PROTECTION IN THE HYDRA OPERATING SYSTEM

Ellis Cohen and David Jefferson[1]
Carnegie-Mellon University
Pittsburgh, Pa.

Abstract

This paper describes the capability based protection mechanisms provided by the Hydra Operating System Kernel. These mechanisms support the construction of user-defined protected subsystems, including file and directory subsystems, which do not therefore need to be supplied directly by Hydra. In addition, we discuss a number of well known protection problems, including Mutual Suspicion, Confinement and Revocation, and we present the mechanisms that Hydra supplies in order to solve them.

Keywords and Phrases: operating system, protection, capability, type, protected subsystem, protection problem, mutual suspicion, confinement, revocation.

1. Introduction

Hydra was designed with the philosophy that protection must be an integral part of any general purpose operating system. A set of protection mechanisms should be part of the lowest level of an operating system, and those mechanisms must be flexible enough to support the wide range of security policies needed by both high level subsystems and user programs. In this paper we will describe the capability based mechanism in the Hydra kernel, its use in constructing protected subsystems and in solving a number of well known protection problems. We expect that the reader is already familiar with the contents of the companion paper [WLP75] which discusses in greater detail our philosophy of what a protection system should be.

When users share access to information, there is inevitably a possibility for malicious or accidental disclosure of that information. It is necessary to restrict the behavior of possible computations in order to guarantee that such mishaps do not occur. In a general sense [Coh75], a *protection problem* is simply a description of some class of restricted behaviors. A protection problem can be solved in a protection system if the system provides some set of mechanisms which, when invoked, guarantee that the behavior of the system will be appropriately restricted.

1. This work was supported by the Defense Advanced Research Projects Agency under Contract F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

In this paper, we will primarily be discussing *access protection* - the permission or denial of access to information. In a companion paper [Lev75], we discuss the related issue of *control protection* which involves guaranteeing or preventing execution of programs at scheduled times for specific durations using specific resources. Problems of access protection can be viewed along two orthogonal dimensions:

1) *Prior vs. Future Decisions*: By a prior decision, we mean deciding in advance the circumstances under which access to certain information will be permitted or denied. By a future decision, we mean deciding at arbitrary times to revoke or restore access permissions, that is, to change the circumstances of access at some time in the future.

2) *Unilateral vs. Negotiated Decisions*: By a unilateral decision we refer to a single user deciding how other users are to be able to access information under his aegis. By a negotiated decision we refer to a user wanting to temper or limit the restrictions to access imposed by another user.

We can give examples to illustrate each of these categories. In parentheses following each example are a list of problems discussed in section 3 that fall into the same category.

Prior and Unilateral - The creator of a file wishes other users to be permitted to read, but not write, the file. (Mutual Suspicion, Modification, Limiting Propogation of Capabilities, Conservation, Confinement, Initialization)

Future and Unilateral - The creator of the file wishes at some later point to revoke some user's permission to read the file. (Revocation)

Prior and Negotiated - A user requires a guarantee in advance that his permission to read the file will never be revoked. (Freezing)

Future and Negotiated - A user requires a guarantee that, if the creator ever attempts to revoke access, the matter will be submitted to (some trusted procedure which implements) binding arbitration. (Accounting and Lost Object Problems)

In section 2 of this paper, we will describe the basic protection mechanisms supplied by Hydra, their use in constructed protected subsystems. We will also see how the mechanisms directly solve certain straightforward problems and can be combined, through *procedural embedding* to solve any of the problems described above.

In section 3, we will look at a number of well known problems and present additional Hydra mechanisms that can be used to solve them directly.

2. Protection Mechanisms

2.1 Introduction

Clearly, there are a huge number of protection problems. Hydra cannot even begin to provide policies that solve each one directly. But that is not Hydra's purpose. In [Lev75], we discuss in some detail a central philosophy of Hydra, that of Policy/Mechanism separation. Briefly, we believe that an operating system should not attempt to provide a fixed set of policies, particularly protection policies. Rather, it should provide a set of mechanisms with which a large set of policies (hopefully including all useful and interesting ones) can be constructed. For example, we do not wire into Hydra a policy that permits the creator of a file to revoke access to it. Instead, there is a mechanism that permits revocation and another (freezing) that prevents it. Yet other mechanisms (procedures, amplification) can be used to build arbitrarily complex policies that determine under what circumstances revocation is to be permitted or prevented.

The particular protection mechanisms provided have a substantial impact on the protection policies obtainable. The mechanisms provided by Hydra are based on five philosophical principles. These principles and the mechanisms they induce are:

1) Information can be divided into distinct objects for purposes of protection. We really want a protection system to control access to and propogation of information. Unfortunately, we do not understand how to do this directly. Instead, Hydra forces users to group information together into a uniform data structure called an *object* and provides protection at the level of the object as a whole.[2] This is often quite natural, as when a user wishes to restrict access to an entire file. In other cases, the distinction makes an enormous difference. For example, it is one thing to revoke access to a file; it is quite another to revoke access to all information in the system derived from or dependent upon the data contained in that file.

2) Objects are distinguished by *type*. Each object is of a particular type, which remains constant for the lifetime of the object. Certain types of objects (e.g. Procedure, Process, Semaphore) and the operations relevant to each (e.g. CALL, START, P) are directly provided by Hydra. Hydra also provides mechanisms for creating new types of objects and defining the operations which manipulate them. At the same time, because all objects have the same structure, regardless of type (the type is used in interpreting the contents of the object), the kernel provides a set of generic operations (for example, reading or writing the object) which are type independent.

3) Access to objects is controlled by *capabilities*. Capabilities may be passed from one user to another and may be retained by a user between terminal sessions. Each capability contains a large (compared to other systems) number of access rights which determine how the object named by the capability can be accessed; some of these rights are type independent (related to the fact that all objects have the same structure) while some are type specific.

   Possession of a capability is the sole determinant of access, although through the use of other mechanisms (procedures, amplification), one can construct policies that take into account other criteria (e.g. who is the accessor?). Objects do not have "owners" as such. All holders of capabilities for an object share control of it in proportion to their rights. (Of course, one user might retain all rights to an object himself without granting any rights to others, becoming the de-facto owner.)

4) Each program should execute with the smallest set of access rights necessary. The protection domain is that set of capabilities which may be exercised by an executing procedure. It changes with each procedure call. *Procedures* (a type of object) in Hydra have access to "own" objects (via capabilities), inaccessible to users with only the right to execute the procedure. Each time a procedure is called, it executes in a completely new environment, determined solely by the procedures "owns" and by capabilities passed as arguments by the caller. This permits a direct solution to the Mutual Suspicion problem and (as we shall see) permits the construction of arbitrary protection policies.

2. Other systems [Din73,Gra72,Lau74] permit differential protection of structured subsets of an object. Actually, in Hydra, objects are divided into two 2 parts, a C-list and a Data-part, and in some ways, each may be protected separately.

5) All knowledge about the representation and implementation of operations for each type of object should be hidden in modules called *subsystems*. In general, users of an object of a particular type cannot access it directly; they can only do so through procedures associated with the subsystem for that type. Hydra supports this through the mechanism of *rights amplification*. Under certain circumstances, when a capability is passed as an argument to a procedure, it will have greater rights (in particular, those necessary to access the contents of the object) in the new domain created for that procedure invocation than it had in the domain of the caller.

In the remainder of this section, we will explore the mechanisms that support these principles in greater detail.

2.2 Objects, Capabilities and LNS's

An *object* is a data structure that represents an instance of a resource, either virtual or physical. It may be thought of as a three-tuple:

< unique-name, type, representation >

The *unique-name* of an object distinguishes it from all other objects that ever existed in the past or will exist in the future. The *type* of the object defines the nature of the resource the object represents. Some example of types might be DEVICE, DIRECTORY, PROCEDURE, PROCESS, SEMAPHORE, FILE, SEQFILE and RANDFILE (the last three representing different kinds of files).

A primary purpose of a protection system is to control access to objects. This is accomplished in Hydra through the use of *capabilities*. Associated with each executing program is a *C-list*, a linearly numbered list of capabilities. Each capability contains both the name of a particular object as well as a set of *access rights*. An access right (for example, read-rights or write-rights to a file) usually represents an operation which the possessor of a capability may legally perform on the object.

The representations of capabilities and rights are manipulated only by the Hydra kernel. It is impossible to 'forge' a capability or gain access to an object without having a capability for it.

In many capability based operating systems, a capability is an attribute only of executors. In such systems, the C-list associated with an executing program defines its protection domain. While this is also true in Hydra, we have generalized the notion of objects and capabilities in an important direction. Regardless of type, all objects have the same structure; the type of an object simply provides an interpretation for the contents of the object. Capabilities are not an attribute of executing programs alone; any object may contain a C-list. This generalization has two important effects.

1) Executing programs may be represented as a type of object. This type is an *LNS*, short for "Local Name Space".

2) New object types (new kinds of resources) may be defined in terms of existing object types. For example, one might imagine a File-directory which contains both a list of files and a semaphore which provides mutual exclusion of operations on the file-directory. In the terminology of Hydra, the C-list of an object of type FILE-DIRECTORY would contain capabilities for objects of type FILE, as well as a capability for an object of type SEMAPHORE.

In addition to a C-list, an object contains a *Data-part*, a block of storage holding relevant information. Together, the C-list and the Data-part constitute the *representation* of the object. Returning to our FILE-DIRECTORY example, the Data-part of a file-directory might be used to hold the string names of the files in the file-directory.

