A PENETRATION ANALYSIS OF THE MICHIGAN TERMINAL SYSTEM

B. Hebbard, P. Grosso, T. Baldridge, C. Chan,
D. Fishman, P. Goshgarian, T. Hilton, J. Hoshen,
K. Hoult, G. Huntley, M. Stolarchuk, L. Warner

Abstract

The successful penetration testing of a major time-sharing operating system is described. The educational value of such a project is stressed, and principles of methodology and team organization are discussed as well as the technical conclusions from the study.

Introduction

Increasing attention is being paid to the problems of computer security and to the security of operating systems in particular. Pending the development of new design concepts (such as the "security kernel"), penetration testing has been recognized as a cost-effective tool for improving operating system security[5]. This paper describes the systematic penetration analysis of a major university computer system in an effort to gain insight into some of the related concepts and problems.

This project was undertaken by a graduate-level computer science class at the University of Michigan in Ann Arbor. It was done at the invitation and with the full cooperation of the University of Michigan Computing Center, which is interested in assessing and improving the security of its operating system, the Michigan Terminal System (MTS). The study was aimed at the software aspects of security and was not concerned with physical entry, penetration techniques requiring human intervention or "spoofing," or the compromising of administrative procedures. The emphasis was on techniques that a knowledgeable and generally authorized but non-privileged user could employ to obtain unauthorized services. The specific goals of the penetration attempts were to gain read or write access to files to which the user was not granted access, and, secondarily, to cause a system crash or employ other denial-of-access techniques.

Current versions of MTS are designed to be secure from penetration by users even if they have full access to all system documentation and listings of the operating system code. For this reason, the Computing Center staff provided all such information to the analysis team and offered to answer any reasonable questions pertaining to system design or operation.

Background

The Michigan Terminal System is a general-purpose operating system for the IBM 360 and 370 and compatible machines. Since its introduction at the University of Michigan in 1966, MTS has been under continuous development there and, in recent years, at the several other universities where it now serves as the primary operating system. As its name suggests, MTS is conversationally oriented, but it supports batch use as well through a command language common to both modes. MTS users have a number of convenient features at their disposal, including an elaborate file system with extensive and flexible sharing and access control facilities. The user's view and some internal description are given in Boettner & Alexander[1].

During the penetration study at the University of Michigan, MTS ran on a six-megabyte Amdahl 470V/6 there. This system serves the diverse research and educational computing needs of the university community and related government agencies, while administrative computing is done at a separate facility. During a Fall or Winter semester, approximately 25,000 user accounts have authorized access to the system and its extensive software library. On a typical afternoon, the system simultaneously supports over 250 terminal users, as well as a number of batch streams from remote campus stations.

The students comprising the analysis team were enrolled in a course formally known as Advanced System Programming (Computer and Communication Sciences 673), which seeks to involve students in a large single group project. Another recent session of the course, for example, produced the prototype for the interactive course registration system (CRISP) now familiar to and used by nearly all University of Michigan students[3]. The students generally control most aspects of the course through an internal management structure, while faculty members participate as advisors and resource persons. The background and level of the students vary somewhat, but all have the common experience of an operating system design course.

Preparation

Before meaningful penetration testing can begin, the team must have a thorough understanding of the system. Depending on the objectives and corresponding ground rules of the study, this knowledge may be at either the user (external) or system (internal) level. The members of this team each had the advantage of several years' experience as users of MTS and had access to all user-level documentation and manuals. Since system documentation was made available to the team as well, the meeting time during the first few weeks of the project was devoted entirely to learning as much as possible about the internal structure of MTS. Members of the Computing Center staff gave lectures on their respective areas of responsibility, explaining in some detail the internal structure of various components, pointing out sources of more detailed documentation, and giving some introduction to the mountains of assembly-code listings. After these orientation sessions, the twelve-member project team was divided into three specialized working groups and one student project manager.

Comprehensiveness

Comprehensiveness is one of the most important and yet one of the most elusive goals of a penetration study. While it is never possible to assure formal completeness, i.e., that all or even a given fraction of actual security flaws have been found, it is possible and necessary to strive for a comprehensive study by covering all parts of the system with balanced attention. This was a major concern of this project.

Communication among the groups was a vital ingredient in pursuit of this goal. Coordination of the entire project was needed to assure that no system component was overlooked and, on the other hand, to avoid wasteful duplication of effort by two or more groups in cases of overlapping or interrelated software domains. In addition to the management committee, weekly team meetings and smaller informal meetings provided these channels. Project documentation was also considered essential. Written reports of suspected flaws, whether eventually confirmed or not, were kept to provide a central record of areas that had been checked and to serve as a history of the project.

