27/
8-8-77
250 NTIS

MASTER

UCID-17422

Lawrence Livermore Laboratory

A TAXONOMY OF OPERATING-SYSTEM SECURITY FLAWS

William L. Konigsford

November 1, 1976

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the laboratory.

Prepared for U.S. Energy Research & Development Administration under contract No. W-7405-Eng-48.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

A TAXONOMY OF OPERATING-SYSTEM SECURITY FLAWS

ABSTRACT

Concern over the privacy and security of computerized data has caused government and the private sector to investigate means of insuring the protection of such data from unauthorized disclosure. Experience with the past breakdown of existing protection mechanisms shows that the computer operating system is a primary cause of such disclosures.

RISOS Project members at Lawrence Livermore Laboratory are engaged in this type of research, and they have developed a systematic approach to the problem of flaw detection. The three main components of this approach use test teams, computerized test tools, and past experience in developing generic classes of security flaws.

This paper addresses the categorization of flaws into generic classes so that the nature of such flaw groups can be understood. This understanding can then be utilized in detecting flaws in existing systems and applied in future system development.

INTRODUCTION

RISOS Project members (Research in Secured Operating Systems) at LLL are engaged in research based on the testing and evaluation of present operating systems. This work is being done under contract to the Advanced Research Projects Agency (ARPA) of the Department of Defense. The RISOS team has been charged with the test and evaluation of a number of large operating systems for the purpose of flaw detection and collaborative correction. It is only one of a number of organizations engaged in operating-system research for the federal government.

In the course of this work, RISOS members have developed a methodology of investigation based on the "test-team" concept, the use of computerized test tools, and the systematic categorizing of all known flaws into generic classes.

MASTER

The "test-team" concept has been highly useful in this type of research. Because an operating system is such a complex entity, no individual can hope to understand its workings and interfaces at a relatively detailed level when working under the constraints of a tight time schedule. The individual members of a test team are assigned broad areas to investigate; these areas generally correspond to functional divisions of the operating system itself. Periodic meetings are held to discuss each member's investigations and to assess team strategies. Goals and techniques are often shifted in the light of interim findings. Team members who become expert in certain aspects of operating-system structure can transfer this knowledge to other operating systems. This type of research is thus both an interactive and a heuristic process.

A number of "analytic tools," i.e., programs to aid the investigator in his flaw-detection analysis of a system, have been developed.¹ These generalized programs are structured to work on all operating systems under investigation. In order to utilize these generalized tools, the assembled listings of an operating system are first reduced to a data base that has a common structure recognizable to the analytic tools. The data reduction is done by parsing the assembly-listing files and restructuring the parts into a fixed and standardized format. The tools then act on these data bases, aiding the investigator in his searching operations and freeing him to devote more time to flaw analysis.

The RISOS team is also developing tools of a more specialized nature to aid the investigator in detecting more sophisticated flaws, such as timing or synchronization problems and flow-of-control errors. These tools involve mathematical and graphical modeling of operating systems so that potential problem areas can be defined and flagged automatically. The human investigator can then perform a thorough analysis of the specific areas identified.²

-
1. D. A. Webb, "Analytical Tools for the Examination of Security Aspects of Operating Systems," in *Proc. IBM Data Sec. Forum, Denver 1974* (IBM, Denver, 1974).
  2. R. W. Conn and R. H. Yamamoto, "A Model Highlighting the Security of Operating Systems," in *Proc. 29th Nat. Conf. ACM, San Diego, 1974* (ACM, San Diego, 1974).

RISOS and a number of other organizations within the operating-systems security community have categorized known security flaws into generic classes and this categorization provides valuable insight into the nature of flaw manifestation.³ Whenever a flaw is confirmed in an operating system, it is documented and compared with known flaws for the purpose of categorization, thus enabling one to better understand the nature of such flaws, i.e., where and how often they would tend to occur and in what form they would be likely to appear. This categorization is also helpful in assessing the relative power of alternative techniques of automating or semi-automating security-flaw detection. A systematic classification of flaws also facilitates communication of security analysis techniques and results.

TAXONOMY OF INTEGRITY FLAWS

In this paper, we present a system for arranging integrity flaws into related groups, and examine one class of integrity flaw in detail — the operating-system security flaw.

Table 1, divided into two segments and one example, presents a taxonomy (i.e., a system of arrangement) of integrity flaws. The syntax segment is a generalized description of those conditions required to exploit a flaw in security. This generalized description clarifies the fact that the mere existence of a flaw renders an installation vulnerable and it is analogous to the engineering concept of "unavailable" potential energy. When an individual (or group) becomes aware of a flaw, an active potential to violate installation integrity is achieved — analogous to "available" potential energy. This potential is transformed into kinetic energy with adequate motivation, skill, resource, and opportunity, and an installation's integrity is penetrated. This penetration of integrity provides the individual with potential access to one or more classes of resources — items of value to an installation or its users. If the individual chooses, this access may be exploited to produce a loss for the installation (such as a loss of information, service, or equipment) and/or a gain for the individual.

Each classification depicted in the syntax can be divided into subclassifications and each subclassification can be further divided into

---

3. D. K. Brandstad, "Privacy and Protection in Operating Systems," *Op. Sys. Review* 1 9-17 (1973).

Table 1. Taxonomy of integrity flaws.

Syntax

A *class of user* acquires the potential to compromise the integrity of an installation via a *class of integrity flaw* that, when used, will result in unauthorized access to a *class of resource*, which the user exploits through *category of method* to *category of exploitation*.

