PROJECT MAC October 27, 1975

Computer Systems Research Division Request for Comments No. 92

CASE STUDIES OF PROTECTION SYSTEM FAILURES

by J. H. Saltzer

The attached material was constructed as an appendix to the subject 6.033 notes on protection. I am interested in receiving comments on accuracy, and also any further anecdotes that might be worth including.

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS 02139

6.033 -- Information Systems

October 7, 1975

APPENDIX G: CASE STUDIES OF PROTECTION SYSTEM FAILURES

Although a system designer must know fundamental protection concepts, such as those explored in chapter six, and should be familiar with details of several examples of real protection systems, another valuable asset is familiarity with a large collection of examples of failures in previously designed systems. In addition to teaching humility, a good collection of case studies provides a ready source of ad hoc tests for a new idea: would my design be vulnerable to some variation of this attack? Further, one can develop from failure case studies an intuition about which approaches are inherently weak or difficult to implement correctly. Finally, they provide evidence of the impressive range of considerations that a designer of a protection system must consider.

The case studies described here all really happened. However, failures are sometimes embarrassing, have legal consequences, or if publicized would jeopardize ongoing production systems that have not yet been repaired or redesigned. For this reason, some of the case studies have been disguised by embedding them in fictional circumstances. Ironically, many failures passed on only after receiving promises of strictest confidence turn out to be duplicates of failures already well known in other systems.

In reviewing each case study, three questions should be kept in mind:

1. What design principle or design principles, if any, were violated by this design? Often there is evidence of more than one design principle being overlooked.
2. Is this a particular example of a more general class of problem that the same system probably exhibits in other forms too?
3. What is the best way to redesign this system?

1. The old garbage analysis trick. Many protection systems have failed because they did not attach sufficient importance to protecting residues, the sometimes analyzable remains of a program or data in storage. On early versions of the M.I.T. Compatible Time-Sharing System (CTSS) this failure took the following form: a user operated in a memory region of an assigned size and he could request a change in current size by a supervisor call. If the user requested a larger size, the supervisor assigned a contiguous block of memory no longer being used by other programs, but it failed to clear the contents of the block of memory, so the residue of some previous program became accessible to any program extending its memory size.

At first glance, this oversight merely provides read-only access to an uncontrollable collection of garbage, and appears fairly hard to systematically exploit. However, an industrious penetrator observed that the system administrator ran a self-rescheduling job every midnight that updated the primary accounting and password files. On the assumption that the password file was processed by the administrator's program by copying it into primary memory, the penetrator wrote a program that extended its own memory size from the minimum to the maximum size, then searched the residue in the newly assigned area for his own password.

If found, that would suggest that other passwords might be stored nearby, so the entire memory residue was copied onto a file for later analysis. This program was scheduled to go into operation just before midnight, and use a timer to try the memory extension trick every few seconds. It worked quite well. The penetrator found in the residue a copy of a section of a file relating user names and passwords.

This general attack has been reported in a variety of other forms, such as reading the contents of newly allocated disk files, tracks, or cylinders, or reading newly assigned magnetic tapes. The potential for such an attack turns up in a slightly different form when a hardware technician is asked to repair a storage device--unless the device is cleared first, the technician can read the residue. [Indeed, in certain data-dependent hardware failures, it may be essential that the technician be allowed to read the residue to help diagnose the failure.]

2. Sophisticated garbage analysis. Related to the previous residue problem is a more sophisticated one encountered when recording on continuous media such as magnetic tape, disk, or drum. If the residue is erased by overwriting, it is no longer readable by programs. But analysis of the recording media in the laboratory will disclose residual magnetic traces of previously recorded data. For this reason, certain U. S. Department of Defense agencies routinely burn magnetic tapes and disk packs, and destroy magnetic drum surfaces, rather than discarding them or returning them to the manufacturer. Further, DoD regulation 5200.28-M calls for overwriting of certain magnetic media 1000 or more times before a medium formerly containing classified information can be considered "declassified".

