degree of trust of our

Program Security

SECURE PROGRAMS

Consider what we mean when we say that a program is "secure." Security implies some degree of trust that the program enforces expected confidentiality, integrity, and availability. From the point of view of a program or a programmer, how can we look at a software component or code fragment and assess its security? This question is, of course, similar to the problem of assessing software quality in general. One way to assess security or quality is to ask people to name the characteristics of software that contribute to its overall security. However, we are likely to get different answers from different people. This difference occurs because the importance of the characteristics depends on who is analyzing the software. For example, one person may decide that code is secure because it takes too long to break through its security controls. And someone else may decide code is secure if it has run for a period of time with no apparent failures. But a third person may decide that *any* potential fault in meeting security requirements makes code insecure.

An assessment of security can also be influenced by someone's general perspective on software quality. For example, if your manager's idea of quality is conformance to specifications, then she might consider the code secure if it meets security requirements, whether or not the requirements are complete or correct. This security view played a role when a major computer manufacturer delivered all its machines with keyed locks, since a keyed lock was written in the requirements. But the machines were not secure, because all locks were configured to use the same key! Thus, another view of security is fitness for purpose; in this view, the manufacturer clearly had room for improvement.

In general, practitioners often look at quantity and types of faults for evidence of a product's quality (or lack of it). For example, developers track the number of faults found in requirements, design, and code inspections and use them as indicators of the likely quality of the final product. Sidebar 3-1 explains the importance of separating the faults—the causes of problems—from the failures—the effects of the faults.

Fixing Faults

One approach to judging quality in security has been fixing faults. You might argue that a module in which 100 faults were discovered and fixed is better than another in which only 20 faults were discovered and fixed, suggesting that more rigorous analysis and testing had led to the finding of the larger number of faults. *Au contraire,* challenges your friend: a piece of software with 100 discovered faults is inherently full of problems and could clearly have hundreds more waiting to appear. Your friend's opinion is confirmed by the software testing literature; software that has many faults early on is likely to have many others still waiting to be found.

Early work in computer security was based on the paradigm of "penetrate and patch," in which analysts searched for and repaired faults. Often, a top-quality "tiger team" would be convened to test a system's security by attempting to cause it to fail.

The test was considered to be a "proof" of security; if the system withstood the attacks, it was considered secure. Unfortunately, far too often the proof became a counterexample, in which not just one but several serious security problems were uncovered. The problem discovery in turn led to a rapid effort to "patch" the system to repair or restore the security. (See Schell's analysis in [SCH79].) However, the patch efforts were largely useless, making the system less secure rather than more secure because they frequently introduced *new* faults. There are three reasons why.

- The pressure to repair a specific problem encouraged a narrow focus on the fault itself and not on its context. In particular, the analysts paid attention to the immediate cause of the failure and not to the underlying design or requirements faults.
- The fault often had nonobvious side effects in places other than the immediate area of the fault.
- The fault could not be fixed properly because system functionality or performance would suffer as a consequence.

Unexpected Behavior

The inadequacies of penetrate-and-patch led researchers to seek a better way to be confident that code meets its security requirements. One way to do that is to compare the requirements with the behavior. That is, to understand program security, we can examine programs to see whether they behave as their designers intended or users expected. We call such unexpected behavior a program security flaw; it is inappropriate program behavior caused by a program vulnerability. Unfortunately, the terminology in the computer security field is not consistent with the IEEE standard described in Side-bar 3-1; there is no direct mapping of the terms "vulnerability" and "flaw" into the characterization of faults and failures. A flaw can be either a fault or failure, and a vulnerability usually describes a class of flaws, such as a buffer overflow. In spite of the inconsistency, it is important for us to remember that we must view vulnerabilities and flaws from two perspectives, cause and effect, so that we see what fault caused the problem and what failure (if any) is visible to the user. For example, a Trojan horse may have been injected in a piece of code—a flaw exploiting a vulnerability—but the user may not yet have seen the Trojan horse's malicious behavior. Thus, we must address program security flaws from inside and outside, to find causes not only of existing failures but also of incipient ones. Moreover, it is not enough to identify these problems. We must also determine how to prevent harm caused by possible flaws.

Program security flaws can derive from any kind of software fault. That is, they cover everything from a misunderstanding of program requirements to a one-character error in coding or even typing. The flaws can result from problems in a single code component or from the failure of several programs or program pieces to interact compatibly through a

shared interface. The security flaws can reflect code that was intentionally designed or coded to be malicious, or code that was simply developed in a sloppy or misguided way. Thus, it makes sense to divide program flaws into two separate logical categories: inadvertent human errors versus malicious, intentionally induced flaws.

Frequently, we talk about "bugs" in software, a term that can mean many different things, depending on context. A "bug" can be a mistake in interpreting a requirement, a syntax error in a piece of code, or the (as-yet-unknown) cause of a system crash. The IEEE has suggested a standard terminology (in IEEE Standard 729) for describing "bugs" in our software products

When a human makes a mistake, called an error, in performing some software activity, the error may lead to a fault, or an incorrect step, command, process, or data definition in a computer program. For example, a designer may misunderstand a requirement and create a design that does not match the actual intent of the requirements analyst and the user. This design fault is an encoding of the error, and it can lead to other faults, such as incorrect code and an incorrect description in a user manual. Thus, a single error can generate many faults, and a fault can reside in any development or maintenance product.

A failure is a departure from the system's required behavior. It can be discovered before or after system delivery, during testing, or during operation and maintenance. Since the requirements documents can contain faults, a failure indicates that the system is not performing as required, even though it may be performing as specified.

Thus, a fault is an inside view of the system, as seen by the eyes of the developers, whereas a failure is an outside view: a problem that the user sees. Not every fault corresponds to a failure; for example, if faulty code is never executed or a particular state is never entered, then the fault will never cause the code to fail.

These categories help us understand some ways to prevent the inadvertent and intentional insertion of flaws into future code, but we still have to address their effects, regardless of intention. That is, in the words of Sancho Panza in *Man of La Mancha*, "it doesn't matter whether the stone hits the pitcher or the pitcher hits the stone, it's going to be bad for the pitcher." An inadvertent error can cause just as much harm to users and their organizations as can an intentionally induced flaw. Furthermore, a system attack often exploits an unintentional security flaw to perform intentional damage. From reading the popular press , you might conclude that intentional security incidents (called cyber attacks) are the biggest security threat today. In fact, plain, unintentional, human errors cause much more damage.

Regrettably, we do not have techniques to eliminate or address all program security flaws. There are two reasons for this distressing situation.

1. Program controls apply at the level of the individual program and programmer. When we test a system, we try to make sure that the functionality prescribed in the requirements is implemented in the code. That is, we take a "should do" checklist

and verify that the code does what it is supposed to do. However, security is also about *preventing* certain actions: a "shouldn't do" list. It is almost impossible to ensure that a program does precisely what its designer or user intended, and nothing more. Regardless of designer or programmer intent, in a large and complex system, the number of pieces that have to fit together properly interact in an unmanageably large number of ways. We are forced to examine and test the code for typical or likely cases; we cannot exhaustively test every state and data combination to verify a system's behavior. So sheer size and complexity preclude total flaw prevention or mediation. Programmers intending to implant malicious code can take advantage of this incompleteness and hide some flaws successfully, despite our best efforts.

Carnegie Mellon University's Computer Emergency Response Team (CERT) tracks the number and kinds of vulnerabilities and cyber attacks reported worldwide. Part of CERT's mission is to warn users and developers of new problems and also to provide information on ways to fix them. According to the CERT coordination center, fewer than 200 known vulnerabilities were reported in 1995, and that number ranged between 200 and 400 from 1996 to 1999. But the number increased dramatically in 2000, with over 1,000 known vulnerabilities in 2000, almost 2,420 in 2001, and an expectation of at least 3,750 in 2002 (over 1,000 in the first quarter of 2002).