Syntax elements

<i>Class of user</i>	<i>Class of integrity flaw</i>	<i>Class of resource</i>
Applications	Physical protection	Information
Service	Personnel	Service
Intruder	Procedural	Equipment
	Hardware	
	Applications software	
	Operating system	
<i>Category of method</i>		<i>Category of exploitation</i>
Interception Scavenging Preemption Possession		Denial of possession/use
		Steal equipment
		Destroy equipment
		Degrade service
		Interrupt service
		Destroy data
		Denial of exclusive possession/use
		Read/transcribe data
		Steal service
		Modification
		Alter data
		Alter equipment

Example

- An "applications" user acquires the potential to compromise the integrity of an installation via an "operating-system" integrity flaw that, when used, will result in unauthorized access to an "information" resource, which the user exploits through "scavenging" to "read/transcribe data."

subclassifications and so on — in descending order from most inclusive to most specific. The second segment of Table 1 depicts the first levels of classification for each syntax element. We will briefly discuss each classification; however, because this document is principally concerned with operating-system security flaws, only that class of flaw will be fully expanded.

Class of User

A user may have various capabilities at various times, and similar users may be granted differing sets of capabilities. However, it is useful to classify users in terms of broad sets of capabilities.

Applications User

Applications users are those who have not been specifically granted special capabilities beyond permission to use the system. They are subdivided into consumers and producers. *Consumers* are the authorized recipients of information products from a computer-based application. *Producers* are the analysts and applications programmers who design and implement the specific applications that produce information products for consumers. (Producers may or may not be part of the consumer's organization. Producers need access to the computer system to develop products; their programs require access to data in the system.)

Service Users

Service users are subdivided into systems and administrative servicers. *Systems servicers* are members of a computer-servicing staff that includes the operators, systems programmers, and maintenance engineers responsible for the maintenance and availability of computer-system resources. Because systems servicers have physical access to the computer, to the operating-system code, or to the data-storage volumes, they have the capability to access any information in or on a system. For example, any operator can replace the installation's protected operating system with a nonprotected one or he may use computer console switches to alter the main storage contents. The hardware vendor's maintenance engineer, in another example, is equipped with a set of diagnostic aids that can be utilized as integrity-penetration tools.

Administrative servicers are members of the systems staff who do not have physical access to the computer room or operating system, but who do have special software privileges that permit access to privileged hardware instructions and special operating-system services, or permit special operations on data. Such users frequently have the capability to access any information in a system.

Intruder

An *intruder* is an unauthorized user — an outsider. This term applies to individuals or organizations who have no authorized access to a computer installation or its products and who have a possibly malicious interest in obtaining unauthorized access.

Class of Integrity Flaw

Physical protection flaws include telecommunications interception, mixed-security-level access to terminals, unauthorized access to a computer room, and exposure to natural disasters. Flaws involving *personnel security* include acts such as sabotage, collusion, and user error. *Procedural* flaws are, of course, installation-dependent. Such flaws may involve tricking (or "spoofing") a system operator into making unauthorized data available to a user, inadequate tape-labeling procedures at an installation, and "Trojan Horse" subversion of an operating system. Trojan Horse refers to the covert implanting of computer instructions in a trusted (system) program so that the trusted program executes its intended functions correctly, but with illegitimate side effects. Policies and procedures (such as default values or access policies), explicitly or implicitly embodied in operating-system software, may also be a source of security flaws.

Examples of *hardware integrity* flaws include a user's terminal disconnect signal that is not passed on to the operating system's software or a CPU design that allows all users access to an instruction such as "disk diagnose," which should have restricted access. The sophistication level of hardware varies and, in a system where the operating system polls terminals, failure to recognize a disconnect could be an operating-system flaw. *Applications software* flaws include inadequate user-user isolation, insufficient control over access to data, and exploitable errors in program logic. Almost all applications-software flaws have direct analogies with operating-system flaws. We discuss *operating-system* flaws in detail later in this paper. It is worth noting here, however, that operating-system flaws can sometimes be "fixed" by means of hardware or procedural changes. Conversely, hardware, procedural, and other flaws can sometimes be "fixed" via changes in operating-system software.

Class of Resource

The resources of value to an installation or its users are information, service, and equipment. *Information* includes all system files (programs, data, and file directories) and all user files. *Service* represents the unimpaired operation of the installation. Service resources include all the capabilities of the operating system. If an applications user obtains access to the hardware wait/idle mode, monitor/master mode, or unauthorized disk-storage space, then a valuable resource has been compromised. *Equipment* resources include all installation equipment relevant to the unimpaired operation of its computers.

Category of Method

Interception is the interruption of communication or connection. A user program masquerading as the system could intercept an unwary user's sign-on password. *Scavenging* is the searching for something of value from discarded information or supplies. If reading of (scratch) tapes is not prevented, a user could search through the data left by a previous user in an attempt to find some valuable information. *Preemption* involves taking something to the exclusion of others, such as a user preempting CPU cycles. *Possession* is taking control of property, such as stealing a magnetic tape containing valuable information.

Category of Exploitation

The categories of exploitation in Table 1 are self-explanatory and are listed here only for ease of referral and completeness.

- Denial of possession/use
  - Steal equipment
  - Destroy equipment
  - Degrade service
  - Interrupt service
  - Destroy data
- Denial of exclusive possession/use
  - Read/transcribe data
  - Steal service
- Modification
  - Alter data
  - Alter equipment

Detailed Description of Operating-System Security Flaws

Operating-system integrity is concerned with the assurance that the operating system works as it was intended to work. An operating-system integrity flaw is any condition that would permit a user (or the user's programs) to cause the operating system to cease reliable and secure operation. *Integrity* is thus concerned with *reliability* (fraud and error) problems and with *security* (resource and privacy-protection) problems.

