20 pages

IN-28442 TM ~~RC~~

# Analyzing the Security of an Existing Computer System

*Matt Bishop*

May, 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 86.13

(NASA-TM-89205) ANALYZING THE SECURITY OF AN EXISTING COMPUTER SYSTEM (NASA) 20 p CSCL 09B
N86-33029
Unclas
G3/60 44587

Research Institute for Advanced Computer Science

# Analyzing the Security of an Existing Computer System

*Matt Bishop*

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

*ABSTRACT*

Most work concerning secure computer systems has dealt with the design, verification, and implementation of provably secure computer systems, or has explored ways of making existing computer systems more secure. The problem of locating security holes in existing systems has received considerably less attention; methods generally rely on "thought experiments" as a critical step in the procedure. The difficulty is that such experiments require that a large amount of information be available in a format that makes correlating the details of various programs straightforward. This paper describes a method of providing such a basis for the "thought experiment" by writing a special manual for parts of the operating system, system programs, and library subroutines.

## 1. Introduction

Published work in the security of computer systems tends to take one of two directions. The work may center on a new, secure (possibly provably so) computer system, and discuss its design, implementation, and verification, or the techniques used to do any (or all) of these steps. Less commonly, the work may report ways of improving the security of an existing system by discussing the known problems and methods to counter these threats. Only a few papers[1] deal with how to analyze an existing computer system in order to locate security

Work reported here was supported by the National Aeronautics and Space Administration under contract NAS2-11530.

problems

At this point, we should remind ourselves what we are trying to do. Users who have legitimate access to the system are *authorized users.* If the permissions on the system are set to allow someone to perform an action, that action is an *authorized action*; if the action is performed in the absence of such permission, it is an *unauthorized action.* A *secure* system is a system which allows only authorized users to perform only authorized actions. For example, if a user is not known to the system administrator (by an entry in the password file), he is not an authorized user and hence should not be able to access the system. Similarly, a *breach of security* occurs whenever an authorized user performs an unauthorized action, or when an unauthorized user obtains access to the system.

There are several reasons to check existing programs. The most important is that the design, implementation, and verification of new code takes quite some time, during which the new code could not be used. When one realizes that most operating systems were not designed with security as the primary consideration, the magnitude of such a task becomes apparent. Existing code, on the other hand, could be used for increasingly privileged tasks as it is examined in stages for security flaws. Second, given that there is already enough existing code to keep a nonsecure system functioning, it may be more cost-effective to check the code for security holes rather than rewriting it completely. Finally, once it is written, new software can be treated like existing software.

Unfortunately, lack of formal verification poses problems. The best way to reduce the number of security problems is "to use formal security verification methods to assure that the mandatory ... security controls employed in the system can effectively protect ... sensitive information stored or processed by the system."[2] To do this, the developers must state their security policy, the axioms used to implement the security policy, and using these axioms present a mathematical proof that the system satisfies the security policy. Then, they must show the implementation of the operating system conforms to the design. (LOCUS[3] and PSOS[4] are examples of proposed operating systems for which mechanisms of formal verification have been described.) Throughout this procedure is an assumption that the system is designed with this type of verification in mind. To submit an existing system to this procedure, one must first decide on a security policy, and then model the system mathematically and show that the system not only satisfies the security policy, but also is accurately represented by the mathematical model. Abstraction of a mathematical description from the operating system is far more difficult than implementing the operating system from the mathematical description.

It is important to realize that no method will provide the same degree of security as formally verifying a system; however, less rigorous methods can reveal security flaws, and make the writing and checking of secure system software easier and less prone to error.

## 2. The Starting Point

Given that mathematical verification is not suitable, let us look at other methods of testing, and improving, system security. The most obvious is an *ad hoc* approach of trying types of attacks that have proven successful on this, or other operating systems in the past. Although doing so is very effective in discovering specific security problems, it does not provide a broad, systematic approach for discovering flaws in the security of computer systems, or for testing new components.

A generalization of this method will provide a foundation for analyzing security problems. One technique for penetrating operating systems involves a formal strategy called the "Flaw Hypothesis Methodology."[5] It consists of four parts: knowing how the target operating system interacts with users, hypothesizing a flaw in that interaction, confirming that the flaw exists (through "thought experiments" and actual testing), and generalizing the flaw, and similar flaws, to a design or implementation deficiency in that operating system. Clearly, the most difficult part is taking the first step, from the knowledge of the operating system to the supposition of flaws.