The decision to divide the representation of an object into two parts, Data-part and C-list, was made primarily on pragmatic rather than theoretical grounds. As Fabry notes [Fab74], data and capabilities could, in principle, be combined into one segment, if there were some way to allow complete freedom to alter the data, while at the same time preventing arbitrary manipulation of the capabilities. This would be possible on a processor with a tagged architecture, such as the Burroughs 5500 [Org73]. However, we were limited to the architectural confines of a PDP-11.

There are other reasons for keeping the Data-part and C-list separate as well. Since any object may potentially contain a capability for any other object, objects may form general directed graph structures which require garbage collection. Gathering capabilities together in a separate C-list simplifies the garbage collection mechanism.

2.3 LNSes and Paths

An LNS defines the instantaneous protection domain of an executing program; its C-list contains capabilities for all of the objects that may be directly accessed (including capabilities for PAGE objects, which define the LNS's address space). All objects accessed must be referenced through the LNS, but since the objects referenced by capabilities may themselves contain capabilities for other objects, it should be clear that the actual protection domain extends to all objects reachable via some capability path rooted in the LNS. Access to such objects is limited though, by the rights in the capabilities along the path.

In addition to acting as the protection domain of an executing program, the LNS also serves a naming function. Objects referenced in the C-list of an LNS are never referred to by their unique names. They are always named by their LNS C-list indices. Similarly, objects indirectly accessible via a capability chain rooted in the LNS can be directly named by the sequence of C-list indices along the path to the object. As a general rule, anywhere (the LNS index of) a capability may appear in the calling sequence of a kernel call or procedure call, a *path* to a more distant object may also appear. The Hydra kernel handles all of the details of following the path to the target object.

2.4 Generic Operations

Hydra supports objects of many different types. Some of the types are implemented by the kernel, for example PROCEDURE, SEMAPHORE, PROCESS and TYPE. Others, such as FILE and DIRECTORY, are defined by user level subsystems. But all objects, regardless of type, whether defined by the kernel or by user software, have a common underlying structural representation, i.e. Data-part and C-list. Furthermore, all operations on objects can be composed from simple manipulations of their Data-parts and C-lists. Therefore the Hydra kernel provides a number of type independent "generic" operations for these manipulations. These operations are kernel calls, k-calls for short, and are implemented by instructions which trap to the kernel.

The k-call *Getdata* provides access to the Data-part of an object. Its first parameter is a path to a capability for the object whose Data-part is to be read. The other parameters specify what part of the Data-part of the object is to be read (i.e. offset and length) and the address in the caller's address space into which the information should be copied.[3] A similar k-call, *Putdata*, allows the user to write into the Data-part of an object. There is a third kernel call, *Addata*, which allows the caller to append data onto the end of the Data-part of the object (extending the length of the Data-part as a side-effect.)

There are similar operations for manipulating the C-lists of objects. For "reading" a C-list, i.e. copying a capability into the current LNS, the user executes a *Load* k-call. *Load*'s first parameter is a path to the capability which is to be "read". The second parameter is the index of a current LNS slot into which the capability is copied.

Of course there is also a *Store* k-call which copies a capability from the current LNS into the C-list of another object or into a slot in the LNS. However, in addition to the source and destination parameters, *Store* takes a third parameter, a rights restriction mask. When a capability is stored, it is often desirable for the copy that is stored to have fewer rights than the original. Storing a capability into a public object (or at any rate an object that at least one other user can access) is the primary mechanism Hydra provides for sharing access rights, and often what is desired is to share some, but not all, of the rights to an object, e,g. read-rights, but not write-rights to a file. The rights restriction mask passed to *Store* acts as a rights "filter", and is "anded" with the set of rights contained in the source capability to produce the restricted rights that are placed in the destination capability. In the case that the source and destination are the same, *Store* simply removes the masked-out rights from the designated capability.

An *Append* k-call, which appends a capability to a C-list, is also provided. As with *Store*, the user can restrict the rights of the appended capability.

The *Delete* k-call removes a capability from the C-list of an object. Deleting a capability from a C-list does not cause it to collapse, with the consequent renumbering of the higher numbered capabilities. The slot in which the deleted capability sat is simply written over with a capability of the special type NULL (whose only use is the indication of empty C-list slots.) Note that the *Delete* operation does not destroy the object referred to by the capability; it only destroys the capability itself. Only if all of the capabilities for an object have been deleted is the object itself eligible for destruction. We are planning to implement a *Destroy* k-call which will destroy an object even though capabilities for it are still outstanding. Attempts to access a destroyed object will signal an error.

Finally, there is a *Copy* k-call which copies the Data-part and C-list of an object, given a capability for it, to a new object, placing a capability for the new object in a designated slot in the caller's LNS. The capability for the new object will contain the same rights as did the original.[4] The *Create* k-call, which creates an entirely new object will be discussed in section 2.11.

The generic operations are important because they form the primitive basis for the definition of "higher level" type specific operations implemented as procedures. The C-list manipulating operations are especially important because they allow the construction of collections of objects which are passed around and manipulated together as a unit.

It should be noted that each of the generic operations described above is implemented indivisibly. This means that two processes cannot operate on the same object at the same time even if they are executing concurrently on different processors. However, this mutual exclusion holds for the duration of a single kernel call only. For mutual exclusion of composite operations requiring more than one kernel call, some other form of synchronization, such as semaphores, is necessary.

We did expect that users would desire certain sequences of k-calls executed indivisibly frequently enough that we packaged them as separate k-calls. For example, the k-calls, *Take* and *Pass* are equivalent to the composites (*Load; Delete*) and (*Store; Delete*) respectively.

2.5 Sharing

Perhaps the major benefit of the very general object/capability structure is the ease with which it permits sharing. If two executing LNS's, User-1 and User-2 both have a capability for some object Comm-1, they can easily share both data and capabilities. User-1 can store data into Comm-1's Data-part and User-2 can then retrieve it (assuming their capabilities contain the rights that allow the necessary k-calls). More interestingly, imagine a situation where User-1 has rights that permit both read and write access for some file and wishes to grant User-2 read access only. As figure 2.1 illustrates, it is possible for User-1 to store a capability for the file in Comm-1, restricting rights so that the capability for the file placed in Comm-1 only contains the rights permitting read access. Through Comm-1, User-2 can then gain read access to the file (steps (1) and (2) in the figure).

3. Hardware limitations prevent mapping the Data-parts of objects directly into the user's address space.

4. This is not strictly true, but will suffice for this discussion.

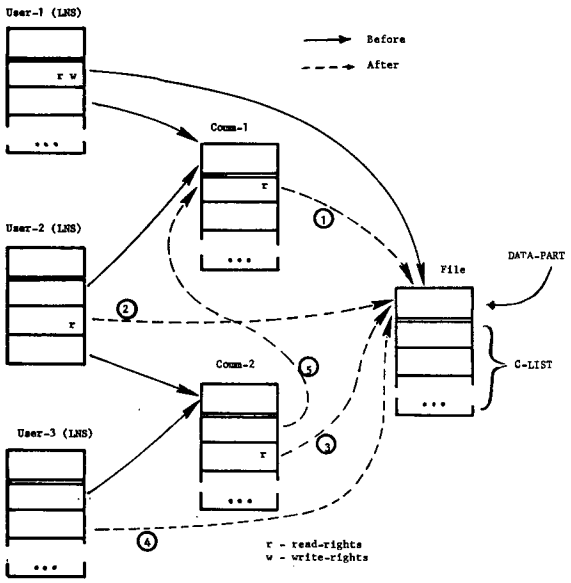

Figure 2.1

If User-2 can communicate with User-3 via Comm-2, then User-2 can then share access of the file with User-3 (steps (3) and (4) in the figure).[5] One can view User-2 as an intermediary, receiving files from User-1 and then passing them on to User-3. Our general object structure actually permits User-2 to withdraw from this arrangement while still arranging for files to be shared between User-1 and User-3. User-2 can simply store his capability for Comm-1 into Comm-2 (step (5) in the figure). User-3 can now access Comm-1 directly through Comm-2 and retrieve any data or capabilities placed in it by User-1.

2.6 Rights

As described above in section 2.2, a capability contains the name of an object and a set of rights to that object. In general, each right corresponds to a class of allowed accesses to the object. The rights list is in fact implemented as a bit vector of length 24. Each bit represents the presence or absence of one right.

Most rights are associated with a single operation. The presence of a particular right in a capability permits the associated operation to be performed on the object, and the absence of the right causes the operation to fail. But some rights, those used directly in the solution of protection problems, are used to permit or deny certain wider classes of behavior. The important property of rights in Hydra is not one-to-one correspondence with operations, but monotonicity: the presence of a right in a capability always allows more (or at least no less) behavioral freedom for the holder of the capability than the absence of that right.

The rights lists in Hydra are divided into two parts: the (16) *generic rights* and the (8) *auxiliary rights*. Generic rights affect type independent behavior, either by permitting the generic operations discussed previously or through their use in the direct solution of problems.

5. If User-1 desires, this can be prevented as we shall see in our discussion of Limiting Propogation of Capabilities.

The interpretation of auxiliary rights is type dependent. Thus, in a capability for an object of type PROCEDURE, one of the rights, known as CALLRTS (call-rights) permits the user to *Call* the procedure. Such a right makes no sense for objects that are not procedures. Similarly, a capability for an object of type FILE may have one of the auxiliary rights bits interpreted as WRITERTS (write-rights). This right makes no sense for objects of type SEMAPHORE or PROCESS or PROCEDURE.