In this section, we discuss the seven major categories of operating-system security flaws listed in Table 2 and give examples of each. Tables associated with the general text description for each operating-system security flaw show each flaw further divided into subcategories along with a brief, descriptive example for each subcategory. Not all subcategories and examples are discussed in the text. A more formal description of each category and subcategory of flaw is presented in Appendix A.

Incomplete Parameter Validation

Whenever a process (or program) with one set of privileges requests service from a second process with another set of privileges, the preservation of system integrity requires that the request be thoroughly validated. The boundary of greatest relevance to system integrity for most operating systems, is that boundary between a control program (with complete hardware and software capabilities) and the user programs (with a limited subset of capabilities). This separation is usually enabled by hardware facilities (such as control/monitor state and storage protection) but is enforced through software.

User programs generally invoke control-program services in a manner similar to subroutine calls, using many parameters. Only the control program

Table 2. Operating system security flaws.

-
- Incomplete parameter validation
  - Inconsistent parameter validation
  - Implicit sharing of privileged/confidential data
  - Asynchronous validation/inadequate serialization
  - Inadequate identification/authorization/authentication
  - Violable prohibition/limit
  - Exploitable logic error
-

is capable of performing the requested services. This separation or isolation between user programs and the control program was created to prevent any user from compromising the functioning of the control program that is performing services for all users (e.g., I/O operations, program initiation, date and time, etc.). If the checking mechanism for each requested parameter is not rigorous or complete, it is possible to "fool" the control program into executing the request in a manner detrimental to secure operations. To be validated rigorously, parameters must be checked for permissible:

- Presence or absence
- Data types and formats
- Number and order
- Value ranges
- Access rights to associated storage locations
- Consistency among parameters (e.g., overlapping storage locations).

Three dangerous results can occur if a user succeeds in getting the control program to accept as a parameter an address outside the memory space allocated to that user:

- The control program may obtain unauthorized data for that user.
- A set of conditions can be generated to cause a system crash.
- Control may be returned in control/monitor state to the user.

A penetration attempt that illustrates the return of control in control/monitor state to a user program is described below and in Figs. 1 and 2.

- An instruction that, when executed, will transfer control to a predetermined point in the user's program is loaded into a register.
- A system call is then made, causing the registers to be saved by the control program in a register save area, located within the storage owned by the operating system (Fig. 1).
- When control is returned to the user, another system call is made. Although it may be "illogical" to issue this particular system call at this point, we have chosen it because one of its parameters is a pointer (address) that indicates a location within the control program. This address will be used in transferring control to the appropriate control-program service routine. Naturally, a person wishing to penetrate operating-system security would choose to supply as an address that location in the register save area where a transfer back to the user's program had been planted by the previous system call (Fig. 2).

Fig. 1. Layout of memory after first system call.

(A) Points to register save area instead of a control-program service routine.

Fig. 2. Layout of memory when preparing to issue second system call.

- All parameters are checked and approved. The instruction planted in the register save area is executed during the second system call and control will be returned to the user in control/monitor state, giving him control of the system.

Table 3 further describes the categories of incomplete parameter validation.

Inconsistent Parameter Validation

Whenever there are multiple definitions of the same construct within an operating system, it is possible that inconsistencies among these definitions will create a security flaw.

This design error goes beyond the incomplete parameter-validation error. A situation may exist where each of several control-program routines checks completely for conditions it considers valid; however, the multiple sets of validity criteria (i.e., conventions) are not completely consistent. An example of this category of flaw follows.

An operating system maintains directories (e.g., catalogs) of the data files used by the system and its users. The contents of these directories are often accessed by as many as a half dozen interface programs. The operating system may be implemented in such a way that each interface program makes assumptions as to what constitutes a valid condition in the file system.

Table 3. Incomplete parameter validation — categories and examples.

System routine does not adequately validate parameter attributes.

- The control program verifies an initial I/O transfer. However, it does not verify that the initial I/O transfer will not cause illegal modification to subsequent I/O transfers.
- A system routine validates that the lower bound of a user's buffer lies within storage to which the user is authorized access, but neglects to verify that the upper bound also lies within authorized storage.

System routine does not properly reiterate parameter validation.

- In a chained list of I/O commands, only the first I/O command is verified or all but the last I/O command are verified.

System routine validates a parameter under some conditions, but not under all conditions of invocation.

- A "confused-deputy" control-program service routine adequately verifies parameters when directly invoked by a user, but not when a user's parameters are indirectly passed to the first service routine by a second service routine.
-

Consider something as basic as the characters in the parameters representing the name(s) of users to be given permission to access a file. The routine that creates a master-file-index entry may accept a character (such as embedded blank) as valid in a specific permission name; all the other interface programs that modify/delete master-file-index entries assume blanks will never be valid and thus do not accept them. Under such conditions, specific file permissions could be created (e.g., shared access to a file) that could not thereafter be deleted under normal operating conditions. Table 4 summarizes inconsistent parameter validation.

Implicit Sharing of Privileged/Confidential Data

To ensure integrity, an operating system must be able to isolate each user from all other users and from the control program. This isolation involves both control flow and information. Whenever information isolation is incomplete, the system may allow information of greater privilege to become accessible to a lesser privileged user or may allow one user to access another user's information against that user's wishes.