Before discussing ways to make this easier, let us try to categorize the main areas in which problems arise, to gain some insight about where to look. Bisbey, Carlstedt, and Hollingsworth at the University of Southern California's Information Sciences Institute have identified several categories of system flaws which can produce security violations. The following list summarizes them by listing main areas, each broken into sub-areas:*

(1) Improper protection (initialization and enforcement):

(1a) improper choice of initial protection domain; for example, an incorrect choice of a protection domain or security partition leading to a user

* This organization is from Peter Neumann[6].

being able to access and change an audit trail;

(1b) improper isolation of implementation detail; for example, allowing users to bypass operating system controls and write to absolute input/output addresses;

(1c) improper change; for example, allowing data to be inconsistent while still in use, by letting one process change a database file while another, different process is accessing that file;

(1d) improper naming; for example, allowing two different programs to have the same name;

(1e) improper deallocation or deletion; for example, leaving old data in memory deallocated by one process and reallocated to another process, enabling the second process to access the information used by the first;

(2) Improper validation; for example, not checking critical conditions and parameters, leading to a process' addressing memory not in its memory space by referencing through an out-of-bounds pointer value;

(3) Improper synchronization:

(3a) improper indivisibility; for example, interrupting atomic operations such as locking;

(3b) improper sequencing; such as allowing race conditions among processes vying for resources;

(4) Improper choice of operand or operation; such as using unfair scheduling algorithms that block certain processes or users from running.

Although certainly not complete, this list provides a means of classifying most security problems, and is quite suitable as an outline of areas in which problems of security will arise.

Now that we have guidelines on where to look, we must consider how to go about looking. Unfortunately, there is no way to do this other than by trial and error. (There has been some discussion of problems leading to, and attacks taking advantage of, security flaws in operating systems generally[4,6], as well as discussions of the security of specific operating systems[7,8,9].) Such methods may be made more effective if the trials are done systematically rather than at random. One technique to systematize the search is to use a dependency graph of the control objects in the operating system to study their interaction and look for possible problems that may enable an attacker to breach security. Among the difficulties with this are the generation of the graph, and its being understood by those not familiar with the layout of the graph.

Before examining another technique, let us analyze the problem of finding security holes a bit further.

## 3. Laying the Groundwork

The key point in looking for security flaws is recognizing that the security problems we are dealing with arise from interactions between the user and the operating system. Specifically, the user creates a condition using one or more programs and then executes another program or programs which cause the operating system to ignore specific protections. For example, to copy a protected file, the user must force the operating system to ignore or override this protection (for example, by running a program at a level of privilege sufficient to cause file protections to be ineffective.)

Unfortunately, any list of methods to do this will contain only a subset of all possible methods, since any new system program would add many new ways to evade protections. Even if such a list could be made, it would be very different for each operating system, because each operating system has its own design and implementation philosophy, and the latter often differ in ways that affect very subtle points of interaction. Similarly, programs perform different tasks, and the work needed to catalogue all of the possible jobs programs may do will be endless. Indeed, the required level of security differs, too; programs executed with special powers (such as *root* or *operator* privileges) must be checked for security violations that need not be looked for in other nonprivileged programs.

But the problems that arise come from the interaction of users with the operating system, as we have said. The only two ways for a user to interact with the operating system are through programs (software) and through equipment attached to the computer (hardware), in the latter case the interface being the kernel. So, in order to examine the way users interact with the operating system, we must study how the programs interact with the operating system, and the device drivers and other routines through which the equipment interacts with the kernel.

Let us deal with individual programs first. To study how they interact with the kernel, we shall try to abstract the functionality of the program from the actual code. This will have two effects. First, it will separate security problems introduced by the coding of the program from those introduced by the design of the program. Then, the design of the program can be checked, both for internal security problems and for security problems arising from interaction with other programs. Once this is done, the implementation can be examined to ensure that it does not introduce other security problems.