Methodology

Despite a familiarity with operating system concepts, the students had no formal experience in penetration techniques or system security concepts as such. As a basis for project methodology, a set of guidelines based on past penetration studies was adopted. This so-called "Flaw Hypothesis Methodology" has been developed by System Development Corporation as a result of its penetration testing of its clients' computer systems. For a complete description, the reader is referred to Linde[5] and Weissman[8]. Briefly, the methodology centers on a three-stage cycle for locating security flaws to be applied within a specified framework of goals, rules, and limits. In the first stage (flaw hypothesis generation), team members draw upon a number of sources (flaw hypothesis generators) and postulate the existence of flaws in the system's security. Suggested flaw hypothesis generators include: historical system weaknesses, unusual or even prohibited commands or functions, and the team's collective experience with the system under analysis. In the second stage (flaw hypothesis confirmation), the suspected flaws are filtered through a series of tests to determine whether or not they actually exist. A suspected flaw is first subjected to simple mental testing, desk-checked against listings and documentation, and, if necessary, put to a "live" machine test. In the third stage (flaw generalization), confirmed flaws are studied to determine to what extent they represent greater design weaknesses. That is, a confirmed flaw may in fact be only one instance of a generic flaw; if this is so, security will not be appreciably enhanced simply by correcting the specific instance. Rather, the resulting family of flaws itself becomes a new flaw hypothesis generator, closing a loop in the Flaw Hypothesis Methodology and assuring a more complete study of the system's overall strengths and weaknesses.

Technical Background

The SDC papers stress the need to identify the control structure of the system. They assert that a dependency hierarchy of control objects can be constructed, making an important tool for penetration analysis. Some knowledge of the internal structure of MTS is necessary to appreciate how this applies here. For a more complete description, the reader is referred to Pirkola & Sanguinetti[7].

Whether in terminal or batch mode, each user job has associated with it exactly one task which represents an activation of certain reentrant programs within the operating system. The virtual memory space of each of these tasks consists of several shared segments, which are common to all such tasks, and a number of private segments which are only accessible to a given task (figure 1). The first of these private segments, called the system work segment, is used by the reentrant system programs to record information specific to that task, such as accounting information, user privilege level, etc. The remaining private segments are user storage areas which are at the disposal of user programs.

These various areas of virtual memory are protected from unauthorized modification, both accidental and deliberate, in

Figure 1. An individual task's virtual memory space in MTS.

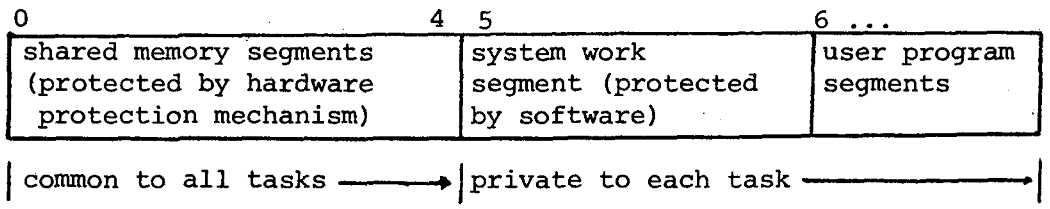

0
4
5
6 ...
shared memory segments (protected by hardware protection mechanism)
system work segment (protected by software)
user program segments
common to all tasks
private to each task

Figure 2. A control dependency diagram for MTS.