The control/program portion of many operating systems shares memory space with particular user programs, either as work space or as a convenient place to put information associated with particular user programs. This policy is deliberately designed to facilitate charging individual users directly for the resources that they use. If the user requires file operations or other kinds of system resources, the system will maintain the information and the work space for his requirement in an area uniquely chargeable to that user. Because the workspace is shared, but in a mode not normally available to the user, operating system implementors are often careless about how they leave the workspace after a user's request has been processed.

Table 4. Inconsistent parameter validation — categories and examples.

Two or more system routines perform adequate parameter verification for their purpose, but the multiple sets of validity criteria are mutually inconsistent.

- The routine that creates a master-file-index entry permits embedded blanks; all the other routines that modify/delete master-file-index entries treat an embedded blank as an error. Thus, once granted, a user may be unable to revoke shared access to a file.
-

The control program may use such a workspace to read in the master index of user files along with their associated passwords as part of a search for data requested by a given user. This function is necessary in order for the system to determine that the request is properly formed and authorized. If the control program finds the request improper, it returns control to the user program originating the request and indicates the nature of the error but — in this example — does nothing about the information remaining in the shared workspace. Consequently, the user can now access that workspace and obtain from it identifiers and authenticators (passwords) for other users that he can then use to masquerade as another user to the system (Fig. 3). Even if the system erases the information before returning control to the user's program as shown under the "asynchronous validation" flaw, the user may be able to obtain the information through some form of concurrent processing, such as an independent I/O operation that reads from the workspace in question. There are other variations of this flaw. Work files and workspace are sometimes not erased when a user releases them and another user can scavenge this "unerased blackboard" when the uncleared file space or buffer space is next assigned.

The full implications of information made available to a user are often not realized by the system's designers. Control programs frequently acknowledge the disposition of user service requests by setting a return-code/status-flag. Various return conditions — "illegal parameter," "segment error," "password OK," etc. — and other forms of interprocess communication (e.g, SEND/RECEIVE acknowledgement) may connote intelligence that enables a user to breach security. The system will implicitly share information if it permits a user to perform a large number of system sign-on (LOGIN) attempts without disconnecting him. This negative feedback information can be valuable to a potential system penetrator. Table 5 summarizes the categories of implied sharing.

Asynchronous Validation/Inadequate Serialization

System integrity requires preservation of the integrity of information passed between cooperating processes or control-program instruction sequences. If serialization is not enforced during the timing window between the storage of a data value and its reference (or between two sequential references), then the consistency of such a data value may be destroyed by an asynchronous process.

Fig. 3. Layout of memory before and after issuing requests to read master-file index.

Table 5. Implicit sharing of privileged/confidential data — categories and examples.

Explicit transfer of information

- While servicing a user request, the control program uses a user-accessible buffer to scan master-file-index entries. The user asynchronously reads the buffer and obtains another user's file-index password during this activity.
- The control program does not erase blocks of storage or temporary file spaces when they are reassigned to another user ("unerased blackboard").
- A user's password is still legible through the overstrike characters on his terminal printout, or a user's password is listed on his batch output if his job command is flushed due to incorrect syntax.

Implicit transfer of information.

- A user decomposes a password piecewise by locating it on a virtual memory page boundary and noting page faults or by precisely timing variations in the execution time required by a password-checking routine.
-

Control information is especially susceptible to modification whenever it is located in storage accessible to a subordinate process. This is sometimes called the "time-of-check-to-time-of-use" problem. As previously described, an operating system may frequently share memory space with user programs. This space may not only be used for passive storage of information, but may also contain system or user parameters that represent data upon which future actions will be based. A potential security flaw is created whenever there is a "timing window" between the time the control program verifies a parameter and the time it retrieves the parameter from shared storage. This is because contemporary operating systems allow a user to execute two or more activities (processes) concurrently which share that user's memory allocation. For example, a user may initiate an I/O operation and then continue executing his program while the I/O operation is being completed.

A timesharing user may achieve such concurrency by temporarily suspending one operation by pressing the "attention" or negative-acknowledgment (NAK) key on his terminal, performing a second operation, and then returning control to the first operation for completion. Some systems permit "multitasking," where two or more programs, sharing a single user's assigned memory (address space), execute concurrently — perhaps each being simultaneously executed by separate CPU's of a multiprocessing computer system.

The following steps describe an example of the asynchronous-validation flaw, depicted in Fig. 4.

- In time frame 1, a user issues an I/O request to the control program. The control program validates all the user's I/O parameters (including the address pointer to a valid buffer within the memory legitimately assigned to the user), enqueues the I/O request (that must wait until the appropriate device is no longer busy), and then returns control to the user.
- In time frame 2, the user replaces the valid address pointer to his buffer with an address that points to a location within the control program.
- When the I/O is performing in time frame 3, the data requested by the user is read into (or out of) the control program instead of his valid buffer. Instructions within the control program can thus be overlaid with instructions supplied by the user, or privileged control-program information can be read and written to the user's file.

In some systems, the control program may use an overflow register-save area (located in user-accessible storage) whenever the control program's primary save area is filled. This saved information generally contains program status and control information and can give rise to another variation of the asynchronous-validation flaw if a user is able to modify this control information. An example of such a penetration attempt follows:

- A user constructs an I/O record that simply contains an address pointing to a desired location in one of his programs.
- Multiple copies of this record are output as a file.
- The user sets up an I/O process to read the file into the overflow register-save area.
- The user then issues a system service request that causes the control program to make a number of nested intramonitor calls, thus overflowing its primary save area. (The repeated issuing of certain service requests may also accomplish this aim.)
- The registers saved by the control program in the overflow area will be overlaid by input records that contain the address pointing to the user's code. (Some timing adjustments may be required for the user to accomplish this.)

