Naval Research Laboratory

Washington, DC 20375-5320

NRL/FR/5542--93-9591

A Taxonomy of Computer Program Security Flaws, with Examples

Carl E. Landwehr
Alan R. Bull
John P. McDermott
William S. Choi

Center for Computer High Assurance Systems
Information Technology Division

November 19, 1993

Abstract

An organized record of actual flaws can be useful to designers, implementors, and evaluators of computer systems. This paper provides a taxonomy for computer program security flaws together with an appendix that carefully documents 50 actual security flaws. These flaws have all been described previously in the open literature, but in widely separated places. For those new to the field of computer security, they provide a good introduction to the characteristics of security flaws and how they can arise. Because these flaws were not randomly selected from a valid statistical sample of such flaws, we make no strong claims concerning the likely distribution of actual security flaws within the taxonomy. However, this method of organizing security flaw data can help those who have custody of more representative samples to organize them and to focus their efforts to remove and, eventually, to prevent the introduction of security flaws.

Subject terms: security and protection, security flaw, access controls, error/defect classification

This report was prepared for electronic distribution in September, 1994. Pagination varies from the originally published report; the Table of Contents and internal references to page numbers have been adjusted accordingly. Please send comments or requests for additional copies of this report, by e-mail to *landwehr@itd.nrl.navy.mil*, or retrieve PostScript via URL

http://www.itd.nrl.navy.mil/itd/5540/publications/index.html

(e.g., using Mosaic), or by anonymous ftp from

chacs.itd.nrl.navy.mil

under directory */pub/chacs*.

CONTENTS

A TAXONOMY OF COMPUTER PROGRAM SECURITY FLAWS, WITH EXAMPLES

INTRODUCTION

Knowing how systems have failed can help us build systems that resist failure. Petroski [1] makes this point eloquently in the context of engineering design, and although software failures may be less visible than those of the bridges he describes, they can be equally damaging. But the history of software failures, apart from a few highly visible ones, [2,3] is relatively undocumented. This report collects and organizes a number of actual security flaws that have caused failures, so that designers may do their work with a more precise knowledge of what has gone before.

Computer security flaws are any conditions or circumstances that can result in denial of service, unauthorized disclosure, unauthorized destruction of data, or unauthorized modification of data [4]. Our taxonomy attempts to organize information about flaws so that, as new flaws are added, users will gain a fuller understanding of which parts of systems and which parts of the system life cycle are generating more security flaws than others. This information should be useful not only to designers, but also to those faced with the difficult task of assessing the security of a system already developed. To accurately assess the security of a computer system, an analyst must find its vulnerabilities. To do this, the analyst must understand the system thoroughly and recognize that computer security flaws that threaten system security may exist anywhere in the system.

There is a legitimate concern that this kind of information could assist those who would attack computer systems. Partly for this reason, we have limited the cases described here to those that already have been publicly documented elsewhere and are relatively old. We do not suggest that we have assembled a representative random sample of all known computer security flaws, but we have tried to include a wide variety. We offer the taxonomy for the use of those who are presently responsible for repelling attacks and correcting flaws. Their data, organized this way and abstracted, could be used to focus efforts to remove security flaws and prevent their introduction.

Other taxonomies [5,6,7] have recently been developed for organizing data about software defects and anomalies of all kinds. These are primarily oriented toward collecting data during software development that will lead to improvements in the development process. We are primarily concerned with security flaws that are detected only after the software has been released for operational use; our taxonomy, while not incompatible with these efforts, reflects this perspective.

What Is a Security Flaw in a Program?

This question is akin to ''what is a bug?''. In fact, an inadvertently introduced security flaw in a program *is* a bug. Generally, a security flaw is a part of a program that can cause the system to violate its security requirements. Finding security flaws, then, demands some knowledge of system security requirements. These requirements vary according to the system and the application, so we cannot address them in detail here. Usually, they concern identification and authentication of users, authorization of particular actions, and accountability for actions taken.

We have tried to keep our use of the term ''flaw'' intuitive without conflicting with standard terminology. The *IEEE Standard Glossary of Software Engineering Terminology* [8] includes definitions of

- error: human action that produces an incorrect result (such as software containing a fault),
- fault: an incorrect step, process, or data definition in a computer program, and
- failure: the inability of a system or component to perform its required functions within specified performance requirements.

A failure may be produced when a fault is encountered. This glossary lists *bug* as a synonym for both *error* and *fault*. We use *flaw* as a synonym for bug, hence (in IEEE terms) as a synonym for fault, except that we include flaws that have been inserted into a system intentionally, as well as accidental ones.

IFIP WG10.4 has also published a taxonomy and definitions of terms [9] in this area. These define faults as the cause of errors that may lead to failures. A system fails when the delivered service no longer complies with the specification. This definition of ‘‘error’’ seems more consistent with its use in ‘‘error detection and correction’’ as applied to noisy communication channels or unreliable memory components than the IEEE one. Again, our notion of flaw corresponds to that of a fault, with the possibility that the fault may be introduced either accidentally or maliciously.

Why Look for Security Flaws in Computer Programs?

Early work in computer security was based on the ‘‘penetrate and patch’’ paradigm: analysts searched for security flaws and attempted to remove them. Unfortunately, this task was, in most cases, unending: more flaws always seemed to appear [10,11]. Sometimes the fix for a flaw introduced new flaws, and sometimes flaws were found that could not be repaired because system operation depended on them (e.g., cases I3 and B1 in the Appendix).

This experience led researchers to seek better ways of building systems to meet security requirements in the first place instead of attempting to mend the flawed systems already installed. Although some success has been attained in identifying better strategies for building systems [12,13], these techniques are not universally applied. More importantly, they do not eliminate the need to test a newly built or modified system (for example, to be sure that flaws avoided in initial specification haven’t been introduced in implementation).

Previous Work

Most of the serious efforts to locate security flaws in computer programs through penetration exercises have used the Flaw Hypothesis Methodology developed in the early 1970s [14]. This method requires system developers to first become familiar with the details of the way the system works (its control structure), then to generate hypotheses as to where flaws might exist in a system; to use system documentation and tests to confirm the presence of a flaw; and finally to generalize the confirmed flaws and use this information to guide further efforts. Although Ref. 14 provides lists of generic system functional flaws and generic operating system attacks, it does not provide a systematic organization for security flaws.

In the mid-70s both the Research in Secured Operating Systems (RISOS) project conducted at Lawrence Livermore Laboratories, and the Protection Analysis project conducted at the Information Sciences Institute of the University of Southern California (USC/ISI), attempted to characterize operating system security flaws. The RISOS final report [15] describes seven categories of operating system security flaws:

incomplete parameter validation,

inconsistent parameter validation,
implicit sharing of privileged/confidential data,
asynchronous validation/inadequate serialization,
inadequate identification/authentication/authorization,
violable prohibition/limit, and
exploitable logic error.

The report describes generic examples for each flaw category and provides reasonably detailed accounts for 17 actual flaws found in three operating systems: IBM OS/MVT, Univac 1100 Series, and TENEX. Each flaw is assigned to one of the seven categories.

The goal of the Protection Analysis (PA) project was to collect error examples and abstract patterns from them that, it was hoped, would be useful in automating the search for flaws. According to the final report [16], more than 100 errors that could permit system penetrations were recorded from six different operating systems (GCOS, MULTICS, and Unix, in addition to those investigated under RISOS). Unfortunately, this error database was never published and no longer exists [17]. However, the researchers did publish some examples, and they did develop a classification scheme for errors. Initially, they hypothesized 10 error categories; these were eventually reorganized into four ''global'' categories:

- domain errors, including errors of exposed representation, incomplete destruction of data within a deallocated object, or incomplete destruction of its context;
- validation errors, including failure to validate operands or to handle boundary conditions properly in queue management;
- naming errors, including aliasing and incomplete revocation of access to a deallocated object; and
- serialization errors, including multiple reference errors and interrupted atomic operations.

Although the researchers felt that they had developed a very successful method for finding errors in operating systems, the technique resisted automation. Research attention shifted from finding flaws in systems to developing methods for building systems that would be free of such errors.

Our goals are more limited than those of these earlier efforts in that we seek primarily to provide an understandable record of security flaws that have occurred. They are also more ambitious, in that we seek to categorize not only the details of the flaw, but also the genesis of the flaw and the time and place it entered the system.

Taxonomy

The taxonomy of security flaws proposed in this report classifies each flaw according to how, when, and where it was introduced into the system. The description of each flaw category refers to applicable cases (listed in the Appendix). Open-literature reports of security flaws are often abstract and fail to provide a realistic view of system vulnerabilities. Where studies do provide examples of actual vulnerabilities in existing systems, they are sometimes sketchy and incomplete lest hackers abuse the information. Our criteria for selecting cases are:

(1) the case must present a particular type of vulnerability clearly enough that a scenario or program that threatens system security can be understood by the classifier, and

(2) the potential damage resulting from the vulnerability described must be more than superficial.

Each case includes the name of the author or investigator, the type of system involved, and a description of the case.

The taxonomy and the cases summarized here can help system builders, administrators, and users beware of the types of security flaws they imply and develop strategies to prevent or counter them. At the same time, we know that the selected cases are but a small sample, and we caution against unsupported generalizations based on the flaws they exhibit. In particular, readers should not interpret the flaws recorded in the Appendix as indications that the systems in which they occurred are necessarily more or less secure than others. In most cases, the absence of a system from the Appendix simply reflects the fact that it has not been tested as thoroughly or had its flaws documented as openly as those we have cited. Readers are encouraged to communicate additional cases to the authors so that we can better understand where security flaws really occur.

FLAW CLASSIFICATION

We distinguish the nature of a flaw from the nature of its exploitation, and we focus on the former. For example, although the covert channel used in the TENEX password compromise scheme (case DT in the Appendix) could be exploited maliciously, it was introduced innocently. Consequently, we place this flaw, and similar covert channels, in the ‘‘Nonmalicious’’ category for genesis. Virtually all exploitations of flaws, except those done as part of penetration testing, could be categorized as malicious to some degree.

A given case may reveal several kinds of security flaws. For example, if a system programmer inserts a Trojan horse that exploits a covert channel to disclose sensitive information, both the Trojan horse and the covert channel are flaws in the operational system; the former will probably have been introduced maliciously, the latter inadvertently. Of course, any system that permits a user to invoke an uncertified program is vulnerable to Trojan horses. Whether the fact that a system permits users to install programs also represents a security flaw is an interesting question. The answer seems to depend on the context in which the question is asked. Permitting users of, say, an air traffic control system or, less threateningly, an airline reservation system, to install their own programs seems intuitively unsafe; it is a flaw. On the other hand, preventing owners of PCs from installing their own programs would seem ridiculously restrictive.

This report classifies security flaws according to *genesis, time of introduction,* and *location* (Figs. 1-3). Note that this classification does not partition the set of possible security flaws: the same flaw will show up at least once in each of these categories. The motive is to provide several different perspectives from which to consider possible sources of flaws in the system under study.

Within each of these categories, divisions and subdivisions are provided. Where feasible, these subdivisions define sets of mutually exclusive and collectively exhaustive categories. Often however, especially at the finer levels, such a partitioning is infeasible, and completeness of the set of categories cannot be assured. In general, we have tried to include categories only where they might help an analyst searching for flaws or a developer seeking to prevent them. A category for hardware flaws is included under ‘‘Location’’ for completeness. We understand that the increasing embedding of programs in hardware may yield increasing numbers of flaws that are in that ‘‘place,’’ but our focus is on flaws in programs, wherever they are found. A flaw in a program that has been frozen in silicon is still a program flaw to us; it would be placed in the appropriate category under ‘‘Operating System’’ rather than under ‘‘Hardware.’’ We reserve the use of the latter category for cases in which hardware exhibits security flaws that did not originate as errors in programs. We solicit proposals for additional categories.

					Count	Case ID's
Genesis	Intentional	Malicious	Trojan Horse	Non-Replicating	2	PC1 PC3
				Replicating (virus)	7	U1,PC2,PC4,MA1, MA2,CA1,AT1
			Trapdoor		(2)	(U1)(U10)
			Logic/Time Bomb		1	I8
		Nonmalicious	Covert Channel	Storage	1	DT1
				Timing	2	I9,D2
			Other		5	I7,B1,U3,U6,U10
	Inadvertent	Validation Error (Incomplete/Inconsistent)			10	I4,I5,MT1,MU2,MU4, MU8,U7,U11,U12,U13
		Domain Error (Including Object Re-use, Residuals, and Exposed Representation Errors)			7	I3,I6,MT2,MT3, MU3,UN1,D1
		Serialization/aliasing (Including TOCTTOU Errors)			2	I1,I2
		Identification/Authentication Inadequate			5	MU1,U2,U4,U5,U14
		Boundary Condition Violation (Including Resource Exhaustion and Violable Constraint Errors)			4	MT4,MU5,MU6,U9
		Other Exploitable Logic Error			4	MU7,MU9,U8,IN1

Fig. 1. Security flaw taxonomy: flaws by genesis. Parenthesized entries indicate secondary assignments.