The representation of rights lists as bit vectors is particularly convenient because it makes the two basic operations on rights lists, namely rights checking and rights restriction, very simple. Testing whether a capability contains a certain set of rights involves a single (24 bit) comparison. Restricting rights is just a single "and" operation.

The fact that rights checking and rights restriction are merely bit vector operations gains more than just speed and simplicity. It means that the mechanism of rights checking and restriction can be implemented without regard to the meaning of the rights. The assignment of meaning to the auxiliary rights is a matter left up to the software defining the types. This is a clear example of the basic Hydra principle of policy/mechanism separation.

There are only a small number of generic rights defined by Hydra, and we can list them here. They are divided into two groups. The first nine rights control the nine generic operations discussed in section 2.4. In each case, the appropriate right must be present in a capability or else, an attempt to perform the corresponding operation on the object it references will fail.

1. GETRTS - to get data from an object's Data-part
2. PUTRTS - to put data into an object's Data-part
3. ADDRTS - to add (append) data onto an object's Data-part
4. LOADRTS - to load a capability from an object's C-list (into an LNS)
5. STORTS - to store a capability into an object's C-list
6. APPRTS - to append a capability onto an object's C-list
7. KILLRTS - to delete a capability from an object's C-list
8. COPYRTS - to copy an object
9. OBJRTS - to destroy an object

The remaining rights, DLTRTS, MDFYRTS, UCNFRTS, ENVRTS, ALLYRTS and FRZRTS are used in the direct solution of protection problems, and with the exception of DLTRTS, discussed below, are left to section 3.

As we mentioned in section 2.4, a version of *Store* allows a user to restrict his own access to an object. A good protection mechanism protects users not only from other users, but from themselves as well. This is especially important when a user is testing a program about which he is still a bit uncertain, and wants to limit the havoc he can wreak among objects he can access.

DLTRTS provides an example of such a safeguard. Normally all capabilities contain DLTRTS, though we will not show them in any of our diagrams. A capability may not be deleted, nor may any right be removed from it, unless it contains DLTRTS.[6] Thus, an uncertain user can remove DLTRTS from capabilities in his own LNS in order to guarantee that he will not mistakenly reduce his own access rights. A user may also find it occasionally useful to restrict DLTRTS when storing a capability in an object before granting KILLRTS for that object to another user.

2.7 Procedures and LNSs

A *procedure* is an object which serves as an abstraction of the ordinary programming notion of procedure or subroutine. Thus, it has some "code" and some "owns" associated with it. It may take capabilities as parameters and it may return a capability to its caller. Procedures may *Call* one another in a potentially recursive manner, because procedure activations (LNSes) are stacked.

However, Hydra procedures go beyond this simple model by including protection facilities. The procedure object actually serves as a prototype or model for the LNS created when the procedure is called. For example, the procedure's C-list contains a capability for each of the objects considered to be "own" to the procedure, and copies of those capabilities are placed in the instantiated LNS during a procedure call. In addition, the C-list of a procedure may contain structures which are not capabilities at all, but "prototype capabilities" called *templates*. There must be one template for each "formal parameter" of the procedure, which specifies the type and rights required of the parameter. During a procedure call these templates are replaced by "actual parameter" capabilities derived from the capabilities passed as arguments.[7]

The Data-part of an LNS contains a variety of useful information which is initialized from the Data-part of the procedure when the LNS is instantiated. This information includes specification of the LNS's address space, software trap and interrupt vectors (hardware traps and interrupts are handled by the kernel) and the location of the first instruction of the procedure.

We believe that ideally, all procedures, including simple subroutines such as *sqrt* or *sine*, should execute in an environment providing the smallest set of access rights necessary. This implies frequent changes in the protection domain. But in much the same way that one might decide whether a subroutine should be called or expanded in line, one must consider the costs involved in switching protection domains. Unfortunately this cost in Hydra is considerable, due to the limitations of hardware and to certain design flaws. As a result, users package routines as Hydra Procedures only when the protection domain must be changed to protect either the caller or the supplier of the routine, or when the routine is so large that the overhead of domain switching is insignificant. For example, a compiler might be packaged as a procedure which takes a capability for a source file as an argument and returns a capability for an object file.

The difference between a procedure and an LNS is an important one even though it is frequently blurred. (We sometimes speak of the executing procedure when we actually mean the LNS created from that procedure during the call operation.) An LNS may change during the course of its execution, for instance by creating new objects and storing capabilities for them in the LNS's C-list. But the procedure object itself is never affected by the LNS's execution. Thus, procedures are potentially reentrant and recursive.

2.8 Processes

Objects of type PROCESS correspond to the usual informal notion of a process, that is, an entity which may be scheduled for execution. The Data-part of a process object contains process state information (e.g. scheduling parameters). The C-list of a process object contains a list of LNS's, treated as a stack. The "top" LNS defines the current protection domain of the process.

The current protection domain of a process may change many times during execution of the process, corresponding to calls and returns of Hydra Procedures. Each time a procedure is called, a new LNS is created, initialized and pushed onto the top of the LNS stack, becoming the current LNS. When the top LNS returns, it is popped from the top of the LNS stack and destroyed. Control returns to LNS below it on the stack.[8]

Holders of a capability for a process object may start and stop it, as well as change the process state. (Additional details can be found in [Lev75]), however no capability for a process contains the generic rights necessary (LOADRTS, etc.) to permit access to the process' C-list. A user scheduling a process does not necessarily have a right to know what the process is doing. This is especially true in the case that a proprietary procedure has been called. If the process' scheduler could access the process' C-list, it would be able to access the proprietary procedure's "owns".

6. This differs from KILLRTS. A user requires a capability containing KILLRTS to delete capabilities in the object referenced by the given capability.

7. Hydra Procedures are very similar to a combination of what CAL-TSS calls domains and gates [Gra72]. The former specifies the procedure "owns", while the latter specifies the form of the procedure arguments. For a number of reasons, including efficiency, we have come to believe that the separation is probably desirable.

8. There is a facility that allows LNS's to be saved after they return. The LNS may then be continued, causing control to be transfered just beyond the return point. This does not require any changes in the LNS stack as described. The stack is used only for control (call/return discipline), rather than for access. Any object accessed is reached through a path rooted in the current LNS, not through any LNS in the stack. Actually, there is a set of capabilities accessible indirectly. The creator of a process, in addition to specifying its initial LNS, also may associate a process base with a process, an object whose C-list can be accessed by any LNS executing under the process.

2.9 Types and Subsystems

We have already mentioned that all objects are a three-tuple:

< unique-name, type, representation >.

In this section, we will discuss the representation of types and its role in defining type *subsystems*.

The type of an object is actually the name of some other object whose type is TYPE. That is, just as there objects of type PROCESS, PROCEDURE and perhaps FILE-DIRECTORY, there are objects of type TYPE, each one of which "represents" the class of objects of that type. For example, the object whose name is SEMAPHORE and whose type is TYPE "represents" all objects whose type is SEMAPHORE.

Of course, all those objects of type TYPE must be represented by a TYPE object whose name is also TYPE. This all can be depicted by a three level tree with the TYPE-TYPE object at the root. Figure 2.2 shows part of the tree containing objects of type TYPE, PAGE, FILE and SEMAPHORE.

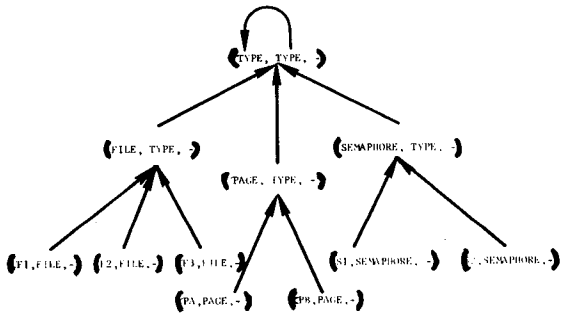

Figure 2.2

Given a capability for some TYPE object, a new object of that type may be created (the details may be found in section 2.11). In particular, new types may be defined by starting with the TYPE-TYPE object.

The Data-part of a TYPE object contains a variety of useful information, such as the maximum permissible sizes of the C-list and Data-parts of objects of that type (enforced by the kernel). As we shall see shortly, the C-list is more interesting.

The concept of "type" in Hydra is closely related to the concept of "data type" in a number of programming languages designed for "structured programming", i.e. "class" in Simula [Dah66], "cluster" in CLU [Lis74] and "form" in Alphard [Wul74b] (this relationship is discussed more thoroughly in [WLP75]). The central notion is that a type is an abstraction of a class of objects, and that the abstraction specifies not only the representation of the objects, but the operations that apply to the objects as well. A key feature of type abstraction is that the representation of an object should not generally be known to users of the object. Manipulation of the object should only be possible by invoking those operation specific to its type.

In the case of user defined types in Hydra, these operations are specified as Hydra Procedures, and the C-list of the TYPE object contains capabilities for these procedures.[9] We call this collection of procedures a *Subsystem*. For example, a File-directory subsystem would likely contain one procedure that would store a file in a given file-directory by symbolic name as well as one that would return a capability for a file given a file-directory and the file's symbolic name.