How does that translate into cyber attacks? The CERT reported 3,734 security incidents in 1998, 9,859 in 1999, 21,756 in 2000, and 52,658 in 2001. But in the first quarter of 2002 there were already 26,829 incidents, so it seems as if the exponential growth rate will continue [HOU02]. Moreover, as of June 2002, Symantec's Norton antivirus software checked for 61,181 known virus patterns, and McAfee's product could detect over 50,000 [BER01]. The Computer Security Institute and the FBI cooperate to take an annual survey of approximately 500 large institutions: companies, government organizations, and educational institutions [CSI02]. Of the respondents, 90 percent detected security breaches, 25 percent identified between two and five events, and 37 percent reported more than ten. By a different count, the Internet security firm Riptech reported that the number of successful Internet attacks was 28 percent higher for Janu-ary–June 2002 compared with the previous six-month period [RIP02].

A survey of 167 network security personnel revealed that more than 75 percent of government respondents experienced attacks to their networks; more than half said the attacks were frequent. However, 60 percent of respondents admitted that they could do more to make their systems more secure; the respondents claimed that they simply lacked time and staff to address the security issues [BUS01]. In the CSI/FBI survey, 223, or 44 percent of respondents, could and did quantify their loss from incidents; their losses totaled over $455,000,000.

It is clearly time to take security seriously, both as users and developers.

2. Programming and software engineering techniques change and evolve far more rapidly than do computer security techniques. So we often find ourselves trying to secure last year's technology while software developers are rapidly adopting today's—and next year's—technology.

Still, the situation is far from bleak. Computer security has much to offer to program security. By understanding what can go wrong and how to protect against it, we can devise techniques and tools to secure most computer applications.

Types of Flaws

To aid our understanding of the problems and their prevention or correction, we can define categories that distinguish one kind of problem from another. For example, Landwehr et al. [LAN94] present a taxonomy of program flaws, dividing them first into intentional and inadvertent flaws. They further divide intentional flaws into malicious and nonmalicious ones. In the taxonomy, the inadvertent flaws fall into six categories:

- validation error (incomplete or inconsistent)
- domain error
- serialization and aliasing
- inadequate identification and authentication
- boundary condition violation
- other exploitable logic errors

This list gives us a useful overview of the ways programs can fail to meet their security requirements. We leave our discussion of the pitfalls of identification and authentication for Chapter 4, in which we also investigate separation into execution domains. In this chapter, we address the other categories, each of which has interesting examples.

Domain error: ⇒ is a process of translating data structure into a format that can be stored

Serialization and aliasing ⇒ it describes a situation in which a data location in memory can be accessed through different symbolic means.

CONTROLS AGAINST PROGRAM THREATS

There are many ways a program can fail and many ways to turn the underlying faults into security failures. It is of course better to focus on prevention than cure; how do we use controls during software development—the specifying, designing, writing, and testing of the program—to find and eliminate the sorts of exposures we have discussed? The discipline of software engineering addresses this question more globally, devising approaches to ensure the quality of software. In this book, we provide an overview of several techniques that can prove useful in finding and fixing security flaws. Here we look at three types of controls: developmental, operating system, and administrative. We discuss each in turn.

Developmental Controls

Many controls can be applied during software development to ferret out and fix problems. So let us begin by looking at the nature of development itself, to see what tasks are involved in specifying, designing, building, and testing software.

The Nature of Software Development

Software development is often considered a solitary effort; a programmer sits with a specification or design and grinds out line after line of code. But in fact, software development is a collaborative effort, involving people with different skill sets who combine their expertise to produce a working product. Development requires people who can

- *specify* the system, by capturing the requirements and building a model of how the system should work from the users' point of view
- *design* the system, by proposing a solution to the problem described by the requirements and building a model of the solution
- *implement* the system, by using the design as a blueprint for building a working solution
- *test* the system, to ensure that it meets the requirements and implements the solution as called for in the design
- *review* the system at various stages, to make sure that the end products are consistent with the specification and design models
- *document* the system, so that users can be trained and supported
- *manage* the system, to estimate what resources will be needed for development and to track when the system will be done
- *maintain* the system, tracking problems found, changes needed, and changes made, and evaluating their effects on overall quality and functionality

One person could do all these things. But more often than not, a team of developers works together to perform these tasks. Sometimes a team member does more than one activity; a tester can take part in a requirements review, for example, or an implementer can write documentation. Each team is different, and team dynamics play a large role in the team's success.

We can examine both product and process to see how each contributes to quality and in particular to security as an aspect of quality. Let us begin with the product, to get a sense of how we recognize high-quality secure software.

Modularity, Encapsulation, and Information Hiding

Code usually has a long shelf-life, and it is enhanced over time as needs change and faults are found and fixed. For this reason, a key principle of software engineering is to create a design or code in small, self-contained units, called components or modules; when a system is written this way, we say that it is modular. Modularity offers advantages for program development in general and security in particular.

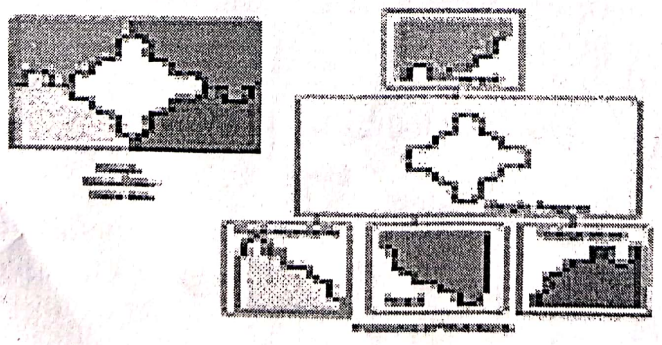
If a component is isolated from the effects of other components, then it is easier to trace a problem to the fault that caused it and to limit the damage the fault causes. It is also easier to maintain the system, since changes to an isolated component do not affect other components. And it is easier to see where vulnerabilities may lie if the component is isolated. We call this isolation encapsulation.

Information hiding is another characteristic of modular software. When information is hidden, each component hides its precise implementation or some other design decision from the others. Thus, when a change is needed, the overall design can remain intact while only the necessary changes are made to particular components.

Modularity

Modularization is the process of dividing a task into subtasks. This division is done on a logical or functional basis. Each component performs a separate, independent part of the task. Modularity is depicted in Figure below. The goal is to have each component meet four conditions:

- *single-purpose*: performs one function
- *small*: consists of an amount of information for which a human can readily grasp both structure and content
- *simple*: is of a low degree of complexity so that a human can readily understand the purpose and structure of the module
- *independent*: performs a task isolated from other modules



Often, other characteristics, such as having a single input and single output or using a limited set of programming constructs, help a component be modular. From a security standpoint, modularity should improve the likelihood that an implementation is correct.

In particular, smallness is an important quality that can help security analysts understand what each component does. That is, in good software, design and program units should be only as large as needed to perform their required functions. There are several advantages to having small, independent components.

- *Maintenance.* If a component implements a single function, it can be replaced easily with a revised one if necessary. The new component may be needed because of a change in requirements, hardware, or environment. Sometimes the replacement is an enhancement, using a smaller, faster, more correct, or otherwise better module. The interfaces between this component and the remainder of the design or code are few and well described, so the effects of the replacement are evident.
- *Understandability.* A system composed of many small components is usually easier to comprehend than one large, unstructured block of code.
- *Reuse.* Components developed for one purpose can often be reused in other systems. Reuse of correct, existing design or code components can significantly reduce the difficulty of implementation and testing.
- *Correctness.* A failure can be quickly traced to its cause if the components perform only one task each.
- *Testing.* A single component with well-defined inputs, output, and function can be tested exhaustively by itself, without concern for its effects on other modules (other than the expected function and output, of course).

Security analysts must be able to understand each component as an independent unit and be assured of its limited effect on other components.

