after all other processing has been successfully completed. If something goes wrong during processing, the database is not updated and so does not become inconsistent. Virtually all e-commerce systems, where you only commit to your purchase on the final screen, work in this way.

User interactions with e-commerce systems usually last a few minutes and involve minimal processing. Database transactions are short, and are usually completed in less than a second. However, other types of systems such as CAD systems and word processing systems involve long transactions. In a long transaction system, the time between starting to use the system and finishing work may be several minutes or hours. If the system fails during a long transaction, then all of the work may be lost. Similarly, in computationally intensive systems such as some e-science systems, minutes or hours of processing may be required to complete the computation. All of this time is lost in the event of a system failure.

In all of these types of systems, you should provide a restart capability that is based on keeping copies of data that is collected or generated during processing. The restart facility should allow the system to restart using these copies, rather than having to start all over from the beginning. These copies are sometimes called checkpoints. For example:

1. In an e-commerce system, you can keep copies of forms filled in by a user and allow them to access and submit these forms without having to fill them in again.
2. In a long transaction or computationally intensive system, you can automatically save data every few minutes and, in the event of a system failure, restart with the most recently saved data. You should also allow for user error and provide a way for users to go back to the most recent checkpoint and start again from there.

If an exception occurs and it is impossible to continue normal operation, you can handle the exception using backward error recovery. This means that you reset the state of the system to the saved state in the checkpoint and restart operation from that point.

Guideline 6: Check array bounds

All programming languages allow the specification of arrays—sequential data structures that are accessed using a numeric index. These arrays are usually laid out in contiguous areas within the working memory of a program. Arrays are specified to be of a particular size, which reflects how they are used. For example, if you wish to represent the ages of up to 10,000 people, then you might declare an array with 10,000 locations to hold the age data.

Some programming languages, such as Java, always check that when a value is entered into an array, the index is within that array. So, if an array A is indexed from 0 to 10,000, an attempt to enter values into elements A [-5] or A [12345] will lead to an exception being raised. However, programming languages such as C and C++ do not automatically include array bound checks and simply calculate an offset from the beginning of the array. Therefore, A [12345] would access the word that was 12345 locations from the beginning of the array, irrespective of whether or not this was part of the array.

The reason why these languages do not include automatic array-bound checking is that this introduces an overhead every time the array is accessed. The majority of array accesses are correct so the bound check is mostly unnecessary and increases the execution time of the program. However, the lack of bound checking leads to security vulnerabilities, such as buffer overflow, which I discuss in Chapter 14. More generally, it introduces a vulnerability into the system that can result in system failure. If you are using a language that does not include array-bound checking, you should always include extra code that ensures the array index is within bounds. This is easily accomplished by implementing the array as an abstract data type, as I have discussed in Guideline 1.

Guideline 7: Include timeouts when calling external components

In distributed systems, components of the system execute on different computers and calls are made across the network from component to component. To receive some service, component A may call component B. A waits for B to respond before continuing execution. However, if component B fails to respond for some reason, then component A cannot continue. It simply waits indefinitely for a response. A person who is waiting for a response from the system sees a silent system failure, with no response from the system. They have no alternative but to kill the waiting process and restart the system.

To avoid this, you should always include timeouts when calling external components. A timeout is an automatic assumption that a called component has failed and will not produce a response. You define a time period during which you expect to receive a response from a called component. If you have not received a response in that time, you assume failure and take back control from the called component. You can then attempt to recover from the failure or tell the system user what has happened and allow them to decide what to do.

Guideline 8: Name all constants that represent real-world values

All non-trivial programs include a number of constant values that represent the values of real-world entities. These values are not modified as the program executes. Sometimes, these are absolute constants and never change (e.g., the speed of light) but more often they are values that change relatively slowly over time. For example, a program to calculate personal tax will include constants that are the current tax rates. These change from year to year and so the program must be updated with the new constant values.

You should always include a section in your program in which you name all real-world constant values that are used. When using the constants, you should refer to them by name rather than by their value. This has two advantages as far as dependability is concerned:

1. You are less likely to make mistakes and use the wrong value. It is easy to mistype a number and the system will often be unable to detect a mistake. For example, say a tax rate is 34%. A simple transposition error might lead to this being mistyped as 43%. However, if you mistype a name (such as Standard-tax-rate), this is usually detected by the compiler as an undeclared variable.
2. When a value changes, you do not have to look through the whole program to discover where you have used that value. All you need do is to change the value associated with the constant declaration. The new value is then automatically included everywhere that it is needed.

KEY POINTS

- Dependability in a program can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault-tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- The use of redundancy and diversity in hardware, software processes, and software systems is essential to the development of dependable systems.
- The use of a well-defined, repeatable process is essential if faults in a system are to be minimized. The process should include verification and validation activities at all stages, from requirements definition through to system implementation.
- Dependable system architectures are system architectures that are designed for fault tolerance. There are a number of architectural styles that support fault tolerance including protection systems, self-monitoring architectures, and N-version programming.
- Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.

- Dependable programming relies on the inclusion of redundancy in a program to check the validity of inputs and the values of program variables.
- Some programming constructs and techniques, such as go-to statements, pointers, recursion, inheritance, and floating-point numbers, are inherently error prone. You should try to avoid these constructs when developing dependable systems.

FURTHER READING

Software Fault Tolerance Techniques and Implementation. A comprehensive discussion of techniques to achieve software fault tolerance and fault-tolerant architectures. The book also covers general issues of software dependability. (L. L. Pullum, Artech House, 2001.)

'Software Reliability Engineering: A Roadmap'. This survey paper by a leading researcher in software reliability summarizes the state of the art in software reliability engineering as well as discussing future research challenges. (M. R. Lyu, *Proc. Future of Software Engineering*, IEEE Computer Society, 2007.) http://dx.doi.org/10.1109/FOSE.2007.24.

EXERCISES

13.1. Give four reasons why it is hardly ever cost effective for companies to ensure that their software is free of faults.

13.2. Explain why it is reasonable to assume that the use of dependable processes will lead to the creation of dependable software.

13.3. Give two examples of diverse, redundant activities that might be incorporated into dependable processes.

13.4. What is the common characteristic of all architectural styles that are geared to supporting software fault tolerance?

