Operating system penetration

by RICHARD R. LINDE
System Development Corporation
Santa Monica, California

INTRODUCTION

One of the favorite diversions of university students involves "beating" the system. In the case of operating systems, this has been a remarkably easy accomplishment. An extensive lore of operating system penetration, ranging from anecdotes describing students who have outsmarted the teacher's grading program to students who captured the system's password list and posted it on one of the bulletin boards,[1] has been collected on college campuses. Private industry has been victimized much more seriously. Here the lore of "system" penetrations contains scenarios involving the loss of tens of thousands of dollars.[2]

The Research and Development organization at SDC has been seriously involved with legitimate operating system penetration efforts. Under contract to government agencies and industry, SDC has assessed the secure-worthiness of their systems by attempts to gain illegal access to their operating systems. As of this date, seven operating systems have been studied. This paper examines the successful penetration methodology employed, and the generic operating system functional weaknesses that have been found. Recommendations are made for improvement that can strengthen the penetration methodology.

THE SDC FLAW HYPOTHESIS METHODOLOGY

In the absence of more formal correctness proof techniques,[3] penetrations are the most cost effective method for assessing vulnerabilities. Exhaustive testing of an operating system's security controls is different than subjecting these controls to a penetration attack. System testing exercises are used to examine a system for implementation errors; whereas, penetration tests are used to examine an implementation, and from these analyses infer areas of possible design weakness.

Peterson and Turn have defined comprehensive system attack strategies, and have proposed a set of countermeasures.[4] This paper focuses more on software attack strategies with respect to generic operating system weaknesses than on countermeasures. In this case, countermeasures entail good system design practices. SDC has formalized a strategy, the Flaw Hypothesis Methodology, based on our experience with operating system penetrations.[5]

Comprehensive flaw finding requires four stages: knowledge of the system control structure; the generation of an inventory of suspected flaws; i.e., "flaw hypotheses"; confirmation of the hypotheses; and making generalizations regarding the underlying system weakness for which the flaw represents a specific instance.

Knowledge of system control structure

Knowledge of the system control structure is an obvious prerequisite to the penetration effort. It is necessary for the penetration analysts to understand how users interact with the system, what services are provided to them, and what constraints are placed on them. In order to gain this familiarity, penetration analysts must read the system manuals pertaining to command language, system debugging, editing, operator's instructions, terminal user's instructions, and the introductory manuals pertaining to system overview and generation. Hence, the penetration analysts are able to list the security objects—i.e., file data, password lists, disk volumes—that are protected by system control objects—i.e., the file cataloger, installation management techniques, and label checking of disk volumes, respectively. Sometimes an object may be both a security object to be protected and a control object that protects other objects. For example, a password list may be viewed as a control object when it is protecting access to files, or it may be viewed as a security object protected recursively by itself, by installation management techniques guaranteeing password integrity, by password changeability, etc.

Security weaknesses found in operating systems contribute to the formulation of a control object hierarchy for a hypothetically "secure" system. This hierarchy of control objects can be represented in graphical form; graphical templates have been produced for each functional area of the hypothetical system (see Figure 1; the arrows between each node reflect a logical dependency). As more "live" systems are studied, each template is changed to reflect more "adequate" controls for the hypothetical system. The templates are applied to the "live" system under analysis by using them to produce graphs for the system. Flaws are postulated based upon control objects that may or may not be present. "Templates" or sequences of faulty generic

code derived from past system studies are used as search parameters when perusing system listings. Cross reference programs for searching a data base of system symbols would be an excellent analytic tool at this phase of the methodology.[6] Conceptually, the hierarchical dependency graph is a representation standing between a protection matrix model[7] and a flow chart of the operating system. In order to understand an operating system well enough to comprehensively penetrate it, one must view the operating system from an abstract level as well as from an implementation level, e.g., the following levels:

- Inter-Module Design: Schematics of operating system modules and layers.[8,9]
- Access Control Mechanism: For example, set theoretic descriptions of the access matrix.[10]
- Control Object Hierarchy: Hierarchical dependency graphs (see Figure 1).
- Intra-Module Design: System flow charts and Logic manuals.
- Implementation: Symbolic listings of system code.