As already noted, the representation of an object, whatever its type, is simply the contents of its C-list and Data-part. There is no explicit declaration of how that representation is to be interpreted; rather it is implicit in how the representation is used by a subsystem's procedures. For example, the procedures in a File-directory subsystem might interpret the Data-part of a file-directory object as containing a map from the symbolic name of a file to the index in the file-directory's C-list where the capability for the file might be found.

Hydra must somehow guarantee that ordinary users cannot access or manipulate an object's representation except by calling subsystem procedures, especially since outside access might threaten the integrity of the assumptions made by the subsystem regarding the format of the representation. This implies that ordinary users do not have capabilities containing the various generic rights (LOADRTS, PUTRTS, etc.) that permit access to an object's representation. Yet, a subsystem procedure must be able to gain these rights when a capability for an object of the type it supports is passed to it as an argument. In other systems, objects are "sealed" [Mor73,Red74a] by a subsystem when presented to a user. The subsystem procedures must then be able to "unseal" them in order to manipulate them directly. In Hydra, sealing simply means the restriction of the appropriate generic rights. Unsealing is accomplished by *Rights Amplification* [Jon73]. The exact mechanism which supports amplification is discussed in section 2.11.

As we noted in section 2.6, the auxiliary rights of user defined types are not specially interpreted by the kernel. Like the generic rights, they may be restricted via the generic operations already discussed (section 2.4) and may not be gained except through rights amplification. It is possible to check whether a capability contains a particular right (section 2.11), and thus a subsystem may use auxiliary rights to allow or disallow calls on various procedures in much the same way that the kernel uses generic rights to allow or disallow the application of various generic operations.

For example, a File-write procedure might require a capability for a file containing auxiliary right #2, while a File-read procedure might require a capability for a file containing auxiliary right #5. If the "owner" of a file holds a capability with both rights, but shares only a capability with auxiliary right #5 with other users, then while other users will be able to read the file, only the "owner" will be able to write it.

9. One does not need a capability for the Type object in order to call one of these procedures. Using the TCALL operation supplied by the kernel, they may be called through any object of the specified type. A similar mechanism may be found in the Plessey system [Cos74].

2.10 Kernel Types

While all types may be though of as defining subsystems, certain types, such as PROCEDURE, LNS and PROCESS, are crucial to the operation of Hydra. These types, plus certain others useful as building blocks for user-defined types, are defined and implemented directly by the Hydra kernel. Operations on these kernel-supported types are implemented as k-calls instead of procedure calls.

The following is a list of the kernel-defined subsystems. In some cases operations specific to the type are mentioned. Each such operation is protected by an auxiliary right.

LNS - An LNS serves as a protection domain and as a dynamic activation of a procedure. It is the root of the tree (graph) of objects accessible to a program and provides a framework for naming them.

Procedure - Procedures are the Hydra analogue of ordinary programming of procedure. However, a procedure call in Hydra causes a change in protection domain. The *Call* operation requires the auxiliary right CALLRTS.

Process - A process object represents an independently schedulable activity, the unit of parallel decomposition. Processes consist of some scheduling data and a stack of LNSes. The primary operations on them are *start* and *stop.*

Page - A page object is an image of one of the 4k word defined by the hardware. One can think of the address space of an LNS as defined by a table found in the Data-part of the LNS; each entry containing an index into the LNS's C-list where the capability for the corresponding page can be found. A more accurate and complete description is included in [Lev75].

Semaphore - These are Dijkstra-style semaphores with P, V and conditional-P operations defined.

Port - Ports are the basic objects of the Hydra interprocess message communication system. They act as message switching centers and synchronization structures. Operations on ports include *connect* and *disconnect* (to form and break channel connections between ports) and other primitives for sending and receiving messages.

Device - A device object is the software representative of a physical i/o device. In Hydra it is treated as a variety of port. The only operations currently defined on devices are *connect* and *disconnect* to ports.

Policy - Policy objects are mailboxes used by the kernel to communicate with *policy systems* responsible for scheduling processes. Details can be found in [Lev75].

Data - The kernel provides data objects as a convenience to users who wish to seal data in protected objects without going to the trouble of defining a formal subsystem. Data objects have Data-parts, but no C-list. Their only use is as simple data carriers.

Universal - Universal objects are similar in concept to data objects except that universal objects do have C-lists, and thus can act as carriers of capabilities as well as data.

Type - Type objects represent entire subsystems. The C-lists of type objects contain capabilities for all of the operations in the subsystem (if the system is not one of these kernel defined subsystems). One auxiliary right defined is TMPLRTS, which permits a template to be made from a Type object (section 2.11).

2.11 Templates

We have delayed discussing until now three mechanisms that Hydra must provide in oder to support Type Subsystems:

1) Creation - A user wishes to create a new object of a specific type.

2) Type and Rights Checking - A user wishes to guarantee that a capability references an object of a specific type and contains required rights. This is particularly important in specifying procedure "formals".

3) Rights Amplification - Given a capability for an object of a particular type, the subsystem for objects of that type wishes to gain the rights necessary to manipulate the object's representation.

Hydra provides a single mechanism, *templates*, which serves all three purposes.

Templates, like capabilities, may appear in the C-list of an object, and through the use of the generic operations already discussed, may be moved from the C-list of one object to another. Unlike capabilities they do not contain a reference to an object. Rather they can be thought of as prototype capabilities for all objects of a given type. Through the use of the generic operation *Template*, a template of a particular type may be created by a user already holding a capability for the TYPE object of the same type. Thus, a File-directory Template could only be created by someone having a capability for the File-directory TYPE Object.

There are three kinds of templates, Creation Templates, Parameter Templates and Amplification Templates, corresponding to the three functions described above. Each template contains a field designating its type, and depending upon which kind of template it is, contains one or both of the two fields *required-rights* and *new-rights.*

1) Creation Templates. Creation templates contain a type field and a new-rights field. Through the use of the generic operation *Create*, the holder of a creation template can create a new object whose type will be the same as that of the template. A capability for the new object will be placed in the creator's LNS with the same rights as as those specified in the new-rights field of the template.

When a template is initially created, its new-rights field contains all rights. Through the use of generic operations, these rights may be selectively removed. A subsystem may choose to make creation templates generally available after first removing those rights from new-rights that would permit direct access of the object (e.g. STORTS, PUTRTS, etc.). Thus, while a user could create a new object, he still would be unable to manipulate its representation without calling subsystem procedures.

Often, when an object is newly created, a subsystem wishes to initialize it in some way. In that case, the subsystem might not choose to make creation templates generally available. It might simply retain a creation template itself and make a procedure available to users, which when called, would both create and initialize the object, returning a capability with appropriately restricted rights to the caller. (Also see section 3.6).

2) Parameter Templates. Parameter templates contain a type field and a required-rights field. They can be compared against a capability to determine whether or not that capability is of the same type and has at least those rights listed in the required-rights field of the template. (Here, Hydra goes beyond type checking generally found in programming languages in that it checks rights as well as type.) It is expected that a subsystem will make these templates generally available to users.[10]

3) Amplification Templates. Amplification templates contain a type field, a required-rights field and a new-rights field. Given a capability of the same type as the template, with all the rights specified in the template's required-rights list, a new capability can be produced, referencing the same object as the original capability, but containing the rights specified in the new-rights field of the template. It is expected that amplification templates will never be made generally available by a subsystem. In particular, amplification templates for kernel-supported types (Process, LNS, etc.) are never made available.

10. A special kind of parameter template, a Null Template, is made available by the kernel. It matches any type and only checks rights.

2.12 The Hydra Procedure Call Mechanism

We have discussed in a general way the effects of a procedure call. Now that we have explained templates, we can discuss in more detail the heart of the Call Mechanism, the initialization of the LNS's C-list. The *Call* operation is a k-call having the following form:

Call (cproc, return-slot, p1, mask1, ... pn, maskn)

The first parameter, *cproc*, must be (a path to) a capability for a Hydra procedure object. The second parameter, *return-slot*, is an index into the current LNS (the calling LNS) indicating where the called procedure should store the capability it returns. The rest of the parameters to *Call* are grouped in pairs. Each pair consists of a path to a capability and a mask used to restrict the rights in the capability passed.

Each capability (and creation template) in the procedure's C-list is then copied into the corresponding slot of the LNS's C-list. These are the procedure "owns" and are said to be inherited from the procedure.

A number of slots in the procedure's C-list will contain amplification and parameter templates. The capabilities passed as arguments to the procedure are bound to these templates in left to right order. Each LNS slot corresponding to an amplification or parameter template is filled by the matching capability passed as an argument.

If a parameter template is encountered in the parameter binding process, the matching argument capability (with rights restricted as specified by its associated mask) is compared against the template. If both type and rights match as required, the capability, with rights restricted according to the associated mask, is placed in the appropriate C-list slot. In the case of an amplification template, the same algorithm is used, except that the capability placed in the LNS will have the rights specified by the new-rights field of the template. If the type or rights of any argument fails to meet the requirements of the template, the LNS is destroyed and control returns to the caller with an indication that the Call failed.

2.13 An Example

We have noted that files are not a kernel supported type. Instead they must be provided through a user defined File subsystem. In one possible implementation, a subsystem might have sole access to a disk (of type Device). It could make file objects available whose Data-part would contain the location on the disk where the file could be found. Of course, the Data-part could only be accessed by the File subsystem procedures. This is similar, in fact, to Hydra's implementation of kernel-supported Page objects.