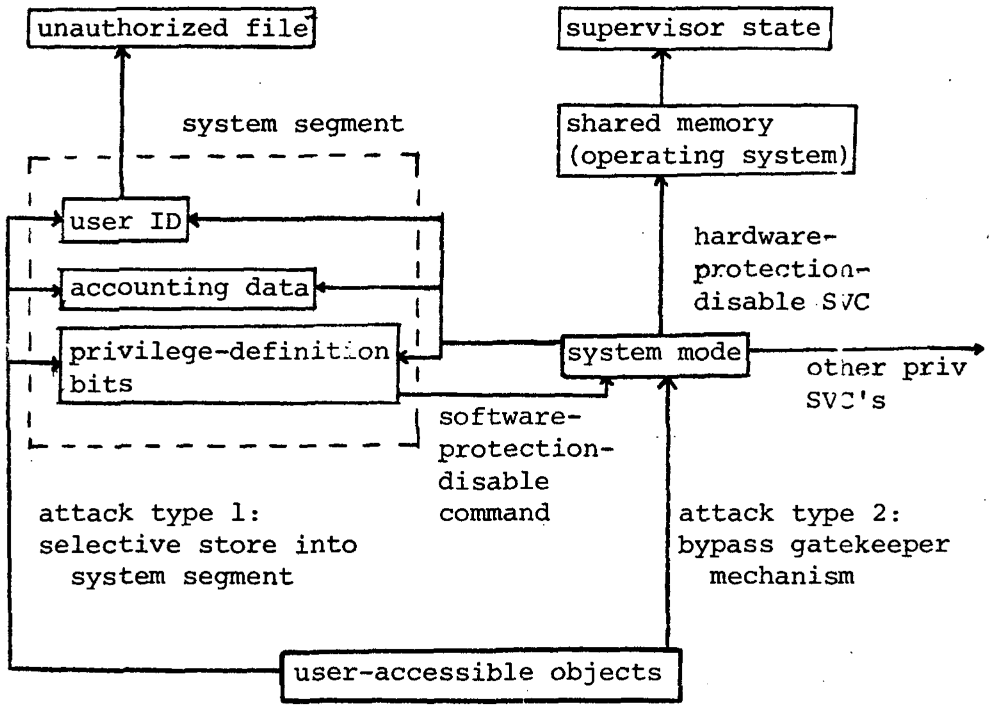
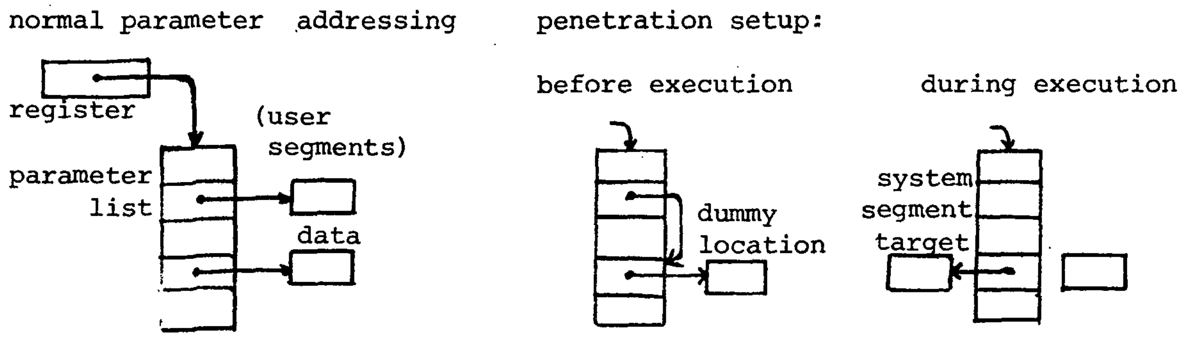

unauthorized file
supervisor state
system segment
shared memory (operating system)
user ID
accounting data
privilege-definition bits
hardware-protection-disable SVC
system mode
other priv SVC's
software-protection-disable command
attack type 1: selective store into system segment
attack type 2: bypass gatekeeper mechanism
user-accessible objects

Figure 3. Parameter-list alteration flaw.



different ways. The shared segments contain the supervisor and system programs and are not normally subject to task modification, so these segments are protected using the hardware storage-protection mechanism. Any task is prevented from accessing other tasks' private segments by the dynamic address translation mapping, under which they are simply not addressible. The system work segment, however, must be available to the task when executing system routines and cannot economically be protected using the regular hardware storage-protection mechanism. But a user should be prevented from tampering with the control information contained there.

Until recently, the system segment was protected only by secrecy, as little or none of the internal documentation was available to the general user. Clearly this was unacceptable, particularly in university communities where knowledgeable people abound and casual computer penetration has become something of a recreational activity. To deal with the problem of safeguarding the system segment and the need to restrict certain supervisor calls to trusted system routines, a software "virtual memory protection system" was developed at Wayne State University (an MTS installation)[7]. This protection system defines two task states termed "system" and "user" modes which form a software analogue of the supervisor and problem hardware states. System mode is the privileged task state; in this mode the task has full read/write access to its own system segment and it may issue almost any defined supervisor call. User mode is the restricted state under which the system segment is removed from the task's virtual address space, making it completely inaccessible. Furthermore, supervisor calls which could potentially affect system integrity are disallowed in user mode.