A penetration consists of an interloper capturing a control object, sometimes starting with a low-valued object on the hierarchical dependency graph and working his way up to complete omniscience—the supervisor bit (i.e., supervisor state). Graphs with badly designed control objects, such as faulty I/O control design decisions, are classified as

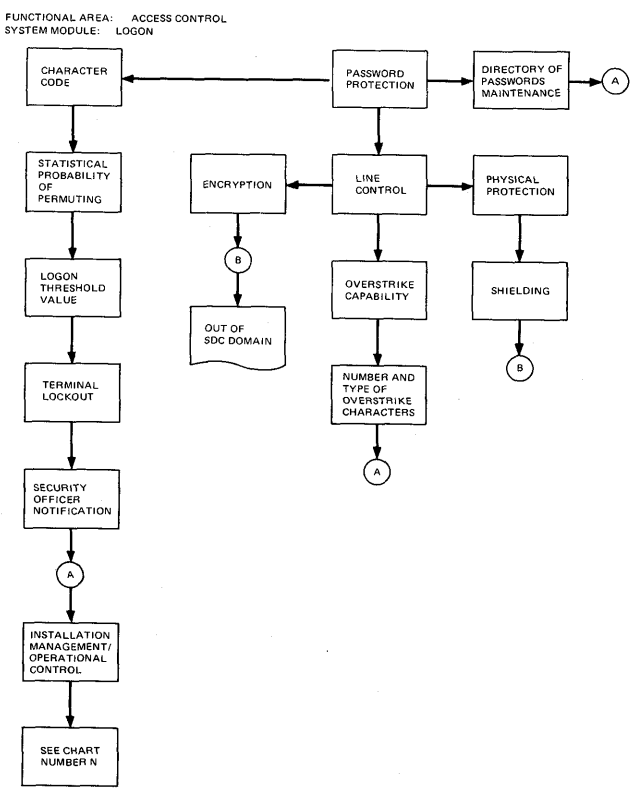

Figure 1—Security Control Object Dependency Graph: A Generic Template to be Applied to the System Under Investigation

TABLE I—Flaw Hypotheses Generators

FLAW HYPOTHESES GENERATORS
HISTORICAL GENERIC SYSTEM WEAKNESSES (SEE APPENDIX A)
SYSTEM PROHIBITIONS AND WARNINGS Timing Dependencies
INTERFACES Man-Man (Operator Messages) Man-System (Commands)
SELDOM USED OR UNUSUAL FUNCTIONS OR COMMANDS Read Backward
CONTROL OBJECT DEPENDENCY GRAPH TEMPLATES
HISTORICAL ATTACK STRATEGIES (See Appendix B)
SYSTEM LISTINGS, LOGIC MANUALS, USERS' GUIDES
COLLECTION OF USER AND SYSTEM PROGRAMMER EXPERIENCES WITH THE SYSTEM UNDER ANALYSIS

generic functional weaknesses when they can be penetrated at many nodes.

Flaw hypothesis generation

The security control object dependency graph is a useful heuristic in that it gives the penetration analysts a visual representation of potential operating system attack points. Many of the flaw hypotheses that are generated result from such graphs. Once a hypothesis is formulated, it is written onto a form that contains a cross reference to the security control object dependency, a priority of investigation based upon the nature of the object and the probability of exploiting it, an attack strategy, and an evaluation summary. System code is also studied and used to generate flaw hypotheses. Specific areas of the operating system are analyzed from different points of view, and kinds of possible security weaknesses are generated for each area. The result is the generation of the flaw hypothesis form. Table I contains a list of flaw hypotheses generators.

Flaw hypothesis confirmation

"Gedanken" (thought) experiments and desk-checking are employed, using existing documentation, program logic manuals, and symbolic listings of the system, to determine the validity of a flaw hypothesis. The Gedanken experi-

ment is the most important phase of the penetration study. This is attested to by the fact that most uncovered flaws are found by "thought" testing.

