Memorandum on the

Protection Mechanisms and Policies in the BCC 500 Operating System

Jack Freeman
January 18, 1974

Introduction

We stated in our recent proposal to ARPA that the protection facilities in the BCC 500 Operating System provide the user with means for granting or denying access to such objects as files, processes, resouce allocations, and all other system-defined objects in a unified, flexible way. The protection mechanisms provide for specification of the accesses to be allowed by any single program, process, or user. They also permit any user to define arbitrarily constituted groups of programs, processes, and users, and to specify the kinds of accesses to objects to be allowed by each of these groups.

The purpose of this report is to amplify and extend this general description of the BCC 500's protection facilities. In particular, we will talk about the following sorts of things.

- How a user concerned about the security of his data and programs would use these mechanisms.

- How the rights to access objects which are specified and controlled by the protection mechanisms are moved into execution environments where they can be exercised by programs. And how the programs are constrained to work within these environments.

- Some general characterization of the implementation of the protection mechanisms.

- A discussion of weaknesses and deficiencies of the 500's protection schemes.

This report is not meant to be a thorough description of the security-related features of the BCC 500 system. Such a description will be issued within the next several weeks. The present report is intended as a sort of introduction to the sub-set of these features that explicitly provide for the controlled sharing of information stored within the system.

I. Using the Protection Mechanisms to Achieve Security

The protection mechanisms provided by the BCC 500 Operating System do not and are not intended to in themselves "secure" anything. It is up to the users of the system to decide what restrictions they wish to place upon accesses to the objects they keep in the system*. The set of decisions one user makes can be called his "security policy." The proper function of the protection mechanisms is to give the user a way of stating the policy he decides on and to enforce it.

We want to list some typical security decisions a user might wish to make and to show that these decisions can be expressed as protection rules within the BCC 500's protection framework. To this end we now give a concise but imcomplete description of the 500's basic protection mechanisms.

A part of the (protected) description of every object kept within the system is a list of the entities (users, etc.) authorized to access the object. Each element in the list both names an authorized entity and specifies the kind(s) of access the entity may exercise on the object. On every attempted access to an object, the protection system verifies that the entity attempting the access appears in some element of the object's list and is authorized to make the specific kind of access it is attempting.

Now consider the following decisions a user might wish to make about accesses by other entities to some file, XYZ.

1. No-one is to have any access to XYZ. This is clearly implementable. If there are no entities currently authorized to access XYZ, nothing has to be done. If there are some, the list entrys which authorize them can be deleted.

* The notion of a user and of his relationship of "ownership" to objects will be made more precise where necessary. Here we can think of a human user and some object he naturally regards as his own, such as a text file created by him.

2. Everyone is to have some access (say READ) to XYZ. This could be implemented by putting on the object's list the names of all possible accessing entities, with READ specified as the access allowed each one. This is absurd, and the system of course contains a special provision for this case. Basically, there is a standard element on every list which specifies the access (if any) to be allowed by "the public."

3. An individual user (or whatever) is to have some access (say EXECUTE) to XYZ. This can clearly be implemented by adding to XYZ's list an element naming the user and specifying EXECUTE access.

The above simple examples show the diversity of the levels of "security" which a user may specify for a file or other object. In more complicated cases, the strict limitation to a list of authorized accessing entities would lead to inefficiency or clumsiness. Without enumerating them, we state that the system includes various ways of avoiding these unpleasantnesses, similar to the provision for "public" access mentioned above.

II. Protection During Program Execution

So far we have discussed the protection mechanism from the point of view of the relatively static representation of the protection information, together with a little about how this information is conveyed to the system by a user and with an indication of the significance the information has to the protection-implementing algorithms. We need now to relate this to program execution and to sketch the means by which the authorization to access an object is brought into action and used to actually manipulate the object.

The environment within which a program executes was called by Lampson [L69] a "domain". A domain consists of a collection of "capabilities", which are simply names of objects together with specifications of the accesses which may be made to the objects.

In a typical, simple case a domain might consist of a few EXECUTE capabilities for pages containing the code of a program and of a READ capability for an input file and a WRITE capability for an output file.

Initiating the execution of a program involves creating a domain for it to operate in. In some cases, all the capabilities a program will need are known ahead of time, and its domain can be completely pre-established before control is given to it. For example, if the names of a simple program's input and output files are known, capabilities for READing the first and WRITEing the second can be moved into the domain being created for it. In such a case, it is natural to think of the program as "encapsulated" within the domain established for it. During execution, the program will be restricted to the realm of reference defined by this domain.

