PROJECT MAC 08/07/74

Computer Systems Research Division Request for Comments No. 58

A KERNEL FOR ACL-BASED PROTECTION MECHANISMS

by Douglas H. Hunt

This RFC explores a specification and part of the interface of a system kernel which can support an ACL-based protection mechanism, such as the Multics mechanism. The proposed kernel does not implement any naming hierarchy, or directories, or even names of segments. Implications of building a system like Multics on such a kernel are explored.

Introduction and Definitions

System certification involves determining whether or not a set of software modules satisfies a specification, or set of properties. The difficulty of certifying a software system is positively correlated with the complexity of the specification. That is, the more properties one must prove about a system, the more difficult the proof procedure is likely to be. Therefore, there is an incentive to limit the scope of the specification, in order to ease the certification process.

A protection specification, in particular, may focus on those objects that are protected by the system and the means for accessing them. Finding a reasonable protection specification is a very difficult problem. A specification such as "the system is secure", while able to be stated simply, is not well-defined. A specification which states that "information flow along control paths must be prohibited" is an example of a specification which is too all-encompassing, since researchers have realized the impossibility of solving the confinement problem. On the other hand, a specification like "the active process table cannot be directly modified by users", while more specific and understandable, is of such a trivial nature that one is unwilling to trust one's private data to a system with such a specification. Accordingly, the first step in the system certification process is to arrive at a specification which is complete enough to be useful, but not so over-constrained that either (1) any system which satisfies it is too restrictive, or

(2) techniques for verifying the system with respect to the specification do not exist.

In order that any non-trivial property be true of a system, a certain amount of (software) mechanism must exist to implement that property. The abstract objects and information channels referenced in a specification are, after all, software constructs. As an example of a property of a system and the software modules which support it, consider the following informal specification about the Multics supervisor:

> The system will not add a segment to a user's address space, nor even inform him of its existence, unless his attributes are in agreement with the access attributes of the segment.

By inspecting some of the lowest levels of the Multics supervisor, say traffic control and page control, one cannot affirm or deny the specification since access to segments is controlled by higher-level modules. By inspecting the entire supervisor (in its current form) one obtains superfluous information, since the action of such supervisor modules as the dynamic linker should not affect the specification. The relevant set of modules to inspect are those "up to and including" the modules responsible for segment control.

If an operating system can be viewed as a collection of (partially) ordered levels, then any object can be considered to be implemented by some collection of levels together with all levels below them. The higher levels do not affect the specification. Given a security specification--a collection of properties--there is a nested family of system modules which, by hypothesis, satisfy it. The certification approach involves determining the least member of this family, and attempting to show that it satisfies the specification. A member of this family is the least member in the sense that ignoring levels which are not in it does not prevent the specification from being proven, while failing to consider any level internal to the member makes it no longer possible to prove at least one of the properties in the specification. This least member of the family will be referred to as a kernel of the operating system, with respect to the security specification. The concept of a kernel is meaningful only when taken together with a specification about the operation of the system; i.e. for each distinct specification there is a distinct kernel. The advantage of attempting to verify the collection of modules which compose the kernel, rather than any of the members of the nested family which contain it, is that the relative effort is likely to be minimal.

A Trial Specification

One goal of this research is to arrive at a specification of a system which protects objects by means of access control lists. Consequently, the ACLs themselves would be kernel-implemented objects. The kernel, which is assumed to be built on top of harcware similar to the current Multics hardware base, should provide sufficient facilities to support a system such as Multics, or another ACL-based system. It is the author's contention that an adequate protection specification can be formulated, such that the kernel need not provide many of the interfaces now provided by Multics directory control.

The first task at hand is then to produce an adequate specification for a protection mechanism. Since there are advantages in keeping the specification as simple as possible, the specification will not deal with control paths in the system, nor with denial of service issues. (1) At this point, there has been only modest effort devoted to choosing a particular specification, since there is a potentially infinite variety of specifications which may seem appropriate for given user communities. The intention is to choose a rather rudimentary specification, one which is probably a subset of a larger custom-tailored class that various users will desire. Having arrived at a specification, this proposal will focus on its implications -- i.e. whether Multics-like systems can be built upon it -- and also on verification methodologies which can be applied to it [Ne74]. It is hoped that the verification methodologies can be extended to kernels satisfying other specifications.

