To: Distribution

From: Andre Bensoussan

Date: 03/12/76

Subject: Removing Directory Control from the Security Kernel

INTRODUCTION.

This document is a proposal to remove directory control from the Air Force security kernel in the Multics System. It is intended to serve as a basis for further discussions with the various groups involved in project Guardian at the Air Force, Mitre, MIT and Honeywell, in order to evaluate its feasibility.

The long range objective is to show that the mathematical model for security, developed by Mitre, can be met by a system for which the top level formal specifications do not refer to directories at all.

The short range objective is to make minor changes to the Multics version known as the "new storage system" in order to obtain a system where the security access rules (i.e., no read up and no write down) are enforced regardless of how directories are manipulated.

This document addresses the short range objective only. The term "security kernel" is used to refer to the set of supervisor procedures and data bases which are necessary to enforce the security access rules.

Removing the management of directories from the security kernel would require restructuring the current ring 0 supervisor into two hierarchical layers. The security kernel would operate in ring 0; it would provide the segment and the process entities and would be responsible for enforcing the security access rules. Directory control would operate in ring 1, under the rules imposed by the kernel, and would use the abstract machine made available by the kernel in the form of a set of ring 0 kernel primitives.

The design of the new storage system has, to a large extent, achieved the separation of directory control and segment control, and would provide a very good basis for implementing this proposal without a major rewriting of the current ring zero supervisor.

THE KERNEL AND THE DIRECTORY TREE STRUCTURE.

One would expect the new kernel to be ignorant of directories. The new storage system has the appropriate modularity to cope with this situation, except for the quota implementation. The quota facility is defined in terms of the directory tree structure, and the procedures that implement this facility must know about the tree structure. Since the implementation of quota is distributed between directory control, segment control and page control procedures, the knowledge of the tree structure, instead of being encapsulated in directory control, has penetrated deeper into segment control and page control. To eliminate the knowledge of the tree structure from the new kernel, one has to reimplement the quota facility in such a way as to perform in directory control those quota manipulations that take place in segment control and page control of the current system (as well as the new storage system). Such an implementation of the quota facility poses no logical problem: the quota information would be stored in directories and managed by directory control; a "page fault" for a page with no disk address would be turned into a "quota fault" handled by directory control; the quota fault handler would check and update the quota information and would call the page fault handler in page control after having authorized this page to use a disk record.

Practically, however, it would require a large effort since the data structures used in the new storage system would have to be changed, a large number of segment control and page control procedures would have to be modified, and new procedures would have to be written. In addition, it would introduce a high overhead since the "quota fault" handler, invoked quite often, would induce new page faults to do its job while, in the current system, all data needed to do the quota checking at page fault time is carefully kept in core memory by page control and segment control.

For a long range project, or for an experimental system, one certainly should consider reimplementing the quota facility in order to simplify the kernel specification as well as its implementation and certification.

For a short range project, what I am proposing instead, is to retain the present implementation of quota, as in the new storage system, and to retain in the security kernel just enough knowledge about the directory tree structure to guarantee that the kernel would never violate the star property when performing a quota operation.

THE BASIC SYSTEM REQUIREMENTS.

In order to remove directory control from the security kernel the system should be endowed with the following properties:

1. Directories must no longer contain any item needed to implement the segment entity or to enforce the security access rules.

2. Access to directories must be subject to the security access rules as if they were user segments.

3. Directory control procedures must execute in a less privileged mode than the kernel.

4. The kernel must guarantee its integrity without using directories and, in particular, without using the ACL protection mechanism provided by directory control

These four requirements are discussed separately below.

SEGMENT ATTRIBUTES REORGANIZATION.

In the old storage system, the first requirement was far from being satisfied. All segment attributes, regardless of their nature, were stored in directories. For example, the file map of a segment, needed to implement the segment entity, was stored in a directory, making segment control vulnerable to directory control since any directory control procedure could modify the file map.

In the new storage system, segment attributes have been reorganized into two groups. The first group consists of those attributes that directory control is responsible for, such as the symbolic names, the access control list, the ring brackets; these attributes are still stored in the branch. The other group consists of those attributes which are needed by segment control and page control to implement the segment entity and the quota facility, as well as the security attributes; these attributes are stored in a data base manipulated exclusively by segment control procedures. This data base is called the VTOC (Volume Table of Contents). There is one VTOC for each disk, describing all segments stored on the disk. The VTOC for a disk is stored in the disk itself, at a conventional location, and consists of an array with one entry for each segment residing on the disk. Each VTOC entry contains the following segment attributes:

- the unique identifier
- the security access class
- the file map
- the current and maximum segment length

- the dates the segment was last used and modified
- the directory switch (*)
- the unique identifier of the parent (*)
- the number of records used (*)
- the quota information (if directory) (*)
- a few other items needed by the salvager

All items marked with an (*) are used by segment control or page control procedures that deal with quota. If the quota facility was entirely implemented by directory control procedures, these items would be moved to the branch (or to some other non-kernel data base), and segment control would totally ignore the existence of directories.