3. Exploiting weaknesses in operational design. Some design choices, while not strictly affecting the internal security properties of a system, can affect operational aspects enough that system security is weakened. In the CTSS system, as mentioned, passwords were stored in a text file together with user names; this file was effectively a master user list and the system administrator therefore, wherever he changed the file, printed a copy for quick reference. He had no interest in the passwords, but the list of user names was needed to avoid duplication of names when adding new users. This copy, including the passwords, was processed by printer controller software, handled by the printer operator, placed in output bins, moved to the system administrator's office, and eventually discarded by his secretary when the next version arrived. At least one penetration of CTSS was accomplished by a student who discovered an old copy of this printed report in a wastebasket. At another time, the system administrator was reviewing and updating the master user list using a standard editing program. The editor, unbeknownst to the administrator, operated by creating an unprotected copy of the file being edited in the current directory, under a name chosen by the editor. Another system operator working simultaneously from another terminal was using the same editor to update another file in the same directory--the "message of the day", a short file printed out whenever a user logs in. The two instances of the editor used the same intermediate file, with the result that the master user list, complete with passwords, was appended to the end of the message of the day.

4. The system programmer attack. A programmer was temporarily given the privilege of modifying the supervisor of a time-sharing system, as the most expeditious way of getting a user problem solved. While he made the changes appropriate to solve the problem, he also added a feature to a rarely-used metering entry of the supervisor: if called with a certain argument value, the metering entry would reset the status of the current user's account to show no usage. This new "feature" was used by the programmer, and his friends, for months afterwards to obtain unlimited quantities of computer time.

5. The supervisor trusts the user. In the first version of CTSS, a short-cut was taken in the design of the supervisor entry that permitted a user to read his own file directory. Rather than remembering in a supervisor data base the current position in the file directory, as part of each read call the supervisor returned to the user an index that the user was to provide in turn when calling for the next record. A curious user printed out the index, concluded that it looked like a disk track address, and wrote a program that specified track address zero, which contained track addresses of key system files. From there he was able to find his way to the master user table containing passwords.

Although the vulnerability seems obvious, many operating systems have been discovered to contain some situation in which the supervisor leaves some critical piece of data in an unprotected user area, and later relies on its integrity. In one large-scale operating system implementation effort, each system module was allocated a limited quota of system protected storage, as a strategy to keep the size of the system down. Since in many cases the quota was too small, system programmers were effectively forced to place system data in unprotected user areas. Despite many later efforts to repair the situation, an acceptable level of protection was never achieved in that system.

6. The supervisor trusts the user without realizing it. As a subtle variation of the previous problem, consider the following supervisor program:

```
delete_file:  procedure ( file_name, code );
              call check_auth ( file_name, user_id, code );
              if code = 0 then
                        call destroy ( file_name );
              return;
              end;
```

This program is a user-callable entry point, and it is apparently correctly checking to see that the user has correct authority before actually destroying the file. Program check_auth will set "code" to some non-zero value if it finds that the user named by "user_id" does not have the necessary authority for file "file_name".

But variables "file_name" and "code" are user-supplied arguments, allocated and stored in user-chosen and user-accessible areas. The user may be able (by use of a second, parallel process, for example) to observe the value of code being set by check_auth, and quickly reset it before delete_file gets a chance to test it. Alternatively, by careful timing, the user may be able to change the name stored in variable file_name between the time "check_auth" examines it and "destroy" uses it. In one time-sharing system, several dozen examples of this penetration route were found.

7. The "open design" penetration. In the process of working out a system design, there is often an argument made by system programmers that there is nothing wrong with letting the user have read access to most supervisor programs and tables, since the algorithms are not secret. The usual reason for this argument is that it makes debugging easier: without special privilege a system programmer can examine system tables following unusual situations and perhaps detect clues

to the cause of the problem. The following article from Computerworld shows the result. (Note that this attack is not unique--it was used successfully on CTSS, too.)

T/S Service Security Cracked by Schoolboy With Series of Tricks

Special to Computerworld

LONDON – A 15-year-old schoolboy with only four months' experience in its Assembly language cracked the security of a major time-sharing service here – while keeping up with his regular homework.

Using the teletypewriter terminal in his school, the student, identified only as "Joe," obtained access to the system's most secret files. He was able to read and change them at will and even affect billing procedures, but Joe said he had never done this.

Joe's trick was to learn the system's highest level account names and passwords, but that required a long series of steps.

First, Joe found there was no read protection on any location in core, so he wrote a dump program and printed out the operating system. From that listing, along with some tips from programmers at the time-sharing service and one obsolete systems manual, he was able to work out much of the system.

The next step was relatively easy. Joe found that he could print out the account name and line number of every terminal logged on to the sytsem. This showed that there was a simple algorithm for assigning the line number to the next user.