Before going on to consider more complicated cases, we must stop and sketch the mechanisms by which capabilities are put into domains.

Recall that every object known to the system has associated with it a list of entities (e.g., users) which are allowed to access it and a specification of the accesses allowed by each entity. For reasons

that will become clear in a moment, the elements of these lists are called "access locks" and the lists themselves are called "lock lists". Associated with every potentially active entity is at least one protected object called an "access key". To move a capability into a domain, a program makes a certain system call. The arguments to this call are

1. The name of the object for which a capability is wanted;
2. A specification of the access(es) wanted for the object; and
3. An access key.

The last argument is presented in the form of a little number which selects a capability in the domain of the program making the system call.

The system call uses the first argument to find the system's internal representation of the named object. In particular, it locates the object's lock list. Then it compares the value of the access key specified by the third argument with the value of the lock portion of each element of the list. Either no lock will match the offered key or exactly one will. In the former case, the system call fails and no capability is acquired. In the latter case, the access specification part of the access lock matched by the calling program's key is ANDed with the calling program's specification of the access it needs, and a capability for the selected object with the access thus computed is moved into the domain of the calling program.

Remember that the goal of the calling program, in the case where it is initializing a domain for another program to run in, is to get the capability into the new program's domain, not into its own. Such a program would next use another system call which implements the transference of capabilities between domains.

The arguments to this system call are:

1. The "name" of the capability to be transferred; and
2. The "name" of the domain to which the capability is to be transferred.

Both of these names are little numbers which select capabilities from the domain of the calling process. The implementation of this system call is such as to allow for a number of modes of transference of the named capability. In particular, the following possibilities exist.

1. The capability may be completely turned over to the target domain, being "erased" from the domain of the calling program;
2. The capability may be shared on an equal basis between the two domains; or
3. The new domain may be given only a subset of the current domain's "rights" to the capability. For example, the new domain might or might not be given the right to pass the capability on to other domains.

Now that we know the basic mechanisms through which capabilities are moved into execution environments and transferred between them, we can show how they may be used to do good things. As an interesting and not overly complicated example, we will consider how one might implement a message-handling program like the TENEX program, SNDMSG.

First we describe some static things. We establish SNDMSG as a system "principle", similar to a human user in that it is an entity possessing an identity known to the system and capable of directing a computation. As was mentioned earlier, every such potentially active entity has associated with it an "access key" which can be used to move capabilities into domains.

So far, we have said nothing about where the representations of protected objects are kept. We note at this time that there are protected data structures called "directories" which serve as repositories for the representations of objects. One way to fit directories in with the concepts mentioned so far is to consider them as sort of "static domains", containing among other things, "static capabilities" which can be moved, as was described above, into execution domains.

Every system principle such as SNDMSG has a directory associated with it. We continue our discussion by enumerating the objects which would appear in SNDMSG's directory. We could get by with only two such objects. Namely the access key assigned to SNDMSG as a system principle and a file containing the code of the SNDMSG program together with instructions about how to place the code in a virtual address space.

Some users of the system may wish to have the service of SNDMSG and some may not. Those that do wish to make it possible for messages to be sent to them will cause to be created in their directory a file with the name MESSAGE.TXT. On the lock list of the representation of this file, the user will put an access lock which matches the access key associated with SNDMSG. The accesses allowed to SNDMSG by this lock would be READ and WRITE. [Note that one really desires to give it APPEND access instead. This could be managed within existing BCC 500 mechanisms, but not in a truly general way.] The only other entry in the lock list of the file would be one giving READ access to the user himself. Thus the user can feel relatively secure that no-one can falsify or otherwise fiddle with his file of messages.

We have as yet said nothing about the accessibility of the objects (an access key and a file) in SNDMSG's directory. In BCC 500 jargon, the file that contains the code of SNDMSG would be called a "program image file", or PIF. All users who are to send messages need to have EXECUTE access to this PIF. Thus the lock list for the PIF might be quite extensive, containing an access lock for every participating user with the specified access being EXECUTE in each case.

Now, some system principle is going to have to be able to create a domain for SNDMSG to run in and to perform the important operation of placing the code pages of the PIF into an address space. This system principle is a program called "the Utility". Like SNDMSG, the Utility has an access key. If we put on the lock list of SNDMSG's PIF an entry which gives READ access to the Utility, this will allow the Utility to get the capabilities it needs to arrange SNDMSG's code in an address space. Finally, if we also put on the lock list of SNDMSG's access key a lock allowing the Utility to move the key into an execution domain, we have

everything we will need to allow the secure use of SNDMSG.