The basic system requirement 1 is entirely satisfied by the new storage system, due to the way segment attributes are split between the directory branch and the VTOC entry.

ACCESS TO DIRECTORIES.

The second requirement states that access to directories must be subject to the security access rules. What is meant by "access" is direct access, hardware access through a segment descriptor word (SDW). A process should never be permitted to modify even a single bit of a directory if the classiciation of the process is not equal to the classification of the directory. A process should never be permitted to read even a single bit of a directory if the classification of the process is not greater than or equal to the classification of the directory.

In the old storage system, there were many instances where a process could not perform under these restrictions:

A process of any classification had to be able to write in a directory of any classification in order to deactivate a segment. The new storage system handles the deactivation of a segment without reading or modifying or locking or having to know anything about the parent.

A high classification process had to be able to modify lower classification directories in order to activate a high classification segment. The new storage system handles the activation of a segment without modifying any bit of any directory.

A high classification process had to be able to modify a lower classification directory in order to lock it since the lock was a word of the directory. In the new storage system, the lock of a directory does not reside in the directory therefore locking and unlocking a directory does not require modifying the directory.

One could probably find other examples where the security access rules had to be violated when accessing a directory. In the old storage system, a simple analysis of the various items stored in a directory would show that they can be classed into three categories, with respect to security: first, those with the same classification as the directory itself, such as the header, the names, the ACLs; second, those with the same classifications as the segments they refer to, such as the size of the segment, the time it was modified, the quota information; and third, those with no classification at all such as the file map, the AST entry pointer. Because of the heterogeneousness of the information recored in a directory, some of the directory manipulations could not be done without reading and modifying the directory regardless of its classification. In the new storage system all items stored in directory are exclusively of the same classification as the directory itself. All items that were not of the same classification as the directory have been eliminated and, as a result, all directory operations that required violating the security rules have been eliminated. Therefore, subjecting directory control to the security access rules would still give all processes enough access to perform all operations they are supposed to perform, and would guarantee that security could not be compromised by a malicious or erroneous directory control procedure.

The basic system requirement 2 is also satisfied by the new storage system.

DIRECTORY CONTROL IN RING 1.

It is clear that, for this proposal to make any sense at all, directory control should not be able to change the kernel. The most natural way in Multics to protect the kernel from directory control is to use the ring mechanism. The kernel would execute in ring zero and directory control in ring 1. This means that all directories would reside in ring 1 and all directory control procedures would execute in ring 1. (The current ring 1 would be moved into ring 2, curently empty).

All supervisor ring 0 gates such as hcs$xxx entry points would become ring 1 gates which may, in turn, call upon a new set of kernel ring 0 gates.

The kernel must present an adequate interface to directory control in the form of a set of kernel primitives to create segments, to delete them, to truncate them, to manipulate those segment attributes which are relevant to directory control but stored in the VTOC entry (such as the quota), to assign segment numbers and to manipulate the access fields of segment descriptor words. The list of these kernel primitives is given in one of the next paragraphs.

The basic system requirement 3 is not satisfied by the new storage system, of course. However, the new storage system provides a very good framework for implementing it since segment control has been made functionally independent of directory control. As a result, the new storage system already exhibits the exact modularity one would expect the system to have for moving directory control into ring 1. In fact, each of the kernel primitives described later is already available as a separate ring 0 procedure.

KERNEL INTEGRITY.

In the current system, the integrity of the kernel is achieved by using the ring protection mechanism and the ACL mechanism. The ring information, as well as the ACL information are stored in directories and manipulated by directory control primitives.

If directory control is no longer part of the kernel, access to all kernel segments must be determined by using information recorded in a kernel data base and manipulated by kernel procedures. The protection mechanism needed to protect kernel segments does not have to be as flexible and sophisticated as the ACL mechanism used to protect user segments because, in most cases, all users are given the same access rights to a given kernel segment, and also because these access rights are not likely to change as often and as freely as the ACL for user segments.

The protection mechanism I am proposing is only one of many possible mechanisms; it is less flexible but simpler than the ACL mechanism and can be described as follows: A kernel segment always has the same ring brackets; it has a single standard access mode used for standard processes, and a single privileged access mode, associated with a single privileged process key, used for privileged processes that have requested and obtained this privilege key at login time. The privileged access mode would allow a trusted system process to call special kernel gates that are not available to normal users. These privileged processes would have to request that a given key be associated with them at login time. This request would be validated the same way the user name, project name, and access class are validated at login time.

This mechanism can easily be implemented in the new storage system. The VTOC entry of a segment could indicate whether or not the segment is a kernel segment. If it is a kernel segment, the access control information would be found in the VTOC entry and would be used by the kernel to manufacture the access field of the segment descriptor word for that segment.

KERNEL PRIMITIVES.

This paragraph provides the list of the primitives the kernel should make available to directory control, in order to make it possible for directory control to perform those functions it is responsible for. All these primitives already exist in the new storage system but they are not implemented as kernel gates and they do not perform any security checking. The purpose of this paragraph is to select from the set of segment control procedures those which need to be a gate, to give a short description of what their functions are, and to give a complete description of the security checking they are responsible for. Special gates available to only privileged processes, such as "declassify", are not relevant to this discussion and have been omitted.