The first step, therefore, is to figure out what the program does, and how it goes about doing it. For the first part, system documentation will provide some guidance, but because documentation very often is incorrect, incomplete, or imprecise, it is not always good to rely on it; hence, for both learning what the program does, and how it does it, one must go through the program code. Second, one must document all interaction with the operating system (such as the files looked at, and how the program uses them.) In particular, one must document all error checking and recovery (or the lack of it.)

As an outline, the following organization for this document would be appropriate:

**Name**

This is the name of the program. If the program may be invoked by any of several names, all should be listed.

**Actions**

Although similar to a specification, this section should conform to the code and not to what the program is supposed to do. This section requires that the implementation be examined and written out in such detail that someone not familiar with the code could understand not only the action of the program, but how it works, and what side effects it has. If library routines or programs already documented in this fashion are used, it is often useful to refer to the appropriate pages rather than recapitulate the actions of those routines or programs.

## Apparent Assumptions

This presents any inherent assumptions. For example, if a file is assumed to be in a specific format, this should be noted here. If an assumption about the meaning of an error condition is made, list it here.

## Files Used

This section names the system and user files used. It also contains a short description of each, any assumptions made about format, and the system calls used to access each.

## System Calls

This lists all the system calls used.

## Execution Modes

This is most useful for programs; it describes who may execute the program and with what privileges the program executes.

## Known Bugs

Any known security problems are listed here. As security holes are found, they should be added. Note that suspicions should be listed (but marked as suspicions) until they are proven or disproven.

## Error Handling

This describes what happens if errors occur. For example, suppose an index into an array is out of bounds; does the program dump core? Suppose a file is not in the correct format? Are there checks to ensure any reading or writing succeeds?

## Library Functions Used

List the names, versions, and dates of any library functions used.

## Manual Page Version

Give the author, date, version of the program, and system for which this document was prepared.

We shall call this document the *security manual page* to distinguish it from the usual manual page. (A sample page, for the UNIX* library routine *getlogin*, follows the references.)

Of course, in the section, one should document any discovered security problems.

This documentation should not be confined to the program only. Very

* UNIX is a Trademark of Bell Laboratories.

often system programs need to perform a task such as looking up a name in a table to obtain associated data. These functions are performed so often that they have been collected into a set of library routines. Since these routines affect the function of each program in which they are used, it would save time and work to document these routines as described above. This would provide one reference for each library routine, rather than having the same routine be checked once for each program in which it is used (and risking a security hole being overlooked once). Similarly, new programs should use library routines whenever possible, and rather than duplicating code amongst several programs, the code should be changed into a library routine which the programs then call.

As an example of why documentation that describes the implementation of a program or library routine is necessary, consider the *getlogin*() bug, which exists on many UNIX systems. According to the manual[10], "[g]etlogin returns a pointer to the login name as found in */etc/utmp*." Although accurate, this description is very imprecise. *Getlogin* actually returns the login name of the user whose terminal is associated with the input, output, or error streams; this may or may not be the same as the login name of the person who executed the program. The security manual page should make this final statement, even though the manual page states *getlogin*'s function as indicated.**

Because of its complexity and function, the kernel must be checked differently than system programs and libraries. The principle is the same — analyze the code and document those parts which interact with other programs

** See the sample security manual page that follows the references.

and equipment — but many security manual pages, not just one, will be written for it. Specifically, at least one page per system call and device driver will be necessary, stating error conditions and precisely how they are handled, as well as how the system calls and device drivers are accessed. Main components of the kernel — the initialization routines, the scheduler, and so forth — must also be documented, as must any routines that rely on files or specific memory locations or any other external factors.

Hence, the first step to checking the security of programs and the operating system is:

*Document each program, system call and device driver, and library routine thoroughly, not just as to purpose but also as to its side effects and error handling.*

## 4. Hypothesizing the Flaws

Once a manual page or set of manual pages have been written, the process of locating security flaws begins. Unfortunately, the only known approach to doing this is largely *ad hoc.*

There are analogies in other fields. For example, the only way communications analysts can assess vulnerabilities of communications systems is to study the system thoroughly, and then draw on their knowledge of that system, their experience, and their knowledge of attacks that worked with other systems, to hypothesize security problems. They then test for these suspected flaws. The situation is precisely the same for computer security.