A modular component usually has high cohesion and low coupling. By cohesion, we mean that all the elements of a component have a logical and functional reason for being there; every aspect of the component is tied to the component's single purpose. A highly cohesive component has a high degree of focus on the purpose; a low degree of cohesion means that the component's contents are an unrelated jumble of actions, often put together because of time-dependencies or convenience.

Coupling refers to the degree with which a component depends on other components in the system. Thus, low or loose coupling is better than high or tight coupling, because the loosely coupled components are free from unwitting interference from other components.

Encapsulation

Encapsulation hides a component's implementation details, but it does not necessarily mean complete isolation. Many components must share information with other

components, usually with good reason. However, this sharing is carefully documented so that a component is affected only in known ways by others in the system. Sharing is minimized so that the fewest interfaces possible are used. Limited interfaces reduce the number of covert channels that can be constructed.

Information Hiding

Developers who work where modularization is stressed can be sure that other components will have limited effect on the ones they write. Thus, we can think of a component as a kind of black box, with certain well-defined inputs and outputs and a well-defined function. Other components' designers do not need to know *how* the module completes its function; it is enough to be assured that the component performs its task in some correct manner.

Information hiding is desirable, because developers cannot easily and maliciously alter the components of others if they do not know how the components work.

These three characteristics—modularity, encapsulation, and information hiding—are fundamental principles of software engineering. They are also good security practices because they lead to modules that can be understood, analyzed, and trusted.

Peer Reviews

We turn next to the process of developing software. Certain practices and techniques can assist us in finding real and potential security flaws (as well as other faults) and fixing them before the system is turned over to the users. Of the many practices available for building what they call "solid software," Pfleeger et al. recommend several key techniques: [PFL01a]

- peer reviews
- hazard analysis
- testing
- good design
- prediction
- static analysis
- configuration management
- analysis of mistakes

Here, we look at each practice briefly, and we describe its relevance to security controls. We begin with peer reviews.

You have probably been doing some form of review for as many years as you have been writing code: desk-checking your work or asking a colleague to look over a routine to ferret out any problems. Today, a software review is associated with several formal process steps to make it more effective, and we review any artifact of the development process, not just code. But the essence of a review remains the same: sharing a product with colleagues

able to comment about its correctness. There are careful distinctions among three types of peer reviews:

- *Review:* The artifact is presented informally to a team of reviewers; the goal is consensus and buy-in before development proceeds further.
- *Walk-through:* The artifact is presented to the team by its creator, who leads and controls the discussion. Here, education is the goal, and the focus is on learning about a single document.
- *Inspection:* This more formal process is a detailed analysis in which the artifact is checked against a prepared list of concerns. The creator does not lead the discussion, and the fault identification and correction are often controlled by statistical measurements.

A wise engineer who finds a fault can deal with it in at least three ways:

1. by learning how, when and why errors occur,
2. by taking action to prevent mistakes, and
3. by scrutinizing products to find the instances and effects of errors that were missed.

Peer reviews address this problem directly. Unfortunately, many organizations give only lip service to peer review, and reviews are still not part of mainstream software engineering activities.

But there are compelling reasons to do reviews. An overwhelming amount of evidence suggests that various types of peer review in software engineering can be extraordinarily effective. For example, early studies at Hewlett-Packard in the 1980s revealed that those developers performing peer review on their projects enjoyed a very significant advantage over those relying only on traditional dynamic testing techniques, whether black-box or white-box. black-box testing, inspections, and software execution. It is clear that inspections discovered far more faults in the same period of time than other alternatives. [GRA87] This result is particularly compelling for large, secure systems, where live running for fault discovery may not be an option.

The effectiveness of reviews is reported repeatedly by researchers and practitioners. For instance, Jones [JON91] summarized the data in his large repository of project information to paint a picture of how reviews and inspections find faults relative to other discovery activities. Because products vary so wildly by size, Table below presents the fault discovery rates relative to the number of thousands of lines of code in the delivered product.

The inspection process involves several important steps: planning, individual preparation, a logging meeting, rework, and reinspection. Details about how to perform reviews and inspections can be found in software engineering books such as [PFL01] and [PFL01a].

During the review process, it is important to keep careful track of what each reviewer discovers and how quickly he or she discovers it. This log suggests not only whether particular reviewers need training but also whether certain kinds of faults are harder to

find than others. Additionally, a root cause analysis for each fault found may reveal that the fault could have been discovered earlier in the process. For example, a requirements fault that surfaces during a code review should probably have been found during a requirements review. If there are no requirements reviews, you can start performing them. If there are requirements reviews, you can examine why this fault was missed and then improve the requirements review process.

The fault log can also be used to build a checklist of items to be sought in future reviews. The review team can use the checklist as a basis for questioning what can go wrong and where. In particular, the checklist can remind the team of security breaches, such as unchecked buffer overflows, that should be caught and fixed before the system is placed in the field. A rigorous design or code review can locate trapdoors, Trojan horses, salami attacks, worms, viruses, and other program flaws. A crafty programmer can conceal some of these flaws, but the chance of discovery rises when competent programmers review the design and code, especially when the components are small and encapsulated. Management should use demanding reviews throughout development to ensure the ultimate security of the programs.

TABLE Faults Found During Discovery Activities.

Discovery Activity	Faults Found (Per Thousand Lines of Code)
Requirements review	2.5
Design review	5.0
Code inspection	10.0
Integration test	3.0
Acceptance test	2.0

Hazard Analysis

Hazard analysis is a set of systematic techniques intended to expose potentially hazardous system states. In particular, it can help us expose security concerns and then identify prevention or mitigation strategies to address them. That is, hazard analysis ferrets out likely causes of problems so that we can then apply an appropriate technique for preventing the problem or softening its likely consequences. Thus, it usually involves developing hazard lists, as well as procedures for exploring "what if" scenarios to trigger consideration of nonobvious hazards. The sources of problems can be lurking in any artifacts of the development or maintenance process, not just in the code, so a hazard analysis must be broad in its domain of investigation; in other words, hazard analysis is a system issue, not just a code issue. Similarly, there are many kinds of problems, ranging from incorrect code to unclear consequences of a particular action. A good hazard analysis takes all of them into account.

Although hazard analysis is generally good practice on any project, it is required in some regulated and critical application domains, and it can be invàluable for finding security

flaws. It is never too early to be thinking about the sources of hazards; the analysis should begin when you first start thinking about building a new system or when someone proposes a significant upgrade to an existing system. Hazard analysis should continue throughout the system life cycle; you must identify potential hazards that can be introduced during system design, installation, operation, and maintenance.

A variety of techniques support the identification and management of potential hazards. Among the most effective are hazard and operability studies (HAZOP), failure modes and effects analysis (FMEA), and fault tree analysis (FTA). HAZOP is a structured analysis technique originally developed for the process control and chemical plant industries. Over the last few years it has been adapted to discover potential hazards in safety-critical software systems. FMEA is a bottom-up technique applied at the system component level. A team identifies each component's possible faults or fault modes; then, it determines what could trigger the fault and what systemwide effects each fault might have. By keeping system consequences in mind, the team often finds possible system failures that are not made visible by other analytical means. FTA complements FMEA. It is a top-down technique that begins with a postulated hazardous system malfunction. Then, the FTA team works backwards to identify the possible pre cursors to the mishap. By tracing back from a specific hazardous malfunction, we can locate unexpected contributors to mishaps, and we then look for opportunities to mitigate the risks.

