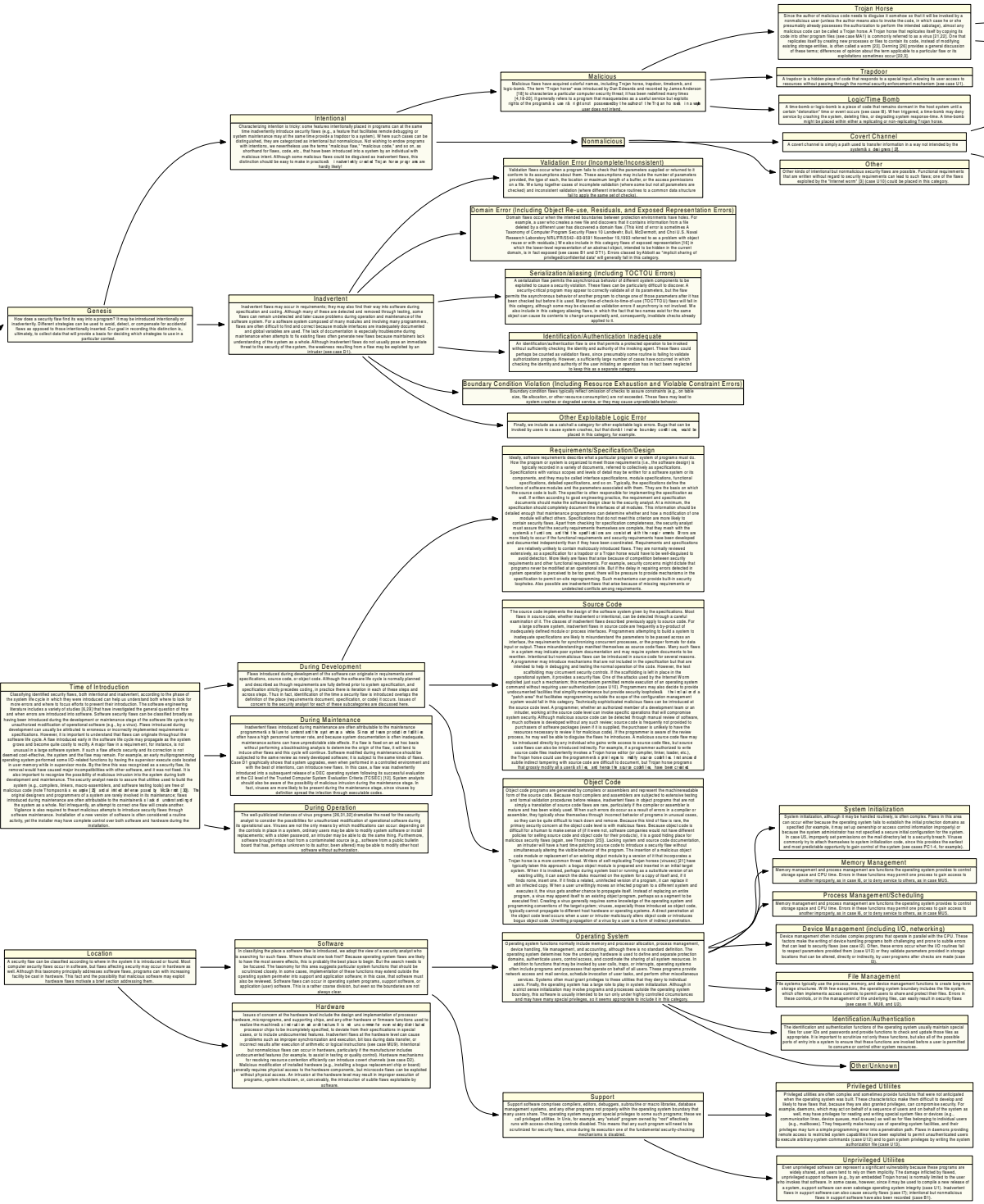
Root
Genesis
How does a security flaw find its way into a program? It may be introduced intentionally or inadvertently. Different strategies can be used to avoid, detect, or compensate for accidental flaws as opposed to those intentionally inserted. Our goal in recording this distinction is, ultimately, to collect data that will provide a basis for deciding which strategies to use in a particular context.