13.5. Imagine you are implementing a software-based control system. Suggest circumstances in which it would be appropriate to use a fault-tolerant architecture, and explain why this approach would be required.

13.6. You are responsible for the design of a communications switch that has to provide 24/7 availability, but which is not safety-critical. Giving reasons for your answer, suggest an architectural style that might be used for this system.

13.7. It has been suggested that the control software for a radiation therapy machine, used to treat patients with cancer, should be implemented using N-version programming. Comment on whether or not you think this is a good suggestion.

13.8. Give two reasons why different versions of a system based around software diversity may fail in a similar way.

13.9. Explain why you should explicitly handle all exceptions in a system that is intended to have a high level of availability.

13.10. The use of techniques for the production of safe software, as discussed in this chapter, obviously includes considerable extra costs. What extra costs can be justified if 100 lives would be saved over the 15-year lifetime of a system? Would the same costs be justified if 10 lives were saved? How much is a life worth? Do the earning capabilities of the people affected make a difference to this judgment?

REFERENCES

Avizienis, A. (1985). 'The N-Version Approach to Fault-Tolerant Software'. *IEEE Trans. on Software Eng.*, **SE-11** (12), 1491–501.

Avizienis, A. A. (1995). 'A Methodology of N-Version Programming'. In *Software Fault Tolerance*. Lyu, M. R. (ed.). Chichester: John Wiley & Sons. 23–46.

Boehm, B. (2002). 'Get Ready for Agile Methods, With Care'. *IEEE Computer,* **35** (1), 64–9.

Brilliant, S. S., Knight, J. C. and Leveson, N. G. (1990). 'Analysis of Faults in an N-Version Software Experiment'. *IEEE Trans. On Software Engineering,* **16** (2), 238–47.

Dijkstra, E. W. (1968). 'Goto statement considered harmful'. *Comm. ACM.*, **11** (3), 147–8.

Hatton, L. (1997). 'N-version design versus one good version'. *IEEE Software,* **14** (6), 71–6.

Knight, J. C. and Leveson, N. G. (1986). 'An experimental evaluation of the assumption of independence in multi-version programming'. *IEEE Trans. on Software Engineering.*, **SE-12** (1), 96–109.

Leveson, N. G. (1995). Safeware: *System Safety and Computers. Reading,* Mass.: Addison-Wesley.

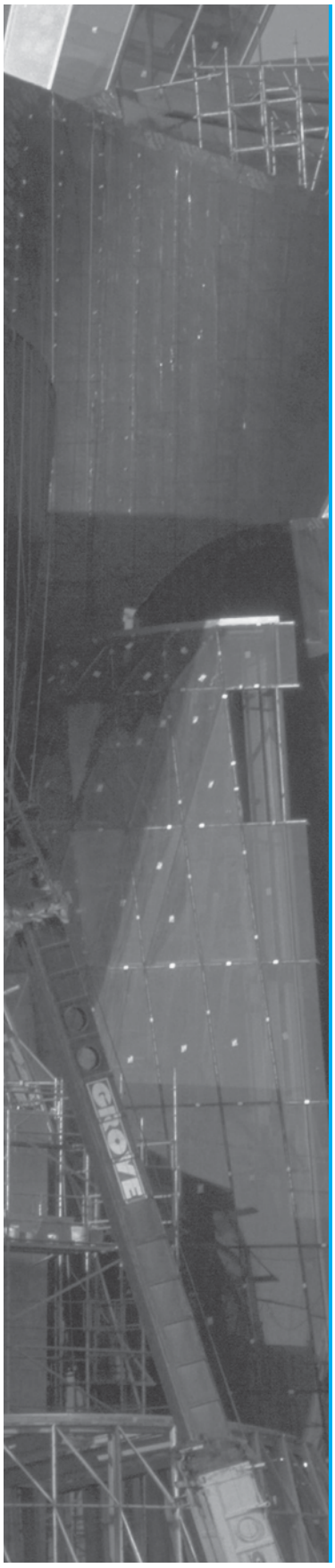
Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. and Kahkonen, T. (2004). 'Agile Software Development in Large Organizations'. *IEEE Computer,* **37** (12), 26–34.

Parnas, D. L., Van Schouwen, J. and Shu, P. K. (1990). 'Evaluation of Safety-Critical Software'. *Comm. ACM,* **33** (6), 636–51.

Pullum, L. L. (2001). *Software Fault Tolerance Techniques and Implementation*. Norwood, Mass.: Artech House.

Storey, N. (1996). *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Torres-Pomales, W. (2000). 'Software Fault Tolerance: A Tutorial.' http://ntrs.nasa.gov/archive/nasa/casi./20000120144_2000175863.pdf.



14

Security engineering

Objectives

The objective of this chapter is to introduce issues that should be considered when you are designing secure application systems. When you have read this chapter, you will:

- understand the difference between application security and infrastructure security;
- know how life-cycle risk assessment and operational risk assessment are used to understand security issues that affect a system design;
- be aware of software architectures and design guidelines for secure systems development;
- understand the notion of system survivability and why survivability analysis is important for complex software systems.

Contents

The widespread use of the Internet in the 1990s introduced a new challenge for software engineers—designing and implementing systems that were secure. As more and more systems were connected to the Internet, a variety of different external attacks were devised to threaten these systems. The problems of producing dependable systems were hugely increased. Systems engineers had to consider threats from malicious and technically skilled attackers as well as problems resulting from accidental mistakes in the development process.

It is now essential to design systems to withstand external attacks and to recover from such attacks. Without security precautions, it is almost inevitable that attackers will compromise a networked system. They may misuse the system hardware, steal confidential data, or disrupt the services offered by the system. System security engineering is therefore an increasingly important aspect of the systems engineering process.

Security engineering is concerned with the development and evolution of systems that can resist malicious attacks, which are intended to damage the system or its data. Software security engineering is part of the more general field of computer security. This has become a priority for businesses and individuals as more and more criminals try to exploit networked systems for illegal purposes. Software engineers should be aware of the security threats faced by systems and ways in which these threats can be neutralized.