Fig. 4. An example of an asynchronous validation flaw.

- When the control program eventually restores registers and status from the overflow area, it will transfer control to the user's program in monitor/control state — thus giving the user full control over the operating system.

An operating system may store information over a period of time in shared auxiliary storage as well as in main memory. For instance, an operating system may have a checkpoint/restart provision to record the state of a running program at convenient restart points as "checkpoint" dumps. These checkpoint dumps frequently contain both user data and control information specifying the control status to be assigned if the program is restarted; they are recorded in a file that the user specifies to the system and are accessible by that user for manipulation. Through such manipulation, the user could cause his program to be restarted with modified state information giving his program greater privileges than that originally specified. This can result in the user gaining control/monitor state privileges. Table 6 further describes the categories of asynchronous-validation and serialization flaws.

Inadequate Identification/Authorization/Authentication

Identification, authorization, and authentication are the essential components of the concept of controlled access. Authorization — the controlled granting of access rights — is ultimately based upon authenticated, unique identification of individuals and resources. An operating system is essentially a resource manager. Thus, an operating system is subject to integrity problems whenever it does not require authorization for an individual or process to access any data or to use any resource that should not be available to all users, or it does not uniquely identify the resources it is dealing with.

A bypass flaw is created whenever a system permits a user possessing one set of privileges/capabilities to legitimately bypass its controlled access security mechanisms and to perform an action only permitted to users with differing privileges/capabilities or whenever it permits all users to perform an action that should be restricted only to users of greater privilege. For example, some systems have extensive authorization checking associated with most, but not all, of the file-access methods and they do not restrict use of those access methods that do not perform authorization checking. Any user who obtains documentation for these latter access methods ("unrequired capabilities") has simply to use them to access any file in the system.

Table 6. Asynchronous validation/inadequate serialization — categories and examples.

Asynchronous modification of user (inferior process) storage.

- A user performs asynchronous I/O into his parameter list to illegally modify a previously validated system call.
- A user performs I/O into a checkpoint/restart file so that his process is given additional, but unauthorized, privileges when restarted.
- Within the calling sequence to invoke control-program services, a user substitutes an indirect addressing instruction, that increments its target address after each execution, for the simple indirect instruction expected by the control program. The control program does not adequately validate the calling sequence and consequently validates one set of parameters on the first indirect access and then utilizes unvalidated parameters on its second indirect access.

Inadequate serialization/control of protected storage.

- A user issues a system call that (in part) sets values in an I/O control table and then returns control to the user. The user then issues a second, different system call that also (in part) stores values in the I/O control table — thus overlaying a portion of the previously set values in such a way as to gain unauthorized I/O privileges for the I/O performed in conjunction with the first system call.
  - A user induces a system routine to overlay its own parameter/storage area. The user supplies an address where code is to be stored by the system routine upon return of control to the user. This user-supplied address overlays the initial word of a buffer where the system routine has stored a return-jump instruction.
-

An inadequate identification/isolation flaw can be created whenever one system routine relies upon mechanisms (implemented elsewhere in the system) to ensure the isolation of system resources and, hence, the adequacy of their identification. This may be a bad policy if the mechanisms are not adequate.

For example, a program must be identified both by program name and by the name of the library from which it was loaded. Otherwise, it is very easy for a user to preload a counterfeit program whose name is the same as some control program routine (that must be dynamically loaded when required) and to have this counterfeit routine used by the control program in place of the authentic routine.

To accomplish this, the user generates an activity that will result in a control program request for this routine. The loader will see that the named

(counterfeit) routine is already loaded (which is legitimate) and will set up the control program to use the counterfeit program.

As another example, the user-ID or password-checking mechanism may be circumvented if it does not effectively limit the number of times a user can attempt to log into the system or if it does not limit the elapsed time permitted for completing a log-in. It may be possible, under such circumstances, for a user to utilize another computer to exhaustively enumerate all password bit combinations and thus break password security. (Categories and additional examples of the inadequate identification, authorization, and authentication flaw are summarized in Table 7.)

Violable Prohibition/Limit

An operating system is described both by its embodiment in computer instructions and by its external documentation. Whenever these two descriptions

Table 7. Inadequate identification/authorization/authentication - categories and examples.

Inadequate resource identification/isolation.

- A user program with the same name as a system program, preloaded by a user, is then accepted and used by the system.
- A system routine assumes the validity of a system-control table whenever the control table is located in system storage to which users do not have direct-write access. In fact, it is possible for a user to create a counterfeit control table and to have it copied into system storage by certain control-program service routines, such as the storage-deallocation routine.

Bypass of controlled-access security.

- A user legally bypasses the file-initialization (open) routine and its security mechanisms by utilizing a basic file-access method.
  - A user obtains system privileges by taking a legal exit from abnormal job-termination (i.e., abort) processing.
  - A user obtains system privileges by discovering and using a "trap door" exit to the system (i.e., unadvertised capability) meant for system-maintenance programmer use.
  - An operating system that does not prevent a user from simulating its log-out and log-in functions permits an unattended (hardwired) terminal to simulate a logged-out terminal and to obtain another user's password during a simulated log-in process.
  - A user is able to obtain unauthorized privileges from the system operator, because the operating system does not prevent a user from simulating control-program messages to the system operator.
-

differ, an integrity flaw may exist. A security flaw is created whenever a documented operating-system limit or procedural prohibition is not enforced.