Time of Introduction
Classifying identified security flaws, both intentional and inadvertent, according to the phase of the system life cycle in which they were introduced can help us understand both where to look for more errors and where to focus efforts to prevent their introduction. The software engineering literature includes a variety of studies [8,29] that have investigated the general question of how and when errors are introduced into software. Software security flaws can be classified broadly as having been introduced during the development or maintenance stage of the software life cycle or by unauthorized modification of operational software (e.g., by a virus). Flaws introduced during development may be the result of incorrect specifications, incomplete specifications, or specifications. However, it is important to understand that flaws can originate throughout the software life cycle. A flaw introduced early in the software life cycle may propagate as the system grows and become quite costly to rectify. A major flaw in a requirement, for instance, is not unusual in a large software system. If such a flaw affects security and its correction is not deemed cost-effective, the system and the flaw may remain. For example, a flaw in a Multiprogramming operating system performed some I/O-related functions by having the supervisor execute code located in user memory while in supervisor mode. By the time this was recognized as a security flaw, its removal would have caused major incompatibilities with other software, and it was not fixed. It is clear that flaws introduced during the development process can persist into the maintenance and development stage. The security analyst needs to assure that utilities used to build the system (e.g., compilers, linkers, macro-assemblers, and software testing tools) are free of malicious code (note Thompson's seminal paper [35] and other more recent work [36]). The original designers and programmers of a system are rarely involved in its maintenance; flaws introduced during maintenance are often attributable to the maintenance staff's lack of understanding of the system as a whole. Not infrequently, an attempt to correct one flaw will create another. Flaws introduced during the operational phase are dominated by security flaws introduced by viruses or trojan horses. Installation of a new version of software is often considered a routine activity, yet the installer may have complete control over both software and hardware during the installation.
Location
A security flaw can be classified according to where in the system it is introduced or found. Most computer security flaws occur in software, but flaws affecting security may occur in hardware as well. Although this taxonomy principally addresses software flaws, programs can with increasing facility be cast in hardware. This fact and the possibility that malicious software may exploit hardware flaws motivate a brief section addressing them.
Intentional
Characterizing intention is tricky: some features intentionally placed in programs can at the same time inadvertently introduce security flaws (e.g., a feature that facilitates remote debugging or system maintenance may at the same time provide a trapdoor to a system). Where such cases can be distinguished, they are categorized as intentional but nonmalicious. Not wishing to endow programs with intentions, we nevertheless use the terms "malicious flaw," "malicious code," and so on, as shorthand for flaws, code, etc., that have been introduced into a system by an individual with malicious intent. Although some malicious flaws could be disguised as inadvertent flaws, this distinction should be easy to make in practice: intentionally created Trojan horse programs are hardly likely
Malicious
Malicious flaws have acquired colorful names, including Trojan horse, trapdoor, timebomb, and logic-bomb. The term "Trojan horse" was introduced by Dan Edwards and recorded by James Anderson [18] to characterize a particular computer security threat. It has been redefined many times [4,16-20]. In general, a Trojan horse is a program containing some code which, when executed, gains the program's user (e.g., a game player) increased privileges. The author of the Trojan horse could, in some cases, use the increased privileges granted to the game player's processes to modify files belonging to other users, for example, or to transmit sensitive information over some network.
Nonmalicious
Inadvertent
Inadvertent flaws may occur in requirements; they may also find their way into software during specification and coding. Although many of these are detected and removed through testing, some may remain undetected for years. Although inadvertent flaws may occur throughout the life cycle of a software system. For a software system composed of many modules and involving many programmers, flaws are often difficult to find and correct because module interfaces are inadequately documented and global variables are used. The lack of documentation is especially troublesome during maintenance when attempts to fix existing flaws often generate new flaws because maintainers lack understanding of the system as a whole. Although inadvertent flaws do not usually pose an immediate threat to the security of the system, the weakness resulting from a flaw may be exploited by an intruder (see case D1).
During Development
Flaws introduced during development of the software can originate in requirements and specifications, source code, or object code. Although the software life cycle is normally planned and described as though requirements are fully defined prior to system specification, and specification strictly precedes coding, in practice there is iteration in each of these steps and movement back and forth among them. Although flaws can arise in the requirements phase, a definition of the phase (requirements document, specification, or code) it occurs, issues of concern to the security analyst for each of these subcategories are discussed here.