Alternately, files could be constructed directly from kernel-supported objects. The C-list of a file object might then contain capabilities for Page or Data objects. We will briefly explore the latter alternative by examining the construction and instantiation of Datafile-Append, a procedure supplied by a Datafile subsystem.

Datafile-Append takes two arguments, a Datafile and a Data object, and appends the data encapsulated in the Data object onto the end of the Datafile. The creator of the Datafile subsystem must store three things in in the C-list of the Datafile-Append procedure when he creates it.

3) A Datafile amplification template. The new-rights field of the template contains those rights necessary so that the procedure can manipulate the representation of the Datafile passed to it. In addition, the template requires the Datafile passed to it have the second auxiliary right set. In essence, this means that the second auxiliary right is interpreted as an "append-right" for Datafile objects; it permits a Datafile capability to be used as an argument to Datafile-Append.

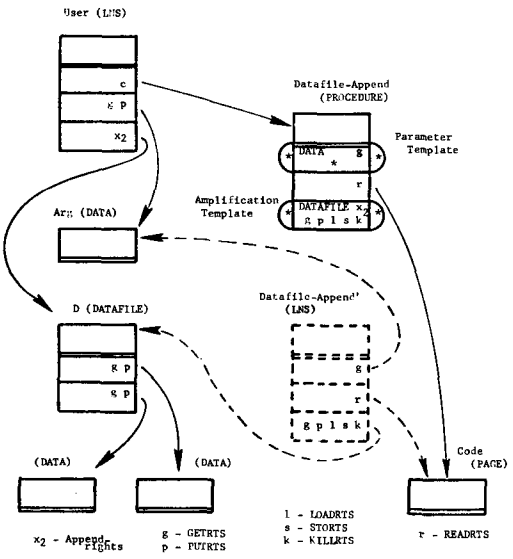

Figure 2.3

Figure 2.3 illustrates what happens when Datafile-Append is called by User (of type LNS) with arguments Arg (of type DATA) and D (of type DATAFILE) instantiating the LNS, Datafile-Append'. The C-list slots in the procedure specifying the Data and Datafile templates have been replaced in the LNS by the corresponding arguments, while the capability for the code page in the procedure has been inherited by the LNS.

2.14 Protection Mechanisms and Protection Problems

The mechanisms of procedure invocation and rights amplification combine to form a powerful tool for the construction of arbitrary protection policies. If direct access to an object can be prevented except through a procedure call, then the procedure can decide when access should be permitted. In general, where mechanisms do not exist to directly solve a protection problem, procedures can be used in constructing a solution. Thus, there are two ways that protection problems can be solved:

1) Direct Solution. The mechanism directly solves the problem. For example, a mechanism that provides separate rights for read and write access to a file, on a user by user basis, directly solves the problem of finding a way to allow some users to read a file while permitting others to both read and write it.

2) Procedural Embedding. The mechanism merely provides an appropriate protected environment for code which implements the solution to the problem. Procedures in Hydra provide just such an environment. Imagine a procedure to which a user could pass a file and a set of "keys" (capabilities for type key objects) as arguments, which the procedure would store in its "own" area. Subsequent callers of this procedure would be permitted access to the file only if they presented one of the designated keys. This kind of arrangement could be used to implement lock and key protection [Lam69] or the Military Clearance Classification system [Wei69,Wal74].

Procedural embedding has other possibilities as well. Imagine almost any common protection problem relating to usage of another user's program. (For example, consider a user who requires a guarantee that a procedure given access to a file will not destroy the file - especially important on a system that, unlike Hydra, does not provide separate rights for read and write access.) Let us suppose that some very trustworthy (and very bright) programmer constructs a procedure which accepts source programs written in a special language that makes it easy (well.. generally possible) to decide whether the program exhibits certain repugnant behavior (e.g. would it destroy a file passed to it as an argument?). This very trustworthy procedure would compile the programs (into Hydra Procedures) and would store them away with their repugnance ratings.

Subsequent callers of this very trustworthy procedure could retrieve procedures along with their repugnance ratings. Using the repugnance information, users could deduce specifications of the program's behavior.

Both the "direct" solutions and the "procedural embedding" solutions that we have described are *dynamic* in the sense that they require overhead during the execution of programs. We could consider, however, going one step further to statically determined protection by complete certification of programs. If users can verify, in advance, that procedures they interact with exhibit only unobjectionable behavior, presumably there would be no need to supply any dynamic protection mechanisms. (Although it is not clear, of course, how a user could know, in advance, of all procedures with which it might interact.) Certification is still an art, and one would not want to rely upon it as the answer to protection in a current system. Further, certification requires a language (or set of languages) in which protection questions are decidable. We did not want to restrict ourselves to such a language, especially since no such language is presently available.

Certification, as a static process, ideally need only be performed once, whereas Hydra's dynamic protection necessitates a continuing overhead. But programs are not always correct or complete, even when their protection properties have been certified, and thus, the tradeoff between recertification and dynamic protection is by no means obvious. In the case of the ideal machine (though unfortunately not the one upon which Hydra runs), dynamic protection of the sort available in Hydra should cause very little overhead at all.

In Hydra, procedural embedding of protection is expensive because of the cost involved in calling a Hydra Procedure. Even supposing a suitable architecture, a procedure would generally have to make many of the same dynamic protection decisions interpretively that are already available directly from Hydra.

There are a set of protection problems (such as Mutual Suspicion, Confinement and Revocation) which are reasonably well understood, and for which users frequently need solutions. It is desirable that Hydra make mechanisms directly available that can be used to provide solutions to these problems, especially if the mechanisms are useful in building other general protection policies (via embedding) and do not "clutter up" the basic system design.

It is not especially difficult for a clever designer to generate such mechanisms. Rotenberg [Rot73], in his thesis, described a large number of interesting protection problems (many for the first time), but he attempted to forge solutions to some of them by positing somewhat awkward or complex additions to the Multics File System.

The basic protection mechanisms already described above go a long way towards providing an ideal environment in which protection problems can be solved. In section 3, we will see that by some simple extensions, primarily through extending the concept of access rights, we have produced what we believe are some elegant direct solutions to some difficult problems.

3. Protection Problems

3.1 Mutual Suspicion

In most operating systems, whenever one user calls a program belonging to another user, or even a utility belonging to the operating system, he takes a risk. He has no way of being sure that the program he calls will not, through maliciousness or error, do something disastrous (such as request that the operating system delete all his files.) Most users simply take such risks for granted and rely on backup systems to aid recovery in the unlikely event that disaster should occur. But in a system in which security is important such faith is not enough. The user needs some way to limit or circumscribe the amount of damage a procedure that he calls can do.

A similar problem is faced by the author of a utility program intended to be called by many different users. The utility presumably has to manipulate certain private files or data structures which it cannot allow its callers to manipulate directly. The author of the utility program needs some guarantee that, except during execution of the program, users cannot access these sensitive data structures.

These two problems together are known as the Mutual Suspicion Problem [Sch72]. Restated in the language of Hydra the problem is this: The caller of a Hydra procedure needs a guarantee that the callee is not granted access to any of his objects except those for which capabilities are explicitly passed as parameters. The callee (i.e. the owner or maintainer of the procedure) needs a guarantee that the caller cannot gain access to any objects private to that procedure except when the procedure explicitly allows it. Note that, in our earlier terminology, mutual suspicion is not a problem of negotiation, but a pair of mutually unilateral problems.

The Hydra procedure call mechanism described in section 2.12 was designed as a direct solution to the Mutual Suspicion problem.

A procedure's execution environment is not determined solely by its caller; procedures may have "own" capabilities, inherited by the LNS incarnated by the procedure and unavailable to the caller. Thus, sensitive or private data structures need never be made available to the caller of the procedure. Furthermore, there is no way that the caller can automatically inherit any capabilities from the called LNS, unless a capability is explicitly returned.

Because an LNS does not inherit access to the capabilities in LNSes deeper in the process's stack, the only capabilities from the caller that are available are those acquired through the parameter binding process of the call mechanism. Thus, even the most malicious procedure could, at worst, access or damage only those objects reachable through the parameters. All files and directories and other sensitive objects that are not passed as parameters are absolutely safe. Of course the procedure might make copies of the capabilities passed to it and store the copies away someplace where they could be used for mischief later. (Section 3.4 indicates how this may be prevented.) But even so, the later damage would still be confined to only those objects passed as parameters. (It should, of course, be clear that the caller's guarantee is still preserved even if the procedure should call another procedure.)

So the Hydra call mechanism solves the Mutual Suspicion problem. It actually does a little more. Not only can the caller control the set of objects that he must allow the callee to access, but by restricting the rights lists of the capabilities he passes, he can actually control the kinds of accesses he risks. He thus has extremely tight access control of his objects.

There is one technical exception to the tight access control: if the procedure in question has an amplification template for some type, then it may be able to acquire more rights to an object than the caller passed (or even had!) This phenomenon, however, should not be viewed as a genuine breach of security. For one thing, it does not affect the class of objects which can be accessed by the procedure. An amplification template does not allow acquisition of new capabilities -- only new rights to objects that were passed as parameters anyway. Thus, the amount of damage that can be done is still limited to the objects actually passed as parameters.

But there is another reason, as well, why the existence of amplification templates does not constitute a security breach. Any procedure having an amplification template must be considered to be part of the defining subsystem for that type. Thus, one generally presumes that it is reliable and does only "correct" things with objects of that type. If a user is unwilling to trust the defining subsystem for a type, he probably should not be using it. If he does use a subsystem that he doesn't completely trust, there are other protection tools available that help guarantee certain aspects of the behavior of any procedure, whether or not it contains amplification templates. The details are discussed in the following sections.