My intention in this chapter is to introduce security engineering to software engineers, with a focus on design issues that affect application security. The chapter is not about computer security as a whole and so doesn't cover topics such as encryption, access control, authorization mechanisms, viruses and Trojan horses, etc. These are described in detail in general texts on computer security (Anderson, 2008; Bishop, 2005; Pfleeger and Pfleeger, 2007).

This chapter adds to the discussion of security elsewhere in the book. You should read the material here along with:

- Section 10.1, where I explain how security and dependability are closely related;
- Section 10.4, where I introduce security terminology;
- Section 12.1, where I introduce the general notion of risk-driven specification;
- Section 12.4, where I discuss general issues of security requirements specification;
- Section 15.3, where I explain a number of approaches to security testing.

When you consider security issues, you have to consider both the application software (the control system, the information system, etc.) and the infrastructure on which this system is built (Figure 14.1). The infrastructure for complex applications may include:

- an operating system platform, such as Linux or Windows;
- other generic applications that run on that system, such as web browsers and e-mail clients;
- a database management system;

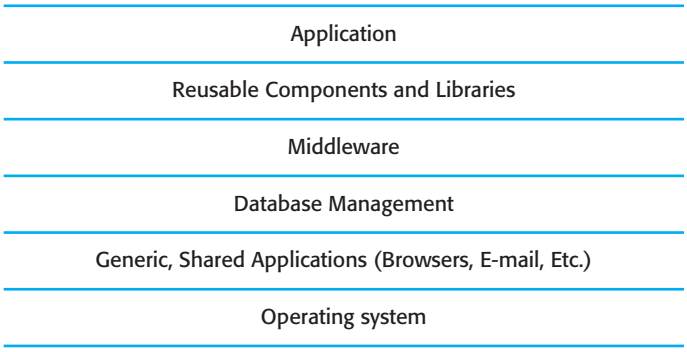


Figure 14.1 System layers where security may be compromised

- middleware that supports distributed computing and database access;
- libraries of reusable components that are used by the application software.

The majority of external attacks focus on system infrastructures because infrastructure components (e.g., web browsers) are well known and widely available. Attackers can probe these systems for weaknesses and share information about vulnerabilities that they have discovered. As many people use the same software, attacks have wide applicability. Infrastructure vulnerabilities may lead to attackers gaining unauthorized access to an application system and its data.

In practice, there is an important distinction between application security and infrastructure security:

1. Application security is a software engineering problem where software engineers should ensure that the system is designed to resist attacks.
2. Infrastructure security is a management problem where system managers configure the infrastructure to resist attacks. System managers have to set up the infrastructure to make the most effective use of whatever infrastructure security features are available. They also have to repair infrastructure security vulnerabilities that come to light as the software is used.

System security management is not a single task but includes a range of activities such as user and permission management, system software deployment and maintenance, and attack monitoring, detection and recovery:

1. User and permission management includes adding and removing users from the system, ensuring that appropriate user authentication mechanisms are in place and setting up the permissions in the system so that users only have access to the resources that they need.
2. System software deployment and maintenance includes installing system software and middleware and configuring these properly so that security vulnerabilities are avoided. It also involves updating this software regularly with new versions or patches, which repair security problems that have been discovered.

Insider attacks and social engineering

Insider attacks are attacks on a system carried out by a trusted individual (an insider) who abuses that trust. For example, a nurse, working in a hospital may access confidential medical records of patients that he or she is not caring for. Insider attacks are difficult to counter because the extra security techniques that may be used would disrupt trustworthy system users.

Social engineering is a way of fooling accredited users into disclosing their credentials. An attacker can therefore behave as an insider when accessing the system.

http://www.SoftwareEngineering-9.com/Web/SecurityEng/insiders.html

3. Attack monitoring, detection and recovery includes activities which monitor the system for unauthorized access, detect, and put in place strategies for resisting attacks, and backup activities so that normal operation can be resumed after an external attack.

Security management is vitally important, but it is not usually considered to be part of application security engineering. Rather, application security engineering is concerned with designing a system so that it is as secure as possible, given budget and usability constraints. Part of this process is 'design for management', where you design systems to minimize the chance of security management errors leading to successful attacks on the system.

For critical control systems and embedded systems, it is normal practice to select an appropriate infrastructure to support the application system. For example, embedded system developers usually choose a real-time operating system that provides the embedded application with the facilities that it needs. Known vulnerabilities and security requirements can be taken into account. This means that an holistic approach can be taken to security engineering. Application security requirements may be implemented through the infrastructure or the application itself.

However, application systems in an organization are usually implemented using the existing infrastructure (operating system, database, etc.). Therefore, the risks of using that infrastructure and its security features must be taken into account as part of the system design process.

14.1 Security risk management

Security risk assessment and management is essential for effective security engineering. Risk management is concerned with assessing the possible losses that might ensue from attacks on assets in the system, and balancing these losses against the

costs of security procedures that may reduce these losses. Credit card companies do this all the time. It is relatively easy to introduce new technology to reduce credit card fraud. However, it is often cheaper for them to compensate users for their losses due to fraud than to buy and deploy fraud-reduction technology. As costs drop and attacks increase, this balance may change. For example, credit card companies are now encoding information on an on-card chip instead of a magnetic strip. This makes card copying much more difficult.

Risk management is a business issue rather than a technical issue so software engineers should not decide what controls should be included in a system. It is up to senior management to decide whether or not to accept the cost of security or the exposure that results from a lack of security procedures. Rather, the role of software engineers is to provide informed technical guidance and judgment on security issues. They are, therefore, essential participants in the risk management process.

As I explained in Chapter 12, a critical input to the risk assessment and management process is the organizational security policy. The organizational security policy applies to all systems and should set out what should and should not be allowed. The security policy sets out conditions that should always be maintained by a security system and so helps to identify risks and threats that might arise. The security policy therefore defines what is and what is not allowed. In the security engineering process, you design the mechanisms to implement this policy.

Risk assessment starts before the decision to acquire the system has been made and should continue throughout the system development process and after the system has gone into use (Alberts and Dorofee, 2002). I also introduced, in Chapter 12, the idea that this risk assessment is a staged process:

1. *Preliminary risk assessment* At this stage, decisions on the detailed system requirements, the system design, or the implementation technology have not been made. The aim of this assessment process is to decide if an adequate level of security can be achieved at a reasonable cost. If this is the case, you can then derive specific security requirements for the system. You do not have information about potential vulnerabilities in the system or the controls that are included in reused system components or middleware.