During Maintenance
Inadvertent flaws introduced during maintenance are often attributable to the maintenance programmer's failure to understand the system as a whole. Since a flaw in a utility or application often have a high personnel turnover rate, and because system documentation is often inadequate, maintenance actions can have unpredictable side effects. If a flaw is fixed on an ad hoc basis without performing a backtracking analysis to determine the origin of the flaw, it will tend to recur as the system ages and undergoes successive modifications. Maintenance personnel may be subjected to the same review as newly developed software. It is subject to the same kinds of flaws. Case D1 graphically shows that system upgrades, even when performed in a controlled environment and with the best of intentions, can introduce new flaws. In this case, a flaw was inadvertently introduced into a subsequent release of a DEC operating system following its successful evaluation at the C2 level of the Trusted Computer System Evaluation Criteria (TCSEC) [10]. System analysts should also be aware of the possibility of malicious intrusion during the maintenance stage. In fact, viruses are more likely to be present during the maintenance stage, since viruses by definition spread the infection through executable codes.
During Operation
The well-publicized instances of virus programs [26,31,32] dramatize the need for the security analyst to consider the possibilities for unauthorized modification of operational software during its operational use. Viruses are not the only means by which modifications can occur: depending on the controls in place in a system, ordinary users may be able to modify system software or install replacements; with a stolen password, an intruder may be able to do the same thing. Furthermore, software brought into a host from a contaminated source (e.g., software from a public bulletin board that has, perhaps unknown to its author, been altered) may be able to modify other host software without authorization.
Software
In classifying the place a software flaw is introduced, we adopt the view of a security analyst who is searching for such flaws. Where should one look first? Because operating system flaws are likely to have far-ranging effects, this is probably the best place to begin. But the search needs to be focused. The taxonomy for this area suggests particular system functions that should be scrutinized closely. In some cases, implementation of these functions may extend outside the operating system perimeter into support and application software; in this case, that software must also be scrutinized closely. In practice, the line between system and application software is not always clear, although in most cases it is clear enough that most users can be informed of which applications (user) software. This is a rather coarse distinction, but even so the boundaries are not always clear.
Hardware
Issues of concern at the hardware level include the design and implementation of processor hardware, microprograms, and supporting chips, and any other hardware or firmware functions used to realize the machine's instruction architecture. It is not unusual for even relatively distributed processor chips to be incompletely specified, to deviate from their specifications in special cases, or to include undocumented features. Inadvertent flaws at the hardware level can cause unexpected, undesirable results or conditions to occur. For example, there have been cases in which incorrect results after execution of arithmetic or logical instructions (see case M3B). Inadvertent but nonmalicious flaws can occur in hardware, particularly if the manufacturer includes undocumented features (for example, to assist in testing or quality control). Hardware mechanisms for resolving resource contention efficiently can introduce covert channels (see case D2). Malicious modification of installed hardware (e.g., installing a bogus replacement chip or board) generally requires physical access to the hardware components, but microcode flaws can be exploited without physical access. An intruder at the hardware level may result in improper execution of programs, system shutdown, or, conceivably, the introduction of subtle flaws exploitable by software.
Trojan Horse
Since the author of malicious code needs to disguise it somehow so that it will be invoked by a nonmalicious user (unless the author means also to invoke the code, in which case he or she presumably already possesses the authorization to perform the intended sabotage), almost any malicious code can be called a Trojan horse. A Trojan horse that replicates itself by copying its code into other program files (see case MA1) is commonly referred to as a virus [21,22]. One that replicates by creating new files containing only its code is called a worm [23]. Instead of replicating existing storage entities, it can also create a new one. Denning [36] provides a general discussion of these terms; differences of opinion about the term applicable to a particular flaw or its exploitation sometimes occur [22,23].
Non-Replicating
Replicating (virus)
Trapdoor
A trapdoor is a hidden piece of code that responds to a special input, allowing its user access to resources without passing through the normal security enforcement mechanism (see case U1).
Logic/Time Bomb
A time-bomb or logic-bomb is a piece of code that remains dormant in the host system until a certain "detonation" time or event occurs (see case B). When triggered, a time-bomb may deny service by crashing the system, deleting files, or degrading system response-time. A time-bomb might be placed within either a replicating or non-replicating Trojan horse.
Storage
Covert channels are frequently classified as either storage or timing channels. A storage channel transfers information through the setting of bits by one program and the reading of those bits by another. Examples would include using a file locking mechanism used by a word processor as a covert signaling channel. Examples would include using a file intended to hold only audit information to convey user passwords. Storage channels are often characterized as resulting from the writing of a file by one process and the reading of a file by another, with the latter process being able to infer something about the contents of the file.