The intent of live tests is to ascertain that a flaw exists and a penetration is possible. A penetration program involves a minimum amount of thought once the flaw is proved. It does require large, straightforward programs to perform input/output, timing, initialization, set-up, etc. (programs written to prove flaws because thought testing has proven inconclusive, can be quite complex, however). Sometimes programs are produced to perform a permutation of all possible operation codes in an attempt to uncover instructions not contained in the System's Principles of Operation. This is an example of where a penetration tool, i.e., a program, can be produced to look for potential flaws, testing for flaws in the absence of the appropriate documentation.

Flaw generalization

It is important for a penetration team to analyze an operating system's security strengths and weaknesses thoroughly, and to delineate the generic weaknesses of the system. In order to describe these weaknesses, the penetration analysts devise a categorization based on families of errors for the study. When one functional area begins to yield a few penetrations, the investigators meet to discuss the nature of the penetration attempts, and to explore other areas of the system, e.g., access control in lieu of I/O control.

GENERIC OPERATING SYSTEM FUNCTIONAL WEAKNESSES: WHICH MODULES TO STUDY FIRST?

Security Control design and implementation weaknesses occurring repeatedly in computer systems can be grouped into a common class based on functionality. This may be a software, hardware, or physical function that is being provided. If it is a software function, the protection mechanism associated with it is usually distributed through more than one layer of system control, and the function itself is generally divided into multiple modules. Often each module is produced by different groups of people. For example, the generic function, physical resource sharing, may be distributed among the Scheduler, Cataloger, Dispatcher, Input/Output Supervisor, Terminal Read/Write Controller, etc. Each module may be part of a different control layer, implementing both security policy and protection mechanism decisions in the system. Systems designed in such a manner are vulnerable to a number of penetration attacks since security design decisions related to a common function have been distributed among several system building teams.

The following describes a set of common operating system functional weaknesses based upon the seven systems that have been studied:

I/O control

Most operating systems allow input/output channel programming in some form. This is a prime target area for potential system penetrators. The complexities of I/O code alone can be a significant factor. Logical errors, inconsistencies, and omissions can often be found by detailed examination of the I/O code. Even where checked it may still be possible for a channel program to modify itself, loop back, and then execute the newly modified code. Also, I/O channels usually act as independent processors and because of this may have unlimited access to memory. The dangers here are that critical code in the system could be modified in an unauthorized manner. Additionally, if an infinite loop can be executed, service to the other users can be intentionally degraded or stopped. This, of course, can be a very dangerous security threat. If a system can be selectively overloaded at critical (real) time periods, the loss of command and control could be severe. This would especially be true in tactical systems, air traffic control systems, or a Navy sea plot environment.

Program and data sharing

Typically operating systems attempt to isolate a user's process and its data from other users' processes and their data. This involves the protection of a number of security objects: hardware resources, files, password lists, operating system routines, etc. However, since dedicated machines or dedicated classified periods of processing are not cost effective to provide computing resources and software services in a completely isolable manner to a community of users, some form of multiprogramming with controlled sharing must take place. These sharing mechanisms involve program and data sharing as well as physical resource sharing, and because of faulty operating system design and implementation techniques, they are prime penetration targets. Incongruously, the operating system must isolate its users from one another yet provide them with controlled communication paths.

Access control

In order to permit sharing of resources, data, and programs, the users must be identified and authenticated by the operating system and the operating system must be identified by the user. To permit this "handshake protocol," a Send/Receive relationship is established between the user and the operating system. Failure to handle this interface communication function correctly provides the opportunity for a user process to masquerade as an operating system process, and for a user to obtain password information through "piece-wise decomposition" and "permutation" type programs (see Appendix B). Less sophisticated but effective penetration techniques involve permuting easy-to-guess passwords, operator spoofs, browsing for poor password overstrike capability, etc.