The following notation is used in this paragraph:

ERR = error
s = segment number
uid = unique identifier
cl (process) = clearance of the current process
cl (uid) = classification of the segment defined by uid
par (uid) = parent of the segment defined by uid

1. create (parent_uid, dirsw, access_class) returns (uid)

This procedure creates an empty segment (i.e., a VTOC entry) with the access_class defined by the input argument "access_class". The parent of the created segment is defined by its unique identifier "parent_uid", and the directory switch "dirsw" defines whether or not the created segment is a directory. A new unique identifier is assigned to the created segment and its value "uid" is returned to the caller.

ERR if parent_uid does not denote a directory
ERR if cl (process) $\neq$ cl (parent_uid)
ERR if (cl (process) $\leq$ access_class) = false
ERR if dirsw = 0 AND access_class $\neq$ cl (parent_uid)

2. delete (uid)

This procedure deletes the segment (i.e., the VTOC entry) defined by its unique identifier "uid".

ERR if uid not found or if it denotes a kernel segment
ERR if cl (process) $\neq$ cl (par (uid))
ERR if cl (par(uid)) < cl (uid) AND segment to be deleted is not empty

This primitive provides a "write-down" channel when deleting an upgraded directory. This channel also exists in the current system and is not due to the fact that directory control is outside the kernel. It could be eliminated by

making deletion of upgraded directories a trusted process function.

3. truncate (uid, n)

This procedure truncates the segment defined by "uid", from the word number "n".

ERR if uid not found or if it denotes a kernel segment
ERR if cl (process) ≠ cl (uid)

4. give_quota (uid, q)

This procedure delegates an amount of quota equal to q (q>0) to the directory whose unique identifier is "uid", from its parent.

ERR if uid does not denote a directory
ERR if cl (process) ≠ cl (par(uid))

5. return_quota (uid, q)

This procedure returns an amount of quota equal to q (q>0) from the directory whose unique identifier is "uid" to its parent.

ERR if uid does not denote a directory
ERR if cl (process) ≠ cl (uid)
ERR if cl (process) ≠ cl (par(uid))

6. read_vtoce_item$XXX (uid) return (v)

This procedure has one entry point for each item XXX located in the VTOC entry, that directory control may have to know the value of. The entry point XXX reads the item XXX from the VTOC entry defined by its unique identifier "uid", and returns its value "v" to the caller.

ERR if uid not found
ERR if (cl(process) $\geq$ cl (uid)) = false

7. write_vtoce_item$XXX (uid, v)

This procedure has one entry point for each item XXX located in the VTOC entry, that directory control may have to change the value of. The entry point XXX selects the VTOC entry defined by its unique identifier "uid", and assigns the value "v" to its item XXX.

ERR if uid not found or denotes a kernel segment
ERR if cl (process) ≠ cl (uid)

8. assign_segno (uid) returns (s)

Assigns a new segment number "s" to the segment whose unique identifier is "uid", and returns the value "s" to the caller.

ERR if uid not found
ERR if (cl (process) $\geq$ cl (uid)) = false

9. release_segno (s)

Makes segment number "s" invalid in the current process.

ERR if segment number "s" has not been assigned by the assign_segno primitive in this process

10. give_access (s, mode, rings)

Sets, in the segment descriptor word for segment number "s", the ring brackets to the values specified by "rings", and the access field to the mode specified by "mode", adjusted according to the star property.

ERR if segment number "s" has not been assigned
ERR if any ring numbers specified by "rings" is zero
ERR if segment defined by "s" is a kernel segment

11. revoke_access (uid)

Revokes any prior access that has been granted to that segment by the "give_access" primitive in any process.

ERR if uid not found or denotes a kernel segment

The first time a process references a segment to which access has been revoked, an "access_must_be_recomputed" fault occurs, transferring control to the "recompute_access" procedure in directory control. This procedure recomputes the access and calls the "give_access" kernel primitive to set the access bits in the segment descriptor word.

12. lock_directory (uid)

Performs a P operation on a binary semaphore associated with the directory defined by uid.

ERR if uid does not denote a directory
ERR if (cl (process) $\geq$ cl (uid)) = false

13. unlock_directory (uid)

Performs a V operation on a binary semaphore associated with the directory defined by uid.

ERR if uid does not denote a directory
ERR if (cl (process) $\geq$ cl (uid)) = false

CONCLUSION.

If the decision is made to implement this proposal, the short range project would consist of (a) giving informal (but precise) specifications of the kernel functions, (b) using these specifications to get a good level of confidence that they represent the mathematical model for security, and (c) modifying the Multics new storage system as proposed in this document.

The long range project would consist of (a) giving formal specifications of the kernel, without refering to directories at all, not even to the tree structure, (b) proving that these formal specifications represent the mathematical model and (c) implement a kernel that meets the formal specifications.

I am very thankful to Jerry Stern for the long discussions he had with me and for his valuable comments and criticisms.