3.2 Modification

Users often want guarantees that an object passed as an argument to a procedure will not be modified as a result of the call. Ordinarily, it is sufficient to restrict those rights that allow modification (PUTRTS,STORTS,etc.) before passing the capability for the object as an argument. However, in the case that a subsystem procedure is called, rights amplification may reinstate those rights.

In general, of course, users must trust that a procedure supplied by a subsystem fulfills its specifications, else there is no reason to use the subsystem. Just as one expects that a subsystem will not promiscuously make amplification templates available, one expects that a subsystem procedure will not modify an object passed as an argument if its specifications declare that it will not.

Unfortunately, this ideal is not always realized in practice. Subsystems believed to be trustworthy may not be, due either to maliciousness or to hardware or software error.

Imagine a user who wishes to list a tediously constructed Datafile. In the nick of time, she is warned that a curious bug has mysteriously appeared, and that the Datafile-List procedure might zero out the Datafile passed to it as an argument. Now, our heroine desperately needs a listing of the Datafile and can't afford to wait until the bug is excised. She can't first make a copy of the Datafile, since that would entail calling the Datafile-Copy procedure, and there is no guarantee that it has not been afflicted with the same ailment troubling Datafile-List.

Hydra solves this problem through the use of MDFYRTS. Each Hydra k-call that modifies an object in any way requires a capability with not only the right that allows the specific modification but MDFYRTS as well. Thus, to store a capability in an object, one must have a capability for the object with both STORTS and MDFYRTS. To put data in the Data-part of an object, one needs a capability for the object with both PUTRTS and MDFYRTS.[11] To solve the Modification Problem though, we must demand that MDFYRTS can never be gained through amplification as other rights are, since a capability lacking MDFYRTS represents an intention that the object it references can never be modified by using that capability. Thus, a capability produced by amplification will only contain MDFYRTS if both the amplification template and the original capability have MDFYRTS. If the caller of the Datafile-List procedure passes a capability for the Datafile restricting MDFYRTS, there is no way that the procedure can modify the Datafile.

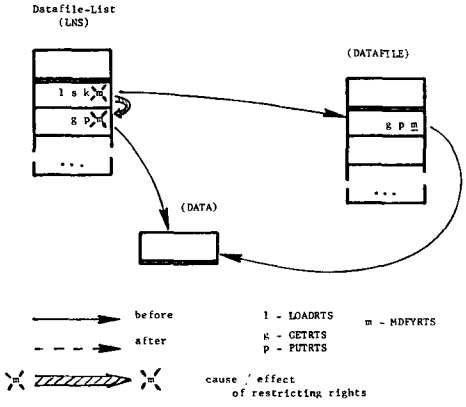

Figure 3.1

In fact, MDFYRTS is more potent. It prevents modification of the representation of an object as well. As figure 3.1 shows, loading a capability into one's LNS through a capability without MDFYRTS masks out MDFYRTS in the loaded capability. If a user calls Datafile-List passing a Datafile capability that does not contain MDFYRTS, she is assured that the Datafile (including the Data objects which, possibly unbeknownst to her, comprise its representation) will not be modified.[12] (This assumes that Datafile-List does not already have "own" access to the Datafile via a different capability which does contain MDFYRTS.[13])

We have stated the Modification problem only in the context of calling a subsystem procedure that might misbehave. Restricting MDFYRTS is of course necessary in calling any procedure that could potentially misbehave, since the called procedure could itself call the subsystem procedure.

This kind of guarantee against modification may force some undesirable constraints on a subsystem. For example, if the Datafile-List procedure wanted to compact or reconfigure the Datafile before listing it, it would not be able to do so if called with a Datafile lacking MDFYRTS. The subsystem could copy the Datafile and modify the copy. The issue here is that the subsystem is prevented from performing internal housekeeping in the original object that could expedite subsequent calls.

We have traded subsystem generality for protection. Of course, a subsystem need not give up such generality. The amplification template in the Datafile-List procedure could require MDFYRTS. Callers of the procedure would then be required to pass a capability with MDFYRTS (else the Call would fail) and could make their own decisions about whether they trusted the Subsystem enough to use it under those circumstances.

3.3 Limiting Propogation of Capabilities

Occasionally, a user wishes to allow another user to access an object but wants to guarantee that the other user cannot share access with yet a third user.

---

11. A k-call that modifies the internal structure of a kernel supported object requires MDFYRTS as well. Thus, P and V (k-Calls that operate on type Semaphore objects) require a capability for the Semaphore with MDFYRTS.

12. In fact, this is actually accomplished in Hydra by a separate right, UCNFRTS. MDFYRTS does not actually cause masking as described. Instead, UCNFRTS masks out both UCNFRTS and MDFYRTS during loads. This division is especially useful when copying an object using a capability in which both MDFYRTS and UCNFRTS are missing. Capabilities for copied objects gain MDFYRTS but not UCNFRTS. This permits a user to modify a copied object (including storing or deleting of capabilities) but prevents objects in the representation of the copied object from being modified where that modification would not have been possible using the original object (due to masking of MDFYRTS in capabilities loaded from the object). For purposes of clarity, we have described MDFYRTS in this paper as including both the functions of MDFYRTS and UCNFRTS as they are used in Hydra.

13. We will discuss how this may be avoided in the section on the Initialization problem.

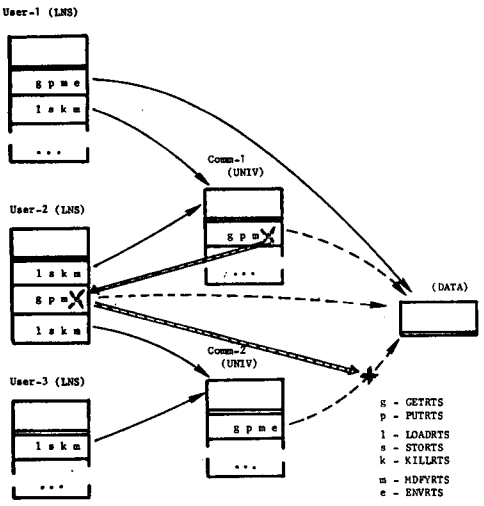

Figure 3.2

Hydra provides ENVRTS in order to solve this problem. Without ENVRTS, a capability may not escape outside of an executing ENVironment (LNS). A capability may only be stored in an object if the capability contains ENVRTS.[14] As figure 3.2 shows, if User-1 stores the capability for the Data object in Comm-1 without ENVRTS, User-2 will be able to load the capability but will not be able to store it into Comm-2. (User-1 must also know that User-2's capability for Comm-1 lacks ENVRTS or else User-2 could simply store a capability for Comm-1 into Comm-2.)

Let us briefly pause here and determine whether there really is a problem. For if User-1 trusts User-2 enough to let her use the Data object, she may as well trust that User-2 not share it with other users. There are a number of answers. Like the use of MDFYRTS in the Modification Problem, ENVRTS is simply an additional small safeguard against error that Hydra can provide. A more important issue though, is one of accountability. If User-1 notices bizarre happenings in the Data object, she knows that User-2 is directly responsible.[15]

It should be clear that, like MDFYRTS, we cannot allow ENVRTS to be gained through amplification, for the absence of ENVRTS also represents a permanent restriction on the use of the capability. And, much like MDFYRTS, ENVRTS masks out ENVRTS in loading a capability. If User-1 had shared with User-2 a list structure, she would want to guarantee that User-2 would be prevented from sharing any sublist of the structure with User-3 as well as a capability for the entire list structure. Thus ENVRTS prevents the storing not only of the given capability but of all capabilities reachable through the C-list of the object referenced by the given capability.

14. Of course, the executing LNS must also contain a capability with STORTS for the object in which the above-mentioned capability is to be stored.

15. This is orthogonal to the issue of revocation. In this case, we could imagine a report to a higher authority instituting some action taken against what/whoever is responsible for User-2's behavior.

3.4 Conservation

In a later section, we will discuss the general Revocation Problem. However, here we will show how ENVRTS provides a solution to a particular revocation problem, the Conservation problem. Often, a user wishes to pass a capability for an object to a procedure. After the procedure returns however, he wants to revoke any accesses retained or propagated by the procedure. There are a number of reasons why he may want to revoke access. Though he expects the procedure to modify the object, he may also want to guarantee that no one will continue to modify the object after the procedure returns. In particular, he wants to guarantee that the executing procedure cannot share the capability with a demonic user who will arbitrarily scribble on the object at unexpected times in the future.

From the previous discussion, we know that without ENVRTS, a Capability may not escape from its executing environment. So, if the capability for an object is passed to a procedure with ENVRTS restricted, the LNS incarnated from the procedure cannot store the capability in any object that another user can access. When the LNS returns execution to the caller, the capability passed as an argument is erased along with the called LNS. Lack of ENVRTS does not prevent a capability from being passed as an argument to another procedure. When execution returns to the original caller, the Call/Return discipline guarantee's that all called LNS's are erased and that the capability escaped from none of them.[16]

If the procedure did not need to modify the object passed as an argument, we might at first think that restricting ENVRTS would not strictly be necessary. For if the object were passed to the procedure simply without MDFYRTS, even though capabilities for the object could be propagated beyond the incarnated LNS's environment, absence of MDFYRTS would guarantee that the object could never be modified by the demonic user.