The proposed specification will deal with a particular information repository -- the segment -- and the methods by which the segment and its attributes can be accessed. The first attempt at an informal specification will include four properties describing access to segments. The first of these properties has

(1) For example, even though the system will return the same message for "not found" and "no access" situations, a penetrator may be able to distinguish between them by timing tests. The penetrator is said to be exploiting a control path in the system.

already been stated in the above example. The other trree are the following:

> The system will not allow a user to reference the access attributes of a segment, nor even inform him of their existence, unless his attributes are in agreement with the access attributes of the access attributes of the segment. (1)
>
> The system will not allow a user to reference the access attributes of the access attributes of a segment, nor even inform him of their existence, unless the user is the locksmith of the segment.
>
> The locksmith attribute of a segment is specified at segment creation time, and cannot be changed.

The locksmith attribute denotes (the process of) the agent who is ultimately responsible for the segment. These properties provide for a limit on the number of levels through which one can recurse before locating the agent ultimately responsible for enforcing access to a segment. The primary reason that this particular specification is given is that these access attributes are self-defining: only the attributes of the desired segment need be examined to determine if it (or any of its attributes) can be referenced. A fundamental difference between this sort of localized authority and the authority hierarchy which is a part of the Multics file system is that in the first case, each segment can potentially have a different locksmith, whereas in the latter case there is at least one individual who is the locksmith for every segment in the system [Ro74]. Since in the

---

(1) The requirement that attributes be in agreement is deliberately left a bit vague at this point. Depending on the kinds of access rights and the rules for comparing them, there can be many implementations which satisfy this property.

proposed scheme all access attributes are associated with the segment, there is no concept of access attributes which are a part of a parent directory. Hence this specification decouples access authority from a naming hierarchy.

Unfortunatley, this specification describes an access control mechanism which is really too inflexible for general application. Administrative control over access is needed in order to remedy problems such as (1) persons who leave a project but who fail to transfer access control rights to shared segments, (2) persons who fail to delete obsolete segments, (3) system errors which result in mangled ACLs, and (4) user errors with ACLs, in which a user accidentally removes a certain access mode and then cannot re-create it. An access control interface such as the one described here can be designed to eliminate the fourth calss of problems, so that the user cannot wipe himself out. However, the first three problems persist. One characteristic of the three problems is that they need to be dealt with only infrequently. Therefore, authority centers, such as the "offices" described in [Ro74], could be invoked to deal with them.

A complete specification of an access control mechanism would describe not only the use of ACLs, but also the use of authority centers in the system. Each segment, with an ACL mechanism like that described here, would be subordinate to some office which is named at segment creation time. An office may have very limited control over a segment; for example it may be able to delete it after a time interval has expired, or after a user is no longer registered on a project. With an augmented specification which includes offices, the ACL mechanism can still operate in much the same manner as described here. Therefore, even though a specification for authority centers is not offered here, some implications of a system in which access authority and naming are decoupled can be explored.

Of course, properties describing the means by which segments can be accessed would form only a part of a reasonable protection specification for an operating system. For example, properties of the terminal I/O mechanism must be stated. Users need to be assured that the characters displayed on their terminals are the same (modulo any documented line control conventions and canonicalization) as those seen by the system. Secondary properties, such as statements regarding how much influence users can have over resource allocation, could be included in the specification. However, this is probably not appropriate. Rather, a high-level specification should emphasize properties of the resources that the operating system makes available to users. Hence, the system makes available a variable number of pseudo-processors, each with a virtual memory composed of a number of segments. The segments are the only objects in the system which can be protected by user-controlled access attributes. These remarks, together with the four properties about segments, will serve as the (very) informal top-level specification. As will be seen, this specification is sufficient to characterize a kernel which can support Multics-like systems. Of course, a high-level specification such as this one may be p.ogressively refined, in the structured programming sense, to obtain more specific properties.

A Kernel Interface for Accessing Segments and their Attributes

Given current hardware architectures, it is evident that procedures for terminal management, processor multiplexing, and memory management will be contained in the kernel corresponding to this specification. The kernel will also contain modules which manage segments and their access attributes. A major purpose of this research is to specify more carefully those parts of the kernel which would correspond to Multics segment control and directory control functions. Therefore it is reasonable to ask which functions the kernel must make available in order to conform to the four properties concerning access to segments.

