REALIZATION OF HIGH SECURITY SHARED-RESOURCE COMPUTER SYSTEMS BASED ON EXECUTE-ONLY SEGMENTS

by

David Rabinowitz

Memorandum No. ERL-M389

June 21, 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

REALIZATION OF HIGH SECURITY SHARED-RESOURCE COMPUTER SYSTEMS BASED ON EXECUTE-ONLY SEGMENTS

by

David Rabinowitz

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

ABSTRACT

A major area of research today concerns provisions for protection of sensitive data and programs. This paper describes a segmented time-sharing security system where each segment is execute-only and is completly protected from access by all other segments. It discusses various mechanisms which can be used to implement such a system and gives an example of how such a system can be used.

Research sponsored by the National Science Foundation Grant GJ-36475.

One of the goals of a multiprogramming or timesharing system with a file system is to provide a means by which files and programs stored wtih the file system can be made available to users according to various criteria defined by their owners. At the same time it is equally important, if not more so, that others be denied access to these entities. When security is of the utmost importance, the ultimate protection is achieved by giving each user his own totally separate computer system, in which case the overhead advantages of timesharing, as well as provisions for sharing resources are lost. This paper proposes a system whereby each user appears to have a totally disjoint computer while actually timesharing a big system with all the advantages in overhead reduction and with full capabilities for sharing all resources of the system while still protecting resources from access and use by people not authorized, at any level desired.

The system proposed is based on a segment-oriented design similar to that of the MULTICS system [1] but differing in one basic way; in this system each segment is totally independent, operating in its own memory space and totally protected from access by any other domain, as if it were running alone on the system. In effect this creates a system with not two modes, such as supervisor and problem modes on IBM's System/360, but rather with n modes, where n is the number of segments allowed in the system, and with each mode having, possibly, special priviledges not available to others.

On this system a segment consists of a fully paged virtual memory space and has a name. The virtual memory space is extremely large and can be used in non-contiguous sections with no storage space wasted for unused pages, but a segment cannot access anything outside its own memory space. Every segment has one owner, the user who created it. The hardware keeps track of who owns each segment and grants the privilege of destruction to the user. The details of the implementation will be discussed later.

Each terminal, or job stream in a multiprogramming system, owns one process which is, at any time, under control of one segment. The process's status is defined by a stateword which contains, in addition to the program counter and working registers, as in all multiprogramming and timesharing systems, the segment-id, the terminal-id (considering from now on the implementation on a timesharing system), the user-id of the user who initiated the current incarnation of the process and the time-of-day this process (actually this session) was initiated. Thus, for example, if user x logged on to teletype y at time t, then x, y and t would be in the hardware-protected stateword. How it gets there will be discussed later. However, once the process is running the segment in control at any time can ascertain who is using it, from what terminal, and in which incarnation of this user-terminal combination.

Thus far it is not clear how resources can be shared in this system. It is evident that, since all segments are totally protected from each other, that every segment must be, in effect, execute-only, and the only way for one segment to communicate with another is by calling it. Therefore one of the keys to the system is the calling mechanism. In this system, every segment can call any other segment whose name it knows. This does <u>not</u> imply that the names of the segments are the keys which must be protected, for what the called segment does when called is not yet specified. A segment calls another segment by executing a CALL instruction, with parameters being the name of the segment being called and the address and length of a string of words in the calling segment's memory space to be passed as parameters. As far as the calling segment is concerned, a call to another segment is similar to a request for output, where the parmeters being passed are elements of the output string. The string is placed on an output buffer, with the output device being the called segment.

A call to another segment handled entirely by hardware. Every segment has a single call entry point, location 1 of page 0, to which every CALL transfers control. On a CALL, the changing parts of the stateword (all but the user-id, terminal-id and log-on time) are pushed onto a hardware-maintained stack for that process, the name of the routine being called is forced into the segment-id portion of the stateword and "1" is forced into the program counter. The segment-id of the calling segment is retained in a special part of the stateword for possible access by the newly-called segment. The system is now executing the new segment at address 1.

All paging operations are handled entirely by hardware (perhaps microcode) which, in effect, considers the segment-id to be part of the program counter. Thus, after the stateword has been modified on a CALL, the hardware tries to execute location 1 of the new segment. If, by some chance, this page of this segment has been executing recently and is still in main memory, the hardware will simply execute from that point. If, however, that segment page is not in main memory, the hardware will treat the situation as a simple page fault and go out and get the page, referring to whatever tables it needs. Note that since all addressing is paged, through page tables, memory is automatically protected, as a segment cannot get at any memory which is not its page table.