TABLE Perspectives for Hazard Analysis

	Known Cause	Unknown Cause
Known effect	Description of system behavior	Deductive analysis, including fault-tree analysis
Unknown effect	Inductive analysis, including failure modes and effects analysis	Exploratory analysis, including hazard and operability studies

Each of these techniques is clearly useful for finding and preventing security breaches. We decide which technique is most appropriate by understanding how much we know about causes and effects. For example, Table above suggests that when we know the cause and effect of a given problem, we can strengthen the description of how the system should behave. This clearer picture will help requirements analysts understand how a potential problem is linked to other requirements. It also helps designers understand exactly what the system should do and helps testers know how to test to verify that the system is behaving properly. If we can describe a known effect with unknown cause, we use deductive techniques such as fault tree analysis to help us understand the likely causes of the unwelcome behavior. Conversely, we may know the cause of a problem but not understand all the effects; here, we use inductive techniques such as failure modes and effects analysis to help us trace from cause to all possible effects. For example, suppose we know that a subsystem is unprotected and might lead to a security failure, but we do not

know how that failure will affect the rest of the system. We can use FMEA to generate a list of possible effects and then evaluate the trade-offs between extra protection and possible problems. Finally, to find problems about which we may not yet be aware, we can perform an exploratory analysis such as a hazard and operability study.

Testing

Testing is a process activity that homes in on product quality: making the product failure free or failure tolerant. Each software problem (especially when it relates to security) has the potential not only for making software fail but also for adversely affecting a business or a life. Thomas Young, head of NASA's investigation of the Mars lander failure, noted that "One of the things we kept in mind during the course of our review is that in the conduct of space missions, you get only one strike, not three. Even if thousands of functions are carried out flawlessly, just one mistake can be catastrophic to a mission." [NAS00] This same sentiment is true for security: The failure of one control exposes a vulnerability that is not ameliorated by any number of functioning controls. Testers improve software quality by finding as many faults as possible and by writing up their findings carefully so that developers can locate the causes and repair the problems if possible.

Testing usually involves several stages. First, each program component is tested on its own, isolated from the other components in the system. Such testing, known as *module testing*, *component testing*, or *unit testing*, verifies that the component functions properly with the types of input expected from a study of the component's design. Unit testing is done in a controlled environment whenever possible so that the test team can feed a predetermined set of data to the component being tested and observe what output actions and data are produced. In addition, the test team checks the internal data structures, logic, and boundary conditions for the input and output data.

When collections of components have been subjected to unit testing, the next step is ensuring that the interfaces among the components are defined and handled properly. Indeed, interface mismatch can be a significant security vulnerability. Integration testing is the process of verifying that the system components work together as described in the system and program design specifications.

Once we are sure that information is passed among components in accordance with the design, we test the system to ensure that it has the desired functionality. A function test evaluates the system to determine whether the functions described by the requirements specification are actually performed by the integrated system. The result is a functioning system.

The function test compares the system being built with the functions described in the developers' requirements specification. Then, a performance test compares the system with the remainder of these software and hardware requirements. It is during the function and performance tests that security requirements are examined, and the testers confirm that the system is as secure as it is required to be.

When the performance test is complete, developers are certain that the system functions according to their understanding of the system description. The next step is conferring with the customer to make certain that the system works according to customer expectations. Developers join the customer to perform an acceptance test, in which the system is checked against the customer's requirements description. Upon completion of acceptance testing, the accepted system is installed in the environment in which it will be used. A final installation test is run to make sure that the system still functions as it should. However, security requirements often state that a system should not do something. As Sidebar 3-6 demonstrates, it is difficult to demonstrate absence rather than presence.

The objective of unit and integration testing is to ensure that the code implemented the design properly; that is, that the programmers have written code to do what the designers intended. System testing has a very different objective: to ensure that the system does what the customer wants it to do. Regression testing, an aspect of system testing, is particularly important for security purposes. After a change is made to enhance the system or fix a problem, regression testing ensures that all remaining functions are still working and performance has not been degraded by the change.

Each of the types of tests listed here can be performed from two perspectives: black box and clear box (sometimes called white box). Black-box testing treats a system or its components as black boxes; testers cannot "see inside" the system, so they apply particular inputs and verify that they get the expected output. Clear-box testing allows visibility. Here, testers can examine the design and code directly, generating test cases based on the code's actual construction. Thus, clear-box testing knows that component

Pfleeger [PFL97] points out that security requirements resemble those for any other computing task, with one seemingly insignificant difference. Whereas most requirements say "the system will do this," security requirements add the phrase "and nothing more." As we pointed out in Chapter 1, security awareness calls for more than a little caution when a creative developer takes liberties with the system's specification. Ordinarily, we do not worry if a programmer or designer adds a little something extra. For instance, if the requirement calls for generating a file list on a disk, the "something more" might be sorting the list into alphabetical order or displaying the date it was created. But we would never expect someone to meet the requirement by displaying the list and then erasing all the files on the disk!

If we could determine easily whether an addition was harmful, we could just disallow harmful additions. But unfortunately we cannot. For security reasons, we must state explicitly the phrase "and nothing more" and leave room for negotiation in requirements definition on any proposed extensions.

It is natural for programmers to want to exercise their creativity in extending and expanding the requirements. But apparently benign choices, such as storing a value in a global variable or writing to a temporary file, can have serious security implications. And sometimes the best design approach for security is counterintuitive. For example, one cryptosystem attack depends on measuring the time to perform an encryption. That is, an

efficient implementation can undermine the system's security. The solution, oddly enough, is to artificially pad the encryption process with unnecessary computation so that short computations complete as slowly as long ones.

In another instance, an enthusiastic programmer added parity checking to a cryptographic procedure. Because the keys were generated randomly, the result was that 255 of the 256 encryptions failed the parity check, leading to the substitution of a fixed key—so that 255 of every 256 encryptions were being performed under the same key!

No technology can automatically distinguish between malicious and benign code. For this reason, we have to rely on a combination of approaches, including human-intensive ones, to help us detect when we are going beyond the scope of the requirements and threatening the system's security.

X uses CASE statements and can look for instances in which the input causes control to drop through to an unexpected line. Black-box testing must rely more on the required inputs and outputs because the actual code is not available for scrutiny.

The mix of techniques appropriate for testing a given system depends on the system's size, application domain, amount of risk, and many other factors. But understanding the effectiveness of each technique helps us know what is right for each particular system. For example, Olsen [OLS93] describes the development at Contel IPC of a system containing 184,000 lines of code. He tracked faults discovered during various activities, and found differences:

- 17.3 percent of the faults were found during inspections of the system design
- 19.1 percent during component design inspection
- 15.1 percent during code inspection
- 29.4 percent during integration testing
- 16.6 percent during system and regression testing

Only 0.1 percent of the faults were revealed after the system was placed in the field. Thus, Olsen's work shows the importance of using different techniques to uncover different kinds of faults during development; it is not enough to rely on a single method for catching all problems.

Who does the testing? From a security standpoint, independent testing is highly desirable; it may prevent a developer from attempting to hide something in a routine, or keep a subsystem from controlling the tests that will be applied to it. Thus, independent testing increases the likelihood that a test will expose the effect of a hidden feature.

Good Design

We saw earlier in this chapter that modularity, information hiding, and encapsulation are characteristics of good design. Several design-related process activities are particularly helpful in building secure software:

- using a philosophy of *fault tolerance*
- having a consistent *policy* for handling failures
- capturing the *design rationale* and history
- using design patterns

We describe each of these activities in turn.

Designs should try to anticipate faults and handle them in ways that minimize disruption and maximize safety and security. Ideally, we want our system to be fault free. But in reality, we must assume that the system will fail, and we make sure that unexpected failure does not bring the system down, destroy data, or destroy life. For example, rather than waiting for the system to fail (called passive fault detection), we might construct the system so that it reacts in an acceptable way to a failure's occurrence. Active fault detection could be practiced by, for instance, adopting a philosophy of mutual suspicion. Instead of assuming that data passed from other systems or components are correct, we can always check that the data are within bounds and of the right type or format. We can also use redundancy, comparing the results of two or more processes to see that they agree before using their result in a task.

If correcting a fault is too risky, inconvenient, or expensive, we can choose instead to practice fault tolerance: isolating the damage caused by the fault and minimizing disruption to users. Although fault tolerance is not always thought of as a security technique, it supports the idea, discussed in Chapter 8, that our security policy allows us to choose to mitigate the effects of a security problem instead of preventing it. For example, rather than install expensive security controls, we may choose to accept the risk that important data may be corrupted. If in fact a security fault destroys important data, we may decide to isolate the damaged data set and automatically revert to a backup data set so that users can continue to perform system functions.