			Count	Case ID's
Time of Introduction	During Development	Requirement/ Specification/ Design	22	I1, I2, I3, I4, I5, I6 I7 ,I9, MT2, MT3, MU4, MU6, B1, UN1 ,U6, U7, U9, U10, U13, U14, D2, IN1
		Source Code	15	MT1, MT4, MU1, MU2, MU5, MU7, MU8, DT1, U2, U3, U4, U5, U8, U11, U12
		Object Code	1	U1
	During Maintenance		3	D1, MU3, MU9
	During Operation		9	I8, PC1, PC2, PC3, PC4, MA1, MA2, CA1, AT1

Fig. 2. Security flaw taxonomy: flaws by time of introduction

				Count	Case ID's
Location	Software	Operating System	System Initialization	8	U5, U13, PC2, PC4, MA1, MA2, AT1 ,CA1
			Memory Management	2	MT3, MU5
			Process Management/ Scheduling	10	I6, I9, MT1, MT2, MU2, MU3 MU4, MU6, MU7, UN1
			Device Management (including I/O, networking)	3	I2, I3, I4
			File Management	6	I1, I5, MU8, U2, U3, U9
			Identification/Authentication	5	MU1, DT1, U6, U11, D1
			Other/Unknown	1	MT4
		Support	Privileged Utilities	10	I7, B1, U4, U7, U8, U10, U12, U14, PC1, PC3
			Unprivileged Utilities	1	U1
		Application		1	I8
	Hardware			3	MU9, D2, IN1

Fig. 3. Security flaw taxonomy: flaws by location

By 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.

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—inadvertently created Trojan horse programs are hardly likely! Inadvertent flaws in requirements or specifications ultimately manifest themselves in the implementation; flaws may also be introduced inadvertently during maintenance.

Malicious flaws presumably are more difficult to detect and are more likely to result in serious degradation of system security than inadvertent ones. Nevertheless, an inadvertent flaw that is exploited by a malicious intruder can be just as damaging to system security as a malicious flaw.

Malicious Flaws

Malicious flaws have acquired colorful names, including *Trojan horse, trapdoor, time-bomb*, 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 [4,18-20]. It generally refers to a program that masquerades as a useful service but exploits rights of the program’s user—rights not possessed by the author of the Trojan horse—in a way the user does not intend.

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 itself by creating new processes or files to contain its code, instead of modifying existing storage entities, is often called a *worm* [23]. Denning [26] provides a general discussion of these terms; differences of opinion about the term applicable to a particular flaw or its exploitations sometimes occur [22,3].

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). For example, a programmer of automated teller machines (ATMs) might be required to check a personal identification number (PIN) read from a card against the number keyed in by the user. If the numbers match, the user is to be permitted to enter transactions. By adding a disjunct to the condition that implements this test, the programmer can provide a trapdoor, shown in italics below:

if PINcard=PINkeyed *OR PINkeyed=9999* **then** {permit transactions}

In this example, 9999 would be a universal PIN that would work with any bank card submitted to the ATM. Of course the code in this example would be easy for a code reviewer, although not an ATM user, to spot, so a malicious programmer would need to take additional steps to hide the code that implements the trapdoor. If passwords are stored in a system file rather than on a user-supplied card, a special password known to an intruder mixed in a file of legitimate ones might be difficult for reviewers to find.

It might be argued that a login program with a trapdoor is really a Trojan horse in the sense defined above, but the two terms are usually distinguished [19]. Thompson [25] describes a method for building a Trojan horse compiler that can install both itself and a trapdoor in a Unix password-checking routine in future versions of the Unix system.

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 I8). 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.

Intentional, Nonmalicious Flaws

A Trojan horse program may convey sensitive information to a penetrator over *covert channels*. A covert channel is simply a path used to transfer information in a way not intended by the system’s designers [27]. Since covert channels, by definition, are channels not placed there intentionally, they should perhaps appear in the category of inadvertent flaws. We categorize them as intentional but nonmalicious flaws because they frequently arise in resource-sharing services that are intentionally part of the system. Indeed, the most difficult ones to eliminate are those that arise in the fulfillment of essential system requirements. Unlike their creation, their exploitation is likely to be malicious. Exploitation of a covert channel usually involves a service program, most likely a Trojan horse. This program generally has access to confidential data and can encode that data for transmission over the covert channel. It also will contain a receiver program that ‘‘listens’’ to the chosen covert channel and decodes the message for a penetrator. If the service program could communicate confidential data directly to a penetrator without being monitored, of course, there would be no need for it to use a covert channel.

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. What distinguishes this case from that of ordinary operation is that the bits are used to convey encoded information. Examples would include using a file intended to hold only audit information to convey user passwords—using the name of a file or perhaps status bits associated with it that can be read by all users to signal the contents of the file. 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 paging rate, the time a certain transaction requires to execute, the time it takes to gain access to a shared bus) and infer protected information.

The distinction between storage and timing channels is not sharp. Exploitation of either kind of channel requires some degree of synchronization between the sender and receiver. It also requires the ability to modulate the behavior of some shared resource. In practice, covert channels are often distinguished on the basis of how they can be detected: those detectable by information flow analysis of specifications or code are considered storage channels.

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’’ [3] (case U10) could be placed in this category.

Inadvertent Flaws

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 flaws can remain undetected and later cause problems during operation and maintenance of the 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).

There are many possible ways to organize flaws within this category. Recently, Chillarege [6] and Sullivan [28] have published classifications of defects (not necessarily security flaws) found in commercial operating systems and databases. Efforts by Bisbey et al., [16] and Abbott [15] provide classifications specifically for security flaws. After trying these classifications, as well as one we developed, on the set of examples included in the Appendix, we found the taxonomy described below, which draws primarily on the work of Bisbey and Abbott, most descriptive. This part of the taxonomy is probably the one with the greatest overlap among its categories.

Inadvertent flaws can be classified as flaws related to

validation errors,
domain errors,
serialization/aliasing errors,
errors of inadequate identification/authentication,
boundary condition errors, and
other exploitable logic errors.

Validation flaws occur 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 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 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 an abstract object, intended to be hidden in the current domain, is in fact exposed (see cases B1 and DT1). Errors classed by Abbott as ‘‘implicit sharing of privileged/confidential data’’ will generally fall in this category.

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.

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 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.

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.

By 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 [6,29] 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 can usually be attributed to erroneous or incorrectly implemented requirements 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, an early 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 also important to recognize the possibility of malicious intrusion into the system during both development and maintenance. 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 example [25] and a limited defense posed by McDermott [30]).

The original designers and programmers of a system are rarely involved in its maintenance; flaws introduced during maintenance are often attributable to the maintainer's lack of understanding of the system as a whole. Not infrequently, an attempt to correct one flaw will create another. Vigilance is also required to thwart malicious attempts to introduce security flaws through software maintenance. 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.

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 across steps. Thus in fact, identification of the time a security flaw is introduced overlaps the definition of the place (requirements document, specification, or code) it occurs. Issues of concern to the security analyst for each of these subcategories are discussed here.

Requirements and Specifications

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 software modules and the parameters associated with them. They are the basis on which the source code is built. The specifier is often responsible for implementing the specification as well.

If 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 security flaws.

Apart from checking for specification completeness, the security analyst 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 programs never be modified at an operational site. But if the delay in repairing errors detected in 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 a large software system, inadvertent flaws in source code are frequently a by-product of 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 scaffolding may circumvent security controls. If the scaffolding is left in place in the 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 inclusion of a ‘‘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 system security. Although malicious source code can be detected through manual review of 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 who gains write access to source code files, but source code flaws can also be introduced indirectly. For example, if a programmer authorized to write source code files inadvertently invokes a Trojan horse editor (or compiler, linker, loader, etc.), the Trojan horse could use the programmer’s privileges to modify source code files. Instances of subtle indirect tampering with source code are difficult to document, but Trojan horse programs that grossly modify all a user's files, and hence the source code files, have been created.

Object Code

Object code programs are generated by compilers or assemblers and represent 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 mature and has been widely used. When such errors do occur as a result of errors in 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 selling source code and object code for their products), it is a good hiding place for malicious security flaws (again, see Thompson [25]).

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 replacement of an existing object module by a version of it that incorporates a 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 replace 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. Creating a virus generally requires some knowledge of the operating system and programming conventions of the target system; 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.

During Maintenance

Inadvertent flaws introduced during maintenance are often attributable to the maintenance programmer's failure to understand the system as a whole. Since software production facilities 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 induce other flaws and this cycle will continue. Software modified during maintenance should 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) [12].

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.

By 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.

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 the most severe 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 reviewed.

Software flaws can occur in *operating system programs*, *support software*, or *application (user) software*. This is a rather coarse division, but even so the boundaries are not always clear.

Operating System Programs

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, authenticate users, control access, and coordinate the sharing of all 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.

We categorize operating system security flaws according to whether they occur in the functions for

system initialization,
memory management,
process management,
device management (including networking),
file management, or
identification/authentication.

We include an *other/unknown* category for flaws that do not fall into any of the preceding classes.

System initialization, although it may be handled routinely, is often complex. Flaws in this area can occur either because the operating system fails to establish the initial protection domains as specified (for example, it may set up ownership or access control information improperly) or because the system administrator has not specified a secure initial configuration for the system. 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 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 I6, or to deny service to others, as in case MU5.

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 I2). 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 I3).

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 controls to 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, MU8, and U2).

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.

Support Software

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. In Unix, for example, any ‘‘setuid’’ program owned by ‘‘root’’ effectively runs with 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.

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 lines, device queues, mail queues) as well as for files belonging to individual users (e.g., mailboxes). They 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 U13).

Even unprivileged software can represent a significant vulnerability because these programs are widely shared, and users tend to rely on them implicitly. The damage inflicted by flawed, unprivileged support software (e.g., by an embedded Trojan horse) is normally limited to 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 I7); intentional but nonmalicious flaws in support software have also been recorded (case B1).

Application Software

We categorize programs that have no special system privileges and are not widely shared as application software. Damage caused by inadvertent software flaws at the application level is usually restricted to the executing process, since most operating systems can prevent one process from damaging another. This does not mean that application software cannot do significant damage to a user’s own stored files, however, as many victims of Trojan horse and virus programs have become painfully aware. An application program generally executes with all the privileges of the user who invokes it, including the ability to modify permissions, read, write, or delete any files that user owns. In the context of most personal computers now in use, this means that an errant or malicious application program can, in fact, destroy all the information on an attached hard disk or writeable floppy disk.

Inadvertent flaws in application software that cause program termination or incorrect output, or can generate undesirable conditions such as infinite looping have been discussed previously. Malicious intrusion at the application software level usually requires access to the source code (although a virus could conceivably attach itself to application object code) and can be accomplished in various ways.

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 set architecture. It is not uncommon for even widely 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 problems such as improper synchronization and execution, bit loss during data transfer, or incorrect results after execution of arithmetic or logical instructions (see case MU9). Intentional 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 intrusion at the hardware level may result in improper execution of programs, system shutdown, or, conceivably, the introduction of subtle flaws exploitable by software.

DISCUSSION

The scatter plots in Figs. 4-7 illustrate the characteristics of the cases included in the Appendix. Because we do not claim that this selection of security flaws is statistically representative, we cannot use these plots to draw strong conclusions about how, when, or where security flaws are most likely to be introduced. However, we believe that the kinds of plots shown would be an effective way to abstract and present information from more extensive, and perhaps more sensitive, data sets. Each symbol plotted in Fig. 4 captures one or more flaws described in the Appendix according to its genesis, location, and time of introduction. Figure 5 plots flaw genesis against location again, but uses symbol size to reflect the number of cases in the appendix for a particular point. Figures 6 and 7 use the same technique, plotting genesis vs time of introduction and location vs time of introduction, respectively.

We also have some observations based on our experiences in creating the taxonomy and applying it to these examples. It seems clear that security breaches, like accidents, typically have several causes. Often, unwarranted assumptions about some aspect of system behavior lead to security flaws. Problems arising from asynchronous modification of a previously checked parameter illustrate this point: the person who coded the check assumed that nothing could cause that parameter to change before its use—when an asynchronously operating process could in fact do so. Perhaps the most dangerous assumption is that security need not be addressed—that the environment is fundamentally benign, or that security can be added later. Both Unix and personal computer operating systems clearly illustrate the cost of this assumption. One cannot be surprised when systems designed without particular regard to security requirements exhibit security flaws. Those who use such systems live in a world of potentially painful surprises.