There is one difference between the way the hardware handles a CALL and a simple memory reference, and that is that a CALL starts a timer which disables all interrupts for up to its duration, that duration being fixed and dependent on the computer being used. This interrupt-free period is granted each segment to allow it to clean house and perform any locking of itself to other references as needed, and can be ended at any time by the segment. It should be mentioned that a segment is an entity and not a text which may be copied. At any time there is only one copy of each page of each segment in existence in the system, so when a segment is called, the segment itself responds to the call, and not some private copy of that segment made by the system. Therefore, a segment must be, in effect, re-entrant, and the interrupt-free period allows it the time to reset pointers and do whatever else is necessary to implement this reentrability.

Once a segment is running it is in complete control. It can read the stateword and determine who the user is, at what terminal he is working, at what time he logged on, and what segment just called it. It can also read the parameter string by issuing an input-like command, GET, as if he were requesting input from an I/O device, giving as a parameter the address in its memory space into which the string is to be placed, as well as a limit on the length to be accepted. Note the similarity to the 'message' system being implemented in PRIME [2] in that parameters are passed through the system. The segment cannot, however, read anything from the process's calling stack, so the user's session history is protected.

Having taken care of the reentrance problems, the called segment now does its thing. In the case of a simple un-protected segment which performs a simple function, like computes the SIN function or such, the segment simply reads the input parameters, computes the function and returns the result using the RETURN instruction, with the parameters again being the location and length of the 'output' string to be returned. The effects of the RETURN are identical to those of the CALL with the exceptions that the new segment name and program counter values, and other changing elements of the stateword, are popped off the calling stack, and there is no interrupt-free period started. Note the similarity between this system and a multi-computer network where each computer communicates with the others by sending messages, each being otherwise totally secure from the others.

In the case of a protected segment, the segment can behave like a FORMULARY CONTROL procedure [3], executing code to verify the user's authority and privileges using whatever algorithm is desired. It should be noted here that every segment can read the time-of-day clock and can do input and output to the device running the process, the terminal listed in the stateword, by giving a simple INPUT or OUTPUT command. It is allowed to use only this one device, and on issuing the command, the character, or string, referenced is transferred by hardware (perhaps microcode) between the address in the segment's virtual memory and the device's I/O buffer. Thus, any segment can interact with the person at the terminal in real time, allowing any level of hand-shaking verification desired.

It was noted above that a segment can do I/O operations only on the device listed in the stateword. This implies that a segment cannot explicitly request to read or write on disks or other mass storage devices, in addition to being denied access to any system line-printers and such. The latter restriction is resolved by allowing only one segment to write on each line-printer or read from each card reader, etc., this protection being provided by having the hardware check the segment-id of the executing segment in the stateword on each such request, and requiring each other segment which wants to use the printer to do so by calling the printer segment. By analogy, the segment-id defines the mode of the system (ala supervisor mode vs. problem mode) and the mode of the printer segment (or of each printer segment in the case of multiple, independently protected line-printers) carries the privilege of performing I/O on its printer. Note that this implies that each I/O device can potentially be protected as well as any other object in the system.

As for the question of requesting access to a mass storage device, it becomes evident that this capability is totally unnecessary for system operation. A segment has its very large virtual memory. Any data belonging to it resides in its virtual memory and is retrieved by doing a simple memory fetch. Any data not belonging to it is not directly accessible to it, which is the goal of the system, and must be obtained by calling the segment to which it does belong. Disk reference are implied in this system, and performed at every page fault, but no explicit references to disk are involved.

The fact that everything accessible to a segment resides in its virtual memory implies that all files must reside in the virtual memories of segments and that these segments must have within them the code necessary to respond to CALLs, determine what is desired and whether the process requesting the file manipulation has the right to do what it wants to do, and finally to do the thing, if necessary, or sound the alarm if that is necessary. This implies that analogs to both the CONTROL and VIRTUAL procedures form the Formulary Model [3] are required by every file segment. Note that unless removable mass storage devices which can be removed from the system and read elsewhere are used, access to the contents of the file segment is not possible except through the code in the segment itself, so SCRAMBLE and UNSCRAMBLE [3] would probably not be necessary, though they can always be used if desired. Finally, since every call to a segment gets the same copy of that segment, as only one copy exists, a file segment can remember who called it recently and what it decided about privileges and authority, and can therefore do the authority and privilege checking only on the first call, when the processing is not data-dependent. The time-on field in the stateword uniquely determines the process incarnation, as the time-of-day clock from which the time-on is generated is of high enough resolution to preclude 2 processes having the same time-on. Thus, if a segment receives a call from a process, verifies the privileges of the process, does the required work and returns the results, it can, before returning, make a note within itself of the process time-on and its privileges, and if it later receives a call from the same process incarnation it already knows the privileges without further computing. I should repeat here that a segment is a virtual memory space and is defined by its name in the stateword. A process is a stream of control defined by the terminal-id, and a process incarnation is a specific session on that process, defined by the user-id and time-on. Thus, at any one time a process is in a certain incarnation and is controlled by some one segment.