2. *Life-cycle risk assessment* This risk assessment takes place during the system development life cycle and is informed by the technical system design and implementation decisions. The results of the assessment may lead to changes to the security requirements and the addition of new requirements. Known and potential vulnerabilities are identified and this knowledge is used to inform decision making about the system functionality and how it is to be implemented, tested, and deployed.

3. *Operational risk assessment* After a system has been deployed and put into use, risk assessment should continue to take account of how the system is

used and proposals for new and changed requirements. Assumptions about the operating requirement made when the system was specified may be incorrect. Organizational changes may mean that the system is used in different ways from those originally planned. Operational risk assessment therefore leads to new security requirements that have to be implemented as the system evolves.

Preliminary risk assessment focuses on deriving security requirements. In Chapter 12, I show how an initial set of security requirements may be derived from a preliminary risk assessment. In this section, I concentrate on life cycle and operational risk assessment to illustrate how the specification and design of a system are influenced by technology and the way that the system is used.

To carry out a risk assessment, you need to identify the possible threats to a system. One way to do this, is to develop a set of 'misuse cases' (Alexander, 2003; Sindre and Opdahl, 2005). I have already discussed how use cases—typical interactions with a system—may be used to derive system requirements. Misuse cases are scenarios that represent malicious interactions with a system. You can use these to discuss and identify possible threats and, therefore also determine the system's security requirements. They can be used alongside use cases when deriving the system requirements.

Pfleeger and Pfleeger (2007) characterize threats under four headings, which may be used as a starting point for identifying possible misuse cases. These headings are as follows:

1. Interception threats that allow an attacker to gain access to an asset. So, a possible misuse case for the MHC-PMS might be a situation where an attacker gains access to the records of an individual celebrity patient.
2. Interruption threats that allow an attacker to make part of the system unavailable. Therefore, a possible misuse case might be a denial of service attack on a system database server.
3. Modification threats that allow an attacker to tamper with a system asset. In the MHC-PMS, this could be represented by a misuse case where an attacker changes the information in a patient record.
4. Fabrication threats that allow an attacker to insert false information into a system. This is perhaps not a credible threat in the MHC-PMS but would certainly be a threat in a banking system, where false transactions might be added to the system that transfer money to the perpetrator's bank account.

Misuse cases are not just useful in preliminary risk assessment but may be used for security analysis in life-cycle risk analysis and operational risk analysis. They provide a useful basis for playing out hypothetical attacks on the system and assessing the security implications of design decisions that have been made.

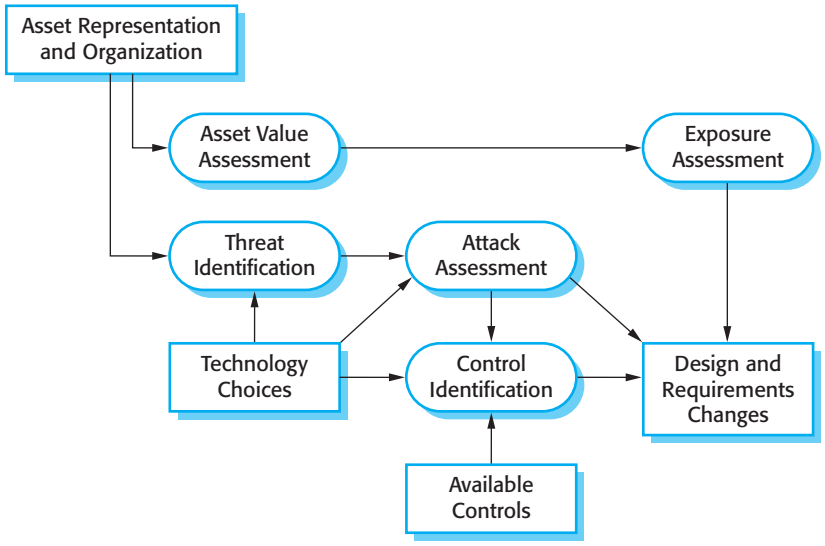


Figure 14.2 Life-cycle risk analysis

14.1.1 Life-cycle risk assessment

Based on organizational security policies, preliminary risk assessment should identify the most important security requirements for a system. These reflect how the security policy should be implemented in that application, identify the assets to be protected, and decide what approach should be used to provide that protection. However, maintaining security is about paying attention to detail. It is impossible for the initial security requirements to take all details that affect security into account.

Life-cycle risk assessment identifies the design and implementation details that affect security. This is the important distinction between life-cycle risk assessment and preliminary risk assessment. Life-cycle risk assessment affects the interpretation of existing security requirements, generates new requirements, and influences the overall design of the system.

When assessing risks at this stage, you should have much more detailed information about what needs to be protected, and you also will know something about the vulnerabilities in the system. Some of these vulnerabilities will be inherent in the design choices made. For example, a vulnerability in all password-based systems is that an authorized user reveals their password to an unauthorized user. Alternatively, if an organization has a policy of developing software in C, you will know that the application may have vulnerabilities because the language does not include array bound checking.

Security risk assessment should be part of all life-cycle activities from requirements engineering to system deployment. The process followed is similar to the preliminary risk assessment process with the addition of activities concerned with design vulnerability identification and assessment. The outcome of the risk assessment is a set of engineering decisions that affect the system design or implementation, or limit the way in which it is used.

A model of the life-cycle risk analysis process, based on the preliminary risk analysis process that I described in Figure 12.9, is shown in Figure 14.2. The most

important difference between these processes is that you now have information about information representation and distribution and the database organization for the high-level assets that have to be protected. You are also aware of important design decisions such as the software to be reused, infrastructure controls and protection, etc. Based on this information, your analysis identifies changes to the security requirements and the system design to provide additional protection for the important system assets.

Two examples illustrate how protection requirements are influenced by decisions on information representation and distribution:

1. You may make a design decision to separate personal patient information and information about treatments received, with a key linking these records. The treatment information is much less sensitive than the personal patient information so may not need as extensive protection. If the key is protected, then an attacker will only be able to access routine information, without being able to link this to an individual patient.