ACKNOWLEDGMENTS

The idea for this report was conceived several years ago when we were considering how to provide automated assistance for detecting security flaws. We found that we lacked a good characterization of the things we were looking for. It has had a long gestation and many have assisted in its delivery. We are grateful for the participation of Mark Weiser (then of the University of Maryland) and LCDR Philip Myers of the Space and Naval Warfare Combat Systems Command (SPAWAR) in this early phase of the work. We also thank the National Computer Security Center and SPAWAR for their continuing financial support. The authors gratefully acknowledge the assistance provided by the many reviewers of earlier drafts of this paper. Their comments helped us refine the taxonomy, clarify the presentation, distinguish the true computer security flaws from the mythical ones, and place them accurately in the taxonomy. Comments from Gene Spafford,

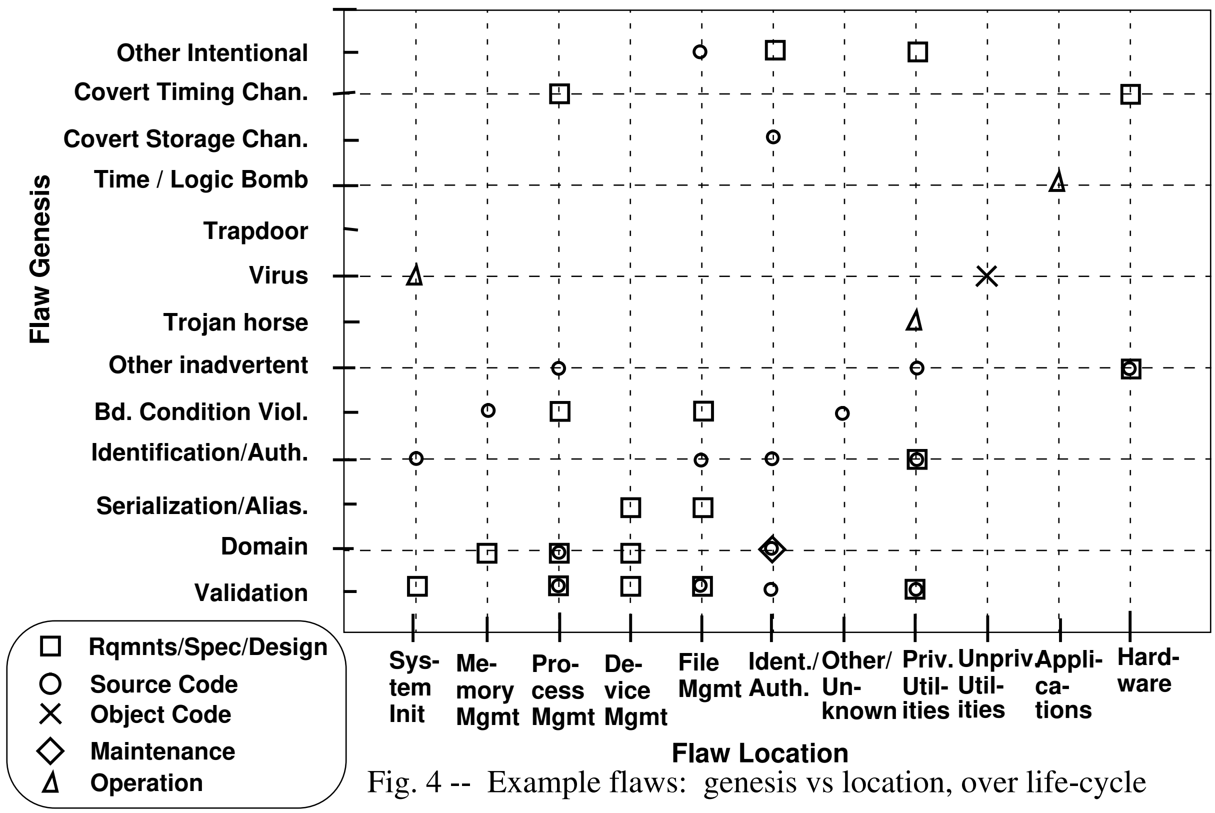

Fig. 4 -- Example flaws: genesis vs location, over life-cycle

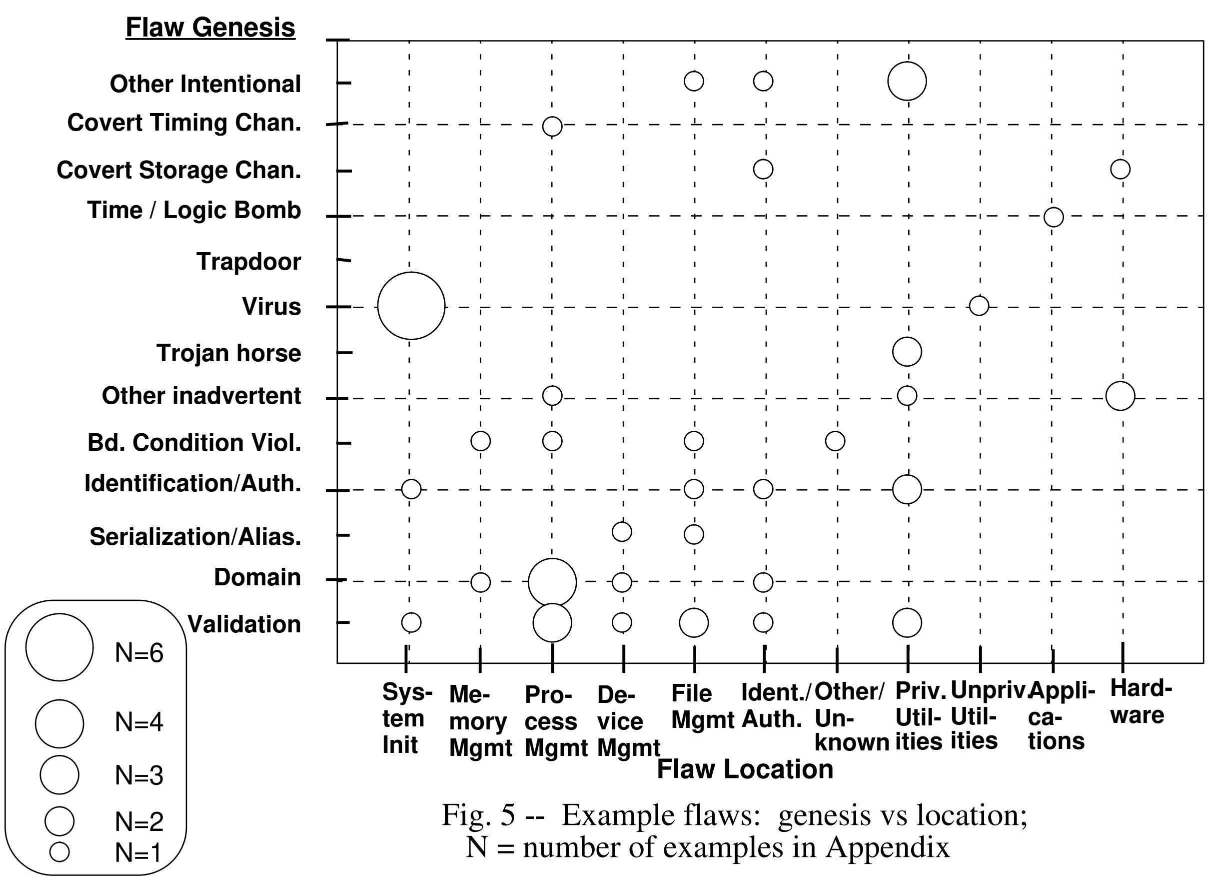

Fig. 5 -- Example flaws: genesis vs location;
N = number of examples in Appendix

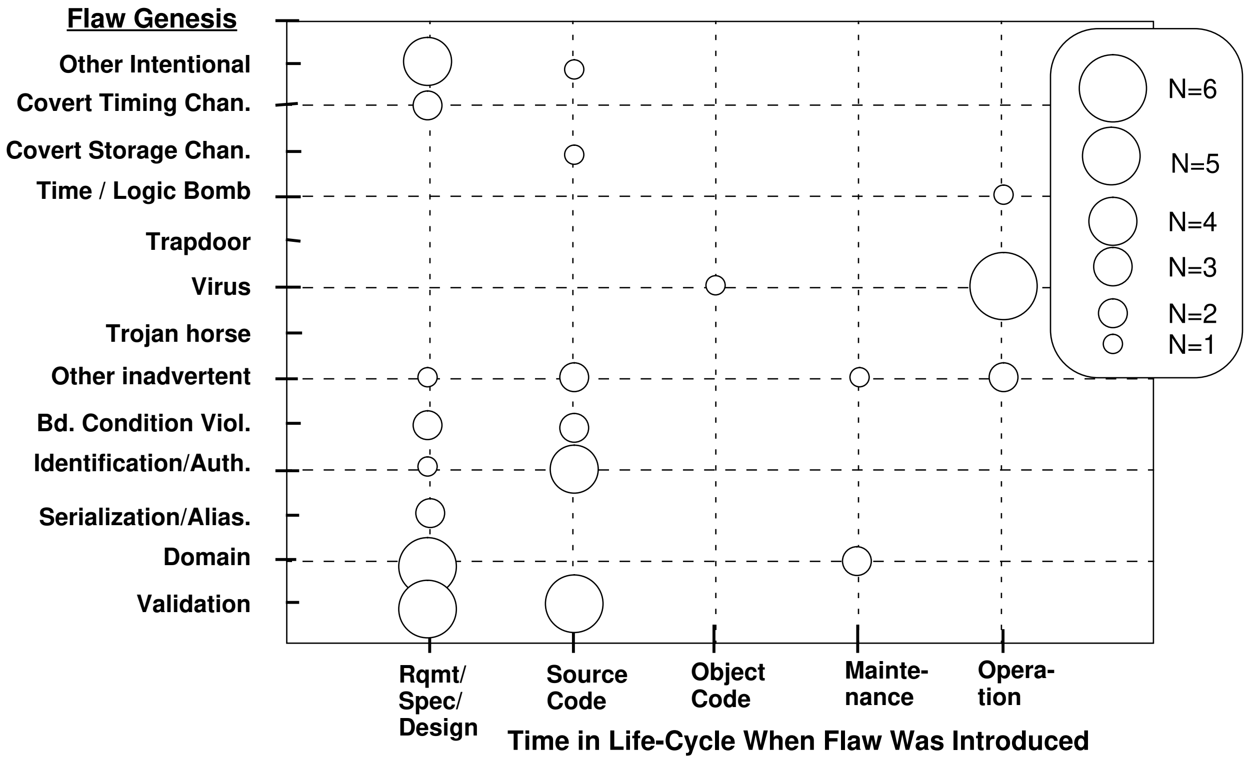

Fig. 6 -- Example flaws: genesis vs time introduced;
N = number of examples in Appendix

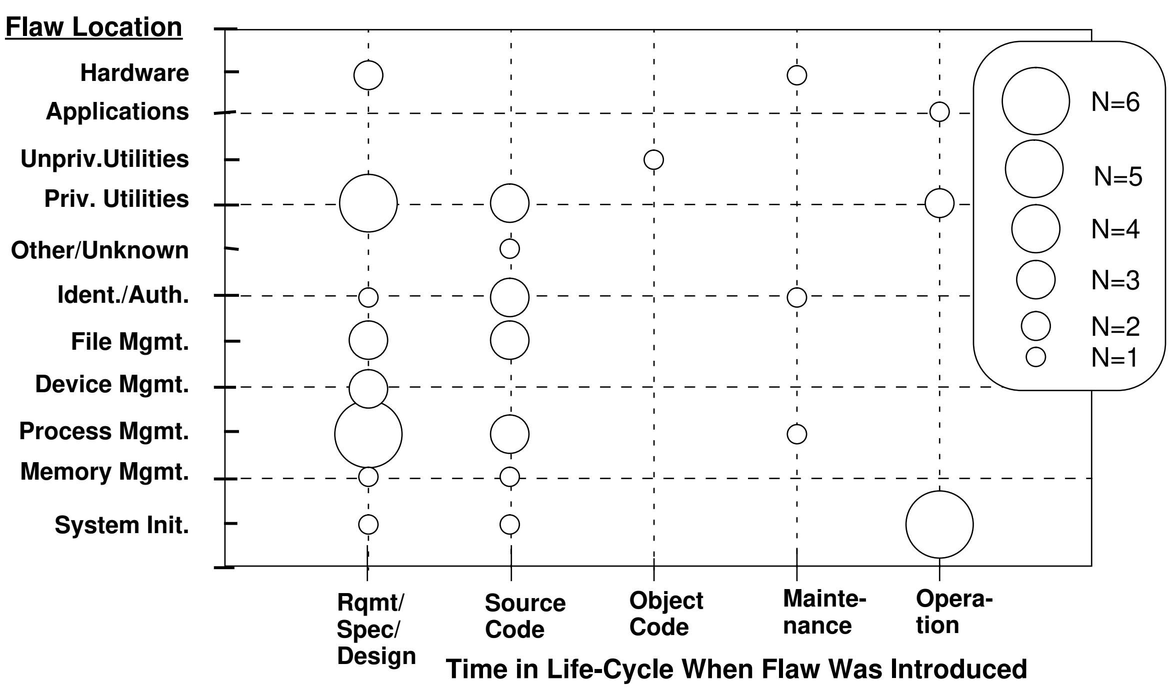

Fig. 7 -- Example flaws: location vs time of introduction;
N = number of examples in Appendix

Matt Bishop, Paul Karger, Steve Lipner, Robert Morris, Peter Neumann, Philip Porras, James P. Anderson, and Preston Mullen were particularly extensive and helpful. Jurate Maciunas Landwehr suggested the form of Fig. 4. Thomas Beth, Richard Bisbey II, Vronnie Hoover, Dennis Ritchie, Mike Stolarchuck, Andrew Tanenbaum, and Clark Weissman also provided useful comments and encouragement; we apologize to any reviewers we have inadvertently omitted. Any remaining errors are, of course, our responsibility.