More generally, we can design or code defensively, just as we drive defensively, by constructing a consistent policy for handling failures. Typically, failures include

- failing to provide a service
- providing the wrong service or data
- corrupting data

We can build into the design a particular way of handling each problem, selecting from one of three ways:

1. *Retrying*: restoring the system to its previous state and performing the service again, using a different strategy
2. *Correcting*: restoring the system to its previous state, correcting some system characteristic, and performing the service again, using the same strategy
3. *Reporting*: restoring the system to its previous state, reporting the problem to an error-handling component, and not providing the service again

This consistency of design helps us check for security vulnerabilities; we look for instances that are different from the standard approach.

Design rationales and history tell us the reasons the system is built one way instead of another. Such information helps us as the system evolves, so we can integrate the design of our security functions without compromising the integrity of the system's overall design.

Moreover, the design history enables us to look for patterns, noting what designs work best in which situations. For example, we can reuse patterns that have been successful in preventing buffer overflows, in ensuring data integrity, or in implementing user password checks.

Prediction

Among the many kinds of prediction we do during software development, we try to predict the risks involved in building and using the system. As we see in depth in Chapter 8, we must postulate which unwelcome events might occur and then make plans to avoid them or at least mitigate their effects. Risk prediction and management are especially important for security, where we are always dealing with unwanted events that have negative consequences. Our predictions help us decide which controls to use and how many. For example, if we think the risk of a particular security breach is small, we may not want to invest a large amount of money, time, or effort in installing sophisticated controls. Or we may use the likely risk impact to justify using several controls at once, a technique called "defense in depth."

Static Analysis

Before a system is up and running, we can examine its design and code to locate and repair security flaws. We noted earlier that the peer review process involves this kind of scrutiny. But static analysis is more than peer review, and it is usually performed before peer review. We can use tools and techniques to examine the characteristics of design and code to see if the characteristics warn us of possible faults lurking within. For example, a large number of levels of nesting may indicate that the design or code is hard to read and understand, making it easy for a malicious developer to bury dangerous code deep within the system.

To this end, we can examine several aspects of the design and code:

- control flow structure
- data flow structure
- data structure

The control flow is the sequence in which instructions are executed, including iterations and loops. This aspect of design or code can also tell us how often a particular instruction or routine is executed.

Data flow follows the trail of a data item as it is accessed and modified by the system. Many times, transactions applied to data are complex, and we use data flow measures to show us how and when each data item is written, read, and changed.

The data structure is the way in which the data are organized, independent of the system itself. For instance, if the data are arranged as lists, stacks, or queues, the algorithms for manipulating them are likely to be well understood and well defined.

There are many approaches to static analysis, especially because there are so many ways to create and document a design or program. Automated tools are available to generate not only numbers (such as depth of nesting or cyclomatic number) but also graphical depictions of control flow, data relationships, and the number of paths from one line of code to another. These aids can help us see how a flaw in one part of a system can affect other parts.

Configuration Management

When we develop software, it is important to know who is making which changes to what and when:

- *corrective changes*: maintaining control of the system's day-to-day functions
- *adaptive changes*: maintaining control over system modifications
- *perfective changes*: perfecting existing acceptable functions
- *preventive changes*: preventing system performance from degrading to unacceptable levels

We want some degree of control over the software changes so that one change does not inadvertently undo the effect of a previous change. And we want to control what is often a proliferation of different versions and releases. For instance, a product might run on several different platforms or in several different environments, necessitating different code to support the same functionality. Configuration management is the process by which we control changes during development and maintenance, and it offers several advantages in security. In particular, configuration management scrutinizes new and changed code to ensure, among other things, that security flaws have not been inserted, intentionally or accidentally.

Four activities are involved in configuration management:

1. configuration identification
2. configuration control and change management
3. configuration auditing
4. status accounting

Configuration identification sets up baselines to which all other code will be compared after changes are made. That is, we build and document an inventory of all components that comprise the system. The inventory includes not only the code you and your colleagues

may have created, but also database management systems, third-party software, libraries, test cases, documents, and more. Then, we "freeze" the baseline and carefully control what happens to it. When a change is proposed and made, it is described in terms of how the baseline changes.

Configuration control and configuration management ensure we can coordinate separate, related versions. For example, there may be closely related versions of a system to execute on 16-bit and 32-bit processors. Three ways to control the changes are separate files, deltas, and conditional compilation. If we use separate files, we have different files for each release or version. For example, we might build an encryption system in two configurations: one that uses a short key length, to comply with the law in certain countries, and another that uses a long key. Then, version 1 may be composed of components $^{A}1$ through ^{A}k and $^{B}1$, while version 2 is $^{A}1$ through ^{A}k and $^{B}2$, where $^{B}1$ and $^{B}2$ do key length. That is, the versions are the same except for the separate key processing files.

Alternatively, we can designate a particular version as the main version of a system, and then define other versions in terms of what is different. The difference file, called a delta, contains editing commands to describe the ways to transform the main version into the variation.

Finally, we can do conditional compilation, whereby a single code component addresses all versions, relying on the compiler to determine which statements to apply to which versions. This approach seems appealing for security applications because all the code appears in one place. However, if the variations are very complex, the code may be very difficult to read and understand.

Once a configuration management technique is chosen and applied, the system should be audited regularly. A configuration audit confirms that the baseline is complete and accurate, that changes are recorded, that recorded changes are made, and that the actual software (that is, the software as used in the field) is reflected accurately in the documents. Audits are usually done by independent parties taking one of two approaches: reviewing every entry in the baseline and comparing it with the software in use or sampling from a larger set just to confirm compliance. For systems with strict security constraints, the first approach is preferable, but the second approach may be more practical.

Finally, status accounting records information about the components: where they came from (for instance, purchased, reused, or written from scratch), the current version, the change history, and pending change requests.

All four sets of activities are performed by a configuration and change control board, or CCB. The CCB contains representatives from all organizations with a vested interest in the system, perhaps including customers, users, and developers. The board reviews all proposed changes and approves changes based on need, design integrity, future plans for the software, cost, and more. The developers implementing and testing the change work with a program librarian to control and update relevant documents and components; they also write detailed documentation about the changes and test results.

Configuration management offers two advantages to those of us with security concerns: protecting against unintentional threats and guarding against malicious ones. Both goals are addressed when the configuration management processes protect the integrity of programs and documentation. Because changes occur only after explicit approval from a configuration management authority, all changes are also carefully evaluated for side effects. With configuration management, previous versions of programs are archived, so a developer can retract a faulty change when necessary.

Malicious modification is made quite difficult with a strong review and configuration management process in place. In fact, as presented in Sidebar 3-7, poor configuration control has resulted in at least one system failure; that sidebar also confirms the principle of easiest penetration from Chapter 1. Once a reviewed program is accepted for inclusion in a system, the developer cannot sneak in to make small, subtle changes, such as inserting trapdoors. The developer has access to the running production program only through the CCB, whose members are alert to such security breaches.

In the 1970s the primary security assurance strategy was "penetration" or "tiger team" testing. A team of computer security experts would be hired to test the security of a system prior to its being pronounced ready to use. Often these teams worked for months to plan their tests.

The U.S. Department of Defense was testing the Multics system, which had been designed and built under extremely high security quality standards. Multics was being studied as a base operating system for the WWMCCS command and control system. The developers from M.I.T. were justifiably proud of the strength of the security of their system, and the sponsoring agency invoked the penetration team with a note of haughtiness. But the developers underestimated the security testing team.

