PROJECT MAC

Computer Systems Research Division

July 13, 1973

Request for Comments No.33

DIFFERENT SMALL IMPROVEMENTS TO ACCESS CONTROL IN MULTICS

by Victor L. Voydock

This RFC presents some new proposals for solving the ACL formation and modification problems discussed in RFC-27. It is intended to be a stimulus for discussion and not necessarily a finished design proposal.

Different Small Improvements to Access Control in Multics

by

Victor L. Voydock

This RFC proposes an alternate solution to the ACL formation and modification problems discussed in RFC-27. By adopting a somewhat different point of view, a simpler solution is possible. It will be argued that the "-p" principal id component is unnecessary to solve the default ACL term problem, and that the standard mode concept which attempts to solve two quite different problems (and is therefore hard to understand and explain) can be replaced by two separate but much simpler mechanisms. Before proceeding, let us list the goals we wish to achieve:

1. Simplify the rules by which the first ACL* of a segment is formed.
2. Allow the contents of the first ACL to be influenced by the type of the segment being created.
3. Allow the ACL of a segment to be suspended while the segment is being modified, and then restored.

First ACL Formation

As pointed out in RFC-27, the first ACL is formed in a somewhat obscure manner. RFC-27 introduced the "-p" principal id component and the standard mode to simplify this formation and to achieve goal 2. This was a case of overkill. A simple first ACL formation method which achieves both goals 1 and 2 is as follows:

The initial ACL will have terms of the form:

<principal id class> <potential access> <absolute access>

When a segment is created, the <potential access> field of each initial ACL term is ANDed with the access argument supplied to the segment creation subroutine. The <absolute access> field is unaffected. The resulting ACL becomes the first ACL of the segment.

---

* By "first ACL" we mean the ACL which is on a segment when control returns from the system subroutine call which created it.

For example, a typical initial ACL (where capital letters indicate absolute modes) might be:

Voydock.*.*	rwe
.SysDaemon.	RW
..*	re

If "re" were specified in the segment creation call then the first ACL would be:

Voydock.*.*	re
.SysDaemon.	rw
..*	re

If "rw" were specified, the first ACL would be:

Voydock.*.*	rw
.SysDaemon.	rw
..*	r

Thus the first ACL is completely specified by the initial ACL and the modes specified in the segment creation call. (Note that the rule which caused a *.SysDaemon.* entry to magically appear on every ACL has been removed. It is cleaner to require that this entry explicitly appear in the initial ACL).

Note that the ANDing of access modes takes place only once - when the segment is created. Thus it is easier to understand and explain than the standard mode which is dynamically ANDed with ACL terms each time access is computed.

RFC-27 introduces the "-p" principal id component and seems to imply that it is needed to solve the default acl term problem (defined in RFC-27). This is not the case. The problem comes about because an ACL entry over which the user has no control (when using system commands) is added to the new segment's ACL by the segment creation primitive. The problem disappears if this primitive is changed to not add this ACL entry. Both this paper and RFC-27 suggest this change. No additional mechanisms are needed to solve the problem. The -p idea does provide additional flexibility. For example, an initial ACL of the form:

```
Voydock.CompSys.-p   rew
*.SysDaemon.*        rew
*.*.*                re
```

would allow multiple protection compartments (see RFC-21) to share a single directory. It is not clear that this flexibility is needed. So, for the sake of simplicity, the -p idea should not be implemented. It could be added later with no compatibility problems.

ACL Modification

The standard mode of RFC-27 only partially achieves goal 3 (and does it in a very unsatisfying manner). In particular, before recompiling an object segmnent one can change the standard mode to "rw" thus preventing other

users from executing it (but those users will still have "r" access to it). However, if one wishes to modify a data base, there is no way to lock out attempted reads using the standard mode.

A more straightforward way of achieving goal 3 is to introduce a mechanism which gives the calling process the access it needs to modify the segment after removing the segment's current ACL and saving it somewhere so that it can later be restored.

If the system never crashed and processes never terminated in uncontrolled ways then this mechanism could be implemented without changing the file system. That is, the ACL could be saved in and later restored from a process's temporary storage. Since this happy state of affairs does not exist, we must modify the file system to provide a place for saving ACLs*. That is, we change the file system to remember the old ACL so that it can later be restored.** For this we provide two new primitives:

suspend_acl (path, access, code)

restore_acl (path, code)

When suspend_acl is called, the ACL of the segment whose pathname is path is suspended. That is, its location is remembered in the branch but it is no longer used to compute access. It is replaced by an ACL with a single term naming the principal of the calling process with an access field as

---

* One could save a segment's ACL in a segment stored in the user's permanent hierarchy but this seems basically unaesthetic and the idea will not be pursued here.