REFERENCES

1. H. Petroski, *To Engineer is Human: The Role of Failure in Successful Design* (Vintage Books, New York, NY, 1992).
2. N.G. Leveson and C.S. Turner, ''An Investigation of the Therac-25 Accidents,'' *UCI TR 92-108*, Inf. and Comp. Sci. Dept, Univ. of Cal.-Irvine, Irvine, CA.
3. E.H. Spafford, ''Crisis and Aftermath,'' *Comm. ACM* **32**(6), 678-687 (June 1989).
4. C.E. Landwehr, ''Formal Models for Computer Security,'' *ACM Computing Surveys* **13**(3), 247-278 (September 1981).
5. C.L. Brehmer and J.R. Carl, ''Incorporating IEEE Standard 1044 into Your Anomaly Tracking Process,'' *CrossTalk, J. Defense Software Engineering,* **40**, 9-16 (January 1993).
6. R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M-Y Wong, ''Orthogonal Defect Classification—A Concept for In-process Measurements,'' *IEEE Trans. on Software Engineering* **18**(11), 943-956 (Nov. 1992).
7. W.A. Florac, ''Software Quality Measurement: A Framework for Counting Problems and Defects,'' *CMU/SEI-92-TR-22* (Software Engineering Institute, Pittsburgh, PA, Sept. 1992.).
8. ANSI/IEEE Standard 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology* (IEEE, New York, NY, 1990).
9. J.C. LaPrie (ed.), *Dependability: Basic Concepts and Terminology*, Vol. 6, Springer-Verlag Series in Dependable Computing and Fault-Tolerant Systems, New York.
10. P.G. Neumann, ''Computer Security Evaluation,'' *1978 National Computer Conference, AFIPS Conf. Proceedings* **47** (1978) pp. 1087-1095.
11. R.R. Schell, ''Computer Security: the Achilles Heel of the Electronic Air Force?,'' *Air University Review* **30**(2), 16-33 (Jan.-Feb. 1979).
12. Department of Defense, *Trusted Computer System Evaluation Criteria, DoD 5200.28-STD*, December 1985.
13. C.E. Landwehr, ''The Best Available Technologies for Computer Security,'' *COMPUTER* **16**(7), 86-100, Los Alamitos, CA (July 1983).
14. R.R. Linde, ''Operating System Penetration,'' *AFIPS National Computer Conference* (1975) pp. 361-368.
15. R.P. Abbott, J.S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, and D.A. Webb, ''Security Analysis and Enhancements of Computer Operating Systems,'' NBSIR 76-1041 (National Bureau of Standards, ICST, April 1976).

16. R. Bisbey II and D. Hollingworth, ''Protection Analysis Project Final Report,'' ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute (May 1978).
17. R. Bisbey II, *private communication*, 26 July 1990.
18. J.P. Anderson, ''Computer Security Technology Planning Study,'' *ESD-TR-73-51, Vols I and II*, NTIS AD758206, Hanscom Field, Bedford, MA (October 1972).
19. D.E. Denning, *Cryptography and Data Security* (Addison-Wesley Publishing Company, Inc., Reading, MA, 1982).
20. M. Gasser, *Building a Secure Computer System* (Van Nostrand Reinhold, New York, NY, 1988).
21. C.P. Pfleeger, *Security in Computing* (Prentice Hall, Englewood Cliffs, NJ, 1989).
22. F. Cohen, ''Computer Viruses: Theory and Experiments,'' *7th DoD/NBS Computer Security Conference*, 240-263, Gaithersburg, MD (Sept. 1984).
23. J.F. Schoch and J.A. Hupp, ''The 'Worm' Programs—Early Experience with a Distributed Computation,'' *Comm. ACM* **25**(3), 172-180 (March 1982).
24. J.A. Rochlis and M.W. Eichen, ''With Microscope and Tweezers: The Worm from MIT's Perspective,'' *Comm. ACM* **32**(6), 689-699 (June 1989).
25. K. Thompson, ''Reflections on Trusting Trust,'' *Comm. ACM* **27**(8), 761-763 (August 1984).
26. P.J. Denning, ''Computer Viruses,'' *American Scientist* **76**, 236-238 (May-June 1988).
27. B.W. Lampson, ''A Note on the Confinement Problem,'' *Comm. ACM* **16**(10), 613-615 (October 1973).
28. M. Sullivan and R. Chillarege, ''A Comparison of Software Defects in Database Management Systems and Operating Systems,'' *Proc. 22nd Int. Symp. on Fault-Tolerant Computer (FTCS-22)* (July 1992).
29. D.M. Weiss and V.R. Basili, ''Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory,'' *IEEE Trans. Software Engineering* **SE-11**(2), 157-168 (February 1985).
30. J.P. McDermott, ''A Technique for Removing an Important Class of Trojan Horses from High Order Languages,'' *Proc. 11th National Computer Security Conference*, NBS/NCSC, Gaithersburg, MD, pp. 114-117, October 1988.
31. P. Elmer-DeWitt, ''Invasion of the Data Snatchers,'' *TIME Magazine*, 62-67 (Sept. 26, 1988).
32. D. Ferbrache, *A Pathology of Computer Viruses* (Springer-Verlag, New York, NY, 1992).

Appendix

SELECTED SECURITY FLAWS

The following case studies exemplify security flaws. Without making claims as to the completeness or representativeness of this set of examples, we believe they will help designers know what pitfalls to avoid and security analysts know where to look when examining code, specifications, and operational guidance for security flaws.

All of the cases documented here (except possibly one) reflect actual flaws in released software or hardware. For each case, a source (usually with a reference to a publication) is cited, the software/hardware system in which the flaw occurred is identified, the flaw and its effects are briefly described, and the flaw is categorized according to the taxonomy.

Where it has been difficult to determine with certainty the time or place a flaw was introduced, the most probable category (in the judgment of the authors) has been chosen, and the uncertainty is indicated by the annotation '?'. In some cases, a flaw is not fully categorized. For example, if the flaw was introduced during the requirements/specification phase, then the place in the code where the flaw is located may be omitted.

The cases are grouped according to the systems on which they occurred (Unix, which accounts for about a third of the flaws reported here, is considered a single system), and the systems are ordered roughly chronologically. Since readers may not be familiar with the details of all of the architectures included here, brief introductory discussions of relevant details are provided as appropriate.

The codes used to refer to systems: .

IBM /360 and /370 Systems

In the IBM System /360 and /370 architecture, the Program Status Word (PSW) defines the key components of the system state. These include the current machine state (problem state or supervisor state) and the current storage key. Two instruction subsets are defined: the problem state instruction set, which excludes privileged instructions (loading the PSW, initiating I/O operations, etc.) and the supervisor state instruction set, which includes all instructions. Attempting to execute a privileged operation while in problem state causes an interrupt. A problem state program that wishes to invoke a privileged operation normally does so by issuing the Supervisor Call (SVC) instruction, which also causes an interrupt.

Main storage is divided into 4K byte pages; each page has an associated 4-bit storage key. Typically, user memory pages are assigned storage key 8, while a system storage page will be assigned a storage key from 0 to 7. A task executing with a nonzero key is permitted unlimited access to pages with storage keys that match its own. It can also read pages with other storage keys that are not marked as fetch-protected. An attempt to write into a page with a nonmatching key causes an interrupt. A task executing with a storage key of zero is allowed unrestricted access to all pages, regardless of their key or fetch-protect status. Most operating system functions execute with a storage key of zero.

The I/O subsystem includes a variety of *channels* that are, in effect, separate, special-purpose computers that can be programmed to perform data transfers between main storage and auxiliary devices (tapes, disks, etc.). These channel programs are created dynamically by device driver programs executed by the CPU. The channel is started by issuing a special CPU instruction that provides the channel with an address in main storage from which to begin fetching its instructions. The channel then operates in parallel with the CPU and has independent and unrestricted access to main storage. Thus, any controls on the portions of main storage that a channel could read or write must be embedded in the channel programs themselves. This parallelism, together with the fact that channel programs are sometimes (intentionally) self-modifying, provides complexity that must be carefully controlled if security flaws are to be avoided.

OS/360 and MVS (Multiple Virtual Storages) are multiprogramming operating systems developed by IBM for this hardware architecture. The Time Sharing Option (TSO) under MVS permits users to submit commands to MVS from interactive terminals. VM/370 is a virtual machine monitor operating system for the same hardware, also developed by IBM. The KVM/370 system was developed by the U.S. Department of Defense as a high-security version of VM/370. MTS (Michigan Terminal System), developed by the University of Michigan, is an operating system designed especially to support both batch and interactive use of the same hardware.

MVS supports a category of privileged, non-MVS programs through its Authorized Program Facility (APF). APF programs operate with a storage key of 7 or less and are permitted to invoke operations (such as changing to supervisor mode) that are prohibited to ordinary user programs. In effect, APF programs are assumed to be trustworthy, and they act as extensions to the operating system. An installation can control which programs are included under APF. RACF (Resource Access Control Facility) and Top Secret are security packages designed to operate as APF programs under MVS.

Case: I1

Source: Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

System: IBM OS/360

Description: In OS/360 systems, the file access checking mechanism could be subverted. When a password was required for access to a file, the filename was read and the user-supplied password was checked. If it was correct, the file name was re-read and the file was opened. It was possible, however, for the user to arrange that the filename be altered between the first and second readings. First, the user would initiate a separate background process to read data from a tape into the storage location that was also used to store the filename. The user would then request access to a file with a known password. The system would verify the correctness of the password. While the password was being checked, the tape process replaced the original filename with a file for which the user did not have the password, and this file would be opened. The flaw is that the user can cause parameters to be altered after they have been checked (this kind of flaw is sometimes called a time-of-check-to-time-of-use (TOCTTOU) flaw). It could probably have been corrected by copying the parameters into operating system storage that the user could not cause to be altered.

Genesis: Inadvertent: Serialization

Time: During development: Requirement/Specification/Design

Place: Operating System: File Management

Case: I2

Source: C.R. Attanasio, P.W. Markstein, and R.J. Phillips, ''Penetrating an operating system: a study of VM/370 integrity,'' *IBM Systems Journal*, 1976, pp. 102-116.

System: IBM VM/370

Description: By carefully exploiting an oversight in condition-code checking (a retrofit in the basic VM/370 design) and the fact that CPU and I/O channel programs could execute simultaneously, a penetrator could gain control of the system. Further details of this flaw are not provided in the cited source, but it appears that a logic error (''oversight in condition-code checking'') was at least partly to blame.

Genesis: Inadvertent: Serialization

Time: During development: Requirement/Specification/Design

Place: Operating System: Device Management

Case: I3

Source: C.R. Attanasio, P.W. Markstein, and R.J. Phillips, ''Penetrating an operating system: a study of VM/370 integrity,'' *IBM Systems Journal*, 1976, pp. 102-116.

System: IBM VM/370

Description: As a virtual machine monitor, VM/370 was required to provide I/O services to operating systems executing in individual domains under its management, so that their I/O routines would operate almost as if they were running on the bare IBM/370 hardware.

Because the OS/360 operating system (specifically, the Indexed Sequential Access Method (ISAM) routines) exploited the ability of I/O channel programs to modify themselves during execution, VM/370 included an arrangement whereby portions of channel programs were executed from the user's virtual machine storage rather than from VM/370 storage. This permitted a penetrator, mimicking an OS/360 channel program, to modify the commands in user storage before they were executed by the channel and thereby to overwrite arbitrary portions of VM/370.

Genesis: Inadvertent: Domain (?) This flaw might also be classed as (Intentional, Non-Malicious, Other), if it is considered to reflect a conscious compromise between security and both efficiency in channel program execution and compatibility with an existing operating system.

Time: During development: Requirement/Specification/Design

Place: Operating System: Device Management

Case: I4

Source: C.R. Attanasio, P.W. Markstein, and R.J. Phillips, ''Penetrating an operating system: a study of VM/370 integrity,'' *IBM Systems Journal*, 1976, pp. 102-116.

System: IBM VM/370

Description: In performing static analysis of a channel program issued by a client operating system for the purpose of translating it and issuing it to the channel, VM/370 assumed that the meaning of a multi-word channel command remained constant throughout the execution of the channel program. In fact, channel commands vary in length, and the same word might, during execution of a channel program, act both as a separate command and as the extension of another (earlier) command, since a channel program could contain a backward branch into the middle of a previous multi-word channel command. By careful construction of channel programs to exploit this blind spot in the analysis, a user could deny service to other users (e.g., by constructing a nonterminating channel program), read restricted files, or even gain complete control of the system.

Genesis: Inadvertent: Validation (?) The flaw seems to reflect an omission in the channel program analysis logic. Perhaps additional analysis techniques could be devised to limit the specific set of channel commands permitted, but determining whether an arbitrary channel program halts or not appears equivalent to solving the Turing machine halting problem. On this basis, this could also be argued to be a design flaw.

Time: During development: Requirement/Specification/Design

Place: Operating System: Device Management

Case: I5

Source: Walter Opaska, ''A security loophole in the MVS operating system,'' *Computer Fraud and Security Bulletin,* May 1990, Elsevier Science Publishers, Oxford, pp. 4-5.