A possible implementation of the segment and directory control functions necessary to support the specification will now be described. Each segment will have associated with it a number of access control attributes, such as an ACL, domain names (or ring brackets), and an access class (or classification). The specification describes how access is granted not only to segments but also to the access attributes themselves. Assuming that access classes last the lifetime of the segment and that the setting of domain names depends on rights in an ACL entry, the mechanism for controlling access to the access attributes boils down to controlling access to ACLs. There are a number of different modes or rights available on an ACL; some of these, such as read, write, and execute, can be interpreted directly by the hardware. There is another ACL , containing "status" and "modify" modes, which controls access to the first ACL. The modes on the second ACL correspond to the "access attributes of the access attributes" mentioned in the propositions above. To avoid confusion, the first ACL, which governs references to the segment, will be called the "reference ACL" and the second ACL, which governs references to the reference ACL, will be called the "administrative ACL." Although the reference ACL and the administrative ACL may be implemented as a single list, they are conceptually separate entities. Each segment also has a locksmith attribute, which looks like a process principal identifier. The access rules should now be evident. A process can add a segment to its address space only if its name, i.e. the process group id, appears on the reference ACL. It can access the reference ACL only if its name appears on the administrative ACL. A process can access the administrative ACL only if it is the process of the locksmith.

A kernel which satisfies the four properties need not implement a hierarchy, or directories, or even names of segments. The kernel will manage access attributes of segments, and will refer to segments by unique id (UID). A hierarchical scheme for reflecting quota usage and other accounting attributes would be discarded, in favor of the ability to associate a segment with a particular accounting pool at segment creation time. The three most important reasons for a hierarchy seem to be to provide (1) a naming structure, (2) an access control "chain of command," and (3) accounting pools. From the point of view of the kernel, each of these needs has been decoupled from a hierarchy, so the kernel need not implement one. Of course, a hierarchical naming structure is very useful and can be implemented outside the kernel using the kernel primitives. For example, a simple directory implementation would be a segment full of name and UID pairs.

The proposed kernel interface offers a number of advantages, when compared to a system kernel which maps access authority onto a naming hierarchy. First, there is the issue of ease of verifiability, which has already been stressed. Since the proposed interface manages simpler objects and is easier to describe, there is good reason to believe that the verification effort will be less. It is worthwhile noting that the Case model describing a simple kernel with no segment hierarchy is also quite simple [Wa74]. Secondly, some protection anomalies vanish under the proposed scheme. One would like to state that only the access attributes of an object are relevant in the access-granting decision. Since Multics wishes to adhere to this rule, it is possible to add directories to the address space of a process, even though the process has no access rights to them. With the proposed kernel, it is always possible to add a segment to an address space without consulting any intervening directories, since segments are referenced by UID. Third, the proposed kernel interface allows users to construct arbitrary directory structures. A user can implement his own hierarchy, invent his own segment attributes, or implement a set of directories which are not tree-structured. Fourth, one may implement not only directories, but other higher-level objects as extended type objects supported by the kernel. The kernel would associate each extended object type with a domain which supports it. This particular facility is not explored further in this RFC.

Before describing the part of the kernel interface which manages segments and address spaces, the segment attributes which the kernel must manage should be itemized. Since the kernel is intended to support Multics-like systems, all attributes in a Multics segment branch are potential candidates. First, the kernel must certainly maintain the file map and the device id of a segment. These attributes are invisible to the user. Secondly, the kernel will manage all access attributes of a segment. These are the reference ACL, the administrative ACL, ring brackets, access class, locksmith, call limiter, and maximum length. (For purposes of illustration, ring brackets are referred to in the description below. In a more general system, they would be replaced by domain names.) Third, the kernel will still gather data for accounting and backup purposes, since this data is accumulated by the paging software. These attributes are date-time-used and date-time-modified, records used, current length, and time-page product.

The kernel interface for managing segments would consist of functions to create and delete segment objects, functions to inspect and modify each access attribute of a segment, and functions to display the accounting attributes. The entry points and their arguments are shown below.

1. create_segment(locksmith, admin_acl, ref_acl, ringbr, acc_class, call_lim, max_length, acct_name, uid, status)
2. delete_segment(uid, status)
3. list_admin_acl(uid, output_structure, status)
4. change_admin_acl(uid, old_entry_structure, new_entry_structure, status)
5. list_ref_acl(uid, output_structure, status)
6. change_ref_acl(uid, old_entry_structure, new_entry_structure, status)
7. list_ringbr(uid, output_array, status)
8. change_ringbr(uid, input_array, status)
9. list_acc_class(uid, acc_class, status)
10. change_acc_class(uid, new_acc_class, status)
11. list_call_lim(uid, call_lim, status)
12. change_call_lim(uid, new_call_lim, status)
13. list_max_length(uid, max_length, status)
14. change_max_length(uid, new_max_length, status)
15. list_acct_info(uid, dtu, dtm, cur_length, rec_used, time-page)

Most of the parameters have already been described. The "acct_name" parameter in the "create_segment" entry is used to specify an accounting pool, fixed for the lifetime of the segment, with which the segment is to be associated. In a protection specification which included offices, the accounting pool would be an office instead, and the (previously negotiated) access modes for the office would also be specified in the "create_segment" call. Of course, many of these parameters have default values. For instance, the default value of the locksmith is the process group id of the creating process. These entries are listed separately for purposes of discussion; in an implementation, some of them might be combined. Some of them, for example the entry for changing an access class, may be available only through a privileged gate. Two other entries which could be made available are an entry to get the effective access mode and an entry to truncate a segment. However, these need not be in the kernel.