Led by Roger Schell and Paul Karger, the team analyzed the code and performed their tests without finding major flaws. Then one team member thought like an attacker. He wrote a slight modification to the code to embed a trapdoor by which he could perform privileged operations as an unprivileged user. He then made a tape of this modified system, wrote a cover letter saying that a new release of the system was enclosed, and mailed the tape and letter to the site where the system was installed.

When it came time to demonstrate their work, the penetration team congratulated the Multics developers on generally solid security, but said they had found this one apparent failure, which the team member went on to show. The developers were aghast because they knew they had scrutinized the affected code carefully. Even when told the nature of the trapdoor that had been added, the developers could not find it. [KAR74, KAR02]

Lessons from Mistakes

One of the easiest things we can do to enhance security is learn from our mistakes. As we design and build systems, we can document our decisions—not only what we decided to do and why, but also what we decided *not* to do and why. Then, after the system is up and

running, we can use information about the failures (and how we found and fixed the underlying faults) to give us a better understanding of what leads to vulnerabilities and their exploitation.

From this information, we can build checklists and codify guidelines to help ourselves and others. That is, we do not have to make the same mistake twice, and we can assist other developers in staying away from the mistakes we made. The checklists and guidelines can be invaluable, especially during reviews and inspections, in helping reviewers look for typical or common mistakes that can lead to security flaws. For instance, a checklist can remind a designer or programmer to make sure that the system checks for buffer overflows. Similarly, the guidelines can tell a developer when data require password protection or some other type of restricted access.

Proofs of Program Correctness

A security specialist wants to be certain that a given program computes a particular result, computes it correctly, and does nothing beyond what it is supposed to do. Unfortunately, results in computer science theory (see [PFL85] for a description) indicate that we cannot know with certainty that two programs do exactly the same thing. That is, there can be no general decision procedure which, given any two programs, determines if the two are equivalent. This difficulty results from the "halting problem," which states that there is no general technique to determine whether an arbitrary program will halt when processing an arbitrary input.

In spite of this disappointing general result, a technique called program verification can demonstrate formally the "correctness" of certain specific programs. Program verification involves making initial assertions about the inputs and then checking to see if the desired output is generated. Each program statement is translated into a logical description about its contribution to the logical flow of the program. Finally, the terminal statement of the program is associated with the desired output. By applying a logic analyzer, we can prove that the initial assumptions, through the implications of the program statements, produce the terminal condition. In this way, we can show that a particular program achieves its goal. Proving program correctness, although desirable and useful, is hindered by several factors.

- Correctness proofs depend on a programmer or logician to translate a program's statements into logical implications. Just as programming is prone to errors, so also is this translation.
- Deriving the correctness proof from the initial assertions and the implications of statements is difficult, and the logical engine to generate proofs runs slowly. The speed of the engine degrades as the size of the program increases, so proofs of correctness are even less appropriate for large programs.
- The current state of program verification is less well developed than code production. As a result, correctness proofs have not been consistently and successfully applied to large production systems.

CH-3

Malicious software

Malware, short for malicious software, is any software used to disrupt computer operations, gather sensitive information, gain access to private computer systems, or display unwanted advertising.[1] Before the term malware was coined by YisraelRadai in 1990, malicious software was referred to as computer viruses. The first category of malware propagation concerns parasitic software fragments that attach themselves to some existing executable content. The fragment may be machine code that infects some existing application, utility, or system program, or even the code used to boot a computer system. Malware is defined by its malicious intent, acting against the requirements of the computer user, and does not include software that causes unintentional harm due to some deficiency.

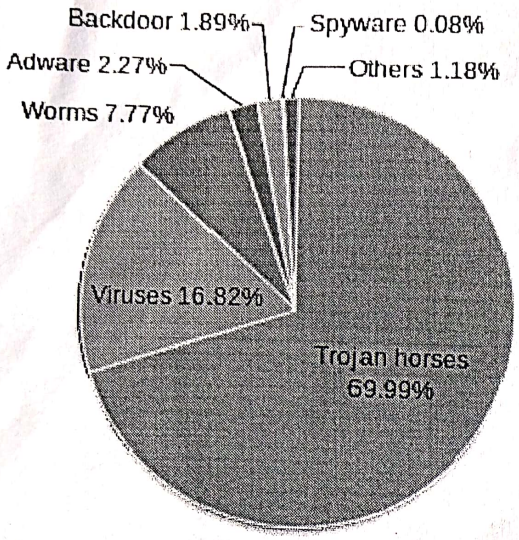
Malware may be stealthy, intended to steal information or spy on computer users for an extended period without their knowledge, as for example Regin, or it may be designed to cause harm, often as sabotage (e.g., Stuxnet), or to extort payment (CryptoLocker). 'Malware' is an umbrella term used to refer to a variety of forms of hostile or intrusive software,[4] including computer viruses, worms, trojan horses, ransomware, spyware, adware, scareware, and other malicious programs. It can take the form of executable code, scripts, active content, and other software.[5] Malware is often disguised as, or embedded in, non-malicious files. As of 2011 the majority of active malware threats were worms or trojans rather than viruses.

In law, malware is sometimes known as a computer contaminant, as in the legal codes of several U.S. states.

Spyware or other malware is sometimes found embedded in programs supplied officially by companies, e.g., downloadable from websites, that appear useful or attractive, but may have, for example, additional hidden tracking functionality that gathers marketing statistics. An example of such software, which was described as illegitimate, is the Sony rootkit, a Trojan embedded into CDs sold by Sony, which silently installed and concealed itself on purchasers' computers with the intention of preventing illicit copying; it also reported on users' listening habits, and unintentionally created vulnerabilities that were exploited by unrelated malware.

Software such as anti-virus, anti-malware, and firewalls are used to protect against activity identified as malicious, and to recover from attacks.

Purposes



Malware by categories March 16, 2011

Malware by categories on 16 March 2011.

Many early infectious programs, including the first Internet Worm, were written as experiments or pranks. Today, malware is used by both black hat hackers and governments, to steal personal, financial, or business information.

Malware is sometimes used broadly against government or corporate websites to gather guarded information,[13] or to disrupt their operation in general. However, malware is often used against individuals to gain information such as personal identification numbers or details, bank or credit card numbers, and passwords. Left unguarded, personal and networked computers can be at considerable risk against these threats. (These are most frequently defended against by various types of firewall, anti-virus software, and network hardware).[14]

Since the rise of widespread broadbandInternet access, malicious software has more frequently been designed for profit. Since 2003, the majority of widespread viruses and worms have been designed to take control of users' computers for illicit purposes.[15] Infected "zombie computers" are used to send email spam, to host contraband data such as child pornography,[16] or to engage in distributed denial-of-serviceattacks as a form of extortion.

Programs designed to monitor users' web browsing, display unsolicited advertisements, or redirect affiliate marketing revenues are called spyware. Spyware programs do not spread like viruses; instead they are generally installed by exploiting security holes. They can also be hidden and packaged together with unrelated user-installed software.[18]

Ransomware affects an infected computer in some way, and demands payment to reverse the damage. For example, programs such as CryptoLockerencrypt files securely, and only decrypt them on payment of a substantial sum of money.

Some malware is used to generate money by click fraud, making it appear that the computer user has clicked an advertising link on a site, generating a payment from the advertiser. It was estimated in 2012 that about 60 to 70% of all active malware used some kind of click fraud, and 22% of all ad-clicks were fraudulent.[19]

Malware is usually used for criminal purposes, but can be used for sabotage, often without direct benefit to the perpetrators. One example of sabotage was Stuxnet, used to destroy very specific industrial equipment. There have been politically motivated attacks that have spread over and shut down large computer networks, including massive deletion of files and corruption of master boot records, described as "computer killing". Such attacks were made on Sony Pictures Entertainment (25 November 2014, using malware known as Shamoon or W32.Disttrack) and Saudi Aramco (August 2012).