System: IBM /370 MVS(TSO)

Description: Time Sharing Option (TSO) is an interactive development system that runs on top of MVS. Input/Output operations are only allowed on allocated files. When files are allocated (via the TSO ALLOCATE function), for reasons of data integrity the requesting user or program gains exclusive use of the file. The flaw is that a user is allowed to allocate files whether or not he or she has access to the files. A user can use the ALLOCATE function on files such as SMF (System Monitoring Facility) records, the TSO log-on procedure lists, the ISPF user profiles, and the production and test program libraries to deny service to other users.

Genesis: Inadvertent: Validation (?) The flaw apparently reflects omission of an access permission check in program logic.

Time: During development: Requirement/Specification/Design (?) Without access to design information, we cannot be certain whether the postulated omission occurred in the coding phase or prior to it.

Place: Operating System: File Management

Case: I6

Source: R. Paans and G. Bonnes, ''Surreptitious security violation in the MVS operating system,'' in *Security, IFIP/Sec '83*, V. Fak, ed., North Holland, 1983, pp. 95-105.

System: IBM MVS(TSO)

Description: Although TSO attempted to prevent users from issuing commands that would operate concurrently with each other, it was possible for a program invoked from TSO to invoke multi-tasking. Once this was achieved, another TSO command could be issued to invoke a program that executed under the Authorized Program Facility (APF). The concurrent user task could detect when the APF program began authorized execution (i.e., with storage key<8). At this point the entire user's address space (including both tasks) was effectively privileged, and the user-controlled task could issue privileged operations and subvert the system. The flaw here seems to be that when one task gained APF privilege, the other task was able to do so as well—that is, the domains of the two tasks were insufficiently separated.

Genesis: Inadvertent: Domain

Time: Development: Requirement/Specification/Design (?)

Place: Operating System: Process Management

Case: I7

Source: R. Paans and G. Bonnes, ''Surreptitious security violation in the MVS operating system,'' in *Security, IFIP/Sec '83*, V. Fak, ed., North Holland, 1983, pp. 95-105.

System: IBM MVS

Description: Commercial software packages, such as database management systems, often must be installed so that they execute under the Authorized Program Facility. In effect, such programs operate as extensions of the operating system, and the operating system permits them to invoke operations that are forbidden to ordinary programs. The software package is trusted not to use these privileges to violate protection requirements. In some cases, however, (the referenced source cites as examples the Cullinane IDMS database system and some routines supplied by Cambridge Systems Group for servicing Supervisor Call (SVC) interrupts) the package may make operations available to its users that do permit protection to be violated. This problem is similar to the problem of faulty Unix programs that run as SUID programs owned by root (see case U5): there is a class of privileged programs developed and maintained separately from the operating system proper that can subvert operating system protection mechanisms. It is also similar to the general problem of permitting ‘‘trusted applications.’’ It is difficult to point to specific flaws here without examining some particular APF program in detail. Among others, the source cites an SVC provided by a trusted application that permits an address space to be switched from non-APF to APF status; subsequently all code executed from that address space can subvert system protection. We use this example to characterize this kind of flaw.

Genesis: Intentional: Non-Malicious: Other (?) Evidently, the SVC performed this function intentionally, but not for the purpose of subverting system protection, even though it had that effect. Might also be classed as Inadvertent: Domain.

Time: Development: Requirement/Specification/Design (?) (During development of the trusted application)

Place: Support: Privileged Utilities

Case: I8

Source: John Burgess, ‘‘Searching for a better computer shield,’’ *The Washington Post*, Nov. 13, 1988, p. H1.

System: IBM

Description: A disgruntled employee created a number of programs that each month were intended to destroy large portions of data and then copy themselves to other places on the disk. He triggered one such program after being fired from his job, and was later convicted of this act. Although this certainly seems to be an example of a malicious code introduced into a system, it is not clear what, if any, technical flaw led to this violation. It is included here simply to provide one example of a ‘‘time bomb.’’

Genesis: Intentional: Malicious: Logic/Time Bomb

Time: During operation

Place: Application (?)

Case: I9

Source: Schaefer, M., B. Gold, R. Linde, and J. Scheid, ‘‘Program Confinement in KVM/370,’’ *Proc. ACM National Conf.*, Oct., 1977.

System: KVM/370

Description: Because virtual machines shared a common CPU under a round-robin scheduling discipline and had access to a time-of-day clock, it was possible for each virtual machine to detect at what rate it received service from the CPU. One virtual machine could signal another by either relinquishing the CPU immediately or using its full quantum; if the two virtual machines operated at different security levels, information could be passed illicitly in this way. A straightforward, but costly, way to close this channel is to have the scheduler wait until the quantum is expired to dispatch the next process.

Genesis: Intentional: Nonmalicious: Covert timing channel.

Time: During Development: Requirements/Specification/Design. This channel occurs because of a design choice in the scheduler algorithm.

Place: Operating System: Process Management (Scheduling)

Case: MT1

Source: B. Hebbard et al., ‘‘A penetration analysis of the Michigan Terminal System,’’ *ACM SIGOPS Operating System Review 14*, 1 (Jan. 1980) pp. 7-20.

System: Michigan Terminal System

Description: A user could trick system subroutines into changing bits in the system segment that would turn off all protection checking and gain complete control over the system. The flaw was in the parameter checking method used by (several) system subroutines. These subroutines retrieved their parameters via indirect addressing. The subroutine would check that the (indirect) parameter addresses lay within the user’s storage area. If not, the call was rejected; otherwise the subroutine proceeded. However, a user could defeat this check by constructing a parameter that pointed into the parameter list storage area itself. When such a parameter was used by the system subroutine to store returned values, the (previously checked) parameters would be altered, and subsequent use of those parameters (during the same invocation) could cause the system to modify areas (such as system storage) to which the user lacked write permission. The flaw was exploited by finding subroutines that could be made to return at least two controllable values: the first one to modify the address where the second one would be stored, and the second one to alter a sensitive system variable. This is another instance of a time-of-check-to-time-of-use problem.

Genesis: Inadvertent: Validation

Time: During development: Source Code (?) (Without access to design information, we can’t be sure that the parameter checking mechanisms were adequate as designed)

Place: Operating System: Process Management

Case: MT2

Source: B. Hebbard et al., ‘‘A penetration analysis of the Michigan Terminal System,’’ *ACM SIGOPS Operating System Review 14*, 1 (Jan. 1980) pp. 7-20.

System: Michigan Terminal System

Description: A user could direct the operating system to place its data (specifically, addresses for its own subsequent use) in an unprotected location. By altering those addresses, the user could cause the system to modify its sensitive variables later so that the user would gain control of the operating system.

Genesis: Inadvertent: Domain

Time: During development: Requirement/Specification/Design

Place: Operating System: Process Management

Case: MT3

Source: B. Hebbard et al., ''A penetration analysis of the Michigan Terminal System,'' *ACM SIGOPS Operating System Review 14*, 1 (Jan. 1980) pp. 7-20.

System: Michigan Terminal System

Description: Certain sections of memory readable by anyone contained sensitive information including passwords and tape identification. Details of this flaw are not provided in the source cited; possibly this represents a failure to clear shared input/output areas before they were re-used.

Genesis: Inadvertent. Domain (?)

Time: During development: Requirement/Specification/Design (?)

Place: Operating System: Memory Management (possibly also Device Management)

Case: MT4

Source: B. Hebbard et al., ''A penetration analysis of the Michigan Terminal System,'' *ACM SIGOPS Operating System Review 14*, 1 (Jan. 1980) pp. 7-20.

System: Michigan Terminal System

Description: A bug in the MTS supervisor could cause it to loop indefinitely in response to a ''rare'' instruction sequence that a user could issue. Details of the bug are not provided in the source cited.

Genesis: Inadvertent: Boundary Condition Violation

Time: During development: Source Code (?)

Place: Operating System: Other/Unknown

Multics (GE-645 and successors)

The Multics operating system was developed as a general-purpose ''information utility'' and successor to MIT's Compatible Time Sharing System (CTSS) as a supplier of interactive computing services. The initial hardware for the system was the specially designed General Electric GE-645 computer. Subsequently, Honeywell acquired GE's computing division and developed the HIS 6180 and its successors to support Multics. The hardware supported ''master'' mode, in which all instructions were legal, and a ''slave'' mode, in which certain instructions (such as those that modify machine registers that control memory mapping) are prohibited. In addition, the hardware of the HIS 6180 supported eight ''rings'' of protection (implemented by software in the GE-645), to permit greater flexibility in organizing programs according to the privileges they required. Ring 0 was the most privileged ring, and it was expected that only operating system code would execute in ring 0. Multics also included a hierarchical scheme for files and directories similar to that which has become familiar to users of the Unix system, but Multics file structures were integrated with the storage hierarchy, so that files were essentially the same as segments. Segments currently in use were recorded in the Active Segment Table (AST). Denial of service flaws like the ones listed for Multics below could probably be found in many current systems.

Case: MU1

Source: Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

System: Multics

Description: Perhaps because it was designed with interactive use as the primary consideration, Multics initially permitted batch jobs to read card decks into the file system without requiring any user authentication. This made it possible for anyone to insert a file in any user's directory through the batch stream. Since the search path for locating system commands and utility programs normally began with the user's local directories, a Trojan horse version of (for example) a text editor could be inserted and would very likely be executed by the victim, who would be unaware of the change. Such a Trojan horse could simply copy the file to be edited (or change its permissions) before invoking the standard system text editor.

Genesis: Inadvertent: Inadequate Identification/Authentication. According to one of the designers, the initial design actually called for the virtual card deck to be placed in a protected directory, and mail would be sent to the recipient announcing that the file was available for copying into his or her space. Perhaps the implementer found this mechanism too complex and decided to omit the protection. This seems simply to be an error of omission of authentication checks for one mode of system access.

Time: During development: Source Code

Place: Operating System: Identification/Authentication

Case: MU2

Source: Paul A. Karger and R.R. Schell, *Multics Security Evaluation: Vulnerability Analysis,* ESD-TR-74-193, Vol II, June 1974.

System: Multics

Description: When a program executing in a less-privileged ring passes parameters to one executing in a more-privileged ring, the more-privileged program must be sure that its caller has the required read or write access to the parameters before it begins to manipulate those paramenters on the caller's behalf. Since ring-crossing was implemented in software in the GE-645, a routine to perform this kind of argument validation was required. Unfortunately, this program failed to anticipate one of the subtleties of indirect addressing modes available on the Multics hardware, so the argument validation routine could be spoofed.

Genesis: Inadvertent: Validation. Failed to check arguments completely.

Time: During development: Source Code

Place: Operating System: Process Management

Case: MU3

Source: Paul A. Karger and R.R. Schell, *Multics Security Evaluation: Vulnerability Analysis,* ESD-TR-74-193, Vol II, June 1974.

System: Multics

Description: In early designs of Multics, the stack base (sb) register could only be modified in master mode. After Multics was released to users, this restriction was found unacceptable, and changes were made to allow the sb register to be modified in other modes. However, code remained in place that assumed the sb register could only be changed in master mode. It was possible to exploit this flaw and insert a trap door. In effect, the interface between master mode and other modes was changed, but some code that depended on that interface was not updated.

Genesis: Inadvertent: Domain. The characterization of a domain was changed, but code that relied on the former definition was not modified as needed.

Time: During Maintenance: Source Code

Place: Operating System: Process Management

Case: MU4

Source: Paul A. Karger and R.R. Schell, *Multics Security Evaluation: Vulnerability Analysis,* EST-TR-74-193, Vol II, June 1974.

System: Multics

Description: Originally, Multics designers had planned that only processes executing in ring 0 would be permitted to operate in master mode. However, on the GE-645, code for the signaler module, which was responsible for processing faults to be signaled to the user and required master mode privileges, was permitted to run in the user ring for reasons of efficiency. When entered, the signaler checked a parameter, and if the check failed, it transferred, via a linkage register, to a routine intended to bring down the system. However, this transfer was made while executing in master mode and assumed that the linkage register had been set properly. Because the signaler was executing in the user ring, it was possible for a penetrator to set this register to a chosen value and then make an (invalid) call to the signaler. After detecting the invalid call, the signaler would transfer to the location chosen by the penetrator while still in master mode, permitting the penetrator to gain control of the system.

Genesis: Inadvertent: Validation

Time: During development: Requirement/Specification/Design

Place: Operating System: Process Management

Case: MU5

Source: Virgil D. Gligor, ‘‘Some thoughts on denial-of-service problems,’’ University of Maryland, College Park, MD, 16 Sept. 1982.

System: Multics

Description: A problem with the Active Segment Table (AST) in Multics version 18.0 caused the system to crash in certain circumstances. It was required that whenever a segment was active, all directories superior to the segment also be active. If a user created a directory tree deeper than the AST size, the AST would overflow with unremovable entries. This would cause the system to crash.

Genesis: Inadvertent: Boundary Condition Violation: Resource Exhaustion. Apparently programmers omitted a check to determine when the AST size limit was reached.

Time: During development: Source Code

Place: Operating System: Memory Management

Case: MU6

Source: Virgil D. Gligor, ‘‘Some thoughts on denial-of-service problems,’’ University of Maryland, College Park, MD, 16 Sept. 1982.