Now that the system is generally described we must get into the real nitty-gritty of how the work is done. Among the things which must be described are how segments get created and destroyed and how their gigantic memory spaces are managed without undue overhead, how the user-id and time-on get into the stateword and how the user is authenticated, and finally how security supervision and threat-monitoring is accomplished.

To begin with, there is a fixed number of processes in existence in the system at all times, one for each terminal. When a terminal is idle, not logged on, its process is under control of the LOGON segment, a segment belonging to the system. This segment is the only one which has the power to modify the fixed parts of the stateword of a process. Remember that there is no heirarchy of authority in this system, and therefore that this segment, while having this one extra power, has no other authorities above other segments. However, the simple power to modify the fixed parts of the stateword is sufficient to require that the code in this segment be blessed, that is, totally verified and guaranteed. The algorithm needed is so simple and short that this is no problem.

When a process is idle its stateword contains the null user-id and time-on, and its terminal-id, which never changes for a process. The process is 'asleep', waiting for something to be entered at the terminal, the LOGON segment having requested an input character and having been blocked when none was forthcoming. When a character comes in, the LOGON segment asks the user to enter his user-id and, on receipt of it, looks it up in its user table. If it is found, it has, associated with it, the name of the user's CONTROL segment, a segment which was defined by the user and created for him by the system administration when he was first granted access to the system. The LOGON segment simply calls this segment and lets it decide whether or not this is the real user. The CONTROL segment interacts with the user in whatever ways required and then

RETURNS with an indication of whether or not the person is approved. If the person is approved, the LOGON segment proceeds to update the state-word with the user-id and time-on, and again calls the user's CONTROL segment, this time with a different parameter, telling the CONTROL segment that the user is logged on, and the CONTROL segment then serves as, or calls, the user's command processor. If the person is not approved, the CONTROL segment can make a note of it for that user's information, can simply return unsuccessfully to the LOGON segment, perhaps after some period of delay to discourage machine-aided infiltration, or sound an alarm to the system by the alarm mechanism to be described. Note that the user's CONTROL segment could not have been created by the user, as he didn't exist at the time it was created, but was rather created by the system administration and is therefore owned by the administration, which has full privilege of ownership and can therefore remove any user from the system at any time.

A segment is created by a hardware command, CREATE, with parameters of location and length of a string in memory, as in all previous commands mentioned. The CREATE command generates a new segment-id, creates necessary tables and initializes the segment's virtual memory to the string passed as the parameter, starting at location 0, making note of the owner of the segment, and returning the assigned segment-id. Note that in many cases a user may want to create a segment with a fixed control routine in it, as in the case of a text file with a common protection mechanism and access routine. In that case, rather than keep a copy of this routine within himself to use to initialize the segment, he can simply call a segment, perhaps provided by the system administration, which creates this segment for him. Note that the <u>user who creates the</u>

segment as identified by the user-id, owns it, regardless of what segment actually executes the CREATE. Alternately, he can always create a segment with whatever code he wants. To destroy a segment a user issues the command DESTROY, with the segment-id as the parameter. The hardware checks the stateword to be sure the command is being issued by the owner of the segment, and if it is, all records of the segment are removed from the system. If the command is issued by other than the owner, an alarm is sounded. Note again that unless the mass storage media can be moved and read elsewhere, there is no need to purge the destroyed pages of the segment from the disk, as there is no way for anyone to access them before having overwritten them with new data, as will be shown.

Since a segment has a potentially very large virtual memory, there must be some way to control the amount of storage required by that memory space, to allocate and free pages depending on their use. A page is allocated to a segment automatically the first time it is referenced. The page is initialized to a constant value, probably all 0, and added to the segment's tables. The segment is not allocated disk space for permanent storage of its new page until that page is first swapped out. Therefore it is impossible for a segment to read the remains of another segment on disk. A segment can release a page by giving a FREE command with parameters being again the location and length of a string in memory. The hardware removes from page tables the entries for all pages entirely contained within the string, retaining all pages overlapping the string. Thus, the segment can free sections of its memory without worrying about page boundaries. Again, the disk sectors associated with the freed pages are freed and need not be purged.

Finally, at any time any segment can sound an alarm. An alarm is sounded by executing the instruction ALARM, which effectively executes a CALL to the ALARM segment. The alarm segment is a system-provided segment which is like every other segment with the addition that it alone is allowed to read from the calling stack of the process it is controlling and is able to inhibit execution of a 'grand escape', to be described later. A parameter passed by the ALARM instruction tells the segment how serious the calling segment considers the incursion to be, and the ALARM segment operates accordingly. Among the things it can do is to make a note in a threat-monitoring file, recording any desired information from the process calling stack, sound an audible alarm at the offending terminal (if one exists- if one does exist, the ALARM segment can be given the privilege to use it the same way the line-printer I/O segments had the privilege to use the line-printers), or, in extreme cases, can use its 'grand escape' prevention to refuse to return, effectively tying up the process and therefore the terminal.