2. Assume that, at the beginning of a session, a design decision is made to copy patient records to a local client system. This allows work to continue if the server is unavailable. It makes it possible for a health-care worker to access patient records from a laptop, even if no network connection is available. However, you now have two sets of records to protect and the client copies are subject to additional risks, such as theft of the laptop computer. You, therefore, have to think about what controls should be used to reduce risk. For example, client records on the laptop may have to be encrypted.

To illustrate how decisions on development technologies influence security, assume that the health-care provider has decided to build a MHC-PMS using an off-the-shelf information system for maintaining patient records. This system has to be configured for each type of clinic in which it is used. This decision has been made because it appears to offer the most extensive functionality for the lowest development cost and fastest deployment time.

When you develop an application by reusing an existing system, you have to accept the design decisions made by the developers of that system. Let us assume that some of these design decisions are as follows:

1. System users are authenticated using a login name/password combination. No other authentication method is supported.

2. The system architecture is client-server, with clients accessing data through a standard web browser on a client PC.

3. Information is presented to users as an editable web form. They can change information in place and upload the revised information to the server.

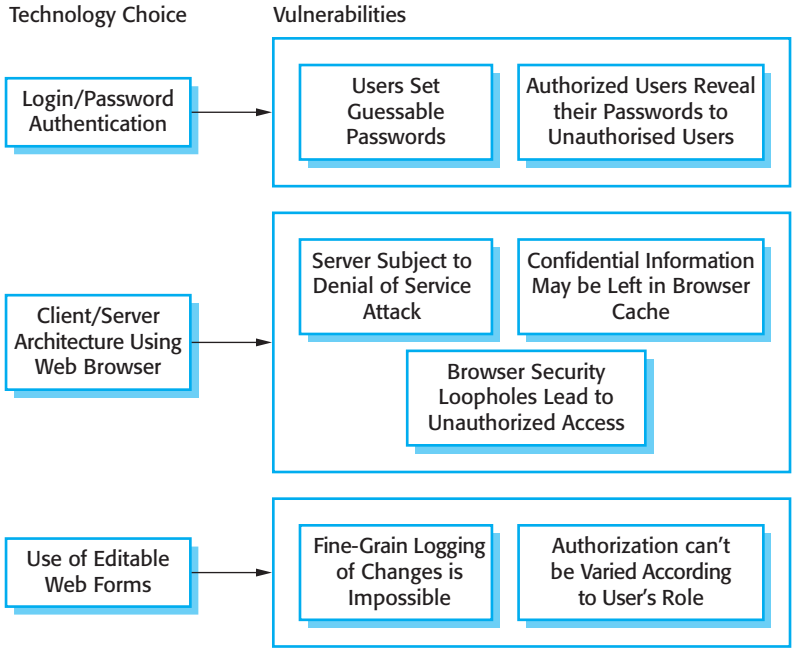


Figure 14.3 Vulnerabilities associated with technology choices

For a generic system, these design decisions are perfectly acceptable, but a life-cycle risk analysis reveals that they have associated vulnerabilities. Examples of possible vulnerabilities are shown in Figure 14.3.

Once vulnerabilities have been identified, you then have to make a decision on what steps that you can take to reduce the associated risks. This will often involve making decisions about additional system security requirements or the operational process of using the system. I don't have space here to discuss all the requirements that might be proposed to address the inherent vulnerabilities, but some examples of requirements might be the following:

1. A password checker program shall be made available and shall be run daily. User passwords that appear in the system dictionary shall be identified and users with weak passwords reported to system administrators.
2. Access to the system shall only be allowed to client computers that have been approved and registered with the system administrators.
3. All client computers shall have a single web browser installed as approved by system administrators.

As an off-the-shelf system is used, it isn't possible to include a password checker in the application system itself, so a separate system must be used. Password checkers analyze the strength of user passwords when they are set up, and notify users if they have chosen weak passwords. Therefore, vulnerable passwords can be identified reasonably

quickly after they have been set up, and action can then be taken to ensure that users change their password.

The second and third requirements mean that all users will always access the system through the same browser. You can decide what is the most secure browser when the system is deployed and install that on all client computers. Security updates are simplified because there is no need to update different browsers when security vulnerabilities are discovered and fixed.

14.1.2 Operational risk assessment

Security risk assessment should continue throughout the lifetime of the system to identify emerging risks and system changes that may be required to cope with these risks. This process is called operational risk assessment. New risks may emerge because of changing system requirements, changes in the system infrastructure, or changes in the environment in which the system is used.

The process of operational risk assessment is similar to the life-cycle risk assessment process, but with the addition of further information about the environment in which the system is used. The environment is important because characteristics of the environment can lead to new risks to the system. For example, say a system is being used in an environment in which users are frequently interrupted. A risk is that the interruption will mean that the user has to leave their computer unattended. It may then be possible for an unauthorized person to gain access to the information in the system. This could then generate a requirement for a password-protected screen saver to be run after a short period of inactivity.

14.2 Design for security

It is generally true that it is very difficult to add security to a system after it has been implemented. Therefore, you need to take security issues into account during the systems design process. In this section, I focus primarily on issues of system design, because this topic isn't given the attention it deserves in computer security books. Implementation issues and mistakes also have a major impact on security but these are often dependent on the specific technology used. I recommend Viega and McGraw's book (2002) as a good introduction to programming for security.

Here, I focus on a number of general, application-independent issues relevant to secure systems design:

1. Architectural design—how do architectural design decisions affect the security of a system?
2. Good practice—what is accepted good practice when designing secure systems?
3. Design for deployment—what support should be designed into systems to avoid the introduction of vulnerabilities when a system is deployed for use?

Denial of service attacks

Denial of service attacks attempt to bring down a networked system by bombarding it with a huge number of service requests. These place a load on the system for which it was not designed and they exclude legitimate requests for system service. Consequently, the system may become unavailable either because it crashes with the heavy load or has to be taken offline by system managers to stop the flow of requests.

http://www.SoftwareEngineering-9.com/Web/Security/DoS.html

Of course, these are not the only design issues that are important for security. Every application is different and security design also has to take into account the purpose, criticality, and operational environment of the application. For example, if you are designing a military system, you need to adopt their security classification model (secret, top secret, etc.). If you are designing a system that maintains personal information, you may have to take into account data protection legislation that places restrictions on how data is managed.