Proliferation

Preliminary results from Symantec published in 2008 suggested that "the release rate of malicious code and other unwanted programs may be exceeding that of legitimate software applications." According to F-Secure, "As much malware [was] produced in 2007 as in the previous 20 years altogether."[1] Malware's most common pathway from criminals to users is through the Internet: primarily by e-mail and the World Wide Web.[24]

The prevalence of malware as a vehicle for Internet crime, along with the challenge of anti-malware software to keep up with the continuous stream of new malware, has seen the adoption of a new mindset for individuals and businesses using the Internet. With the amount of malware currently being distributed, some percentage of computers are currently assumed to be infected. For businesses, especially those that sell mainly over the Internet, this means they need to find a way to operate despite security concerns. The result is a greater emphasis on back-office protection designed to protect against advanced malware operating on customers' computers.[25] A 2013 Webroot study shows that 64% of companies allow remote access to servers for 25% to 100% of their workforce and that companies with more than 25% of their employees accessing servers remotely have higher rates of malware threats.

On 29 March 2010, Symantec Corporation named Shaoxing, China, as the world's malware capital.[27] A 2011 study from the University of California, Berkeley, and the Madrid Institute for Advanced Studies published an article in *Software Development Technologies*, examining how entrepreneurial hackers are helping enable the spread of malware by offering access to computers for a price. Microsoft reported in May 2011 that one in every 14 downloads from the Internet may now contain malware code. Social media, and Facebook in particular, are seeing a rise in the number of tactics used to spread malware to computers.

A 2014 study found that malware is being increasingly aimed at mobile devices such as smartphones as they increase in popularity.

Infectious malware: viruses and worms

The best-known types of malware, viruses and worms, are known for the manner in which they spread, rather than any specific types of behavior. The term *computer virus* is used for a program that embeds itself in some other executable software (including the operating system itself) on the target system without the user's consent and when that is run causes the virus to spread to other executables. On the other hand, a *worm* is a stand-alone malware program that *actively* transmits itself over a network to infect other computers. These definitions lead to the observation that a virus requires the user to run an infected program or operating system for the virus to spread, whereas a worm spreads itself.[30]

Concealment: Viruses, trojan horses, rootkits, backdoors and evasion

These categories are not mutually exclusive, so malware may use multiple techniques.[31] This section only applies to malware designed to operate undetected, not sabotage and ransomware.

A computer program usually hidden within another seemingly innocuous program that produces copies of itself and inserts them into other programs or files, and that usually performs a malicious action (such as destroying data).[32]

Trojan horses

In computing, Trojan horse, or Trojan, is any malicious computer program which misrepresents itself to appear useful, routine, or interesting in order to persuade a victim to install it. The term is derived from the Ancient Greek story of the wooden horse that was used to help Greek troops invade the city of Troy by stealth.[33][34][35][36][37]

Trojans are generally spread by some form of social engineering, for example where a user is duped into executing an e-mail attachment disguised to be unsuspicious, (e.g., a routine form to be filled in), or by drive-by download. Although their payload can be anything, many modern forms act as a backdoor, contacting a controller which can then have unauthorized access to the affected computer.[38] While Trojans and backdoors are not easily detectable by themselves, computers may appear to run slower due to heavy processor or network usage.

Unlike computer viruses and worms, Trojans generally do not attempt to inject themselves into other files or otherwise propagate themselves.[39]

Rootkits

. Once a malicious program is installed on a system, it is essential that it stays concealed, to avoid detection. Software packages known as *rootkits* allow this concealment, by modifying the host's operating system so that the malware is hidden from the user. Rootkits can prevent a malicious process from being visible in the system's list of processes, or keep its files from being read.[40]

Some malicious programs contain routines to defend against removal, not merely to hide themselves. An early example of this behavior is recorded in the Jargon File tale of a pair of programs infesting a Xerox CP-V time sharing system:

> Each ghost-job would detect the fact that the other had been killed, and would start a new copy of the recently stopped program within a few milliseconds. The only way to kill both ghosts was to kill them simultaneously (very difficult) or to deliberately crash the system.[41]

Backdoors

A backdoor is a method of bypassing normal authentication procedures, usually over a connection to a network such as the Internet. Once a system has been compromised, one or more backdoors may be installed in order to allow access in the future,[42] invisibly to the user.

The idea has often been suggested that computer manufacturers preinstall backdoors on their systems to provide technical support for customers, but this has never been reliably verified. It was reported in 2014 that US government agencies had been diverting computers purchased by those considered "targets" to secret workshops where software or hardware permitting remote access by the agency was installed, considered to be among the most productive operations to obtain access to networks around the world.[43] Backdoors may be installed by Trojan horses, worms, implants, or other methods.[44][45]

Evasion

Since the beginning of 2015, a sizable portion of malware utilizes a combination of many techniques designed to avoid detection and analysis.[46]

- The most common evasion technique is when the malware evades analysis and detection by fingerprinting the environment when executed.[47]
- The second most common evasion technique is confusing automated tools' detection methods. This allows malware to avoid detection by technologies such as signature-based antivirus software by changing the server used by the malware.[48]
- The third most common evasion technique is timing-based evasion. This is when malware runs at certain times or following certain actions taken by the user, so it executes during certain vulnerable periods, such as during the boot process, while remaining dormant the rest of the time.
- The fourth most common evasion technique is done by obfuscating internal data so that automated tools do not detect the malware.

- An increasingly common technique is adware that uses stolen certificates to disable anti-malware and virus protection; technical remedies are available to deal with the adware.[50]

Nowadays, one of the most sophisticated and stealthy ways of evasion is the use information hiding techniques, namely Stegomalware.

Vulnerability to malware

Security defects in software

Malware exploits security defects (security bugs or vulnerabilities) in the design of the operating system, in applications (such as browsers, e.g. older versions of Microsoft Internet Explorer supported by Windows XP[51]), or in vulnerable versions of browser plugins such as Adobe Flash Player, Adobe Acrobat or Reader, or Java SE.[52][53] Sometimes even installing new versions of such plugins does not automatically uninstall old versions. Security advisories from plug-in providers announce security-related updates.[54] Common vulnerabilities are assigned CVE IDs and listed in the US National Vulnerability Database. SecuniaPSI[55] is an example of software, free for personal use, that will check a PC for vulnerable out-of-date software, and attempt to update it.

Malware authors target bugs, or loopholes, to exploit. A common method is exploitation of a buffer overrun vulnerability, where software designed to store data in a specified region of memory does not prevent more data than the buffer can accommodate being supplied. Malware may provide data that overflows the buffer, with malicious executable code or data after the end; when this payload is accessed it does what the attacker, not the legitimate software, determines.

Insecure design or user error

Early PCs had to be booted from floppy disks; when built-in hard drives became common the operating system was normally started from them, but it was possible to boot from another boot device if available, such as a floppy disk, CD-ROM, DVD-ROM, USB flash drive or network. It was common to configure the computer to boot from one of these devices when available. Normally none would be available; the user would intentionally insert, say, a CD into the optical drive to boot the computer in some special way, for example to install an operating system. Even without booting, computers can be configured to execute software on some media as soon as they become available, e.g. to autorun a CD or USB device when inserted.

Malicious software distributors would trick the user into booting or running from an infected device or medium; for example, a virus could make an infected computer add autorunnable code to any USB stick plugged into it; anyone who then attached the stick to another computer set to autorun from USB would in turn become infected, and also pass on the infection in the same way.[56] More generally, any device that plugs into a USB port—"including gadgets like lights, fans, speakers, toys, even a digital microscope"—can be

used to spread malware. Devices can be infected during manufacturing or supply if quality control is inadequate.[56]

This form of infection can largely be avoided by setting up computers by default to boot from the internal hard drive, if available, and not to autorun from devices.[56] Intentional booting from another device is always possible by pressing certain keys during boot.

Older email software would automatically open HTML email containing potentially malicious JavaScript code; users may also execute disguised malicious email attachments and infected executable files supplied in other ways.