Installation management/operational control

In a temporal sense, operating system controls are only effective during the period of system execution and must be augmented by static procedural controls such as:

directory maintenance
password assignment
system generation parameters
system changes
building of software utilities
system checkout
enforcing system prohibitions
period processing requirements
disk and tape degaussing, labeling, inventory

Because control of this function is apart from the operating system in a temporal and physical sense, it is particularly vulnerable to interdiction and corruption of system code by planting trap doors and Trojan Horse routines. Personnel inefficiency and inadequate procedural controls account for this susceptibility.

Auditing and surveillance

There is a need to insure that system security controls are working properly. An audit mechanism can be built to record all security transactions, such as file OPEN requests, LOGON requests, resource expenditures, and other security related events. Surveillance can take place dynamically within the operating system through the monitoring of well-designed thresholds. Also, "friendly" programs can be built to periodically assault the system's security controls to insure that they are working properly.[10,11,12] Unfortunately, many systems lack a comprehensive audit and surveillance capability, employing an accounting mechanism instead.

Externally, a security officer introduces a human decision-making capability into the surveillance mechanism. The security officer's function can include the ability to log-off suspicious users and to monitor the security safeguards to insure they are functioning as intended.[11,12] Penetrator entrapment (also called counter-intelligence) can be used with respect to the security officer. Bogus listings with "magic passwords" inside can be used to trap a penetrator using those passwords.

In part, the value of audit and surveillance mechanisms must be weighed against the skill of the penetrator. Clearly, the more skilled penetrator will disable these mechanisms first in order to work undetected. These mechanisms work best against the casual attack by the less skilled penetrator who confronts the system protection matrix attempting to acquire "booty."

As system protection mechanisms are improved, the "penetration work factor"—the amount of effort and resources expended to gain unauthorized access to data, procedure, or machine resources—could become so great that other methods, such as buying off an employee with access to the information will be less expensive.[13] Audit and surveillance mechanisms increase the "penetration work factor" for the skilled penetrator. Penetration studies may be used to quantify this work factor.

Non-software weaknesses

The security aspects of hardware and physical constraints on the computer system have received less attention in our analysis of operating systems than have the software and administrative protection mechanisms. A list of hardware and physical threats have been compiled and have been reported by other studies.[14,15,16]

GENERIC SYSTEM FLAWS

Operating system functional areas are vulnerable to attack strategies because they contain specific flaws. Flaws that repeatedly occur in Computer Systems may be grouped into common classes. These flaws are described more fully in Appendix A. Because of these weaknesses, the capture of design control objects (i.e., a password list) is possible. Conceptually, generic flaws may exist at the nodes of the generic functional dependency graph templates (see Figure 1).

GENERIC ATTACKS

A generic attack is a hostile action that is found to have been repeatedly successful in penetrating operating systems. Typically a security control object (i.e., the storage protection bit) is attacked and captured because an operating system design or implementation mechanism is flawed in a specific functional area. The primary attack may lead to secondary attacks aimed at higher security control objects until a security object (i.e., Top Secret File) is captured.

ATTACK SCENARIOS

In order to illustrate the relationship between generic functional flaws and their proneness to an Appendix B penetration attack, a few representative attack scenarios are presented. One or more of the generic system flaws described in Appendix A provide a focal point for each attack:

Input/output control

- In a so-called chained command, the data channel interprets the address and count fields but discards the operation code and performs the operation defined in the predecessor command. On the hardware, any bit pattern can be formed in the ignored field. Place a branch in such a program attempting to reuse the formerly chained command in an unchained mode.[17]

Program and data sharing

- Transfer a copy of one's own EDIT to the private files belonging to User A. If private files are searched before public files, pseudo-EDIT will be loaded for User A.[18]
- Take advantage of the system's failure to validate user-supplied addresses to where a store or fetch is to be executed, compromising the system's and user's address spaces. For example, the location for storing return parameters from a service call may not be validated.
- A borrowed service routine sends data from the address space of the borrower to the address space of the service routine's owner, using the system's interprocess communication facilities.
- If the system has interlocks that prevent files from being opened for reading and writing at the same time, the service can leak data if it is allowed to read files which can be written by its owner. The interlocks allow a file to simulate a shared Boolean variable which one program can set and the other can test.[19] Or eight files can be used simultaneously for ASCII encoding of short character messages.
- Place a program in a loop that continually requests spooling or terminal read/write buffers.
- Although memory is cleared prior to use, the system fails to clear the state vector general registers until a program has been loaded. Execute a dump routine prior to loading a program and browse for residue.