System: Multics

Description: Because Multics originally imposed a global limit on the total number of login processes, but no other restriction on an individual’s use of login processes, it was possible for a single user to login repeatedly and thereby block logins by other authorized users. A simple (although restrictive) solution to this problem would have been to place a limit on individual logins as well.

Genesis: Inadvertent: Boundary Condition Violation: Resource Exhaustion

Time: During development: Requirement/Specification/Design

Place: Operating System: Process Management

Case: MU7

Source: Virgil D. Gligor, ‘‘Some thoughts on denial-of-service problems,’’ University of Maryland, College Park, MD, 16 Sept. 1982.

System: Multics

Description: In early versions of Multics, if a user generated too much storage in his process directory, an exception was signaled. The flaw was that the signaler used the wrong stack, thereby crashing the system.

Genesis: Inadvertent: Other Exploitable Logic Error

Time: During development: Source Code

Place: Operating System: Process Management

Case: MU8

Source: Virgil D. Gligor, ‘‘Some thoughts on denial-of-service problems,’’ University of Maryland, College Park, MD, 16 Sept. 1982.

System: Multics

Description: In early versions of Multics, if a directory contained an entry for a segment with an all-blank name, the deletion of that directory would cause a system crash. The specific flaw that caused a crash is not known, but, in effect, the system depended on the user to avoid the use of all-blank segment names.

Genesis: Inadvertent: Validation

Time: During development: Source Code

Place: Operating System: File Management (in Multics, segments were equivalent to files)

Case: MU9

Source: Paul A. Karger and R.R. Schell, *Multics Security Evaluation: Vulnerability Analysis,* ESD-TR-74-193, Vol II, June 1974.

System: Multics

Description: A piece of software written to test Multics hardware protection mechanisms (called the Subverter by its authors) found a hardware flaw in the GE-645: if an execute instruction in one segment had as its target an instruction in location zero of a different segment, and the target instruction used index register, but *not* base register modifications, then the target instruction executed with protection checking disabled. By judiciously choosing the target instruction, a user could exploit this flaw to gain control of the machine. When informed of the problem, the hardware vendor found that a field service change to fix another problem in the machine had inadvertently added this flaw. The change that introduced the flaw was in fact installed on all other machines of this type.

Genesis: Inadvertent: Other

Time: During Maintenance: Hardware

Place: Hardware

Burroughs B6700

Burroughs advocated a philosophy in which users of its systems were expected never to write assembly language programs, and the architecture of many Burroughs computers was strongly influenced by the idea that they would primarily execute programs that had been compiled (especially ALGOL programs).

Case: B1

Source: A.L. Wilkinson et al., ‘‘A penetration analysis of a Burroughs large system,’’ *ACM SIGOPS Operating Systems Review 15,* 1 (Jan. 1981) pp. 14-25.

System: Burroughs B6700

Description: The hardware of the Burroughs B6700 controlled memory access according to bounds registers that a program could set for itself. A user who could write programs to set those registers arbitrarily could effectively gain control of the machine. To prevent this, the system implemented a scheme designed to assure that only object programs generated by authorized compilers (which would be sure to include code to set the bounds registers properly) would ever be executed. This scheme required that every file in the system have an associated type. The loader would check the type of each file submitted to it in order to be sure that it was of type ‘‘code-file’’, and this type was only assigned to files produced by authorized compilers. Thus it would be possible for a user to create an arbitrary file (e.g., one that contained malicious object code that reset the bounds registers and assumed control of the machine), but unless its type code were also assigned to be ‘‘code-file’’, it still could not be loaded and executed. Although the normal file-handling routines prevented this, there were utility routines that supported writing files to tape and reading them back into the file system. The flaw occurred in the routines for manipulating tapes: it was possible to modify the type label of a file on tape so that it became ‘‘code-file’’. Once this was accomplished, the file could be retrieved from the tape and executed as a valid program.

Genesis

Intentional: Non-Malicious: Other. System support for tape drives generally requires functions that permit users to write arbitrary bit-patterns on tapes. In this system, providing these functions sabotaged security.

Time: During development: Requirement/Specification/Design

Place: Support: Privileged Utilities

Univac 1108

This large-scale mainframe provided timesharing computing resources to many laboratories and universities in the 1970s. Its main storage was divided into ‘‘banks’’ of some integral multiple of 512 words in length. Programs normally had two banks: an instruction (I-) bank and a data (D-) bank. An I-bank containing a re-entrant program would not be expected to modify itself; a D-bank would be writable. However, hardware storage protection was organized so that a program would either have write permission for both its I-bank and D-bank or neither.

Case: UN1

Source: D. Stryker, ‘‘Subversion of a ‘‘Secure’’ Operating System,’’ NRL Memorandum Report 2821, June, 1974.

System: Univac 1108/Exec 8

Description: The Exec 8 operating system provided a mechanism for users to share re-entrant versions of system utilities, such as editors, compilers, and database systems, that were outside the operating system proper. Such routines were organized as ‘‘Reentrant Processors’’ or REPs. The user would supply data for the REP in his or her own D-bank; all current users of a REP would share a common I-bank for it. Exec 8 also included an error recovery scheme that permitted any program to trap errors (i.e., to regain control when a specified error, such as divide by zero or an out-of-bounds memory reference, occurs). When the designated error-handling program gained control, it would have access to the context in which the error occurred. On gaining control, an operating system call (or a defensively coded REP) would immediately establish its own context for trapping errors. However, many REPs did not do this. So, it was possible for a malicious user to establish an error-handling context, prepare an out-of-bounds D-bank for the victim REP, and invoke the REP, which immediately caused an error. The malicious code regained control at this point with both read and write access to both the REP’s I-and D-banks. It could then alter the REP’s code (e.g., by adding Trojan horse code to copy a subsequent user’s files into a place accessible to the malicious user). This Trojan horse remained effective as long as the modified copy of the REP (which is shared by all users) remained in main storage. Since the REP was supposed to be re-entrant, the modified version would never be written back out to a file, and when the storage occupied by the modified REP was reclaimed, all evidence of it would vanish. The flaws in this case are in the failure of the REP to establish its error handling and in the hardware restriction that I- and D-banks have the same write-protection. These flaws were exploitable because the same copy of the REP was shared by all users. A fix was available that relaxed the hardware restriction.

Genesis: Inadvertent: Domain. It was possible for the user's error-handler to gain access to the REP's domain.

Time: During development: Requirements/Specification/Design

Place: Operating System: Process Management. (Alternatively, this could be viewed as a hardware design flaw.)

DEC PDP-10

The DEC PDP-10 was a medium-scale computer that became the standard supplier of interactive computing facilities for many research laboratories in the 1970s. DEC offered the TOPS-10 operating system for it; the TENEX operating system was developed by Bolt, Beranek, and Newman, Inc. (BBN), to operate in conjunction with a paging box and minor modifications to the PDP-10 processor also developed by BBN.

Case: DT1

Source: Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987, and R.P. Abbott et al, ''Security Analysis and Enhancements of Computer Operating Systems, Final Report of the RISOS Project,'' National Bureau of Standards NBSIR-76-1041, April, 1976 (NTIS PB-257 087), pp. 49-50.

System: TENEX

Description: In TENEX systems, passwords were used to control access to files. By exploiting details of the storage allocation mechanisms and the password-checking algorithm, it was possible to guess the password for a given file. The operating system checked passwords character-by-character, stopping as soon as an incorrect character was encountered. Furthermore, it retrieved the characters to be checked sequentially from storage locations chosen by the user. To guess a password, the user placed a trial password in memory so that the first unknown character of the password occupied the final byte of a page of virtual storage resident in main memory, and the following page of virtual storage was not currently in main memory. In response to an attempt to gain access to the file in question, the operating system would check the password supplied. If the character before the page boundary was incorrect, password checking was terminated before the following page was referenced, and no page fault occurred. But if the character just before the page boundary was correct, the system would attempt to retrieve the next character and a page fault would occur. By checking a system-provided count of the number of page faults this process had incurred just before and again just after the password check, the user could deduce whether or not a page fault had occurred during the check, and, hence, whether or not the guess for the next character of the password was correct. This technique effectively reduces the search space for an *N*-character password over an alphabet of size m from N^m to Nm. The flaw was that the password was checked character-by-character from the user's storage. Its exploitation required that the user also be able to position a string in a known location with respect to a physical page boundary and that a program be able to determine (or discover) which pages are currently in memory.

Genesis: Intentional: Non-Malicious: Covert Storage Channel (could also be classed as Inadvertent: Domain: Exposed Representation)

Time: During development: Source Code

Place: Operating System: Identification/Authentication

Unix

The Unix operating system was originally developed at Bell Laboratories as a ‘‘single user Multics’’ to run on DEC minicomputers (PDP-8 and successors). Because of its original goals—to provide useful, small-scale, interactive computing to a single user in a cooperative laboratory environment—security was not a strong concern in its initial design. Unix includes a hierarchical file system with access controls, including a designated owner for each file, but for a user with userID ‘‘root’’ (also known as the ‘‘superuser’’), access controls are turned off. Unix also supports a feature known as ‘‘setUID’’ or ‘‘SUID’’. If the file from which a program is loaded for execution is marked ‘‘setUID’’, then it will execute with the privileges of the owner of that file, rather than the privileges of the user who invoked the program. Thus a program stored in a file that is owned by ‘‘root’’ and marked ‘‘setUID’’ is highly privileged (such programs are often referred to as being ‘‘setUID to root’’). Several of the flaws reported below occurred because programs that were ‘‘setUID to root’’ failed to include sufficient internal controls to prevent themselves from being exploited by a pene trator. This is not to say that the setUID feature is only of concern when ‘‘root’’ owns the file in question: any user can cause the setUID bit to be set on files he or she creates. A user who permits others to execute the programs in such a file without exercising due caution may have an unpleasant surprise.

Case: U1

Source: K. Thompson, ‘‘Reflections on trusting trust,’’ *Comm ACM 27*, 8 (August, 1984), pp. 761-763.

System: Unix

Description: Ken Thompson’s ACM Turing Award Lecture describes a procedure that uses a virus to install a trapdoor in the Unix login program. The virus is placed in the C compiler and performs two tasks. If it detects that it is compiling a new version of the C compiler, the virus incorporates itself into the object version of the new C compiler. This ensures that the virus propagates to new versions of the C compiler. If the virus determines it is compiling the login program, it adds a trapdoor to the object version of the login program. The object version of the login program then contains a trapdoor that allows a specified password to work for a specific account. Whether this virus was ever actually installed as described has not been revealed. We classify this according to the virus in the compiler; the trapdoor could be counted separately.

Genesis: Intentional: Replicating Trojan horse (virus)

Time: During Development: Object Code

Place: Support: Unprivileged Utilities (compiler)

Case: U2

Source: Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

System: Unix

Description: The ‘‘lpr’’ program is a Unix utility that enters a file to be printed into the appropriate print queue. The -r option to lpr causes the file to be removed once it has been entered into the print queue. In early versions of Unix, the -r option did not adequately check that the user invoking lpr -r had the required permissions to remove the specified file, so it was possible for a user to remove, for instance, the password file and prevent anyone from logging into the system.

Genesis: Inadvertent: Identification and Authentication. Apparently, lpr was a SetUID (SUID) program owned by root (i.e., it executed without access controls) and so was permitted to delete any file in the system. A missing or improper access check probably led to this flaw.

Time: During development: Source Code

Place: Operating System: File Management

Case: U3

Source: Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

System: Unix

Description: In some versions of Unix, ‘‘mkdir’’ was an SUID program owned by root. The creation of a directory required two steps. First, the storage for the directory was allocated with the ‘‘mknod’’ system call. The directory created would be owned by root. The second step of ‘‘mkdir’’ was to change the owner of the newly created directory from ‘‘root’’ to the ID of the user who invoked ‘‘mkdir.’’ Because these two steps were not atomic, it was possible for a user to gain ownership of any file in the system, including the password file. This could be done as follows: the ‘‘mkdir’’ command would be initiated, perhaps as a background process, and would complete the first step, creating the directory, before being suspended. Through another process, the user would then remove the newly created directory before the suspended process could issue the ‘‘chown’’ command and would create a link to the system password file with the same name as the directory just deleted. At this time the original ‘‘mkdir’’ process would resume execution and complete the ‘‘mkdir’’ invocation by issuing the ‘‘chown’’ command. However, this command would now have the effect of changing the owner of the password file to be the user who had invoked ‘‘mkdir.’’ As the owner of the password file, that user could now remove the password for root and gain superuser status.

Genesis: Intentional: Nonmalicious: other. (Might also be classified as Inadvertent: Serialization.) The developer probably realized the need for (and lack of) atomicity in mkdir, but could not find a way to provide this in the version of Unix with which he or she was working. Later versions of Unix (Berkeley Unix) introduced a system call to achieve this.

Time: During development: Source Code

Place: Operating System: File Management. The flaw is really the lack of a needed facility at the system call interface.

Case: U4

Source: A.V. Discolo, ‘‘4.2 BSD Unix security,’’ Computer Science Department, University of California - Santa Barbara, April 26, 1985.