There is a close relationship between dependability and security. The use of redundancy and diversity, which is fundamental for achieving dependability, may mean that a system can resist and recover from attacks that target specific design or implementation characteristics. Mechanisms to support a high level of availability may help the system to recover from so-called denial of service attacks, where the aim of an attacker is to bring down the system and stop it working properly.

Designing a system to be secure inevitably involves compromises. It is certainly possible to design multiple security measures into a system that will reduce the chances of a successful attack. However, security measures often require a lot of additional computation and so affect the overall performance of a system. For example, you can reduce the chances of confidential information being disclosed by encrypting that information. However, this means that users of the information have to wait for it to be decrypted and this may slow down their work.

There are also tensions between security and usability. Security measures sometimes require the user to remember and provide additional information (e.g., multiple passwords). However, sometimes users forget this information, so the additional security means that they can't use the system. Designers therefore have to find a balance between security, performance, and usability. This will depend on the type of system and where it is being used. For example, in a military system, users are familiar with high-security systems and so are willing to accept and follow processes that require frequent checks. In a system for stock trading, however, interruptions of operation for security checks would be completely unacceptable.

14.2.1 Architectural design

As I have discussed in Chapter 11, the choice of software architecture can have profound effects on the emergent properties of a system. If an inappropriate architecture is used, it may be very difficult to maintain the confidentiality and

integrity of information in the system or to guarantee a required level of system availability.

In designing a system architecture that maintains security, you need to consider two fundamental issues:

1. Protection—how should the system be organized so that critical assets can be protected against external attack?
2. Distribution—how should system assets be distributed so that the effects of a successful attack are minimized?

These issues are potentially conflicting. If you put all your assets in one place, then you can build layers of protection around them. As you only have to build a single protection system, you may be able to afford a strong system with several protection layers. However, if that protection fails, then all your assets are compromised. Adding several layers of protection also affects the usability of a system so it may mean that it is more difficult to meet system usability and performance requirements.

On the other hand, if you distribute assets, they are more expensive to protect because protection systems have to be implemented for each copy. Typically, then, you cannot afford as many protection layers. The chances are greater that the protection will be breached. However, if this happens, you don't suffer a total loss. It may be possible to duplicate and distribute information assets so that if one copy is corrupted or inaccessible, then the other copy can be used. However, if the information is confidential, keeping additional copies increases the risk that an intruder will gain access to this information.

For the patient record system, it is appropriate to use a centralized database architecture. To provide protection, you use a layered architecture with the critical protected assets at the lowest level in the system, with various layers of protection around them. Figure 14.4 illustrates this for the patient record system in which the critical assets to be protected are the records of individual patients.

In order to access and modify patient records, an attacker has to penetrate three system layers:

1. *Platform-level protection* The top level controls access to the platform on which the patient record system runs. This usually involves a user signing on to a particular computer. The platform will also normally include support for maintaining the integrity of files on the system, backups, etc.
2. *Application-level protection* The next protection level is built into the application itself. It involves a user accessing the application, being authenticated, and getting authorization to take actions such as viewing or modifying data. Application-specific integrity management support may be available.
3. *Record-level protection* This level is invoked when access to specific records is required, and involves checking that a user is authorized to carry out the requested operations on that record. Protection at this level might also involve

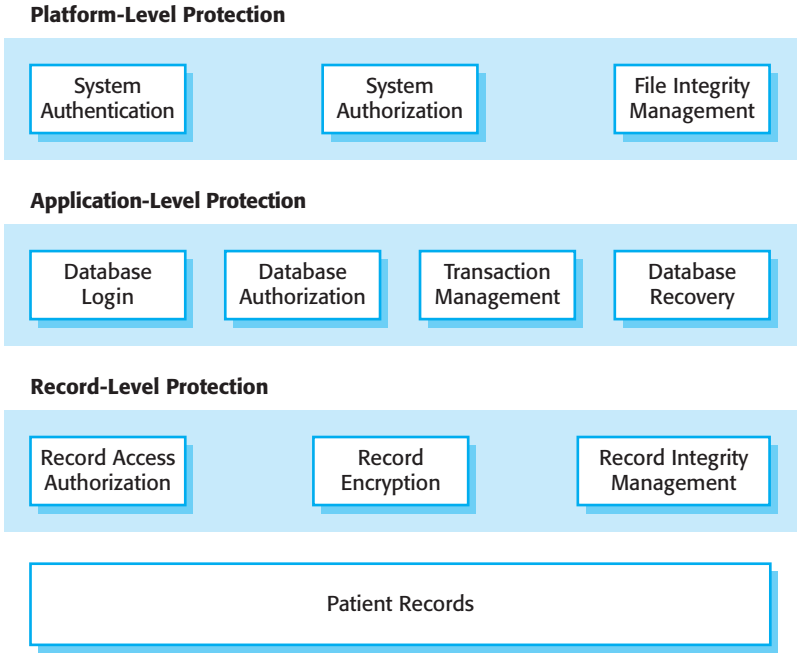


Figure 14.4 A layered protection architecture

encryption to ensure that records cannot be browsed using a file browser. Integrity checking using, for example, cryptographic checksums, can detect changes that have been made outside the normal record update mechanisms.

The number of protection layers that you need in any particular application depends on the criticality of the data. Not all applications need protection at the record level and, therefore, coarser-grain access control is more commonly used. To achieve security, you should not allow the same user credentials to be used at each level. Ideally, if you have a password-based system, then the application password should be different from both the system password and the record-level password. However, multiple passwords are difficult for users to remember and they find repeated requests to authenticate themselves irritating. You often, therefore, have to compromise on security in favor of system usability.

If protection of data is a critical requirement, then a client–server architecture should be used, with the protection mechanisms built into the server. However, if the protection is compromised, then the losses associated with an attack are likely to be high, as are the costs of recovery (e.g., all user credentials may have to be reissued). The system is vulnerable to denial of service attacks, which overload the server and make it impossible for anyone to access the system database.