Access control

- Have one's LOGON routine masquerading as the system's at a hard wired terminal. Simulate a system crash after capturing the password.
- Divert operator's attention with a MOUNT message. Imbed multiple line feeds in a message buffer followed by the system prefix character with a simulated system message comprising the text.

Installation management/operational control

- Although User's Guides, etc., warn against operating self-modifying channel programs, the installation fails to restrict their use. Write a self-modifying channel program and penetrate address boundaries.

Audit and surveillance

- One "secure" system places too much reliance on audit and surveillance not realizing it will be the first mechanism disabled by the interloper. Disable the mechanism and proceed in undetected fashion with another penetration attack.

Physical/hardware

- Tap a minicomputer into a communications line to masquerade as a legitimate central system. This is a method used to acquire user ID's and passwords.

SUMMARY AND OBSERVATIONS

SDC's operating system penetration methodology consists of:

(1) Knowledge of System Control Structure
(2) Flaw Hypothesis Generation
(3) Flaw Hypothesis Confirmation
(4) Flaw Generalization

This methodology is useful in providing a formal strategy for penetrating an operating system, as well as for isolating generic system functional flaws that can be used for determining functional areas of operating system design that need strengthening.* Seven systems were penetrated during the study and the results of these penetrations were used to strengthen this methodology and to isolate specific areas of operating system weakness.

One of the last operating systems that was studied had a high exposure rate (65 percent) with respect to flaw hypotheses generated, indicating the effectiveness of the penetration method and team. I/O control and Installation Management/Operational Control were the system functional targets for the most penetrations; "Unexpected Parameters" and "Denial of Access Programming" were the most frequently used penetration attack strategies.

The penetration methodology will be strengthened by collecting, analyzing, and classifying more contemporary system functional design weaknesses, by developing analytic tools for locating security control objects in system listings,[6] and by producing more permutation type programs. The next class of tools that need to be developed are programs that search for security flaws in the symbolic listings of the operating system.[6,22] The construction of such programs will be guided by collecting more generic code templates or signatures common to the generic functional weaknesses of contemporary operating systems.

ACKNOWLEDGMENT

The author wishes to acknowledge the valuable editorial assistance provided by Clark Weissman, who originated the idea for the Flaw Hypothesis Methodology, and to Dr. R. Stockton Gaines for his most helpful critique. It is next to impossible to provide proper attribution for a specific

* A contemporary solution to the design of secure operating systems exists with respect to secure kernels and virtual machine monitor architecture studies.[17,18,20,21]

attack; however, I would like to acknowledge the fact that Richard Bisbee, Dan Edwards, and several of the Multics people, whom I have not referenced herein, should be given credit for devising a number of the penetration attacks. To all others whom I have not referenced or acknowledged, please forgive me.