System: Unix

Description: By using the Unix command ‘‘sendmail’’, it was possible to display any file in the system. Sendmail has a -C option that allows the user to specify the configuration file to be used. If lines in the file did not match the required syntax for a configuration file, sendmail displayed the offending lines. Apparently sendmail did not check to see if the user had permission to read the file in question, so to view a file for which he or she did not have permission (unless it had the proper syntax for a configuration file), a user could give simply the command ‘‘sendmail -Cfile_name’’.

Genesis: Inadvertent: Identification and Authentication. The probable cause of this flaw is a missing access check, in combination with the fact that the sendmail program was an SUID program owned by root, and so was allowed to bypass all access checks.

Time: During development: Source Code

Place: Support: Privileged Utilities

Case: U5

Source: M. Bishop, ‘‘Security problems with the UNIX operating system,’’ Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

System: Unix

Description: Improper use of an SUID program and improper setting of permissions on the mail directory led to this flaw, which permitted a user to gain full system privileges. In some versions of Unix, the mail program changed the owner of a mail file to be the recipient of the mail. The flaw was that the mail program did not remove any pre-existing SUID permissions that file had when it changed the owner. Many systems were set up so that the mail directory was writable by all users. Consequently, it was possible for a user X to remove any other user’s mail file. The user X wishing superuser privileges would remove the mail file belonging to root and replace it with a file containing a copy of /bin/csh (the command interpreter or shell). This file would be owned by X, who would then change permissions on the file to make it SUID and executable by all users. X would then send a mail message to root. When the mail message was received, the mail program would place it at the end of root’s current mail file (now containing a copy of /bin/csh and owned by X) and then change the owner of root’s mail file to be root (via Unix command ‘‘chown’’). The change owner command did not, however, alter the permissions of the file, so there now existed an SUID program owned by root that could be executed by any user. User X would then invoke the SUID program in root’s mail file and have all the privileges of superuser.

Genesis: Inadvertent: Identification and Authentication. This flaw is placed here because the programmer failed to check the permissions on the file in relation to the requester’s identity.

Other flaws contribute to this one: having the mail directory writeable by all users is in itself a questionable approach. Blame could also be placed on the developer of the ‘‘chown’’ function. It would seem that it is never a good idea to allow an SUID program to have its owner changed, and when ‘‘chown’’ is applied to an SUID program, many Unix systems now automatically remove all the SUID permissions from the file.

Time: During development: Source Code

Place: Operating System: System Initialization

Case: U6

Source: M. Bishop, ‘‘Security problems with the UNIX operating system,’’ Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

System: Unix (Version 6)

Description: The ‘‘su’’ command in Unix permits a logged-in user to change his or her userID, provided the user can authenticate himself by entering the password for the new userID. In Version 6 Unix, however, if the ‘‘su’’ program could not open the password file it would create a shell with real and effective UID and GID set to those of root, providing the caller with full system privileges. Since Unix also limits the number of files an individual user can have open at one time, ‘‘su’’ could be prevented from opening the password file by running a program that opened files until the user’s limit was reached. By invoking ‘‘su’’ at this point, the user gained root privileges.

Genesis: Intentional: Nonmalicious: Other. The designers of ‘‘su’’ may have considered that if the system were in a state where the password file could not be opened, the best option would be to initiate a highly privileged shell to allow the problem to be fixed. A check of default actions might have uncovered this flaw. When a system fails, it should default to a secure state.

Time: During development: Design

Place: Operating System: Identification/Authentication

Case: U7

Source: M. Bishop, ‘‘Security problems with the UNIX operating system,’’ Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

System: Unix

Description: Uux is a Unix support software program that permits the remote execution of a limited set of Unix programs. The command line to be executed is received by the uux program at the remote system, parsed, checked to see if the commands in the line are in the set uux is permitted to execute, and if so, a new process is spawned (with userID uucp) to execute the commands. Flaws in the parsing of the command line, however, permitted unchecked commands to be executed. Uux effectively read the first word of a command line, checked it, and skipped characters in the input line until a ‘‘;’’, ‘‘^’’, or a ‘‘|’’ was encountered, signifying the end of this command. The first word following the delimiter would then be read and checked, and the process would continue in this way until the end of the command line was reached. Unfortunately, the set of delimiters was incomplete (‘‘&’’

and ‘‘‘’’ were omitted), so a command following one of the ignored delimiters would never be checked for legality. This flaw permitted a user to invoke arbitrary commands on a remote system (as user uucp). For example, the command

uux ‘‘remote_computer!rmail rest_of_command & command2’’

would execute two commands on the remote system, but only the first (rmail) would be checked for legality.

Genesis: Inadvertent: Validation. This flaw seems simply to be an error in the implementation of ‘‘uux’’, although it might be argued that the lack of a standard command line parser in Unix, or the lack of a standard, shared set of command termination delimiters (to which ‘‘uux’’ could have referred) contributed to the flaw.

Time: During development: Requirement/Specification/Design (?) Determining whether this was a specification flaw or a flaw in programming is difficult without examination of the specification (if a specification ever existed) or an interview with the programmer.

Place: Support: Privileged Utilities

Case: U8

Source: M. Bishop, ‘‘Security problems with the UNIX operating system,’’ Computer Science Dept., Purdue University, West Lafayette, Indiana, March 31, 1982.

System: Unix

Description: On many Unix systems it is possible to forge mail. Issuing the following command:
mail user1 <message_file >device_of_user2

creates a message addressed to user1 with contents taken from message_file but with a FROM field containing the login name of the owner of device_of_user2, so user1 will receive a message that is apparently from user2. This flaw is in the code implementing the ‘‘mail’’ program. It uses the Unix ‘‘getlogin’’ system call to determine the sender of the mail message, but in this situation, ‘‘getlogin’’ returns the login name associated with the current standard output device (redefined by this command to be device_of_user2) rather than the login name of the user who invoked the ‘‘mail’’. Although this flaw does not permit a user to violate access controls or gain system privileges, it is a significant security problem if one wishes to rely on the authenticity of Unix mail messages. [Even with this flaw repaired, however, it would be foolhardy to place great trust in the ‘‘from’’ field of an e-mail message, since the Simple Mail Transfer Protocol (SMTP) used to transmit e-mail on the Internet was never intended to be secure against spoofing.]

Genesis: Inadvertent: Other Exploitable Logic Error. This flaw apparently resulted from an incomplete understanding of the interface provided by the ‘‘getlogin’’ function. While ‘‘getlogin’’ functions correctly, the values it provides do not represent the information desired by the caller.

Time: During development: Source Code

Place: Support: Privileged Utilities

Case: U9

Source: Unix Programmer's Manual, Seventh Edition, Vol. 2B, Bell Telephone Laboratories, 1979.

System: Unix

Description: There are resource exhaustion flaws in many parts of Unix that make it possible for one user to deny service to all others. For example, creating a file in Unix requires the creation of an ''i-node'' in the system i-node table. It is straightforward to compose a script that puts the system into a loop creating new files, eventually filling the i-node table, and thereby making it impossible for any other user to create files.

Genesis: Inadvertent: Boundary Condition Violation: Resource Exhaustion (or Intentional: Nonmalicious: Other). This flaw can be attributed to the design philosophy used to develop the Unix system, namely, that its users are benign—they will respect each other and not abuse the system. The lack of resource quotas was a deliberate choice, and so Unix is relatively free of constraints on how users consume resources: a user may create as many directories, files, or other objects as needed. This design decision is the correct one for many environments, but it leaves the system open to abuse where the original assumption does not hold. It is possible to place some restrictions on a user, for example by limiting the amount of storage he or she may use, but this is rarely done in practice.

Time: During development: Requirement/Specification/Design

Place: Operating System: File Management

Case: U10

Source: E. H. Spafford, ''Crisis and Aftermath,'' *Comm. ACM 32*, 6 (June 1989), pp. 678-687.

System: Unix

Description: In many Unix systems the sendmail program was distributed with the debug option enabled, allowing unauthorized users to gain access to the system. A user who opened a connection to the system's sendmail port and invoked the debug option could send messages addressed to a set of commands instead of to a user's mailbox. A judiciously constructed message addressed in this way could cause commands to be executed on the remote system on behalf of an unauthenticated user; ultimately, a Unix shell could be created, circumventing normal login procedures.

Genesis: Intentional: Non-Malicious: Other (? — Malicious, Trapdoor if intentionally left in distribution). This feature was deliberately inserted in the code, presumably as a debugging aid. When it appeared in distributions of the system intended for operational use, it provided a trapdoor. There is some evidence that it reappeared in operational versions after having been noticed and removed at least once.

Time: During development: Requirement/Specification/Design

Place: Support: Privileged Utilities

Case: U11

Source: D. Gwyn, UNIX-WIZARDS Digest, Vol. 6, No. 15, Nov. 10, 1988.

System: Unix

Description: The Unix *chfn* function permits a user to change the full name associated with his or her userID. This information is kept in the password file, so a change in a user's full name entails writing that file. Apparently, *chfn* failed to check the length of the input buffer it received, and merely attempted to re-write it to the appropriate place in the password file. If the buffer was too long, the write to the password file would fail in such a way that a blank line would be inserted in the password file. This line would subsequently be replaced by a line containing only ‘‘::0:0:::’’, which corresponds to a null-named account with no password and root privileges. A penetrator could then log in with a null userID and password and gain root privileges.

Genesis: Inadvertent: Validation

Time: During development: Source Code

Place: Operating System: Identification/Authentication. From one view, this was a flaw in the *chfn* routine that ultimately permitted an unauthorized user to log in. However, the flaw might also be considered to be in the routine that altered the blank line in the password file to one that appeared valid to the login routine. At the highest level, perhaps the flaw is in the lack of a specification that prohibits blank userIDs and null passwords, or in the lack of a proper abstract interface for modifying /etc/passwd.

Case: U12

Source: J. A. Rochlis and M. W. Eichin, ‘‘With microscope and tweezers: the worm from MIT's perspective,’’ *Comm. ACM 32*, 6 (June 1989), pp. 689-699.

System: Unix (4.3BSD on VAX)

Description: The ‘‘fingerd’’ daemon in Unix accepts requests for user information from remote systems. A flaw in this program permitted users to execute code on remote machines, bypassing normal access checking. When fingerd read an input line, it failed to check whether the record returned had overrun the end of the input buffer. Since the input buffer was predictably allocated just prior to the stack frame that held the return address for the calling routine, an input line for fingerd could be constructed so that it overwrote the system stack, permitting the attacker to create a new Unix shell and have it execute commands on his or her behalf. This case represents a (mis-)use of the Unix ‘‘gets’’ function.

Genesis: Inadvertent: Validation.

Time: During development (Source Code)

Place: Support: Privileged Utilities

Case: U13

Source: S. Robertson, Security Distribution List, Vol. 1, No. 14, June 22, 1989.

System: Unix

Description: Rwall is a Unix network utility that allows a user to send a message to all users on a remote system. /etc/utmp is a file that contains a list of all currently logged in users. Rwall uses the information in /etc/utmp on the remote system to determine the users to which the message will be sent, and the proper functioning of some Unix systems requires that all users be permitted to write the file /etc/utmp. In this case, a malicious user can edit the /etc/utmp file on the target system to contain the entry:

../etc/passwd

The user then creates a password file that is to replace the current password file (e.g., so that his or her account will have system privileges). The last step is to issue the command:

rwall hostname < newpasswordfile

The rwall daemon (having root privileges) next reads /etc/utmp to determine which users should receive the message. Since /etc/utmp contains an entry ../etc/passwd, rwalld writes the message (the new password file) to that file as well, overwriting the previous version.

Genesis: Inadvertent: Validation

Time: During development: Requirement/Specification/Design. The flaw occurs because users are allowed to alter a file on which a privileged program relied.

Place: Operating System: System Initialization. This flaw is considered to be in system initialization because proper setting of permissions on /etc/utmp at system initialization can eliminate the problem.

Case: U14

Source: J. Purtilo, *RISKS-FORUM Digest, Vol. 7*, No. 2, June, 2, 1988.

System: Unix (SunOS)

Description: The program *rpc.rexd* is a daemon that accepts requests from remote workstations to execute programs. The flaw occurs in the authentication section of this program, which appears to base its decision on userID (UID) alone. When a request is received, the daemon determines if the request originated from a superuser UID. If so, the request is rejected. Otherwise, the UID is checked to see whether it is valid on this workstation. If it is, the request is processed with the permissions of that
UID. However, if a user has root access to any machine in the network, it is possible for him to create requests that have any arbitrary UID. For example, if a user on computer 1 has a UID of 20, the impersonator on computer 2 becomes root and generates a request with a UID of 20 and sends it to computer 1. When computer 1 receives the request it determines that it is a valid UID and executes the request. The designers seem to have assumed that if a (locally) valid UID accompanies a request, the request came from an authorized user. A stronger authentication scheme would require the user to supply some additional information, such as a password. Alternatively, the scheme could exploit the Unix concept

of ‘‘trusted host.’’ If the host issuing a request is in a list of trusted hosts (maintained by the receiver) then the request would be honored; otherwise it would be rejected.