Unlimited propogation has other effects that will be explored in more detail in a discussion of the Lost Object Problem. But, in addition, there is another reason for Conservation that only the ENVRTS solution addresses. A object passed as an argument to a procedure may, at times, contain certain sensitive information. When a user calls the procedure, he may know that there is no sensitive information in the object. But, after the procedure returns, he may once again wish to store sensitive information in it. Thus he wants to guarantee that no capability to the object can be retained by a spy.

3.5 Confinement

While ENVRTS is useful in preventing propogation of capabilities, it is of limited usefulness in preventing propagation (disclosure) of information. Even though a capability lacking ENVRTS may not escape outside of its execution environment, nothing prevents a user from creating a new object (which will have ENVRTS and MDFYRTS), copying data from the old object to the new one, and sharing a capability for the newly created object.

16. Hydra does provide a mechanism by which LNS's may be retained and subsequently continued, even by another process, after they return. Capabilities lacking ENVRTS may not be used in incarnating these LNSes.

The problem of guaranteeing that no information initially contained in some selected subset of objects can escape outside of its execution environment is called the Selective Confinement problem.[17] HYDRA makes no attempt to solve the Selective Confinement problem but concentrates instead on the less general, but still important Confinement problem, which requires a guarantee that no information at all may escape from a suitably called procedure (incarnating a confined LNS) except to objects designated by the caller.

The best example illustrating the use of selective confinement is the problem faced by a confined LNS producing a bill for services rendered. Either the system must provide a facility for producing prix fixe bills under such circumstances (with minor variations as described by [Rot73]) or the caller runs the risk that the bill can be used to encode information that should remain confined. With selective confinement, the caller can allow the confined LNS to produce a bill that may vary depending upon the non-sensitive arguments only.

Figure 3.3 illustrates a well known example of the need for confinement and shows how Hydra solves it. Consider a user (Nelson) who wants to execute a Tax Procedure. He passes in a capability for an object containing all relevant data concerning his income, expecting that when the procedure returns, the same object will contain a completed tax form. Unfortunately, the tax program potentially could communicate with a spy and Nelson wants to prevent that communication. Even a single bit of leakage might be harmful (it might, for example, encode whether or not Nelson has controlling interests in more than 10 major banks).

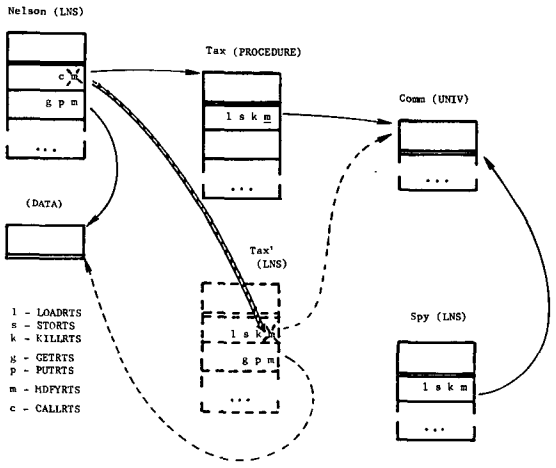

Figure 3.3

Nelson guarantees confinement by calling the Tax procedure through a capability from which Nelson has removed MDFYRTS.[18] Just as MDFYRTS masks MDFYRTS on loads, it masks MDFYRTS in all capabilities in the incarnated LNS inherited from the procedure (but not those passed as arguments). This guarantees that information cannot be leaked to a spy, as figure 3.3 shows, since such leakage would require modification of some object (such as Comm in the figure) inherited from the procedure.[19]

The use of MDFYRTS to confine LNSes is a little different than its usage as explained under the Modification section. However, there are some interesting effects of this marriage of usage. First, we must guarantee (in general) that any procedure called from a confined LNS incarnates a confined LNS as well, else a confined LNS could leak information through a procedure it calls. However, as figure 3.4 illustrates, all inherited procedure capabilities in a confined LNS automatically have MDFYRTS removed and thus, when called must produce confined LNSes. Lampson calls this property Transitivity [Lam73].

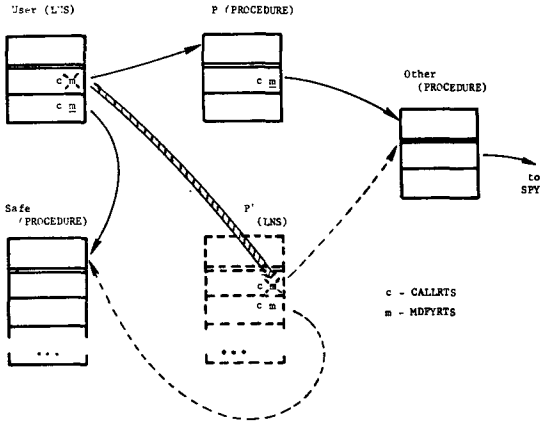

Figure 3.4

Note though, that only the inherited capabilities and not those passed as arguments have MDFYRTS removed. The objects passed as parameters represent safe channels through which data and capabilities may be returned to the caller. In particular, the caller may provide as a parameter, a capability for a procedure with MDFYRTS. In such a case, the procedure when called, will not be confined. This is acceptable because the original caller, in passing such a capability, has effectively vouched for its safety. Thus, transitivity of confinement need not be absolute as required by Lampson.[20]

17. Jones and Lipton [JL75] produce a solution to a version of this problem that corresponds to a situation in which objects in general contain only data but not capabilities. Their formulation can be extended, although one must be careful to avoid the "sneaky signalling" problems detailed by Rotenberg [Rot73]. The reader is invited to construct a solution to the Selective Confinement problem by positing additional rights and combining the solution to the Confinement problem given here with the results of Jones and Lipton.

18. UCNFRTS in the actual implementation.

19. This solution does not prevent (nor need to prevent) the confined LNS from copying information which is to be confined into a new object created by the LNS (which will have MDFYRTS) since no capability for the new object can escape the confined LNS. But, it must be noted that Hydra's solution does not solve the confinement problem completely. Covert channels may still be used to leak information [Lam73], though at a low bandwidth. For example, the pattern of memory access of a confined LNS may cause certain memory interference patterns that could be detected by a spy.

3.6 Initialization

In our earlier discussion of the Conservation problem we showed how a procedure could be prevented from storing away or sharing a capability for an object (or objects in its representation) passed to it. This solution depended upon an assurance that the procedure did not already have "own" access to the object or some object in its representation. This expectation may especially be violated in the initialization of the object.

Initializing a newly created object entails the generation of its representation. Suppose that Datafile-Init initialized a Datafile passed to it by creating a Data object and storing it in the Datafile. We want to prevent Datafile-Init from either making the Datafile or the newly created Data object available to a demonic user, else that demonic user might scribble on it at unexpected times in the future. Restricting ENVRTS when passing the Datafile to Datafile-Init will not suffice, since a capability for the newly created Data object could be made available to the demonic user at the same time it is used to initialize the Datafile. This can be prevented by confining Datafile-Init (calling it without MDFYRTS). In that way, no capability for either the Datafile or the newly created Data object can be propagated beyond Datafile-Init's environment.

Unfortunately, confinement alone is not enough. Instead of initializing the Datafile with a newly created Data object, Datafile-Init might use a Data object that it already shares with the demonic user. Hydra solves this problem by restricting the inheritance of ENVRTS across amplification in the same manner as for MDFYRTS. A capability produced by amplification will contain ENVRTS only if both the given capability as well as the amplifying template contain ENVRTS.

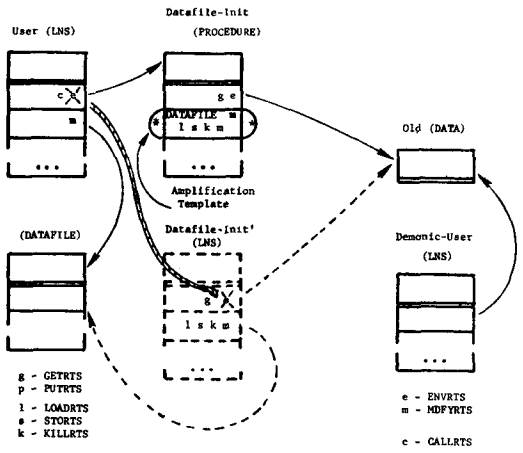

Figure 3.5

As illustrated in figure 3.5, when a procedure is called through a capability lacking ENVRTS, all capabilities in the incarnated LNS inherited from the procedure have ENVRTS removed. In the example above, no object already available to the Datafile-Init procedure could be stored in the Datafile, only newly created objects (or capabilities passed to Datafile-Init with ENVRTS) may be stored in it.

To safely initialize an object (or to completely solve the Modification Problem whenever we pass a capability for an object containing MDFYRTS to a procedure), it is necessary to call the procedure via a capability containing neither ENVRTS nor MDFYRTS. In that way, we guarantee that any new capabilities placed in the object will be for newly created objects and that the entire representation of the object will be unavailable to the demonic user.

3.7 Revocation and Guarantees

Users do not always correctly predict what rights should be extended to other users. Forgetful book borrowers, drunk drivers and unscrupulous business partners are but a few of the real world instances where some kind of revocation is desirable.

Protection systems have an edge over the real world. We can provide mechanisms to support revocation more efficient than the courts, yet less bloody than police states or organized crime. However, revocation without recourse evokes an additional set of problems. A protection system that provides revocation has a responsibility to provide mechanisms that can prevent revocation as well.