REFERENCES

1. Organick, Elliott I., *The MULTICS System: An Examination of Its Structure*, the MIT Press, Cambridge, Massachusetts, 1972.
2. Palme, Jacob, "Software Security," *Datamation*, Vol. 20, No. 1, January 1974, pp. 51-55.
3. Linden, T. A., "A Summary of Progress Toward Proving Program Correctness," *AFIPS Conference Proceedings*, 1972 Fall Joint Computer Conference, Vol. 41, pp. 201-211.
4. Peterson, H. E., and R. Turn, "System Implications of Information Privacy," *Spring Joint Computer Conference Proceedings*, 1967, AFIPS Press, pp. 291-300.
5. Weissman, C., *System Security Analysis/Certification Methodology and Results*, SDC SP-3728, 8 October 1973.
6. Webb, Douglas A., *Analytic Tools for the Examination of Security Aspects of Operating Systems*, Lawrence Livermore Laboratory, UCRL-76016, September 1974.
7. Lampson, B. W., "Protection," *Proceedings Fifth Annual Princeton Conference on Information Sciences and Systems*, Department of Electrical Engineering. Princeton University, Princeton, New Jersey, March 1971, pp. 437-443.
8. Dijkstra, E. W., "The Structure of THE—Multiprogramming System," *Comm. ACM*, II, 5, May 1968, pp. 341-346.
9. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.
10. Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," *Fall Joint Computer Conference Proceedings*, 1969, pp. 119-134.
11. Linde, R. R., C. Weissman, and C. E. Fox, "The ADEPT-50 Time-Sharing System," *Fall Joint Computer Conference Proceedings*, 1969, pp. 39-50, AFIPS Press.
12. Linde, R. R., "Operational Management of Time-Sharing Systems, *Proceedings 1966 National ACM Conference*, pp. 149-159.
13. Lackey, R. D., "Penetration of Computer Systems–An Overview," *Honeywell Computer Journal*, pp. 81-85, September 1974, Vol. 8, No. 2.
14. Molho, Lee M., "Hardware Aspects of Secure Computing," *Spring Joint Computer Conference Proceedings*, 1970, AFIPS Press.
15. Van Tassel, Dennis, *Computer Security Management*, Prentice-Hall, Inc., 1972.
16. Hollingworth, D., *Enhancing Computer System Security*, P-5064, The Rand Corporation, August 1973.
17. Belady, L. A., and C. Weissman, *Experiments with Secure Resource Sharing for Virtual Machines*, SDC SP-3769, 15 May 1974.
18. Popek, G. J., and C. S. Kline, "Verifiable Secure Operating System Software," *AFIPS Conference Proceedings*, 1974, National Computer Conference, Vol. 43, pp. 145-151.
19. Lampson, Butler W., "A Note on the Confinement Problem," *Comm. ACM*, Vol. 16, No. 10, October 1973, pp. 613-615.
20. Weissman, C., *Secure Computer Operation with Virtual Machine Partitioning*, SDC SP-3790, 6 September 1974.
21. Attanasio, C. R., "Virtual Machines and Data Security," *Proceedings ACM SIGOPS/SIGARCM Workshop on Virtual Computer Systems*, Cambridge, Massachusetts, 1973.
22. Hollingworth, D., S. Glaseman, M. Hopwood, *"Security Test and Evaluation Tools: An Approach to Operating System Security Analysis*, P-5298, The Rand Corporation, September 1974.
23. Brinch Hansen, Per, "The Nucleus of a Multiprogramming System," *Comm. ACM*, 13, 4, April 1970, pp. 238-241.
24. Branstad, Dennis K., "Privacy and Protection in Operating Systems," *IEEE Computer*, January 1973, pp. 43-46.