The task normally executes in system mode but is put into user mode whenever user-supplied code may be executing. Since user programs may call system subroutines which in turn may need to issue privileged supervisor calls or access the system segment, transitions are also provided for calls to these routines. Since the mode switching requires changing the address translation tables, the supervisor performs it via two supervisor calls defined for this purpose. The user-to-system supervisor call is necessarily non-privileged, but it is accepted only when issued from within a pre-defined address range, called a gate, in the shared segments. When a user program calls a system subroutine, control is first passed to the gatekeeper routines which perform the transition to system mode. A return from the subroutine again passes through the gate mechanism, which effects the transition back to user mode before returning to the user's program. This assures that only trusted system routines may execute in system mode. Those subroutines must also perform parameter checking so that they are not tricked by the user into performing unauthorized activity while in system mode. Adding to the protection scheme's complexity are facilities which allow certain privileged programs to be run entirely in system mode.

Initial Progress

Each group turned to its assigned set of software components, studying them in detail and attempting to apply the Flaw Hypothesis Methodology. Considering the immense size of the system, there was some doubt that a team with limited internal knowledge would be able to find flaws that were not evident to those who work on the system regularly, especially given the time and resource constraints of a one-semester course. Nevertheless, each group gained familiarity with its assigned components and probed for flaws, frequently running small test programs under controlled conditions to confirm or deny flaw hypotheses.

Control Structure

At an early stage in the project, a flaw was discovered in the parameter checking of certain system subroutines that would allow a user to trick them into storing arbitrary bit strings into the system segment (see Example below). At first the full significance of this was not realized. As the project continued, however, this ability was found to be sufficient to give a penetrator complete control over the system (figure 2). Since the system segment contains all of the variable task information (except for low-level scheduling information of interest to the supervisor), a user able to change his own system segment at will could alter his accounting data or elevate himself to higher privilege levels. The protection system maintains its own control information in the system segment and thus in a sense protects itself. If a penetrator could manage to change even one particular bit in the system segment, he could then turn off the protection system completely. User programs could then run in system mode with complete access to the system segment and to privileged supervisor calls which permit control over various system functions. In particular, one of these supervisor calls unconditionally turns off hardware storage protection, allowing modification of the shared memory segments. Since the supervisor code resides there, this implies that the penetrator could take over the machine in supervisor state. This is an important theoretical goal -- the "checkmate" of a penetration attempt since it represents complete omnipotence. Thus a chain of "control dependency" had been established: a knowledgeable penetrator, once able to tamper with the system segment, would be able to topple all security structures and assume unlimited system control.

In fact, once the system segment is vulnerable, most penetration goals could be easily reached through more direct methods. A penetrator in control of the system segment could assume the identity of any other user by an appropriate change to the user identity code contained there. This would allow access to the user's files, among other things. In this way, all barriers to file access could be bypassed without disturbing the file routines at all. Also, MTS contains privileged commands for use by system programmer accounts, which permit remote entry of operator console commands and editing of the system code itself, among other things. Altering the appropriate privilege indicators in the system segment would put these facilities into the hands of a penetrator.

It thus became dramatically apparent that the system segment was central to the security of the system. Once this dependency had been recognized and confirmed, "vertical" testing was no longer necessary and attention was shifted to a "horizontal" search for equivalent flaws. Achieving controllable write access to the system segment or an equivalent capability was thereafter considered a clear demonstration of a total penetration.

As work progressed on various parts of the system, a study of control dependency often indicated that only limited penetration progress could be hoped for without first assuming system-mode privileges. But if these could be attained, the whole system was known to be vulnerable. This all-or-nothing aspect of penetration attempts allowed them to be split into two classes: "total" penetrations which attempted to defeat the main security, and other attempts which assumed these barriers to be intact. Work continued on both of these fronts.

This distinction helped to simplify later work. Any line of attack predicated on privileges beyond the security wall could be safely disregarded. For example, I/O code has historically been a fertile ground for penetrators since it is complex and offers the potential for unlimited memory access. However, MTS services user I/O requests only through system subroutines and shields the user from channel program construction. In order to be able to affect channel programs directly, the user would have to escape memory protection and hence would already have achieved a total penetration. Therefore I/O code was not given intensive study, a fact which allowed many tedious hours to be saved for more productive efforts. There are many similar examples, since

MTS generally provides user services indirectly and isolates the user from internal operations.

As the common importance of the main security structure became evident, the original software assignments of the groups became less and less important. While continuing direct work on the assigned components, all of the groups also worked on ways to defeat the main security. The most promising targets were the safeguards on the system segment and the transitions between system and user modes, since these are at the bottom of the dependency hierarchy (i.e., are closest to the user). This concentrated effort resulted in a number of successful total penetrations.