The kernel interface to manage address spaces would consist of entries to allocate and free segment numbers, and a fault handler to generate a segment descriptor for a process, assuming proper access mode. The call side consists of the following two entries.

1. initiate(uid, segno, status)
2. terminate(uid, status)

The initiate entry merely allows the kernel to associate a UID with a segment number; there is no access checking. The address space manager is also invoked to handle a segment fault. The segment fault handler will map the faulting segment number into a UID, using a per-process table, similar to the Known Segment Table in Multics. (1) The segment fault handler can map the UID into the segment attributes and file map using a paged, system-wide UID mapping table. If the access comparison is successful, a SDW is connected to the AST entry for the segment faulted upon.

Building Sophisticated Systems on the Proposed Kernel

The next section of this proposal is devoted to some of the implications of providing a kernel with the interface described. First to be explored are the consequences of implementing directories, i.e. name-to-UID bindings, outside of the kernel. To begin with, since names are implemented outside the kernel, the binding of names to UIDs cannot be guaranteed by the kernel. The kernel deals with segment UIDs, but users rely on names both for their private use and for describing shared information. Even the system software may rely on a naming structure to establish "home directories" for users. Thus, name-UID bindings must not be changed without the knowledge of those relying on them. A user who changes a name-UID binding potentially has the same power as the locksmith of the segment, since he may trick the locksmith into performing the "right" operation on the "wrong" segment. One who references a segment via a directory may eliminate the possibility of unauthorized renaming by making

(1) This description will suffice for a segment fault handler which is invoked synchronously in a process, or a handler which runs in another process. The first phase of the segment fault, mapping the segment number into a UID, can be carried out in a separate process if that process can access the "segno ---> UID" mapping table in the faulting process (by absegs, for example), or if there are system-wide segment numbers.

his directory read-only to others. On the other hand, he can allow renaming by selected parties by giving them write access to a user-domain directory or rename access to a privileged-domain directory. An important observation here is that these controls on renaming are analogous to those currently available on Multics. This proposed kernel interface does not present any problems for name-UID binding that do not already exist in the Multics file system.

In a system which supports private hierarchies, there is a need for obtaining UIDs of the segments which comprise the initial process environment. In addition, there is a need for obtaining UIDs of shared information made available by other users. Therefore there must be repositories of UIDs which processes can use to bootstrap their environments. As a result, many installations may use file system hierarchies, but the hierarchies will be implemented outside the kernel. It is possible to build a naming hierarchy out of segments which can be referenced in a user ring (domain). The top few levels of the hierarchy would be readable to everyone, but writeable only to system administrators. Thus the same mechanism which allows read-only segments can also preserve a binding of names to UIDs.

Users who wish to ensure the integrity of the name-UID bindings which they use have the option of constructing their own directories and directory- management procedures. They need to verify correct operation of only their own directory-management procedures, and no others. Of course, they also rely upon a kernel which operates as specified.

User-designed directory management subsystems may well be designed to run in a more privileged domain in order to implement useful forms of interpreted access. For example, append access could be implemented in such a privileged-domain subsystem. Also, locking (for the sake of consistency) would require interpreted access provided by such a subsystem, since a user who has only status access to list segment names must nonetheless set and reset the lock. (The locking problem can be eliminated by allowing writers to increment version numbers, and by allowing readers to inspect them.) A privileged-domain directory subsystem would require more verification effort than a user-domain subsystem, since it implements forms of interpreted access to the directory segments.

The management of address spaces will not differ greatly from the way that function is currently performed in Multics. The operation of initiating a segment will be reduced to its bare essentials: associating a segment number with a UID. The role of initiation is just resource allocation (of segment numbers); there is neither access checking nor interpretation of UIDs. At segfault time, the existence of a segment with a given UID and proper access modes will be determined. The checking of access at segfault time is preferable to checking at initiate time, since in steady-state system operation there is a possibility that access may need to be re-calculated on any segment reference. Since the segfault mechanism must be able to calculate access, it should also be invoked on the first reference to a segment by a process. There is, however a drawback to this scheme. The user cannot use the initiate entry in the way that the "open file" operation on other systems is used -- to ascertain access rights to a segment before referencing it. Thus, an error in naming a segment could abort a computation part way through a critical section. Of course with immediate access revocation such an event could always occur, but adopting the present proposal may increase the likelihood of such an occurrence. One solution to this problem is to add an "effective mode" entry to the kernel, to return effective mode for a segment. Prior to entering a critical section, a process could calculate its effective mode to each segment about to be referenced.