If you think that denial of service attacks are a major risk, you may decide to use a distributed object architecture for the application. In this situation, illustrated in Figure 14.5, the system's assets are distributed across a number of different platforms, with separate protection mechanisms used for each of these. An attack on one node might mean that some assets are unavailable but it would still be possible to

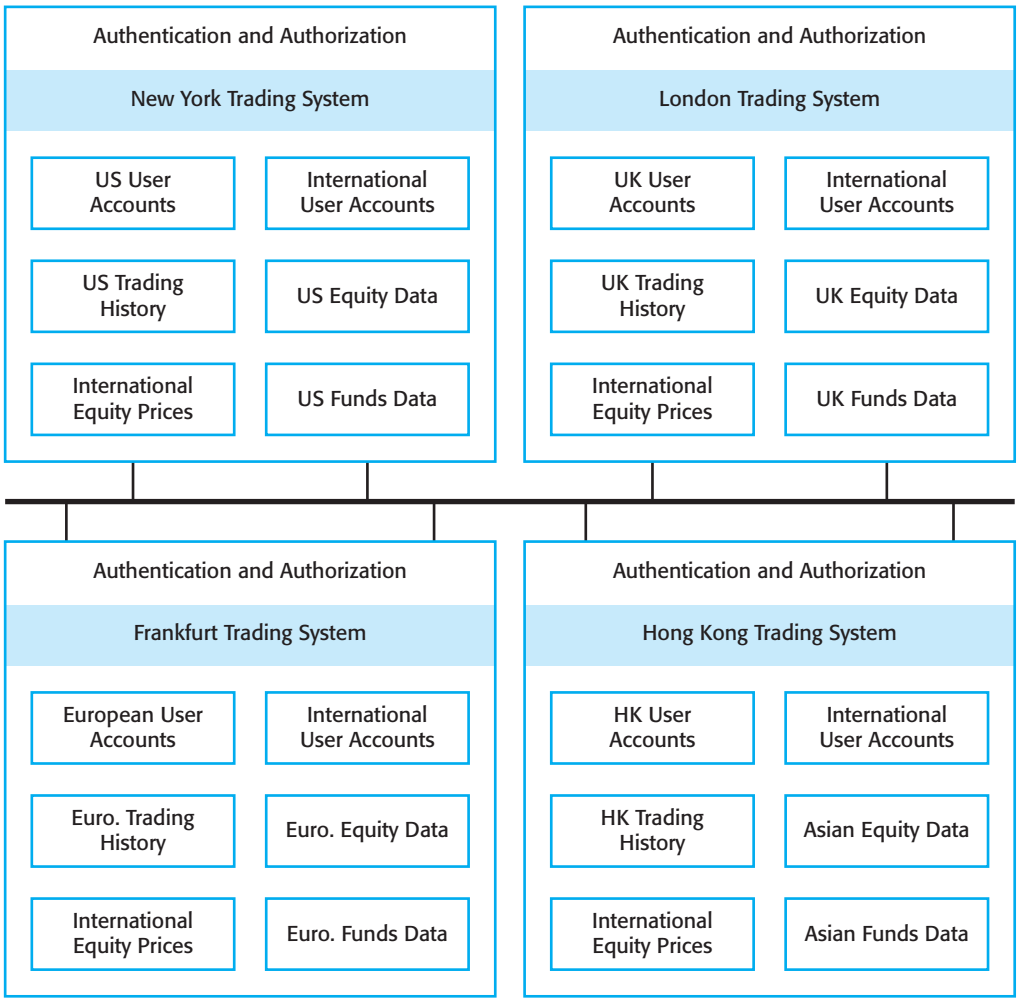


Figure 14.5 Distributed assets in an equity trading system

provide some system services. Data can be replicated across the nodes in the system so that recovery from attacks is simplified.

Figure 14.5 shows the architecture of a banking system for trading in stocks and funds on the New York, London, Frankfurt, and Hong Kong markets. The system is distributed so that data about each market is maintained separately. Assets required to support the critical activity of equity trading (user accounts and prices) are replicated and available on all nodes. If a node of the system is attacked and becomes unavailable, the critical activity of equity trading can be transferred to another country and so can still be available to users.

I have already discussed the problem of finding a balance between security and system performance. A problem of secure system design is that in many cases, the architectural style that is most suitable for meeting the security requirements may not be the best one for meeting the performance requirements. For example, say an

application has one absolute requirement to maintain the confidentiality of a large database and another requirement for very fast access to that data. A high level of protection suggests that layers of protection are required, which means that there must be communications between the system layers. This has an inevitable performance overhead, thus will slow down access to the data. If an alternative architecture is used, then implementing protection and guaranteeing confidentiality may be more difficult and expensive. In such a situation, you have to discuss the inherent conflicts with the system client and agree on how these are to be resolved.

14.2.2 Design guidelines

There are no hard and fast rules about how to achieve system security. Different types of systems require different technical measures to achieve a level of security that is acceptable to the system owner. The attitudes and requirements of different groups of users profoundly affect what is and is not acceptable. For example, in a bank, users are likely to accept a higher level of security, and hence more intrusive security procedures than, say, in a university.

However, there are general guidelines that have wide applicability when designing system security solutions, which encapsulate good design practice for secure systems engineering. General design guidelines for security, such as those discussed, below, have two principal uses:

1. They help raise awareness of security issues in a software engineering team. Software engineers often focus on the short-term goal of getting the software working and delivered to customers. It is easy for them to overlook security issues. Knowledge of these guidelines can mean that security issues are considered when software design decisions are made.
2. They can be used as a review checklist that can be used in the system validation process. From the high-level guidelines discussed here, more specific questions can be derived that explore how security has been engineered into a system.

The 10 design guidelines, summarized in Figure 14.6, have been derived from a range of different sources (Schneier, 2000; Viega and McGraw, 2002; Wheeler, 2003). I have focused here on guidelines that are particularly applicable to the software specification and design processes. More general principles, such as ‘Secure the weakest link in a system’, ‘Keep it simple’, and ‘Avoid security through obscurity’ are also important but are less directly relevant to engineering decision making.