Those who implement an operating system may not treat the situation in which the upper limit in size of tables or buffers is reached or when queue space becomes saturated. Documentation may specify precisely these upper limits and prohibit exceeding the limits, but should a user deliberately or accidentally cause an overflow or overload, then various results may occur -- a system crash may result, system operation may be degraded, sensitive data can be lost, and in some instances, such data could be compromised. In one such example, a user may create an "illogical request" consisting of a set of I/O instructions that create an uninterruptable execution loop in an I/O channel. Table 8 gives additional examples of this flaw.

Exploitable Logic Error

Any major operating system has, at any point in time, some "bugs" or logic errors. Many of these errors depend upon statistically improbable timing situations and are not under the control of any individual user. Some logic errors can, however, be intentionally exploited by a user to compromise the integrity of a system.

One example involves incorrect error handling. The system may, for instance, perform an illegal action before signaling an error condition. For example, a user may request that a series of modifications be made to the

Table 8. Violable prohibition/limit -- categories and examples.

Violable system limit.

- A user is supposedly constrained to operate only within an assigned partition of main storage, while, in fact, the user may access data beyond this partition.
- A user is supposedly constrained in the amount of system queue space available to his process, when, in fact, the user may create an "illogical request" consisting of an uninterruptable, endless loop on a system call that eventually uses up all the control program's queue space. This causes the system to crash.

Violable system procedural prohibitions.

- A user is able to obtain unauthorized privileges by omitting notification to the operating system of an exit from an I/O error-processing routine, although the documentation requires such notification.
-

file-directory entry of another user, to which the first user has read-only authorization. If the system performs the requested actions and then determines that the actions exceed the requesting user's authorization, the security of the system has been compromised through a logic error. In a similar example, the system may inform one user that a certain file name *does not exist* within another user's file library, without checking the first user's authorization. Such negative information can amount to positive intelligence. A compromise may also happen if a system service, such as a storage dump, is initiated concurrently with the checking of a user's authorization to request the specified service for the specified storage areas. By the time the error is detected, forbidden areas may have already been listed.

On a more subtle level, a user may discover that half-word arithmetic instructions are used to improperly process a half-word return-address parameter. If the largest possible half-word number is used as an address, an unanticipated overflow may occur. This may result in an address pointing to location 0001 in control-program memory, which may cause a system crash.

A user, by pressing the "attention" (or NAK) interrupt button on his terminal during the printing of the log-in error message, may be able to cause the system to erroneously accept a new log-in attempt without advancing the counter set to record the number of previous log-in attempts. This system error permits automation of an exhaustive enumeration of passwords, with no indication to system operators that the automation is taking place.

A last example of incorrect error handling can occur when protection mechanisms are disabled or modified as a result of a (deliberate) user error and may not be reset when the control program later returns control to the user. This can result in the user obtaining unauthorized privileges.

Table 9 lists three categories of logic-error flaws and presents some additional examples. Two additional categories of exploitable logic flaws are listed here for completeness and without examples:

- Incorrect process/function initiation or termination.
- Control-state software-error trap.

Table 9. Exploitable logic error -- categories and examples.

Incorrect error-handling sequencing.

- The operating system fails to update a count of unsuccessful log-in attempts if a user presses the interrupt key (NAK) on his terminal just after submitting a password guess.

Instruction side-effects.

- The operating system uses full-word arithmetic on a half-word return address supplied by the user. If the value supplied is -1, this causes an overflow into the index field of the word and a return to the user in control state.
- An operating system uses a particular indirect-addressing instruction in user space to access some parameters. The user substitutes a similar indirect instruction that increments an index register after each execution, thus creating a flaw.

Incorrect resource allocation/deallocation.

- The same tape or core block is assigned to two users at the same time.
-

APPENDIX A: TAXONOMY OF OPERATING-SYSTEM INTEGRITY FLAWS

This appendix contains my attempt to describe each element of the taxonomy in terms of an abstract, generalized pattern. (The opposite method of generalizing a pattern is "instantiating" it by listing examples.)

Each generalized flaw pattern consists of a set of statements. Each statement represents a necessary (but insufficient) condition for the creation of that flaw class. Together, the set of statements represent the necessary and sufficient conditions for the creation of a particular class of flaw. Within these statements, classes of similar objects are represented symbolically. That is, the names of objects are replaced by parameters.

I have chosen to define the terms used in this appendix and then to delineate two levels of flaws. It is my hope that this will represent a first step in describing software security flaws in a system-independent fashion.

Definitions Used in This Appendix

<i>Parameter</i>	- a data value explicitly contained in or pointed to by the calling or invocation sequence when one routine requests service from another.
<i>Storage</i>	- any media capable of retaining information. Categories of storage are: executable storage (address space), machine-readable storage, input or output media.
<i>Superior Process</i>	- any process that possesses at least one capability to a greater degree than another (inferior) process.

The First Flaw Level

Incomplete parameter validation

- A superior process does not adequately validate parameter attributes.
- A superior process does not properly reiterate parameter validation.
- A superior process validates a parameter under some but not all conditions of invocation.

Notes on Generalized Patterns — Incomplete Parameter Validation — To exhaustively analyze a system for such a flaw, one must:

- Identify all operating system routines with superior capabilities.
- Determine all methods and instances by which user routines may invoke routines of greater capabilities. (There may be nonsystematic invocation methods.)
- Determine all parameters associated with each instance of invocation. (The implemented code and the documentation may not be consistent.)
- Determine that there are values "λ" such that an invocation may result in the invoker acquiring unauthorized capabilities.

Inconsistent parameter validation

- Mutually inconsistent, but individually adequate, sets of validity criteria.