The system has a unique buffer for each line to store data being input on that line. With a little bit of trouble, Joe was able to locate the buffers. He then wrote a program to eavesdrop on whatever was being typed on a terminal simply by printing out the contents of the buffer.

With his "who's-logged-in" program, Joe was able to find which lines were already in use and thus predict which one would be used next. With his "buffer-watching" program he then waited until someone signed on, gave his account name and password, and then Joe printed out the content of the buffer.

In practice, there were a few snags. Joe's account had a low priority, and the system therefore did not like him just sitting in a loop checking a section of core. Joe had to pretend to do something beside looping and thus ran the risk of losing information from the buffer on which he was eavesdropping.

He did, in the end, get the privileged passwords – but he never did much with them. He wrote to the time-sharing service and explained how he cracked the security – but he never received a reply.

His teacher did ban him temporarily from the terminal, however.

Shortly after receiving Joe's letter, the time-sharing service introduced a new version of the operating system. Joe doubts that it has corrected his route into the system and wants to try his method again.

But that will have to wait a little while, because he is in the middle of exams at the moment.

Computerworld IX, 5, January 29, 1975.

8. The incomplete check of parameters. A fairly common method of penetrating systems has been to examine the code at the supervisor entry points, looking for places that unexpected, out-of-range parameter values might cause trouble. An interesting example occurred in a system that, like CTSS, allowed the user to request a different memory size. In that system, a penetrator discovered that if the user requested a negative memory size, the supervisor would blindly assign additional contiguous storage at the wrong end of the user's program. That particular area happened to contain critical supervisor information regarding that user, and the user could, by modifying it, obtain control of the supervisor.

9. Spoofing the operator. Many operating systems include a system feature to transmit a message to the system operator, for example to ask a question or to provide supplementary information to a request to mount a user-supplied tape. This message is printed at the operator's terminal, intermixed with messages from the operating system. The operating system normally prints a warning banner ahead of the user's message so that the operator knows its source. In CTSS, the supervisor placed no constraint on either the length or content of such messages, so a user could send a single message that, first, printed several blank lines, to push the warning banner out of sight, then print a line that looks like a system-provided message, such as an instruction to mount a tape or to shut down the system. Other systems have also been discovered to be vulnerable to this trick.

10. The system release trick. A Department of Defense time-sharing system was claimed to be secured well enough to process military classified information. A (fortunately) friendly penetration team looked over the system for a short time and tried the following strategy: they constructed on another, similar computer, a modified version of the operating system with some extra

entry points that permitted any user to "take over" the supervisor. They then mailed to the DoD installation a copy of a tape containing the modified system together with a modified copy of the most recent "new system version" distribution letter from the computer manufacturer. The letter and tape were received, and the tape installed as the standard operating system. A few days later the team proceeded to demonstrate system takeover by any user.

11. Signalling with clandestine channels. Once information has been released to a program it can be very difficult to be sure that the program is not passing the information along to someone else. Even though non-discretionary controls may be operating, the program may be able to signal using a clandestine channel. In an experiment with a virtual memory system that allows shared library procedures, an otherwise confined program used the following signalling technique: for the first bit of the message to be transmitted, it touched (if the bit value was one) or failed to touch (if the bit value was zero) a previously agreed-upon page of a large, infrequently used computer program in the library. It then waited a while, and repeated the procedure for the second bit of the message. A receiving process observed the presence of the agreed-upon page in memory by measuring the time required to read from a variable stored there. A short (microsecond) time meant that the page was already in memory and a one value was recorded for that bit. Using a single page for data transmission, and other pages to signal in the reverse direction that the bit had been received, a data rate of about one bit per second was attained.

12. Unintentional signalling with clandestine channels. If a supervisor entry is trying not to release a piece of information, it may be possible to infer its value from externally observed behavior, such as the time it takes for the supervisor to execute, or the pattern of user data in memory after it finishes.

An example of this attack occurred on a time-sharing system that used demand paging for user memory areas, and allowed a program to acquire the privileges of another user if the program could supply that user's secret password. The supervisor routine that examined the user-supplied password did so by comparing it, one character at a time with the corresponding entry in the password table. As soon as a mismatch was detected, the supervisor password checking routine stopped and returned an error code.