Example

A specific example of a critical flaw may help to illustrate the type of effort involved. The flaw described here is typical of those found and may be applicable to other systems.

As noted before, all user-callable system subroutines which operate in system mode must perform whatever parameter checking is necessary to ensure that a user program cannot obtain services beyond those legally available to it. The case in which parameters are communicated via an indirect addressing scheme (S-type convention) makes this checking difficult. Here, the calling program passes the address of a parameter list in a register. This variable-length list contains the addresses for retrieval or storage of the parameters rather than the values themselves. This two-level indirection poses problems with regard to the verification of addresses. At the time of the penetration study, most such routines initially referred to the parameter list and simply checked each address to see that it lay within the user-storage areas. If so, execution was allowed to proceed normally.

At first this scheme might appear to work. However, since the parameter list (that is, the list of addresses) is supplied by the user, it lies within user storage and may itself be the target of a returned parameter. That is, one entry in the parameter list may point to another one, and if it specifies the location for storage of a returned parameter, the address may be modified during execution. In this case the static legality checking may be ineffective since a penetrator could instruct the subroutine to modify the parameters after they are checked.

A simple test program was sufficient to confirm the existence of this flaw. Proving its significance required the use of a routine which used the indirect calling convention, returned at least two parameters, and allowed sufficient flexibility to enable selection of particular return values. A number of routines were found to meet these criteria. In particular, one routine to read input lines returns not only the text of the line but also an associated line number and a line length integer indicating the number of characters in the input line (figure 3).

In order to store a particular value into the system segment, a small program was set up along with a file containing a dummy line to be read by the input routine. The parameter list was constructed so that the address for return of the line number actually pointed to another entry on the list, namely, the location of the address for return of the line length. This address was originally set to a dummy value located within user storage. When called, the subroutine checked all parameter addresses on the list, and, finding them to be in user areas, allowed execution to proceed. After reading the input line, the routine stored the line number in the indicated location within the parameter list itself. The line, having been constructed by the user, was set up so that the line number was equal to the target address in the system segment. When the routine subsequently went to store the line length, it did so using the just-modified address now pointing within the system segment. Naturally, the line length had been set to the desired value to be planted there.

This flaw was found to apply to a number of subroutines in essentially the same form. On a more general level, it illustrates the problem of synchronization of legality checking of parameters and their use.

Results

All successful total penetrations resulted from two forms of attacks on the protection system. Attacks of the first type caused routines which legitimately run in system mode to modify the system segment in an unauthorized manner. Flaws which permitted this type of attack fell into two groups: 1) the type just described, i.e., faulty static parameter checking on several system-mode subroutines, and 2) design oversights in which the system could be made to store its data in an unprotected segment where the user could modify it. In this case the data were addresses for future storage references, and these could be changed to cause storage into the system segment.

Attacks of the second type exploited weaknesses in the system/user mode transitions to cause routines to branch directly to user-supplied code while still in system mode. Flaws which permitted this type of attack were also of the "implied sharing"[5] type in which the system stored its data -- in this case internal branch addresses -- in user storage.

Only limited effects could be obtained in the "non-total" category. The file system, in particular, withstood all high-level attacks directed at it. A number of ways were found to obtain free computer time, usually by forcing a user job into a highly abnormal termination, thereby preventing a normal accounting update. Certain areas of shared memory to which all users have read access were found to contain potentially sensitive information, such as passwords, tape identification, etc. In the denial-of-access category, one flaw found was a bug which caused the supervisor to loop indefinitely in response to a rare instruction sequence, effectively crashing the system.

Conclusions

The goal of computer security has been expressed in terms of raising the work factor to the point of deterring a would-be penetrator by making the expected cost of a penetration greater than the potential gain. A penetration analysis, properly done, provides an actual test of this work factor. However imprecise and subjective, this can give a valid assessment of a system's overall security and its areas of strengths and weaknesses. It is, of course, also helpful to locate specific flaws so that these can be repaired. In acting as penetrators, the analysts may catch flaws hidden from the view of the system designers. In this role they are likely to find those flaws that would be most obvious to actual penetrators, which in a sense are the most dangerous flaws.