As with analyses of the vulnerability of communications systems, we can draw on past experience. There have been a number of studies of operating system security in general and of specific penetrations of various operating systems (some of these have been referred to earlier.) These studies provide knowledge of attacks that worked with many different systems. Combined with the knowledge gleaned from mathematical analyses of other systems and the weaknesses uncovered using those tools, all this experience provides a very solid background for hypothesizing security flaws.

The security manual described in the previous section will provide both the means of studying the system thoroughly and a reference guide useful in formulating hypotheses. As each program or routine algorithm is considered by itself, flaws may become apparent. (In fact, this happened with the *getlogin* manual page attached to this report. The second of the section was found by noticing the assumption made in step 4 of the algorithm, comparing it to step 3, and wondering what would happen if the assumption was invalid.) Correlating programs which use the same system files may reveal that the interaction of some such programs presents attackers with opportunities to subvert the system, or that these programs make inconsistent assumptions (or invalid assumptions) about the data in the file, or the way the file is used. A similar comment holds for programs and system calls; special attention should be paid to those system calls used to access and manipulate system files. The section on error handling should be quite fruitful for hypothesizing flaws. Many error conditions are not adequately handled, not handled correctly, or simply ignored. Very often this produces unusual situations that may present security holes which a clever attacker can exploit[11,12].

Hence, the second step to checking the security of programs and the operating system is:

*Drawing on the documentation, past experience, and general knowledge of operating system vulnerabilities, hypothesize security flaws in the computer system, and test either to confirm or to deny that those flaws exist.*

## 5. Summary

When checking an existing computer system for security, both the operating system kernel and the system libraries and privileged programs must be examined. (If none of these has security flaws, applications programs will not be able to breach security.) They should be examined in the above order; note that this will ensure that the operating system calls, which are the basis for system library routines and system programs, will be examined before the code using them is examined.

Within each of these aspects, the steps of the "Flaw Hypothesis Method" as described in sections 2, 3, and 4 should be used to locate security flaws, paying special attention to the problem areas described in section 2. For each aspect, a security manual of the sort described in section 3 should be written and used as the basis for examining the interaction of the various components of the kernel, the libraries, and the system programs as discussed in section 4.

While this method will not ensure perfect security of a computer system, it will significantly increase the difficulty of an attacker penetrating the system.

*Acknowledgements:* My deepest thanks to Barry Leiner and Peter Neumann, who both made very valuable suggestions towards improving this paper; to Mike Long, Bill Wall, and George Hays, for their incisive comments; and to Larry Hofman and Bob Brown.

## 6. References

[1] Denning, Dorothy E., *An Intrusion-Detection Model*, Technical Report CSL-149, SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, CA 94025 (Nov. 1985)

[2] –, *Trusted Computer System Evaluation Criteria*, CSC-STD-001-83, Department of Defense Computer Security Center, Fort George G. Meade, MD 20755 (Aug. 1983)

[3] Walker, Bruce, *et al.*, *Specification and Verification of the UCLA UNIX Security Kernel,* CACM **23**(2), pp. 118-131 (Feb. 1980)

[4] Neumann, Peter G., *et al.*, *A Provably Secure Operating System: The System, Its Applications, and Proofs*, Computer Science Laboratory Report CSL-116, SRI International, Computer Science Laboratory, Menlo Park, CA (May 1980)

[5] Linde, Richard R., *Operating System Penetration,* in the 1975 National Computer Conference Proceedings (AFIPS Conference Proceedings **44**), pp. 361-368 (May 1975)

[6] Neumann, Peter G., *Computer System Security Evaluation,* in the 1978 National Computer Conference Proceedings (AFIPS Conference Proceedings

**47**), pp. 1087-1095 (Jun. 1978)

[7] Attanasio, C. R., Markstein, P. W., and Phillips, R., *Penetrating an Operating System: a Study of VM/370 Integrity*, IBM Systems Journal **15**(1), International Business Machines Corp., pp. 102 - 116 (1979)

[8] Grampp, F. T., and Morris, R. H., *"UNIX Operating System Security"*, AT&T Bell Laboratories Technical Journal **63**(8), pp. 1649-1672 (Oct. 1984)