Over-privileged users and over-privileged code

In computing, privilege refers to how much a user or program is allowed to modify a system. In poorly designed computer systems, both users and programs can be assigned more privileges than they should be, and malware can take advantage of this. The two ways that malware does this is through overprivileged users and overprivileged code.

Some systems allow all users to modify their internal structures, and such users today would be considered over-privileged users. This was the standard operating procedure for early microcomputer and home computer systems, where there was no distinction between an *administrator* or *root*, and a regular user of the system. In some systems, non-administrator users are over-privileged by design, in the sense that they are allowed to modify internal structures of the system. In some environments, users are over-privileged because they have been inappropriately granted administrator or equivalent status.

Some systems allow code executed by a user to access all rights of that user, which is known as over-privileged code. This was also standard operating procedure for early microcomputer and home computer systems. Malware, running as over-privileged code, can use this privilege to subvert the system. Almost all currently popular operating systems, and also many scripting applications allow code too many privileges, usually in the sense that when a user executes code, the system allows that code all rights of that user. This makes users vulnerable to malware in the form of e-mail attachments, which may or may not be disguised.

Use of the same operating system

- Homogeneity: e.g. when all computers in a network run the same operating system; upon exploiting one, one worm can exploit them all:[57] For example, Microsoft Windows or Mac OS X have such a large share of the market that concentrating on either could enable an exploited vulnerability to subvert a large number of systems. Instead, introducing diversity, purely for the sake of robustness, could increase short-term costs for training and maintenance. However, having a few diverse nodes could deter total shutdown of the network as long as all the nodes are not part of the same directory service for authentication, and allow those nodes to help with recovery of the infected nodes. Such separate, functional redundancy could avoid

the cost of a total shutdown, at the cost of increased complexity and reduced usability in terms of single sign-on authentication.

Anti-malware strategies

As malware attacks become more frequent, attention has begun to shift from viruses and spyware protection, to malware protection, and programs that have been specifically developed to combat malware. (Other preventive and recovery measures, such as backup and recovery methods, are mentioned in the computer virus article).

Anti-virus and anti-malware software

A specific component of anti-virus and anti-malware software, commonly referred to as an on-access or real-time scanner, hooks deep into the operating system's core or kernel and functions in a manner similar to how certain malware itself would attempt to operate, though with the user's informed permission for protecting the system. Any time the operating system accesses a file, the on-access scanner checks if the file is a 'legitimate' file or not. If the file is identified as malware by the scanner, the access operation will be stopped, the file will be dealt with by the scanner in a pre-defined way (how the anti-virus program was configured during/post installation), and the user will be notified.[citation needed] This may have a considerable performance impact on the operating system, though the degree of impact is dependent on how well the scanner was programmed. The goal is to stop any operations the malware may attempt on the system before they occur, including activities which might exploit bugs or trigger unexpected operating system behavior.

Anti-malware programs can combat malware in two ways:

1. They can provide real time protection against the installation of malware software on a computer. This type of malware protection works the same way as that of antivirus protection in that the anti-malware software scans all incoming network data for malware and blocks any threats it comes across.
2. Anti-malware software programs can be used solely for detection and removal of malware software that has already been installed onto a computer. This type of anti-malware software scans the contents of the Windows registry, operating system files, and installed programs on a computer and will provide a list of any threats found, allowing the user to choose which files to delete or keep, or to compare this list to a list of known malware components, removing files that match.[58]

Real-time protection from malware works identically to real-time antivirus protection: the software scans disk files at download time, and blocks the activity of components known to represent malware. In some cases, it may also intercept attempts to install start-up items or to modify browser settings. Because many malware components are installed as a result of browser exploits or user error, using security software (some of which are anti-malware, though many are not) to "sandbox" browsers (essentially isolate the browser from the computer and hence any malware induced change) can also be effective in helping to restrict any damage done.[citation needed]

Examples of Microsoft Windows antivirus and anti-malware software include the optional Microsoft Security Essentials[1] (for Windows XP, Vista, and Windows 7) for real-time protection, the Windows Malicious Software Removal Tool (now included with Windows (Security) Updates on "Patch Tuesday", the second Tuesday of each month), and Windows Defender (an optional download in the case of Windows XP, incorporating MSE functionality in the case of Windows 8 and later).[1] Additionally, several capable antivirus software programs are available for free download from the Internet (usually restricted to non-commercial use). Tests found some free programs to be competitive with commercial ones.[62] Microsoft's System File Checker can be used to check for and repair corrupted system files.

Some viruses disable System Restore and other important Windows tools such as Task Manager and Command Prompt. Many such viruses can be removed by rebooting the computer, entering Windows safe mode with networking,[63] and then using system tools or Microsoft Safety Scanner.

Hardware implants can be of any type, so there can be no general way to detect them.

Website security scans

As malware also harms the compromised websites (by breaking reputation, blacklisting in search engines, etc.), some websites offer vulnerability scanning.[65][66][67][68] Such scans check the website, detect malware, may note outdated software, and may report known security issues.

"Air gap" isolation or "Parallel Network"

As a last resort, computers can be protected from malware, and infected computers can be prevented from disseminating trusted information, by imposing an "air gap" (i.e. completely disconnecting them from all other networks). However, malware can still cross the air gap in some situations. For example, removable media can carry malware across the gap. In December 2013 researchers in Germany showed one way that an apparent air gap can be defeated.

"AirHopper", "BitWhisper",[1] "GSMem" and "Fansmitter" are four techniques introduced by researchers that can leak data from air-gapped computers using electromagnetic, thermal and acoustic emissions.

Grayware

Grayware is a term applied to unwanted applications or files that are not classified as malware, but can worsen the performance of computers and may cause security risks.[74]

It describes applications that behave in an annoying or undesirable manner, and yet are less serious or troublesome than malware. Grayware encompasses spyware, adware, fraudulent dialers, joke programs, remote access tools and other unwanted programs that

harm the performance of computers or cause inconvenience. The term came into use around 2004.

Another term, PUP, which stands for *Potentially Unwanted Program* (or PUA *Potentially Unwanted Application*),[1] refers to applications that would be considered unwanted despite often having been downloaded by the user, possibly after failing to read a download agreement. PUPs include spyware, adware, and fraudulent dialers. Many security products classify unauthorised key generators as grayware, although they frequently carry true malware in addition to their ostensible purpose.

Software maker Malwarebytes lists several criteria for classifying a program as a PUP. Some adware (using stolen certificates) disables anti-malware and virus protection; technical remedies are available.

History of viruses and worms

Before Internet access became widespread, viruses spread on personal computers by infecting the executable boot sectors of floppy disks. By inserting a copy of itself into the machine code instructions in these executables, a virus causes itself to be run whenever a program is run or the disk is booted. Early computer viruses were written for the Apple II and Macintosh, but they became more widespread with the dominance of the IBM PC and MS-DOS system. Executable-infecting viruses are dependent on users exchanging software or boot-able floppies and thumb drives so they spread rapidly in computer hobbyist circles.

The first worms, network-borne infectious programs, originated not on personal computers, but on multitasking Unix systems. The first well-known worm was the Internet Worm of 1988, which infected SunOS and VAXBSD systems. Unlike a virus, this worm did not insert itself into other programs. Instead, it exploited security holes (vulnerabilities) in network server programs and started itself running as a separate process.

This same behavior is used by today's worms as wellWith the rise of the Microsoft Windows platform in the 1990s, and the flexible macros of its applications, it became possible to write infectious code in the macro language of Microsoft Word and similar programs. These *macro viruses* infect documents and templates rather than applications (executables), but rely on the fact that macros in a Word document are a form of executable code.

Today, worms are most commonly written for the Windows OS, although a few like Mare-D and the L10n worm[1] are also written for Linux and Unix systems. Worms today work in the same basic way as 1988's Internet Worm: they scan the network and use vulnerable computers to replicate. Because they need no human intervention, worms can spread with incredible speed. The SQL Slammer infected thousands of computers in a few minutes in 2003.