Now suppose user SMITH wishes to use SNDMSG to send a message to user JONES. We assume that SMITH has EXECUTE access to SNDMSG and that JONES has a file called MESSAGE.TXT to which SNDMSG has APPEND access (actually READ and WRITE, as noted above). SMITH is using a program that implements a command processor. This program is operating in a domain which contains some subset, probably small, of the capabilities available to SMITH. Among the capabilities in this domain will be SMITH's access key.

The command processor receives the command "SNDMSG" from SMITH. It processes this enough to understand what is wanted and then makes what it thinks of as a system call. This system call transfers control into an otherwise inaccessible ring of the address space wherein resides the code of the Utility. You may wonder how the Utility suddenly got into the act. The fact is that it is always present in every process. That is, when a process is initially created, a domain is set up in it for the Utility. This domain contains of course, the Utility's access key. When execution is initiated in a process, it begins in the Utility in this (obviously quite powerful) domain. The Utility immediately creates a domain such as that for the command processor which contains much less extensive capabilities and passes control to that domain.

Though the Utility has a domain of its own, when it is entered by a system call it continues to operate in the domain of the program which executed the call. So, in our case, the Utility finds itself in SMITH's domain (so to speak). It uses the system call (into the ring which implements protection) described earlier to acquire an EXECUTE capability for SNDMSG. Note that this capability is not needed by the Utility itself. The exercise of acquiring it successfully merely insures the Utility that the program from which it was called is authorized to execute SNDMSG.

The Utility now assumes its own identity (by another system call to the basic protection module) so that it can use its own access key. It creates a new domain, and acquires its capabilities to SNDMSG's PIF and access key. Using these, it maps the code of SNDMSG into the address

space associated with the domain and transfers the capability for the access key into the domain. It also gives the domain a limited right to use the capability for the terminal SMITH is using. It divests its domain of any capabilities it has now finished using. It transfers a CALL capability for the new domain to the domain of the command processor, re-assumes the identity of the command processor, and returns control to that program.

We now have two separate domains (besides the Utility's) - that of the command processor and that of SNDMSG. Control is currently in the command processor, which can now invoke SNDMSG by exercising its newly acquired CALL capability. When it does so, SNDMSG will operate in a domain which contains the capability it needs to communicate SMITH's message to JONES. Strictly speaking, SNDMSG initially finds itself with a capability only for its access key. It must exercise this capability to acquire the capability actually need to access JONES' file.

As an aside to the above discussion, we observe that with the rights to EXECUTE SNDMSG and the rights to APPEND to message files distinct, it is trivial to have send-only and receive-only participants in the SNDMSG community. This might be useful for participants like programs which record or analyze system performance or for "users" which are only repositories for archived messages.

It should also be noted that the above is in no way a description of how we feel SNDMSG should be implemented. It merely shows how the current implementation could be wrapped in a security blanket.

III. How the BCC 500 Protection Mechanisms Measure Up

Here we want to compare the BCC 500 mechanisms with the mechanisms of other existing systems; to discuss them in the light of recent research; and to point out the things about them that we consider unsatisfactory.

Comparison with TENEX

There are at least two important features in the BCC 500 protection scheme which are lacking in TENEX.

1. The lock lists of the 500 have no parallel in Tenex. This means that there's no general way to grant access(es) to an object to a single user or program. The TENEX "group" mechanism can sometimes be used to get the desired effect. It could, in fact, be used to give SNDMSG (and only SNDMSG) APPEND access to the MESSAGE.TXT files of users of a TENEX system. We may describe later how this could be done, but we note here that the solution is not really satisfactory because it would preclude the use of the group mechanism for any other purpose in the system.

2. As far as I can see, there is no implementation in TENEX, except at the job level, of anything one could call a domain. Although a job can consist of a number of independantly executable processes (forks) it seems that the only "capabilities" that are in any way private to a process are those which it has to the pages of its own virtual memory. This is not quite true. One of the arguments to GTJFN is a flag which means "no access by other forks". This would seem to mean that a process can acquire file capabilities private to itself. For some purposes, this feature is probably useful.

Perhaps the basic problem with respect to domains in Tenex is that there is no general way to set up a fork with an "access key" separate from that of other forks of its job. Stating this in Tenex terminology, there is no way for two forks of a job to be CONNECTed to different directories.