APPENDIX A—GENERIC SYSTEM FUNCTIONAL FLAWS

- Authentication—It is important for the user to be able to authenticate that the operating system and hardware he is executing on is what they purport to be. Little attention has been paid to the software identification of software modules. This becomes more important in a shared segmented system because of the ease with which a module can be substituted or replaced. Systemwide schemes are lacking, whereby major hardware components (i.e., storage controllers, processors, peripheral controllers, communication controllers, and remote terminals), can be identified by each other.
- Documentation—Security documentation may be written in a complex manner or be deficient in several areas.
- Encryption—Communication lines must be encrypted if shielding is inadequate. Passwords stored in memory are not encrypted in many systems.
- Error Detection—Protection mechanisms may be disabled or modified as a result of an error, and after subsequent return to the routine causing the error, they may not be reset.
- Implementation—All the well thought out design requirements may be reversed by a bad implementation. Condition codes may be tested improperly, etc. More importantly, implementation in many systems means that more definitive design decisions are to be made by the implementor.
- Implicit Trust—Routine B assumes routine A's parameters are correct because Routine A is a system process.
- Implied Sharing—The system stores its data or references user parameters in the user's address space because memory is in critical demand.
- Interprocess Communication—To facilitate sharing, users are permitted to request hardware and software resources from the operating system using a SEND/RECEIVE type mechanism.[23] Various return conditions (password OK, segment error, illegal parameter, etc.) may give the penetrator "significant" information, especially when this design weakness is combined with others such as Implied Sharing.
- Legality Checking—User parameters may not be checked adequately; addresses may overlap each other or refer to system areas; condition code checks may be omitted; and unusual or extraordinary parameters may not be anticipated by the designer or implementor.
- Line Disconnect—The operating system hardware must make the system cognizant of line disconnects prompting an automatic logout or a RELOG request (i.e., RELOG with correct password).
- Modularity—Many systems were designed so that different groups of people were responsible for the design of various modules embodying common functions at the same system level (e.g., channel command

translation and execution; spooling simulation). Thus, when design changes were necessary, they were made by different, isolated system designers.[9]

- Operator Carelessness—Operators may be tricked into mounting bogus operating system packs for the penetrator. Many systems do not adequately perform label checks and provide correct warning and error messages to the operator.
- Parameter Passing by Reference vs. Passing by Value—It's much safer to pass parameters in general registers than to use the registers to point to the parameter's location. Passing by Reference can lead to an Implied Sharing design weakness through careless implementation since the parameters may not be moved out of the user address space before legality checking occurs.
- Passwords—Passwords for operating system, terminal, and file access that are supplied by users may be easy to guess. Passwords that are made up of a limited set of characters or syntax rules (e.g., digits 0 to 9 only or A to Z only) are subject to permutation attempts.
- Penetrator Entrapment—This design practice sometimes called Counter Intelligence is useful for the capture of the unskilled penetrator. Generally, pseudo-system weaknesses are implemented in the system code to bait the would-be interloper; they rely on the premise that more subtle recording/parameter checking techniques (the trap) will escape his attention. Many systems have inadequate penetrator entrapment and audit and surveillance mechanisms; others employ none of these mechanisms.
- Personnel Inefficiency—The operational management of operating systems (storage volume control, password assignment, virtual memory allocation, etc.) is dependent upon the care and competence of the people operating and maintaining the system. Redundancy checks, etc., to check the accuracy of the operational manager are lacking in most systems.
- Privity—This is a potential design weakness in many systems. Too many system programs and subsystems are given supervisor privileges to facilitate access to certain system tables. The more modules that operate in a supervisor state, the greater the chance for a serious penetration. Moreover, the hardware is inadequate, lacking descriptor based hardware, multiple execution states, and ring structures.[1]
- Program Confinement—Borrowed programs can act as Trojan Horses in that the borrower's files may be pilfered and his programs altered. Alternately, the borrower may copy a proprietary program for his own use. Lampson[19] has shown that borrowed programs may leak information through subtle "communication" channels.
- Prohibitions—System precautionary measures or prohibitions contained in User's Guides and other documentation may lead to system penetrations if the personnel responsible for operational management have not been fully cognizant of the prohibition's potential. For the penetrator, this weakness can become an easy exposure vehicle since the potential weaknesses are readily advertised by the manufacturer in the form of "Prohibitions", etc.
- Residue—Residue refers to unauthorized information that has been made available to the penetrator through poor housekeeping practices and system design carelessness. Trash baskets may be searched for log-on password or user identification. Trash also yields lists, notes, discarded teleprinter ribbons or platens, and data on paper, magnetic tapes, or disc packs which can be searched for sensitive information. In short, anything not erased or overwritten before being discarded is of potential use to a penetrator.
- Magnetic tape, disc space, and core residue can often be easily read and searched for sensitive information; temporary files and buffers are the most common sources.
- Security Design Omissions—If the security design is so complex, that many design decisions must be left to the implementor, the chances for a security design omission in the form of inadequate legality checking are enhanced.
- Shielding—In the absence of encryption techniques, computer rooms are shielded to counteract electromagnetic pickup.
- Threshold Values—Many internal surveillance checks in the form of threshold values leading to the entrapment of a penetrator are not present in many systems. For example, after a user has attempted to use a password that has failed N times, he should be locked from the system and the security officer should be notified.[12]
- Use of Test and Set—Many system designers were not cognizant in their design of an asynchronous attack potential by another processor. Hence, critical region parameters can be checked, then changed prior to the systems use by a concurrent process.
- Utilities—The operational management and control of system utilities is often neglected by the system manager. This area is fraught with Trojan Horse attack potential. Frequently, utility designers are not aware of security design issues. For example, inadequate boundary checking at compile time may allow a user to execute machine code disguised as data in a data area.