Covert Channel
A covert channel is simply a path used to transfer information in a way not intended by the system's designers [8].
Timing
Covert channels are frequently classified as either storage or timing channels. Timing channels convey information by modulating some aspect of system behavior over time, so that the program receiving the information can observe system behavior (e.g., the system's response time) and, in certain transaction requires to execute, the time it takes to gain access to a shared bus) and infer protected information.
Other
Other kinds of intentional but nonmalicious security flaws are possible. Functional requirements that are written without regard to security requirements can lead to such flaws; one of the flaws exploited by the "Internet worm" [9] (case U10) could be placed in this category.
Validation Error (Incomplete/Inconsistent)
Validation flaws arise when a program fails to check that the parameters supplied or returned to it conform to its assumptions about them. These assumptions may include the number of parameters provided, the type of each, the location or maximum length of a buffer, or the access permissions on a file. We lump together cases of incomplete validation (where some but not all parameters are checked) and inconsistent validation (where different interface routines to a common data structure fail to apply the same set of checks).
Domain Error (Including Object Re-use, Residuals, and Exposed Representation Errors)
Domain flaws occur when the intended boundaries between protection environments have holes. For example, a user who creates a new file and discovers that it contains information from a file deleted by a different user has discovered a domain flaw. (This kind of error is sometimes A Taxonomy of Computer Program Security Flaws 10 Landwehr, Bull, McDermott, and Choi U.S. Naval Research Laboratory NRL/FR/5540--93-9591 November 19,1993 referred to as a problem with object reuse or with residuals.) We also include in this category flaws of exposed representation [16] in which the lower-level representation of a data structure is made visible in the current domain, as in fact happened in cases B1 and D2(1). Errors caused by a failure to "implicit sharing of privileged/confidential data" will generally fall in this category.
Serialization/aliasing (Including TOCTOU Errors)
A serialization flaw permits the asynchronous behavior of different system components to be exploited to cause a security violation. These flaws can be particularly difficult to discover. A security-critical program may appear to correctly validate all of its parameters, but the flaw permits the asynchronous behavior of another program to change one of those parameters after it has been checked but before it is used. Many time-of-check-to-time-of-use (TOCTTOU) flaws will fall in this category, although some may be classed as validation errors if asynchrony is not involved. We also include in this category aliasing flaws, in which the fact that two names exist for the same object can cause its contents to change unexpectedly and, consequently, invalidate checks already applied to it.
Identification/Authentication Inadequate
An identification/authentication flaw is one that permits a protected operation to be invoked without sufficiently checking the identity and authority of the invoking agent. These flaws could perhaps be counted as validation flaws, since presumably some routine is failing to validate authorizations properly. However, a sufficiently large number of cases have occurred in which checking the identity and authority of the user initiating an operation has in fact been neglected to keep this as a separate category.
Boundary Condition Violation (Including Resource Exhaustion and Violable Constraint Errors)
Boundary condition flaws typically reflect omission of checks to assure constraints (e.g., on table size, file allocation, or other resource consumption) are not exceeded. These flaws may lead to system crashes or degraded service, or they may cause unpredictable behavior.
Other Exploitable Logic Error
Finally, we include as a catchall a category for other exploitable logic errors. Bugs that can be invoked by users to cause system crashes, but that don't involve boundary conditions, would be placed in this category, for example.
Requirements/Specification/Design
Ideally, software requirements describe what a particular program or system of programs must do. How the program or system is organized to meet those requirements (i.e., the software design) is typically recorded in a variety of documents, referred to collectively as specifications. Specifications with various scopes and levels of detail may be written for a software system or its components, and they may be called interface specifications, module specifications, functional specifications, detailed specifications, and so on. Typically, the specifications define the functions of the software, the environment in which it operates, the inputs it is to accept, the outputs in which the source code is built. The specifier is often responsible for implementing the specification as well. Written according to good engineering practice, the requirement and specification documents should make the software design clear to the security analyst. At a minimum, the specification should completely document the interfaces of all modules. This information should be detailed enough that maintenance programmers can determine whether and how a modification of one module will affect others. Specifications that do not meet this criterion are more likely to contain flaws than those that do. Although specifications are intended to be complete, they do not must assure that the security requirements themselves are complete, that they mesh with the system's functions, and that the specifications are consistent with the requirements. Errors are more likely to occur if the functional requirements and security requirements have been developed and documented independently than if they have been coordinated. Requirements and specifications are relatively unlikely to contain maliciously introduced flaws. They are normally reviewed extensively, so a specification for a trapdoor or a Trojan horse would have to be well-disguised to avoid detection. More likely are flaws that arise because of competition between security requirements and other functional requirements. For example, security concerns might dictate that a program should access only the resources that it is to use in accordance with a certain policy. If system operation is perceived to be too great, there will be pressure to provide mechanisms in the specification to permit on-site reprogramming. Such mechanisms can provide built-in security loopholes. Also possible are inadvertent flaws that arise because of missing requirements or undetected conflicts among requirements.