Implicit sharing of privileged/confidential data

- Privileged information located in storage accessible to an inferior process.
- Indirect transfer of privileged information to an inferior process via acknowledgement or timing.

Asynchronous validation/inadequate serialization

- Asynchronous modification of storage shared by inferior and superior processes.
- Inadequate serialization of storage access among (trusted) superior processes.

Inadequate identification/authorization/authentication

- Inadequate resource identification.
- Bypass of controlled access security.

Violable prohibition/limit

- Violable system limit (illogical request).
- Violable system procedural prohibition.

Exploitable logic error

- Incorrect error handling.
- Instruction side-effects.
- Incorrect resource allocation/deallocation.
- Incorrect process/function initiation or termination.
- Control-state error trap.
- Situations not categorized above.

The Second Flaw Level

Incomplete parameter validation

- Inadequate attribute validation:
  - Procedure "A" invokes procedure "B," with user-specified parameter "Y."
  - There is at least one value "λ" of attribute "μ" of parameter "Y," such that its acceptance by procedure "B" will result in procedure "A" acquiring some capability to an unauthorized degree.
  - Procedure "B" accepts parameter "Y" when it has the value "λ."
- Improper reiteration of parameter validation:
  - Procedure "A" invokes procedure "B," with user-specified parameter "Y."
  - Multiple instances of parameter "Y" may be included in the calling sequence of procedure "A."
  - There is at least one value "λ" of parameter "Y," such that its acceptance by routine "B" would result in procedure "A" acquiring some capability to an unauthorized degree.
  - Procedure "B" validates at least one instance, but not every instance of parameter "Y."
- Intermittent parameter validation:
  - Procedure "A" may invoke procedure "C" with user-specified parameter "Y."
  - Procedure "A" may invoke procedure "B" which may, in turn, invoke procedure "C," with user-specified parameter "Y."
  - There is at least one value "λ" of parameter "Y," such that its acceptance by routine "C" would result in the invoking procedure acquiring some capability to an unauthorized degree.
  - Procedure "C" validates parameters when invoked by routine "A," but bypasses parameter validation when invoked by procedure "B."

Inconsistent parameter validation

- Mutually inconsistent validity criteria:
  - Procedure "A" and procedure "B" both use the identifier "λ" in association with controlling/monitoring access to system resource "X."
  - The conventions adopted by procedures "A" and "B" as to what constitutes a valid bit pattern for identifier "λ" are only partially intersecting.

- There is at least one value of identifier " λ ," such that access to resource "X" cannot be diminished once granted, or such that access to resource "X" can continue after the monitoring/recording of such access is terminated.

Implicit sharing of privileged/confidential data

- Access to storage containing privileged/confidential information:

Pattern 1

- Operator "F" (of procedure "A") writes a value " λ " into storage cell "X."
- Procedure "B" has access to storage cell "X."
- Procedure "B" reads value " λ " from storage cell "X."
- Operator "G" (of procedure "A" or procedure "C") obliterates value " λ ."
- Operator "F" precedes procedure "B" which precedes operator "G"/procedure "C."
- Value " λ " is privileged/confidential information not authorized to procedure "B."

Pattern 2

- Procedure "A" writes a value " λ " into storage cell "X."
- Procedure "A" releases storage cell "X."
- Storage cell "X" is assigned to procedure "B."
- Procedure "B" reads value " λ " from storage cell "X."
- Procedure "A" precedes procedure "B."
- Value " λ " is privileged/confidential information not authorized to procedure "B."

- Information leakage through acknowledgment or timing:

Pattern 1

- Procedure "A" validates a name/password submitted by user procedure "B."
- Procedure "A" validates each character of name " λ " in discrete steps.
- Each discrete step has an execution time greater than the effective resolution of the user-interval timer.
- Procedure "A" returns a negative acknowledgment " μ " to procedure "B" whenever character " λ_n " is incorrect.

- User procedure "B" or "C" initiates interval timer operator "T" concurrent with invoking procedure "A" and terminates it concurrent with receiving acknowledgment " μ ."
- User procedure "B" or "C" piecewise decomposes " λ " by measuring variations in the elapsed time returned by interval timer operator "T."

Pattern 2

- Procedure "A" validates a name/password submitted by user procedure "B" in storage cell "X."
- Procedure "A" accesses and validates each character of name " λ ."
- Storage cell "X" is located on an addressing accessibility boundary such that procedure "C" returns a negative acknowledgment " μ " to user procedure "B" whenever character " λ_n " is accessed.
- User procedure "B" piecewise decomposes name " λ " by shifting name " λ " one character position each time negative acknowledgment " μ " is received.

Asynchronous validation/inadequate serialization

- Asynchronous modification of shared storage:

Pattern 1

- Procedure "A" invokes procedure "B" with parameter "Y" located in user-accessible storage cell "X."
- Operator "F" (of procedure "B") reads and validates parameter "Y."
- There is at least one value " λ " of parameter "Y," such that its acceptance by procedure "B" will result in procedure "A" acquiring some capability to an unauthorized degree.
- There may be an operator "S" of user-invoked procedure "C" such that operator "F" precedes operator "S," which precedes operator "G" (of procedure "B").
- Operator "S" (of procedure "C") stores user-specified value " λ " into storage cell "X."
- Operator "G" (of procedure "B") accesses and accepts value " λ " from storage cell "X."