APPENDIX B—GENERIC OPERATING SYSTEM ATTACKS

- Asynchronous—Multiple processes overlapping each other's execution; one process attempts to change the parameters that the other process has legally checked but has not yet used.
- Browsing—A search by an authorized/unauthorized user for privileged or classified information in the computer system.
- Between Lines—To employ a special terminal tapped into the communication channels to affect "between

lines" entry to the system when a legitimate user is inactive but still holds the communications channel.

- Clandestine Code—The submission of "patch" code containing trap doors used to repair an operating system or utility program error, allowing the repairman to subsequently enter the computer system in an unauthorized fashion.
- Denial of Access—A program written to usurp a preponderant share of the system's physical resources or written to cause a system crash, the ultimate goal being the denial of legitimate users access to the computer system's resources in a timely manner.
- Error Inducement—A program written to cause errors within itself so that its subsequent state vector may contain information altered by the systems error recovery routine.
- Interacting Synchronized Processes—Processes that synchronize their execution through the system's synchronization primitives, sharing and passing information to each other through a common data base or subtle communication channel.[19]
- Line Disconnect—The attempt to gain access to another user's job subsequent to his disconnect but prior to hardware or software acknowledgment.
- Masquerade—To assume the identity of a legitimate user or process after having obtained proper identification through wiretapping or other means.
- "NAK" attack—This probabilistic attack exercises operating system weaknesses for not properly handling user generated asynchronous interrupts. The name comes from a user interrupt being commonly generated with the teletypewriter NAK (Negative Acknowledge) key. A system is often designed so that a user can interrupt a process, perform an operation, and then either return to continue the process or begin another. Poor designs often leave the system in an unprotected state during these times: partially written files left open, improper writing of a protection infraction message, etc. All possible situations of asynchronous interrupts are very difficult to analyze.[24]
- Operator Spoof—The attempt to circumvent installation management procedures by "spoofing" the computer system operator into an action that compromises the security of the system.
- Permutation Programming—A program written to process the total number of changes in position or order possible within a group. For example, to search for "new and useful" operation code.
- Piecewise Decomposition—A technique for cracking password and other Secret character strings by decomposing the string into its constituent characters and testing each in turn.
- Piggy Back—To employ a special terminal tapped into a communication channel to effect "piggy back" entry into the system by selective interception of communications between a user and the processor, and then releasing these with modifications or substituting entirely new messages while returning an "error" message.
- Trojan Horse—This term—first coined by Dan Edwards—refers to planting an entry point or "trap door" in the computer system to allow subsequent unauthorized access to the system. The Trojan Horse may be planted on a temporary or permanent basis. Trojan Horse also refers to any unexpected and malicious side effect, i.e., a program which executes a desired function correctly, but has illegitimate side effects.
- Unexpected Operations—To invoke seldom used system primitives or macros in an unusual manner taking advantage of the system's design and implementation weaknesses.
- Unexpected Parameters—To submit unusual or illegal parameters in a supervisor call attempting to circumvent or capitalize on the system's legality checking procedures.
- Wire Tapping—To cut in one or tap a communication channel to intercept a message.