Source Code
The source code implements the design of the software system given by the specifications. Most flaws in source code, whether inadvertent or intentional, can be detected through a careful examination of it. The classes of inadvertent flaws described previously apply to source code. For large programs, the interfaces among modules may be complex and poorly understood due to inadequately defined module or process interfaces. Programmers attempting to build a system to inadequate specifications are likely to misunderstand the parameters to be passed across an interface, the requirements for synchronizing concurrent processes, or the proper formats for data input or output. These misunderstandings manifest themselves as source code flaws. Many such flaws in a system may indicate poor system documentation and may require system documents to be rewritten. Intentional but nonmalicious flaws can be introduced in source code for several reasons. A programmer may introduce mechanisms that are not included in the specification but that are intended to help in debugging and testing the normal operation of the code. However, the test code may inadvertently compromise the security of the system in which it is installed. In its operational system, it provides a security flaw. One of the attacks used by the Internet Worm exploited just such a mechanism; this mechanism permitted remote execution of an operating system command without requiring user authentication (case U10). Programmers may also decide to provide undocumented facilities that simplify maintenance but provide security loopholes. The infamous "patch area" that facilitates reprogramming outside the scope of the configuration management system would fall in this category. Technically sophisticated malicious flaws can be introduced at the source code level. A programmer, whether an authorized member of a development team or an intruder, working at the source code level can invoke specific operations that will compromise security later when the program containing the flaw is compiled and put into use. In a well-managed software, much software is developed without any such review; source code is frequently not provided to purchasers of software packages (even if it is supplied, the purchaser is unlikely to have the resources necessary to review it for malicious code). If the programmer is aware of the review process, he may well be able to disguise the flaws he introduces. A malicious source code flaw may be introduced directly by any individual with write access to source code files, but such source code flaws can also be introduced indirectly. For example, if a programmer authorized to write source code for a compiler inserts a Trojan horse or other malicious code into the compiler, then the Trojan horse could also be inserted into a new compiler with no such code in its source. A subtle indirect tampering with source code are difficult to document, but Trojan horse programs that grossly modify all a user's files are hardly in keeping with the spirit of the Trojan horse concept.
Object Code
Object code programs are generated by compilers, assemblers, and linkers. Although most of the machine-readable form of the source code. Because most compilers and assemblers are subjected to extensive testing and formal validation procedures before release, inadvertent flaws in object programs that are not simply a translation of source code flaws are rare, particularly if the compiler or assembler is well-known and in widespread use. In fact, the more widely used and well-tested a compiler or assembler, they typically show themselves through incorrect behavior of programs in unusual cases, so they can be quite difficult to track down and remove. Because this kind of flaw is rare, the primary security concern at the object code level is with malicious flaws. Because object code is difficult for a human to make sense of (if it were not, software companies would not have different policies for distributing source code and object code for their products), it is a particularly tempting place for malicious security flaws (again, see Thompson [35]). Lacking system and source code documentation, an intruder will have a hard time patching source code to introduce a security flaw without simultaneously altering the visible behavior of the program. The insertion of a malicious object code module or modification of an existing object code module to introduce a Trojan horse or other malicious Trojan horse is a more common threat. Writers of self-replicating Trojan horses (viruses) [21] have typically taken this approach: a bogus object module is prepared and inserted in an initial target system. When it is invoked, perhaps during system boot or running as a substitute version of an existing utility, it can search the disks mounted on the system for a copy of itself and, if it finds none, insert one. If it finds a related, uninfected version of a program, it can infect it with an infected copy. When a user unwittingly moves an infected program to a different system and executes it, the virus gets another chance to propagate itself. Instead of replacing an entire program, a virus may append itself to an existing object program, perhaps as a segment to be executed first. Because of the widespread use of standard utility programs by most system programmers, viruses, especially those introduced as object code, typically cannot propagate to different host hardware or operating systems. A direct penetration at the object code level occurs when a user or intruder maliciously alters object code or introduces bogus object code. Unwitting propagation of a virus by a user is a form of indirect penetration.