The address space manager must also construct segment descriptors for "attribute segments." The attribute segments are repositories which contain attributes of segments. (Whether or not an attribute segment will contain its own attributes is still unsettled.) The reason that segments are chosen to contain attributes of segments is that segments are good places to store variable-length information, such as ACLs. Attribute segments are invisible to users, except that they require entries in the descriptor segment of the process, just as any other segment does. Alternatively, the descriptors for attribute segments could be kept in a separate descriptor segment, accessible only from kernel domains.

In this design, directory managing procedures and a dynamic linking mechanism would both be regarded as non-kernel procedures. Consequently, users can potentially supply their own versions of either of these two facilities. In order that a system default linker can rely on a user-supplied directory manager, these subsystems must cooperate through a standard interface. The point is that users must observe certain protocols in writing their own non-kernel subsystems, just as a Multics user writing his own command must do presently.

The proposed supervisor interface has a major effect on the operation of a backup facility. Backup cannot operate on a system-protected naming hierarchy. Rather, it must work with the UID's of segments. The interface to the dumper and the retriever will take a segment UID (or a list of them) as an input argument.

An issue to be explored here is whether or not any part of the backup function must take place within the kernel. The backup dumping process, for example, must be informed about any segments which have been modified in the recent past. If segment dumping is to take place only on user request, the dumper does not need any special kernel entry. However, if automatic dumping is desired, then it is preferable to allow the dumper to invoke a kernel entry to obtain a list of recently-modified segments. Obtaining this list is the only special privilege the dumper requires; otherwise it is subject to normal access controls. It cannot dump a segment unless it appears on the appropriate ACL.

The backup function for retrieving segments from removable media should not require any special kernel interface. A user retrieval request names a segment or collection of segments. This set of names is mapped into a set of UIDs by a directory-management subsystem. The set of UIDs is in turn passed to the retriever.

All references by the backup facility to on-line segments are subject to the access rules which have been described. However, even though the backup process is subject to the access rules enforced by the kernel, the backup procedures must be verified to insure that they effect an accurate, memoryless transfer of information between on-line segments and the removable media.

By drawing rather firm modular boundaries around some of the lower-level system functions, a system built on the proposed kernel can exhibit a rather high degree of information hiding [Pa72]. This may result in performance degradation. For example, the Multics kernel is able to save on storage for ACL names of segments which it knows are in the same directory. The proposed kernel could not make such assumptions. The hope is that a reasonable modularization will greatly facilitate kernel verification, while not significantly affecting performance.

Implications for System Verification

A system built on this kernel will implement directory objects at a higher level of abstraction than segment objects. What is really important is that the way in which the implementation of directories depends on segments should be easy to describe. Since the domain which implements directories cannot affect the domain which implements segments, auditors can verify the operation of directory-management software, assuming that the properties of ségments are true.

Isolating components of an operating system in a reasonable manner seems to be a promising approach to the problem of verifying its operation, with respect to a specification. This proposal sketches how a naming hierarchy for segments can be constructed upon a particular kernel. As a result, certain system functions have been isolated in a well-defined way. This same technique for isolating functional units can be applied to the kernel itself. For example, the part of the kernel which manages ACLs is at a lower level of abstraction than the part which handles segment faults.

Further work in this area can proceed along several fronts. First, more complete security specifications can be formulated. Secondly, there needs to be a more complete understanding of how parts of a system depend on each other. Finally, there is the problem of matching a kernel to its specification. A piece of the system must be identified, which is both necessary and sufficient to support the specification.

References

1. [Ne74]. Neumann, P. G., Fabry, R. S., Levitt, K. N., et. al., "On the Design of a Provably Secure Operating System," Stanford Research Institute Computer Science Group, Menlo Park, California.

2. [Pa72]. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM, December, 1972, pp. 1053,1058.

3. [Ro74]. Rotenberg, L. J., "Making Computers Keep Secrets," Project MAC TR-115, February 1974.

4. [Wa74]. Walter, K. G., Ogden, W. F., Rounds, W. C., et. al., "Primitive Models for Computer Security," Air Force Electronic Systems Division Technical Report ESD-TR-74-117, January 23, 1974.