** The primitives discussed below are functionally complete but do not necessarily provide the optimal user interface. Further study is needed to determine exactly what this interface should be.

specified by the access argument. An attempt to suspend an ACL which is already suspended will not be allowed. An appropriate error code value will be returned. An attempt to use other ACL manipulating primitives on a suspended ACL will also result in the return of an appropriate error code value.

When restore_acl is called, the single term ACL placed on the segment by suspend_acl is replaced by the suspended ACL. To apply suspend_acl and restore_acl to a segment, modify access is needed in that segment's parent directory.

If the system crashes while an ACL is suspended, no harm is done. The suspended ACL is preserved in the branch and will be restored the next time the segment is compiled. If the user examines such a segment using one of the status commands he will be told that its ACL is suspended and therefore that the segment is probably in an inconsistent state.

The above two primitives are sufficient if the segment in question already exists. However, there are cases when one wants to create a segment and immediately suspend its ACL. For example, when one is compiling a program for which an object segment does not previously exist. With the current primitives there is a window between the time a segment created and the time its ACL is suspended.* Thus we introduce a new primitive:

make_seg_and_suspend_acl (path, final_access, temp_access, ptr, code)

which behaves functionally as if it indivisibly (i.e., without unlocking

---

* If this is not considered important, the rest of this section can be ignored.

the directory in between) performs the following two calls:

```
call make_seg (path, final_access, ptr, code)

call suspend_acl (path, temp_access, code)
```

These primitives (and the necessary changes to the directory format) would be easy to implement and they achieve goal 3 in a straightforward manner.

The Dynamic Updating Problem

Finally, there is a class of problems not considered by RFC-27. This class is best described by an example -- recompiling a segment while other processes are using it. If a process is executing an object segment when another process begins to recompile it, the executing process will take an access violation fault. This is disconcerting but seldom fatal. A much more serious problem is that when a process uses an object segment, it squirrels away the location of various logical components of the segment, such as entry points. When an object segment is recompiled, the location of these components can change, thus invalidating the saved information in all processes using the segment. Nothing is done to update this saved information (except in the process which performed the recompilation). Thus executing an object segment after it has been recompiled, in a process which was linked to it before it was recompiled (other than the process which performed the recompilation), can lead to unpredictable and potentially disasterous results. A complete solution to this problem, if one exists, is beyond the scope of this paper. A solution which at least causes the process to take a well-defined fault, rather than plunging ahead to disaster is described below.*

---

* One could take the attitude that one should never destroy a segment that is potentially shared, but instead use techniques similar to those used to perform online installations to preserve the old version of the segment. This may well be true, but accidents happen, and it is better to have an accident cause a process to blow up in a well-defined way than cause unpredictable results.

The just described disasterous effects of the dynamic updating problem occur because the system has chosen to truncate a segment which is about to be recompiled rather than deleting it. If the segment were deleted, processes with links snapped to it would take an "invalid segment number" fault the next time they referenced it, instead of continuing with unpredictable results. Truncation was chosen over deletion both for efficiency and because the branch of the trancated segment is a convenient place in which to save those segment attributes which will not be changed by the recompilation. To eliminate the problem and still maintain the convenience of the current mechanism, a new primitive is introduced:

replace_seg_and_suppress_acl (path, final_access, temp_access, ptr, code)

If the segment whose pathname is path does not exist, then this primitive behaves the same as the make_seg_and_suppress_acl primitive above.

If the segment does exist, it is deleted and a new segment is created whose attributes are the same as those of the segment just deleted. The ACL of this segment is suspended by a call of the form:

call suspend_acl (path, temp_access, code)

and a pointer to the segment is returned to the caller.

Thus a caller of this primitive will get a pointer to the desired segment whether it existed previously or not, the ACL of the segment will be suspended, the suspended ACL will be the ACL which is desired to be on the segment when the compilation is complete, and the dynamic updating problem will be "taken care of".

Summary

Goals 1 and 2 were achieved by introducing a simple formation rule for first ACLs. Goal 3 was achieved by introducing the idea of suspended ACL's which allow access information to be saved across crashes and process terminations. These two mechanisms take care of the problems discussed in RFC-27 in a simpler way than the mechanisms of RFC-27. Finally, a mechanism was introduced which reduces the damage caused by the dynamic updating problem.