We note that a projected version of Tenex (version 1.34, for release

in March of 1975) is to have a new feature called "Access Control Lists". Presumably these lists will serve the same function as the similarly named things in MULTICS and as the BCC 500's lock lists. We also note that other projected changes to Tenex seem to indicate that jobs and protection domains will continue to be equated.

Comparison with MULTICS

I don't know enough about MULTICS to say much about how it and the BCC 500 are related. It should be noted that the access control lists of MULTICS are quite similar in function to the lock lists of the 500.

At the moment I am totally ignorant of how MULTICS implements protected domains of execution.

From Jones' Point of View

I believe it's correct to say that the terms "domain" and "capability" are exactly equivalent to Jones' terms "environment" and "right".

Without too much effort we can think of the BCC 500 file directories as "storage environments" and its sub-processes as "execution environments".

Our representation of storage environments is relatively straightforward. It is a representation of the kind Jones discusses in connection with "object site enforcement of protection" [J73, pp 23-24]. To describe an access lock in Jones' terminology we would call the lock field an environment name and the access field an access name. The access field is "bit coded" [J73, p 22].

Thinking about this, I wonder whether Jones' schema really fits our system. In the normal, simple case it seems to. But remember that we have these things called access keys which are used in moving rights from storage environments to execution environments. These keys are the things that match the locks in lock lists. If we are to think of these locks as environment names, then it seems that the keys must be thought of the same way. However, keys are themselves objects for which rights exist in both storage and execution environments. It is possible for an execution environment to contain rights to more than one access key. Confusion.

Our representation of execution environments is basically as a capability list as discussed under "Execution Site Enforcement" [J73, pp 20, 21].

Other Characterizations of the BCC 500 Protection Mechanisms

Schroeder [S72, p 33] describes a division of protection schemes into two types called "list oriented" mechanisms and "ticket oriented" mechanisms.

List oriented schemes, are those in which "control of access to an object is specified in lists maintained by the protected subsystem that is the custodian of the object". It seems to me that our storage environments (directories) fall into this category.

In ticket oriented schemes, "permission to access objects in the system is embodied in unalterable tickets which may be distributed to the domains in the system". It seems to me that the capabilities that make up our execution environments must be thought of as such "tickets".

IV. Shortcomings of the BCC 500 Protection Mechanisms

1. The mechanisms are complex and difficult to describe and therefore probably not very credible. Although the basic access key / access lock scheme is simple enough, the many variations of it provided primarily for efficiency obscure this. Also, there are some means of access, based on the notion of the "owner" of an object, which are implemented outside of the key / lock mechanism.

   One strange feature mentioned earlier in this report is the ability of the special program called the Utility to switch between two domains of protection. While this and other things not described here may be good, they give the BCC 500's protection system an unpleasantly motley appearance.

2. The representation of a procedure in Jones' sense [J73, Chapter IV] is costly in the BCC 500 system. The description of a possible implementation of SNDMSG showed that one must create a system principle so as to have a directory in which to record the "declared rights" of the procedure. It would be much better if the basic protection modules explicity knew about objects of type procedure.

3. The BCC 500's means for passing control from one environment to another contain no provision for passing capabilities (or any other parameters). Parameters, including the "names" of capabilities, can be "passed" only through shared pages of memory. This deficiency is particularly irritating because it could have been avoided at a moderate cost in the 500's micro-coded CPUs.

4. The concept of "rights amplification" is foreign to the basic design of the BCC 500. In the SNDMSG example, we saw how the Utility somewhat clumsily got the effect of such amplification when it first checked to see that its caller had EXECUTE access to SNDMSG, then switched into a much more powerful domain so it could acquire READ access. This technique is not extendable to

Jones' kind of amplification since it depends on the pre-existence (in a directory) of the required right to the specific file, SNDMSG. (Recall that in Jones' model, rights amplification depends only on the type of the object.)

5. The BCC 500 is severely restrictive as to the number of different kinds of access which may be specified for an object. In fact, there are only four possible kinds.

6. The creation of new types of objects is not usefully supported by the 500. A token effort was made in this direction, but the protection system allows the existence of only sixteen distinct object types, approximately half of which have meanings pre-assigned by the system.

Bibliography

[J73] Jones, A. K., Protection in Programmed Systems.
Ph.D. Thesis, Carnegie-Mellon University, 1973.

[L69] Lampson, B. W., Dynamic Protection Structures.
Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press,
Montvale, N. J., pp. 27-38.

[S72] Schroeder, M. D., Cooperation of Mutually Suspicious
Subsystems in a Computer Utility.
Ph.D. Thesis, Mac Tr-104, M. I. T., 1972.