A clever user noticed that the user-supplied password could be placed anywhere in user memory, for example at a boundary between two pages such that only the first character of the password was at the end of the first page. The user then waited long enough for both pages to be "paged out" of memory, then called the supervisor entry asking for the other user's privileges and giving the address of the strategically placed password. When the supervisor returned with a mismatch report, the user program then measured the real time required to read a variable stored in the page containing the second half of the password. A long (millisecond) clock reading meant that that second page was not already in memory, presumably because the supervisor had not touched it, implying that the first character of the password had not been correctly guessed. By cycling through the letters of the alphabet looking for one that produced a short (microsecond) clock reading, the program could systematically search for the first letter of the password. Then, the password could be moved up one character position, and the second character searched for. Continuing in this fashion, the entire password could quickly be exposed.

13. The case of the undeleteable data. It is a common practice for a time-sharing system to periodically make backup copies of all user files on magnetic tape in various formats. One format might allow quick reloading of all files,

while another might allow efficient searching for a single file. Several backup copies, perhaps representing user files at one week intervals for a month, and at one month intervals for a year, might be kept. The administrator of such a time-sharing system was served with an official government request to destroy all copies of a certain file belonging to a user who had compiled an on-line list of secret telephone access codes, which could be used to place free long distance calls. Destroying the on-line file was straightforward, but the potential expense involved in locating and destroying all of the backup copies was enormous. (A compromise was reached, in which the backup tapes received special protection until they were due to be recycled, up to a year later.)

14. The special case that failed. In a large-scale processor designed for maximum speed, the circuitry to check read and write permission was invoked as early in the instruction cycle as possible. When the instruction turned out to be a request to execute an instruction in another location, the execution of the second instruction was carried out with later timing, so the standard circuitry to check read and write permission was not used--a special case version of the circuit was used instead. Although originally designed correctly, a later field change to the processor accidentally disabled one part of the special case protection checking circuitry. Since instructions to execute other instructions are rarely encountered, the accidental disablement was not discovered until a penetration team began a systematic study and found the problem. The disablement was dependent on the address of both the executed instruction and its operand, and was therefore unlikely to have ever been noticed by anyone not intentionally looking for security holes.

15. The buggy password transformer. In a system that performed a "one-way transformation" on passwords for storage purposes, a penetration team mathematically examined the one-way transformation algorithm and discovered an inverse transformation. When the inverse transformation was tried, however, it did not work. After much analysis, the team discovered that the system procedure that did the supposedly one-way transformation used a library mathematical subroutine that contained an error; the passwords were being incorrectly transformed, although since the error was consistent it did not interfere with operation. The erroneous algorithm was reversible, too, so the system was successfully penetrated. An interesting sidelight arose when the error in the mathematical subroutine was reported and a fix developed. If the fixed routine had been installed, the password transforming algorithm would have begun working correctly, which would have meant that correctly-supplied passwords would transform to values that did not match the stored values that had been created using the incorrect algorithm. Thus no one would be able to log in. A creative solution (which the reader may attempt to reinvent) was found for the dilemma.

16. The uncheckable data channel. A common architecture for input/output channel processors is the following: channel command programs refer to absolute memory addresses without any hardware protection, and they may modify themselves by reading data in over part of the channel command program. If, in addition, the operating system permits the user to directly create channel command programs it becomes very difficult to enforce protection constraints. Even if the channel programs are reviewed by the supervisor to make sure that all memory addresses refer to areas assigned to the user who created them, if the channel program makes use of the self-modifying feature, the checks of its original content are meaningless. In the case

of CTSS this problem led to a prohibition on timing dependent and self modifying channel programs. The trouble is, there was no way to enforce the ban, and a battle of wits resulted: for every ingenious technique developed to discover that a channel command program contained an obscure self-modification feature, some clever user discovered a still more obscure way to conceal self-modification in channel command programs.

17. A thorough system penetration job. One particularly thorough system penetration operation went as follows: first, computer time was obtained at a site different from the one to be penetrated, but running the same hardware and same operating system. On that system many experiments were performed, and an obscure error in protecting a supervisor routine was found. The error permitted general changing of any supervisor-accessible variable, so it could be used to modify the current job's principal identifier. After perfecting the technique, the penetrator moved his operation to the site being used for development of the operating system itself. He used the privilege of the new principal identifier to modify one program in a directory containing copies of object programs of the operating system. The change he made was a one line revision to omit a crucial protection check at a supervisor entry point. Having installed this change in the program, he than covered his trail by changing the directory record of date-last-modified, thereby leaving behind no traces except for one changed line of code in the supervisor library. The next version of the system to be distributed to customers contained the penetrator's revision, which could now be exploited at a third target site.