Guideline 1: Base security decisions on an explicit security policy

A security policy is a high-level statement that sets out fundamental security conditions for an organization. It defines the ‘what’ of security rather than the ‘how’, so the policy should not define the mechanisms to be used to provide and enforce security. In principle, all aspects of the security policy should be reflected in the system

Security guidelines
1 Base security decisions on an explicit security policy
2 Avoid a single point of failure
3 Fail securely
4 Balance security and usability
5 Log user actions
6 Use redundancy and diversity to reduce risk
7 Validate all inputs
8 Compartmentalize your assets
9 Design for deployment
10 Design for recoverability

Figure 14.6 Design guidelines for secure systems engineering

requirements. In practice, especially if a rapid application development process is used, this is unlikely to happen. Designers, therefore, should consult the security policy as it provides a framework for making and evaluating design decisions.

For example, say you are designing an access control system for the MHC-PMS. The hospital security policy may state that only accredited clinical staff may modify electronic patient records. Your system therefore has to include mechanisms that check the accreditation of anyone attempting to modify the system and that reject modifications from people who are not accredited.

The problem that you may face is that many organizations do not have an explicit systems security policy. Over time, changes may have been made to systems in response to identified problems, but with no overarching policy document to guide the evolution of a system. In such situations, you need to work out and document the policy from examples, and confirm it with managers in the company.

Guideline 2: Avoid a single point of failure

In any critical system, it is good design practice to try to avoid a single point of failure. This means that a single failure in part of the system should not result in an overall systems failure. In security terms, this means that you should not rely on a single mechanism to ensure security, rather you should employ several different techniques. This is sometimes called ‘defense in depth’.

For example, if you use a password to authenticate users to a system, you might also include a challenge/response authentication mechanism where users have to pre-register questions and answers with the system. After password authentication, they must then answer questions correctly before being allowed access. To protect

the integrity of data in a system, you might keep an executable log of all changes made to the data (see Guideline 5). In the event of a failure, you can replay the log to re-create the data set. You might also make a copy of all data that is modified before the change is made.

Guideline 3: Fail securely

System failures are inevitable in all systems and, in the same way that safety-critical systems should always fail-safe, security critical systems should always 'fail-secure'. When the system fails, you should not use fallback procedures that are less secure than the system itself. Nor should system failure mean that an attacker can access data that would not normally be allowed.

For example, in the patient information system, I suggested a requirement that patient data should be downloaded to a system client at the beginning of a clinic session. This speeds up access and means that access is possible if the server is unavailable. Normally, the server deletes this data at the end of the clinic session. However, if the server has failed, then there is the possibility that the information will be maintained on the client. A fail-secure approach in those circumstances is to encrypt all patient data stored on the client. This means that an unauthorized user cannot read the data.

Guideline 4: Balance security and usability

The demands of security and usability are often contradictory. To make a system secure, you have to introduce checks that users are authorized to use the system and that they are acting in accordance with security policies. All of these inevitably make demands on users—they may have to remember login names and passwords, only use the system from certain computers, and so on. These mean that it takes users more time to get started with the system and use it effectively. As you add security features to a system, it is inevitable that it will become less usable. I recommend Cranor and Garfinkel's book (2005) that discusses a wide range of issues in the general area of security and usability.

There comes a point where it is counterproductive to keep adding on new security features at the expense of usability. For example, if you require users to input multiple passwords or to change their passwords to impossible-to-remember character strings at frequent intervals, they will simply write down these passwords. An attacker (especially an insider) may then be able to find the passwords that have been written down and gain access to the system.

Guideline 5: Log user actions

If it is practically possible to do so, you should always maintain a log of user actions. This log should, at least, record who did what, the assets used, and the time and date of the action. As I discuss in Guideline 2, if you maintain this as a list of executable commands, you have the option of replaying the log to recover from failures. Of course, you also need tools that allow you to analyze the log and detect potentially anomalous actions. These tools can scan the log and find anomalous actions, and thus help detect attacks and trace how the attacker gained access to the system.

Apart from helping recover from failure, a log of user actions is useful because it acts as a deterrent to insider attacks. If people know that their actions are being logged, then they are less likely to do unauthorized things. This is most effective for casual attacks, such as a nurse looking up patient records, or for detecting attacks where legitimate user credentials have been stolen through social engineering. Of course, this is not foolproof, as technically skilled insiders can also access and change the log.

Guideline 6: Use redundancy and diversity to reduce risk

Redundancy means that you maintain more than one version of software or data in a system. Diversity, when applied to software, means that the different versions should not rely on the same platform or be implemented using the same technologies. Therefore, a platform or technology vulnerability will not affect all versions and so lead to a common failure. I explained in Chapter 13 how redundancy and diversity are the fundamental mechanisms used in dependability engineering.

I have already discussed examples of redundancy—maintaining patient information on both the server and the client, firstly in the mental health-care system, and then in the distributed equity trading system shown in Figure 14.5. In the patient records system, you could use diverse operating systems on the client and the server (e.g., Linux on the server, Windows on the client). This ensures that an attack based on an operating system vulnerability will not affect both the server and the client. Of course, you have to trade off such benefits against the increased management cost of maintaining different operating systems in an organization.

Guideline 7: Validate all inputs

A common attack on a system involves providing the system with unexpected inputs that cause it to behave in an unanticipated way. These may simply cause a system crash, resulting in a loss of service, or the inputs could be made up of malicious code that is executed by the system. Buffer overflow vulnerabilities, first demonstrated in the Internet worm (Spafford, 1989) and commonly used by attackers (Berghel, 2001), may be triggered using long input strings. So-called ‘SQL poisoning’, where a malicious user inputs an SQL fragment that is interpreted by a server, is another fairly common attack.

As I explained in Chapter 13, you can avoid many of these problems if you design input validation into your system. Essentially, you should never accept any input without applying some checks to it. As part of the requirements, you should define the checks that should be applied. You should use knowledge of the input to define these checks. For example, if a surname is to be input, you might check that there are no embedded spaces and that the only punctuation used is a hyphen. You might also check the number of characters input and reject inputs that are obviously too long. For example, no one has a family name with more than 40 characters and no addresses are more than 100 characters long. If you use menus to present allowed inputs, you avoid some of the problems of input validation.