[9] Ritchie, Dennis M., "On the Security of UNIX", in *UNIX System Manager's Manual, 4.2 Berkeley Software Distribution, Virtual VAX*-11 Version,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (Mar. 1984), as reprinted by the USENIX Association

[10] –, *UNIX Programmer's Manual Reference Guide, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version,* Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (Mar. 1984), as reprinted by the USENIX Association

[11] Bishop, Matt, *How to Write a Setuid Program* (extended abstract), Proceedings of the Spring 1986 Cray User Group (May 1986)

[12] Darwin, Ian, and Collyer, Geoff, *Can't Happen or /* NOTREACHED */ or Real Programs Dump Core*, 1985 Winter USENIX Proceedings (January 1985)

* **VAX is a Trademark of Digital Equipment Corporation.**

# Appendix — Security Manual Page

## NAME

getlogin – get login name

## INVOCATION

**char *getlogin();**

## ACTIONS

*Getlogin* returns the user believed to be using the controlling terminal. It does this as follows:

1. Find the first of the file descriptors 0, 1, 2 associated with a terminal by running an *ioctl*(2) on each and seeing which one succeeds; if all fail, return 0.
2. Find the device/inode pair corresponding to that terminal by using *fstat*(2), and scan the files in the directory */dev/* until one is found with that device/inode pair. If none is found, return 0.
3. Search the file */etc/ttys* for that file name, and count the number of lines *N* skipped before it is found. If not found, return 0.
4. Read the *N*th record in */etc/utmp*; this corresponds to the user currently using that terminal. It is in the format of *utmp*(5).
5. Return the contents of the *ut_name* field of that record. Note it is kept in a static area, and is overwritten the next time *getlogin* is called.

## APPARENT ASSUMPTIONS

The first of the file descriptors 0, 1, and 2 that is associated with a terminal is associated with the terminal the user logged in on.

The number of the (text) line in */etc/ttys* describing a terminal corresponds to the offset into the file */etc/utmp* for that terminal.

## FILES USED

| | |
|---|---|
| /etc/ttys | List of terminal names, one per line; *open*(2), *read*(2), *close*(2) |
| /etc/utmp | List of logged-in users; assumes each record corresponds to a line in */etc/ttys* and that the records have the same order; *open*(2), *lseek*(2), *read*(2), *close*(2) |
| /dev/ | Directory containing files corresponding to terminals; used to determine the name of the controlling terminal; *open*(2), *read*(2), *close*(2) |

## SYSTEM CALLS

*close*(2), *fstat*(2), *ioctl*(2), *lseek*(2), *open*(2), *read*(2), *sbrk*(2), *stat*(2)

## EXECUTION MODES

This is a system library function.

## KNOWN BUGS

If the first file descriptor found to be associated with a terminal is not associated with the controlling terminal, the name of the user at the associated terminal will be returned, and not the name of the user at the controlling terminal.

If a line is added to or deleted from */etc/ttys*, the algorithm used to associate users with their terminal names fails miserably. This problem can be corrected by looking in the *ut_term* field of the record and comparing it with the name obtained from */etc/ttys*.

## ERROR HANDLING

On error, it is supposed to return 0.

No error check to be sure the *lseek*(2) to the record in */etc/utmp* succeeds.

No error check to be sure the record in */etc/utmp* corresponds to the name of the terminal.

Silently assumes names which are shorter than the space allocated in the record for user names are blank padded.

## LIBRARY FUNCTIONS USED

| *NAME* | *VERSION* | *DATE* |
|---|---|---|
| getlogin.c | 4.2 | 11/14/82 |
| isatty.c | 4.1 (Berkeley) | 12/21/80 |
| ttyslot.c | 4.1 (Berkeley) | 12/21/80 |
| ttyname.c | 4.3 (Berkeley) | 5/7/82 |
| closedir.c | 4.5 (Berkeley) | 7/1/83 |
| opendir.c | 4.5 (Berkeley) | 7/1/83 |
| readdir.c | 4.5 (Berkeley) | 7/1/83 |

## MANUAL PAGE VERSION

| | |
|---|---|
| AUTHOR | Matt Bishop |
| DATE | December 1, 1985 |
| SYSTEM | 4.2 BSD |
| VERSION | getlogin.c 4.2 (11/14/82) |