This brings up the question of how a user can escape from a segment which has entered an endless loop or otherwise refuses to release control. As in all systems there has to be an 'escape' key on the keyboard to allow a user to terminate an uncontrolled process. In this system there are two such keys, a 'minor escape', and a 'grand escape'. The escape is accomplished through an interrupt mechanism. An interrupt looks to the segment being interrupted like a CALL except that control is transferred to location 0 instead of 1, with the parameters being the relevant interrupt information. The difference between a minor escape and a grand escape is that the minor escape causes an interrupt to the segment being executed, while the grand escape causes the process calling stack to be purged before interrupting, thus effectively placing the user back under the control of the LOGON segment. Note that though the process is under control of the LOGON segment, its stateword still contains the user-id and time-on, and the LOGON segment is aware of the fact that it was entered by a grand escape and will turn control over to the user's CONTROL segment, as when the user first logged on. Note that every segment can be interrupted and must therefore be prepared to handle interrupts. Most segments would probably want to pass the interrupt back to the segment which called it, as for example the SIN function segment. This capability is provided with the INTERRUPT command, which effectively backs up one level into the calling stack and then generates an interrupt, carrying the original interrupt information. Thus, such a segment would contain an INTERRUPT instruction in location 0 and would require no other interrupt-handling code.

Note that the grand escape effectively wipes out the user's work and would only be used in rare cases. Note also, however, that any segments the user may have created or modified in the aborted process are not lost, as there are no temporary work copies unless the user explicitly creates them, all work being done with the actual segments. Note also that as the ALARM segment is impervious to the grand escape, it can therefore completely ignore a terminal.

Let us now consider how a typical job of editing, compiling and running a program could be done on this system. There are several different approaches which would work, the methods varying in overhead and security. One method would be the following:

To create and edit a program, the user, once logged onto the system, would call the EDITOR, which would create a new segment for the user with a copy of its (the EDITOR's) routines in it. The user would then use his personal copy of the EDITOR to create and edit the new text file, all of it residing in the virtual memory of this new segment. When the user is happy with his program, he can request that his copy of the EDITOR release itself, freeing the pages it occupied but leaving the necessary control and access routines, and then ask the remaining code to call on the compiler-copying segment to pass it a copy of the compiler it wants. This copy of the COMPILER can be read into the segment's virtual memory and executed there, producing object code, again in the segment's virtual memory. The COMPILER can then self-destruct, if desired, and the code can be run. Note that there is no need to separate the object code from the source text; both can be kept in different parts of the same segment. There is no need to load a resident package either, as whenever the program wants to execute a system subroutine it can simply call that routine's segment; an inter-segment call is almost no more costly than a page fault.

The above is only one way of accomplishing the work. The user could have retained copies of the EDITOR and COMPILER while the program was running, to allow himself to make quick corrections and recompile if the program didn't work, and could have loaded in a debugger if desired, or he could have created a new segment with only basic text update and retrieval routines and then have the EDITOR run within its own segment, calling the new segment to access and update the text file, etc.

The one remaining detail to be discussed is the log-off method, which is very simple and straightforward. A user logs off under control of his command segment, which could also be his control segment but needn't be, thus allowing several users to share a command processor. When the command segment receives a LOGOFF from the terminal it returns control to the LOGON segment, passing a parameter that specifies it wants to log-off. The LOGON segment clears out the stateword, updates account files, if desired, and then issues an INPUT command, requesting a character from the terminal. Until a character arrives the process is blocked and inactive. The arrival of a character restarts the logon procedure.

DISCLAIMER

This is a first attempt to define the structures and techniques necessary to implement this type of system, the original goal being to design a system consisting entirely of execute-only segments. While it does appear to present a straightforward hardware-supported security scheme, it has not been implemented and the design certainly cannot be claimed to be complete. However, it does have some new features, perhaps the most attractive being the simplified inter-segment calls and the concept of an n-mode system where each mode has some privileges, but these privileges are not assigned heirarchically as in traditional ring-type systems, but rather disjointly. We think the concept merits further study.

REFERENCES

[1] Bensoussan and Clingen, The Multics Virtual Memory, ACM 2nd Symposium on Operating System Principles, Oct. 1969, pp. 30-42.

[2] M. Ruschitzka and R. S. Fabry, The Prime Message System, COMPCON 73, pp. 125-128.

[3] Hoffman, L. J., The Formulary Method for Flexible Privacy and Access Controls, FJCC 1971, pp. 587-601.