Operating System
Operating system functions normally include memory and processor allocation, process management, device handling, file management, and accounting, although there is no standard definition. The operating system determines how the underlying hardware is used to define and separate protection domains and to control access by processes to these domains and to other system resources. In addition to functions that may be invoked by user calls, traps, or interrupts, operating systems often include programs and processes that operate on behalf of all users. These programs provide network access and mail service, schedule invocation of user tasks, and perform other miscellaneous services. Systems often must grant privileges to these utilities that they deny to individual users. Finally, the operating system has a large role to play in system initialization. Although in a strict sense initialization may involve programs and processes outside the operating system boundary, this software is usually intended to be run only under highly controlled circumstances and may have many special privileges, so it seems appropriate to include it in this category.
Support
Support software comprises compilers, editors, debuggers, subroutine or macro libraries, database management systems, and any other programs not properly within the operating system boundary that many users share. The operating system may grant special privileges to some such programs; these we call privileged utilities. Flaws in privileged utilities can be exploited to violate system security. Flaws in utilities that do not require access-checking controls disabled. This means that any such program will need to be scrutinized for security flaws, since during its execution one of the fundamental security-checking mechanisms is disabled.
System Initialization
System initialization, although it may be handled routinely, is often complex. Flaws in this area may cause the operating system to be initialized improperly, with some of its access protections as specified (for example, it may set up ownership or access control information improperly) or with some of its critical tables improperly set up or with some critical system utilities not in place. In case U5, improperly set permissions on the mail directory led to a security breach. Viruses commonly try to attach themselves to system initialization code, since this provides the earliest and most predictable opportunity to gain control of the system (see cases PC1-4, for example).
Memory Management
Memory management and process management are functions the operating system provides to control storage space and CPU time. Errors in these functions may permit one process to gain access to another improperly, as in case B5, or to deny service to others, as in case M2S.
Process Management/Scheduling
Memory management and process management are functions the operating system provides to control storage space and CPU time. Errors in these functions may permit one process to gain access to another improperly, as in case B5, or to deny service to others, as in case M2S.
Device Management (including I/O, networking)
Device management often includes complex programs that operate in parallel with the CPU. These factors make the writing of device handling programs both challenging and prone to subtle errors that can lead to security flaws (see case D1). Often, these errors occur when the I/O routines fail to respect parameters provided them (case U12) or they validate parameters provided in storage locations that can be altered, directly or indirectly, by user programs after checks are made (case B5).
File Management
File systems typically use the process, memory, and device management functions to create long-term storage structures. With few exceptions, the operating system boundary includes the file system, which often implements access control permit users to share and protect their files. Errors in these controls, or in the management of the underlying files, can easily result in security flaws (see cases I1, M2B, and U2).
Identification/Authentication
The identification and authentication functions of the operating system usually maintain special files for user IDs and passwords and provide functions to check and update those files as appropriate. It is important to scrutinize not only these functions, but also all of the possible ports of entry into a system to ensure that these functions are invoked before a user is permitted to consume or control other system resources.
Other/Unknown
Privileged Utilities
Privileged utilities are often complex and sometimes provide functions that were not anticipated when the operating system was built. These characteristics make them difficult to develop and likely to have flaws that, because they are also granted privileges, can compromise security. For example, daemons, which may act on behalf of a sequence of users and on behalf of the system as well, may have privileges for reading and writing special system files or devices (e.g., communication ports, printer queues, special files containing mail or network packets). Unprivileged users (e.g., mailers) may frequently make heavy use of operating system facilities, and their privileges may turn a simple programming error into a penetration path. Flaws in daemons providing remote access to restricted system capabilities have been exploited to permit unauthenticated users to execute arbitrary system commands (case U12) and to gain system privileges by writing the system authorization file (case U10).
Unprivileged Utilities
Even unprivileged software can represent a significant security capability because these programs are widely shared, and users tend to rely on them implicitly. The damage inflicted by flawed, unprivileged software can be, in the worst case, proportional to the privileges granted the user who invokes that software. In some cases, however, since it may be used to compile a new release of a system, support software can even sabotage operating system integrity (case U1). Inadvertent flaws in support software can also cause security flaws (case I1); intentional but nonmalicious flaws in support software have also been recorded (case B1).