Genesis: Inadvertent: Identification and Authentication

Time: During development: Requirement/Specification/Design

Place: Support: Privileged Utilities

DEC VAX Computers

DEC’s VAX series of computers can be operated with the VMS operating system or with a UNIX-like system called ULTRIX; both are DEC products. VMS has a system authorization file that records the privileges associated with a userID. A user who can alter this file arbitrarily effectively controls the system. DEC also developed SKVAX, a high-security operating system for the VAX based on the virtual machine monitor approach. Although the results of this effort were never marketed, two hardware-based covert timing channels discovered in the course of its development it have been documented clearly in the literature and are included below.

Case: D1

Source: ‘‘VMS code patch eliminates security breach,’’ *Digital Review*, June 1, 1987, p. 3

System: DEC VMS

Description: This flaw is of particular interest because the system in which it occurred was a new release of a system that had previously been closely scrutinized for security flaws. The new release added system calls that were intended to permit authorized users to modify the system authorization file. To determine whether the caller has permission to modify the system authorization file, that file must itself be consulted. Consequently, when one of these system calls was invoked, it would open the system authorization file and determine whether the user was authorized to perform the requested operation. If the user was not authorized to perform the requested operation, the call would return with an error message. The flaw was that when certain second parameters were provided with the system call, the error message was returned, but the system authorization file was inadvertently left open. It was then possible for a knowledgeable (but unauthorized) user to alter the system authorization file and eventually gain control of the entire machine.

Genesis: Inadvertent: Domain: Residuals. In the case described, the access to the authorization file represents a residual.

Time: During Maintenance: Source Code

Place: Operating System: Identification/Authentication

Case: D2

Source: W-M, Hu, ‘‘Reducing Timing Channels with Fuzzy Time,’’ *Proc. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, 1991, pp. 8-20.

System: SKVAX

Description: When several CPUs share a common bus, bus demands from one CPU can block those of others. If each CPU also has access to a clock of any kind, it can detect whether its requests have been delayed or immediately satisfied. In the case of the SKVAX, this interference permitted a process executing on a virtual machine at one security level to send information to a process executing on a different virtual machine, potentially executing at a lower security level. The cited source describes a technique developed and applied to limit this kind of channel.

Genesis: Intentional: Nonmalicious: Covert timing channel

Time: During development: Requirement/Specification/Design. This flaw arises because of a hardware design decision.

Place: Hardware

Intel 80386/80387 Processor/CoProcessor Set

Case: IN1

Source: ''EE's tools & toys,'' *IEEE Spectrum, 25*, 8 (Aug. 1988), pp. 42.

System: All systems using Intel 80386 processor and 80387 coprocessor.

Description: It was reported that systems using the 80386 processor and 80387 coprocessor may halt if the 80387 coprocessor sends a certain signal to the 80386 processor when the 80386 processor is in paging mode. This seems to be a hardware or firmware flaw that can cause denial of service. The cited reference does not provide details as to how the flaw could be evoked from software. It is included here simply as an example of a hardware flaw in a widely marketed commercial system.

Genesis: Inadvertent: Other Exploitable Logic Error(?)

Time: During development: Requirement/Specification/Design (?)

Place: Hardware

Personal Computers: IBM PCs and Compatibles, Apple Macintosh, Amiga, and Atari

This class of computers poses an interesting classification problem: can a computer be said to have a security flaw if it has no security policy? Most personal computers, as delivered, do not restrict (or even identify) the individuals who use them. Therefore, there is no way to distinguish an authorized user from an unauthorized one or to discriminate an authorized access request by a program from an unauthorized one. In some respects, a personal computer that is always used by the same individual is like a single user's domain within a conventional time-shared interactive system: within that domain, the user may invoke programs as desired. Each program a user invokes can use the full privileges of that user to read, modify, or delete data within that domain.

Nevertheless, it seems to us that even if personal computers don't have explicit security policies, they do have implicit ones. Users normally expect certain properties of their machines—for example, that running a new piece of commercially produced software should not cause all of one's files to be deleted.

For this reason, we include a few examples of viruses and Trojan horses that exploit the weaknesses of IBM PCs, their non-IBM equivalents, Apple Macintoshes, Atari computers, and Commodore Amiga. The fundamental flaw in all of these systems is the fact that the operating system, application packages, and user-provided software user programs inhabit the same protection domain and therefore have the same privileges and information available to them. Thus, if a user-written program goes astray, either accidentally or maliciously, it may not be possible for the operating system to protect itself or other programs and data in the system from the consequences. Effective attempts to remedy this situation generally require hardware modifications, and some such modifications have been marketed. In addition, software packages capable of detecting the presence of certain kinds of malicious software are marketed as ''virus detection/prevention'' mechanisms. Such software can never provide complete protection in such an environment, but it can be effective against some specific threats.

The fact that PCs normally provide only a single protection domain (so that all instructions are available to all programs) is probably attributable to the lack of hardware support for multiple domains in early PCs, to the culture that led to the production of PCs, and to the environments in which they were intended to be used. Today, the processors of many, if not most, PCs could support multiple domains, but frequently the software (perhaps for reasons of compatibility with older versions) doesn't exploit the hardware mechanisms that are available.

When powered up, a typical PC (e.g., running MS-DOS) loads (''boots'') its operating system from pre-defined sectors on a disk (either floppy or hard). In many of the cases listed below, the malicious code strives to alter these boot sectors so that it is automatically activated each time the system is re-booted; this gives it the opportunity to survey the status of the system and decide whether or not to execute a particular malicious act. A typical malicious act that such code could execute would be to destroy a file allocation table, which will delete the filenames and pointers to the data they contained (although the data in the files may actually remain intact). Alternatively, the code might initiate an operation to reformat a disk; in this case, not only the file structures, but also the data, are likely to be lost.

MS-DOS files have two-part names: a filename (usually limited to eight characters) and an extension (limited to three characters), which is normally used to indicate the type of the file. For example, files containing executable code typically have names like ''MYPROG.EXE''. The basic MS-DOS command interpreter is normally kept in a file named COMMAND.COM. A Trojan horse may try to install itself in this file or in files that contain executables for common MS-DOS commands, since it may then be invoked by an unwary user. (See case MU1 for an related attack on Multics).

Readers should understand that it is very difficult to be certain of the complete behavior of malicious code. In most of the cases listed below, the author of the malicious code has not been identified, and the nature of that code has been determined by others who have (for example) read the object code or attempted to ''disassemble'' it. Thus the accuracy and completeness of these descriptions cannot be guaranteed.

IBM PCs and Compatibles

Case: PC1

Source: D. Richardson, *RISKS FORUM Digest*, Vol. 4, No. 48, 18 Feb. 1987.

System: IBM PC or compatible

Description: A modified version of a word-processing program (PC-WRITE, version 2.71) was found to contain a Trojan horse after having been circulated to a number of users. The modified version contained a Trojan horse that both destroyed the file allocation table of a user's hard disk and initiated a low-level format, destroying the data on the hard disk.

Genesis: Malicious: Non-Replicating Trojan horse

Time: During operation

Place: Support: Privileged Utilities

Case: PC2

Source: E. J. Joyce, ''Software viruses: PC-health enemy number one,'' *Datamation*, Cahners Publishing Co., Newton, MA, 15 Oct. 1988, pp. 27-30.

System: IBM PC or compatible

Description: This virus places itself in the stack space of the file COMMAND.COM. If an infected disk is booted, and then a command such as TYPE, COPY, DIR, etc., is issued, the virus will gain control. It checks to see if the other disk contains a COMMAND.COM file, and if so, it copies itself to it and a counter on the infected disk is incremented. When the counter equals four every disk in the PC is erased. The boot tracks and the File Access Tables are nulled.

Genesis: Malicious: Replicating Trojan horse (virus)

Time: During operation.

Place: Operating System: System Initialization

Case: PC3

Source: D. Malpass, *RISKS FORUM Digest*, Vol. 1, No. 2, 28 Aug., 1985.

System: IBM-PC or compatible

Description: This Trojan horse program was described as a program to enhance the graphics of IBM programs. In fact, it destroyed data on the user's disks and then printed the message ''Arf! Arf! Got You!''.

Genesis: Malicious: Nonreplicating Trojan horse

Time: During operation

Place: Support: Privileged Utilities (?)

Case: PC4

Source: Y.Radai, *Info-IBM PC Digest*, Vol. 7, No. 8, 8 Feb., 1988, also *ACM SIGSOFT Software Engineering Notes, 13*, 2 (Apr. 1988), pp.13-14

System: IBM-PC or compatible

Description: The so-called ''Israeli'' virus, infects both COM and EXE files. When an infected file is executed for the first time, the virus inserts its code into memory so that when interrupt 21h occurs the virus will be activated. Upon activation, the virus checks the currently running COM or EXE file. If the file has not been infected, the virus copies itself into the currently running program. Once the virus is in memory it does one of two things: it may slow down execution of the programs on the system or, if the date it obtains from the system is Friday the 13th, it is supposed to delete any COM or EXE file that is executed on that date.

Genesis: Malicious: Replicating Trojan horse (virus)

Time: During operation

Place: Operating System: System Initialization

Apple Macintosh

An Apple Macintosh application presents quite a different user interface from from that of a typical MS-DOS application on a PC, but the Macintosh and its operating system share the primary vulnerabilities of a PC running MS-DOS. Every Macintosh file has two ‘‘forks’’: a data fork and a resource fork, although this fact is invisible to most users. Each resource fork has a type (in effect, a name) and an identification number. An application that occupies a given file can store auxiliary information, such as the icon associated with the file, menus it uses, error messages it generates, etc., in resources of appropriate types within the resource fork of the application file. The object code for the application itself will reside in resources within the file’s resource fork. The Macintosh operating system provides utility routines that permit programs to create, remove, or modify resources. Thus any program that runs on the Macintosh is capable of creating new resources and applications or altering existing ones, just as a program running under MS-DOS can create, remove, or alter existing files. When a Macintosh is powered up or rebooted, its initialization may differ from MS-DOS initialization in detail, but not in kind, and the Macintosh is vulnerable to malicious modifications of the routines called during initialization.

Case: MA1

Source: B. R. Tizes, ‘‘Beware the Trojan bearing gifts,’’ *MacGuide Magazine 1,* (1988) Denver, CO, pp. 110-114.

System: Macintosh

Description: NEWAPP.STK, a Macintosh program posted on a commercial bulletin board, was found to include a virus. The program modifies the System program located on the disk to include an INIT called ‘‘DR.’’ If another system is booted with the infected disk, the new system will also be infected. The virus is activated when the date of the system is March 2, 1988. On that date the virus will print out the the following message:
‘‘RICHARD BRANDOW, publisher of MacMag, and its entire staff would like to take this opportunity to convey their UNIVERSAL MESSAGE OF PEACE to all Macintosh users around the world.’’

Genesis: Malicious: Replicating Trojan horse (virus)

Time: During operation

Place: Operating System: System Initialization

Case: MA2

Source: S. Stefanac, ‘‘Mad Macs,’’ *Macworld 5,* 11 (Nov. 1988), PCW Communications, San Francisco, CA, pp. 93-101.

System: Macintosh

Description: The Macintosh virus, commonly called ‘‘scores’’, seems to attack application programs with VULT or ERIC resources. Once infected, the scores virus stays dormant for several days and then begins to affect programs with VULT or ERIC resources, causing attempts to write to the disk to fail. Signs of infection by this virus include an extra CODE resource of size 7026, the existence of two invisible files titled Desktop and Scores in the system folder, and added resources in the Note Pad file and Scrapbook file.

Genesis: Malicious: Replicating Trojan horse (virus)

Time: During operation

Place: Operating System: System Initialization (?)

Commodore Amiga

Case: CA1

Source: B. Koester, *RISKS FORUM Digest*, Vol. 5, No. 71, 7 Dec. 1987; also *ACM SIGSOFT Software Engineering Notes, 13*, 1 (Jan. 1988), pp. 11-12.

System: Amiga personal computer

Description: This Amiga virus uses the boot block to propagate itself. When the Amiga is first booted from an infected disk, the virus is copied into memory. The virus initiates the warm start routine. Instead of performing the normal warm start, the virus code is activated. When a warm start occurs, the virus code checks to determine if the disk in drive 0 is infected. If not, the virus copies itself into the boot block of that disk. If a certain number of disks have been infected, a message is printed revealing the infection; otherwise the normal warm start occurs.

Genesis: Malicious: Replicating Trojan horse (virus)

Time: During operation

Place: Operating System: System Initialization

Atari

Case: AT1

Source: J. Jainschigg, ‘‘Unlocking the secrets of computer viruses,’’ *Atari Explorer 8*, 5 (Oct. 1988), pp. 28-35.

System: Atari

Description: This Atari virus infects the boot block of floppy disks. When the system is booted from a infected floppy disk, the virus is copied from the boot block into memory. It attaches itself to the function *getbpd* so that every time *getbpd* is called the virus is executed. When executed, the virus first checks to see if the disk in drive A is infected. If not, the virus copies itself from memory onto the boot sector of the uninfected disk and initializes a counter. If the disk is already infected the counter is incremented. When the counter reaches a certain value the root directory and file access tables for the disk are overwritten, making the disk unusable.

Genesis: Malicious: Replicating Trojan horse (virus)

Time: During operation

Place: Operating System: System Initialization