Pattern 2

- Operator "F" (of procedure "A") stores value " λ " in storage cell "X."
  - There is at least one value of " λ ," such that its acceptance by procedure "B" or by operator "G" (of procedure "A") will result in user procedure "C" acquiring some capability to an unauthorized degree.
  - There may be an operator "S" (of user-invoked procedure "D"), such that operator "F" precedes operator "S" which precedes operator "G"/procedure "B."
  - Operator "S" (of procedure "D") stores user-specified value " λ " into storage cell "X."
  - Procedure "B" or operator "G" accesses and accepts value " λ " from storage cell "X."
- Inadequate serialization control of protected storage:
    - User procedure "A" invokes procedure "B" with user-specified parameter "Y."
    - Operator "F" (of procedure "B") and operator "G" (of procedure "C") validate and store parameter "Y" in storage cell "X."
    - Operator "I" (of procedure "B") reads but does not revalidate parameter "Y" from storage cell "X."
    - There is at least one value " λ " of parameter "Y," valid for procedure "C," but such that its acceptance and use by operator "I" (of procedure "B") will result in user procedure "A" acquiring some capability to an unauthorized degree.
    - There may be a user procedure "D" or an operator "H" (of procedure "A"), such that operator "F" precedes operator "H"/procedure "D" which precedes operator "G" which precedes operator "I."
    - User procedures "A"/"D" do not have write-access to storage cell "X."
    - User procedure "D" or operator "H" (of user procedure "A") invokes procedure "C" which stores user-specified value " λ " into storage cell "X."
    - Operator "I" (of procedure "C") accesses and accepts value " λ " from storage cell "X."

Inadequate identification/authorization/authentication

● Inadequate resource identification/isolation:

Pattern 1

- User procedure "A" does not have write access to storage cell "X."
- There is a procedure "B" that will write user-specified value "λ" into storage cell "X."
- User procedure "A" invokes procedure "B" that writes user-specified value "λ" into storage cell "X."
- Procedure "C" invokes procedure "D" with user-specified parameter "Y" to identify system (data) element "R," located in storage cell "X."
- There is at least one value "λ" of system element "R," such that its acceptance by procedure "D" will result in procedure "A" acquiring some capability to an unauthorized degree.
- Procedure "D" assumes (as a system policy) that the location of system element "R" in storage cell "X" is sufficient to guarantee isolation from user procedure "A."
- Procedure "D" accepts and uses element "R" as an authenticated system element.

Pattern 2

- Procedure "A" invokes procedure "B" to locate system element/ procedure "R," identified by parameter "Y."
- Parameter "Y" is necessary but not sufficient to uniquely identify element/procedure "R."
- There may be a (procedure "B" located) user-supplied element/ procedure "S" which satisfies the identity requirements of parameter "Y."
- There may be at least one value "λ" of element/procedure "S," such that its acceptance and use by procedure "A" would result in procedure "C" acquiring some capability to an unauthorized degree.
- Procedure "B" returns the location of element/procedure "S" to procedure "A."
- Procedure "A" accepts and uses element/procedure "S" as an authenticated system element.

- Bypass of controlled-access security:

- Pattern 1

- There is at least one procedure "B" that will reject a request from user procedure "A" to access resource "X," on the basis of inadequate authorization.
    - User procedure "A" may legitimately invoke procedure "C" (or operator "S" of procedure "B").
    - Procedure "C" assumes (as policy) that the capability to invoke procedure "C" is sufficient authorization to acquire its capabilities.
    - The invocation of procedure "C" (or operator "S") results in user procedure "A" obtaining access to resource "X."

- Pattern 2

- There is a system procedure "B," such that user procedure "A" may request it to transfer control to user procedure "C" under specified conditions.
    - The transfer of control to User Procedure "C" results in Procedure "C" acquiring some capability to which it would not otherwise be authorized.

Violable prohibition/limit

- Violable system limit:

- User procedure "A" executes repeated invocations of procedure "B."
  - During each execution, procedure "B" acquires some incremental quantity of limited system resource "X."
  - As more of limited system resource "X" is acquired on behalf of user procedure "A," system service to other users is degraded or interrupted.

- Violable system procedural prohibition

- User procedure "A" invokes procedure "B" or accesses value "λ" from storage cell "X."
  - The invocation of procedure "B" or the access to value "λ" results in procedure "A" acquiring some capability to an unauthorized degree.
  - System policy (as expressed in its documentation) asserts that procedure "A" should not, or may not, invoke procedure "B" or value "λ."

Exploitable logic error

- Incorrect error-handling sequencing:
  - User procedure "A" submits an identifier in a request for authorization from system procedure "B."
  - System procedure "B" notifies procedure "A" of incorrect identification before recording the error.
  - System procedure "B" is interruptable during the time frame preceding recording of the error.
  - Procedure "A" interrupts the execution of system procedure "B" after receiving an error acknowledgment, but before the error is recorded.
  - User procedure "A" repeats this process until an acceptable identifier is submitted.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research & Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights.

NOTICE

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Energy Research & Development Administration to the exclusion of others that may be suitable.

Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Price: Printed Copy \$; Microfiche \$3.00

Page Range	Domestic Price	Page Range	Domestic Price
001-025	\$ 3.50	326-350	10.00
026-050	4.00	351-375	10.50
051-075	4.50	376-400	10.75
076-100	5.00	401-425	11.00
101-125	5.50	426-450	11.75
126-150	6.00	451-475	12.00
151-175	6.75	476-500	12.50
176-200	7.50	501-525	12.75
201-225	7.75	526-550	13.00
226-250	8.00	551-575	13.50
251-275	9.00	576-600	13.75
276-300	9.25	601-up	*
301-325	9.75		

*Add \$2.50 for each additional 100 page increment from 601 to 1,000 pages;
add \$4.50 for each additional 100 page increment over 1,000 pages.