The viewpoint of a penetrator also allows a unique look at the major problems of operating system security, which have been described elsewhere. A large operating system frequently depends on a number of security structures, each of which assumes the others will work correctly. This interdependence, as represented by the control dependency relationships, means that a flaw in any such component may render the others useless. And with the complexity of large operating systems, the existence of such flaws becomes a virtual certainty. In the case of MTS this is particularly dramatic because of a number of "implicit trust"[5] relationships where routines running with some system privileges are assumed to be trusted, and as such are allowed ever-increasing levels of privilege. This means that the security of the whole system hinges on the lowest security layer, which leads to the idea of "total" penetrations discussed earlier.

It was the opinion of the project team that identifying these control dependency relationships was the most important part of the study, both during preparation and as an ongoing part of the search. This allows the team to infer the probable significance of a potential flaw. In order to reduce an otherwise immense work load to manageable proportions, it is necessary to distinguish between those places which are relevant to security and those which can be safely disregarded so that limited resources can be focused on critical areas. A thorough understanding of the system was found to be far more useful than attempts at exhaustive studies of code.

In retrospect, we achieved initial familiarity with this control structure without deliberately attempting to do so. But it was evidenced by an increase in team efficiency: flaws were found at an increasing rate until time had run out. The team also showed increasing confidence in discussing the system's overall security and there was a feeling that the team could attack another system with much improved efficiency, knowing better what to look for and where and how to find it.

The division of the team into small working groups was a practical necessity. The idea of specialization by software component was useful during the early stages of the project. The intensive study of a single part of the system allowed each group to become expert in its respective area, and each could then serve as a resource for other groups. Future studies should allow flexibility and cooperation among groups, since weak spots often tend to be located in the interfaces between components rather than in the modules themselves. Furthermore, a subsequent reorganization based on the control structure should be encouraged. As noted before, the software assignments of our groups became less important as the implications of the MTS control hierarchy emerged.

Cooperation and mutual trust between the installation and the analysis team are essential for a productive search. The team appreciated the initial invitation of the University of Michigan Computing Center and its ongoing cooperation and assistance. Project members, in return, were trusted to act in a professional manner and to share flaw information only with the rest of the team and the Computing Center. One of the discovered flaws did leak, which caused us to be more careful for the remainder of the project. This underscores the need for security in the project itself.

In considering any of the potential weaknesses in MTS, it is important to note that probably no large operating system using current design technology can withstand a determined and well-coordinated attack, and that most such documented penetrations have been remarkably easy[9]. In view of the fact that only a few critical flaws were found after such an intensive effort, it appears that MTS compares favorably with similar systems in this respect. No significant security flaws were found in the high-level structures (e.g., command processor, file routines) and only someone with an intimate knowledge of the system's internal (low-level) structure and storage locations would be able to exploit a "total" penetration, even if another such flaw could be found. Considering the long-range futility of "hole-patching," resources may now be better spent on penetration detection and recovery rather than on prevention. With these considerations in mind, it appears that MTS affords its users excellent security, which has been further enhanced as a result of this study.

Acknowledgements

The project team appreciated the cooperation and assistance of the University of Michigan Computing Center and its staff. We would also like to thank professors Bernard A. Galler and Larry K. Flanigan for their guidance during the course and encouragement during the writing of this paper. Although all facts about MTS were as correct as possible at the time of writing, they may no longer reflect the current state of the system. In particular, all known flaws have now been removed. All inaccuracies in the text remain the responsibilities of the authors.

References

1 Boettner, Donald W. and Michael T. Alexander. 1975. The Michigan Terminal System. Proceedings of the IEEE, Vol 63, No. 6.

2 Dinardo, C. T., ed. 1978. Computers and Security. AFIPS Press, New Jersey.

3 Galler, B. A., R. Wagman, J. Bravatto, G. Lift, G. Kem, V. Berstis, and E. Munn. 1973. CRISP: An Interactive Student Registration System. Proceedings 1973, ACM Annual Computer Conference. p 283-289.

4 Hoffman, Lance J. 1977. Modern Methods for Computer Security and Privacy. Prentice Hall, New Jersey.

5 Linde, Richard R. 1975. Operating System Penetration. Proceedings 1975, National Computer Conference, AFIPS, pp 361-368.

6 Parker, Donn B. 1976. Crime by Computer. Charles Scribner's Sons, New York.

7 Pirkola, Gary C. and John W. Sanguinetti. 1977. The Protection of Information in a General Purpose Time-Sharing Environment. Symposium Proceedings, Trends and Applications 1977: Computer Security and Integrity, pp 106-114.

8 Weissman, Clark. 1973. System Security Analysis/Certification Methodology and Results. System Development Corporation.

9 Whiteside, Thomas. 1978. Computer Capers. Thomas Y. Crowell Company, New York.