We again note how this conflict supports the validity of the Hydra philosophy of policy/mechanism separation. Hydra provides mechanisms that support both revocation and its prevention. The matter of providing policies that determine whether either mechanism can be used in a specific instance is not provided. Rather, that is left to user subsystems which can be used to legislate arbitrarily complex sets of rules.

3.7.1 Revocation

Users may want the ability to revoke access to objects they have shared with others. There are a number of different kinds of revocation. Any system (in particular Hydra) will probably attempt to solve only some of them. Some issues are:

- Immediate Revocation. Does revocation occur immediately? If not, is there a way to know when it has taken place?
- Permanent Revocation. Can access be permanently revoked - can it be guaranteed that some class of users/programs will never be able to gain access to an object?
- Selective Revocation. Must we revoke everyone's access to an object or can we be more selective?
- Partial Revocation. Can some subset of access privileges (e.g. MDFYRTS but not LOADRTS) be revoked?

20. This overly strict requirement of Lampson's is an instance of a more general issue - one of Sufficiency. How are we to insure that a solution to a protection problem does not unnecessarily exclude perfectly acceptable behaviors (such as unconfined calls to safe procedures by confined LNS's)? A complete discussion of this issue is beyond the scope of this paper but may be found in [Coh75]. There it is shown that even the Hydra solution is insufficient (though only slightly).

- Temporal Revocation. Can access be revoked and then granted again?

- Sharing and Revoking the right to revoke. If a user has the right to revoke some access, can that right be shared, and if so, can that right itself be revoked?

Redell [Red74a] has described a system in which users could share capabilities indirectly in such a way that the original holder of the capability could at any time revoke one or more rights from any capabilities propagated from the original. His solution provides for immediate, selective and partial revocation, and can be extended readily to provide the ability to share and revoke the right to revoke [Red74b], and provide temporal revocation.

We believed that immediate, selective and temporal revocation were the most necessary and implemented a variant of Redell's scheme using a mechanism of "Aliases".[21]

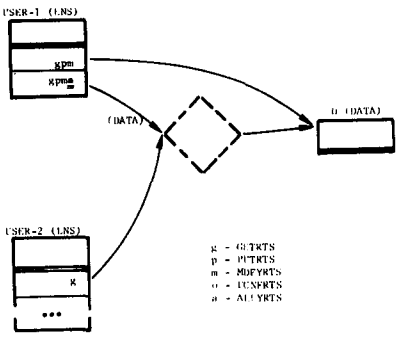

Figure 3.6

The important issue in selective revocation is how to selectively specify the class of users that have their accesses revoked. The solution adopted by Hydra was to allow an Alias (a new kind of entity) to be interposed between a capability and the object it referenced. Figure 3.6 shows how an Alias may be created for an object and a capability for the Alias shared with another user. A new capability references the Alias which contains a pointer to the original object. The Alias is ordinarily completely transparent and all accesses proceed as if the Alias were not there at all. The capability created for the Alias contains a new right, ALLYRTS, which may be exercised to break the link between the Alias and the object it points to, thus effecting revocation.[22]

The Temporal Revocation problem is solved since, with ALLYRTS, the link between an Alias and an object may be re-established. Re-allying requires a capability both for the alias (with ALLYRTS) and a capability for the object originally referenced by the alias before revocation took place.

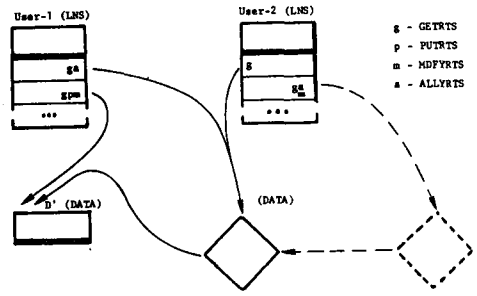

Figure 3.7

Figure 3.7 illustrates how Aliases may be nested. In the diagram, User-1 has shared an Alias capability with User-2. But User-2 (perhaps not even aware that his capability is for an Alias) has interposed yet another Alias so that he may additionally revoke access from anyone with whom he shares the capability. User-2's ALLYRTS will allow him to break or re-ally the link between the old Alias and the new Alias, but not between the old Alias and the actual object.

We promised early in this paper that our solution to various protection problems would not introduce additional mechanisms other than additional interpretation of Kernel rights. We admit that Aliases are an exception to that statement. However, we don't believe that they contradict the more important criterion that additional mechanism not "clutter up" the system design. With additional effort we could have implemented partial revocation as well. We were not completely convinced that partial revocation was necessary, and the additional mechanism that would have been required to implement it would have caused what we felt was unnecessary clutter.

On the other hand, we could envisage a number of uses for temporal revocation. Access to sensitive data might be revoked except from 9 a.m. to 5 p.m. on weekdays. Or, access to a generally available list structured library might be temporarily revoked while the structure was being garbage collected or reorganized.

We were skeptical about partial revocation because we believed that it would be most useful in Hydra when a user erroneously made an object available to an untrustworthy user with rights allowing modification rather than read access alone.[23] Unfortunately, in Hydra, aliases involve a certain overhead and in the situation above it seemed equally likely that the user would have neglected to use Aliases in the first place.

Even if such an error were made, the original holder of the capability could simply make a copy of the object and then destroy the original, effectively causing revocation. The only question that remains is how to disperse the new copy to those who legitimately could read the original.

21. The implementation of aliases is not yet complete. Unfortunately, neither is Destroy, the k-call which would effect non-selective revocation.

22. Since breaking or not breaking a link can be used to encode information, an absence of MDFYRTS (actually UCNFRTS) masks ALLYRTS as well as MDFYRTS.

23. Partial revocation is useful in implementing revocation of the right to revoke. However, we felt that there would be little need for such a feature.

Fortunately, early experience with Hydra indicates that users will not hold tightly onto capabilities for shared objects. Rather, we expect that some kind of Directory subsystem will be made generally available (a primitive version currently available is used very heavily) and that the Directory subsystem will supply procedures to store, retrieve and control the access to various objects. Users will likely only retrieve a capability from the Directory subsystem when needed, retaining the capability only as long as it is useful. Thus, a copy of a destroyed object can be dispersed simply by having a capability for it replace one for the destroyed object in the appropriate directory. Users still holding onto a capability for the destroyed object will eventually find they can no longer use it and can retrieve its replacement from the directory.

The introduction of Aliases was not without difficulties. It raised an interesting issue (also discussed by Redell) that clearly points out the tradeoff between permitting and preventing revocation.

Operations that manipulate the representation of non-kernel-supported objects generally take place in procedures (like Datafile-Init) that comprise the protected subsystem for that object's type. During execution of such procedures, it is likely that for some amount of time the object may be in an inconsistent state. If access to the object is revoked during that time, subsequent calls on the protected subsystem, especially with that object as an argument, may produce undefined results.

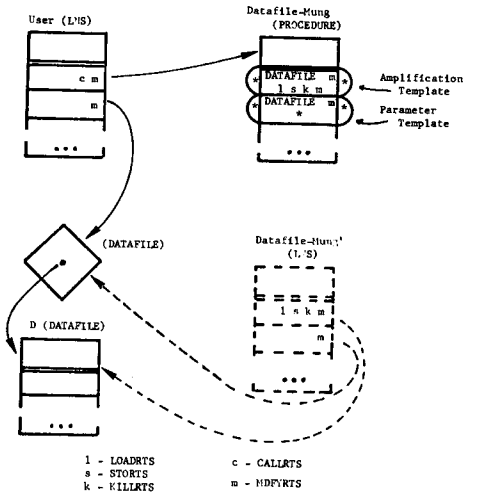

Figure 3.8

Hydra's solution to this problem is illustrated in figure 3.8. When amplification takes place, the new capability references the actual object rather than the Alias. Thus, revocation will not actually take effect while the protected subsystem is accessing the object. Since it is not possible in Hydra to tell the revoker when effective revocation has occurred (when the subsystem procedure has returned), it seems possible that the following situation might occur: User-1 revokes User-2's access to a file, while User-2 is writing via a call to a File subsystem procedure. User-1 then reads the file while it is still being written on behalf of User-2, producing an inconsistent result.

The error in the argument is that User-1's reading of the file will be effected by a call on a File subsystem procedure as well, and that procedure can be constructed so as to wait until a write of the same file is completed. Admittedly this is not a completely adequate answer for a system specified in a way that allows multiple simultaneous updates of a file.

Allowing amplification to circumvent revocation has another interesting side-effect - it allows an unsavory subsystem to subvert revocation. If the unsavory subsystem makes an amplifying template generally available, a user given an Aliased capability can acquire a capability for the actual object and thus revocation will have no effect. Clearly, we don't expect this kind of behavior on behalf of a subsystem. This problem points up yet again the delicate balance between trusting a subsystem and trying to find mechanisms that force it to fulfill certain of its specifications.

Finally, we emphasize that Hydra revocation only revokes access, not information. Users may realize too late that they have made read access available to information that they considered confidential. Revocation as described here does not solve that problem. Another user may have already copied that information onto a listing device and no mechanism residing purely within the computer system can bring it back.

3.7.2 Guaranteeing Access - Freezing

Users will often want assurance that access to an object will not be revoked at awkward times. At first there may appear to be a large class of such problems, but some analysis indicates otherwise. First, the case of guaranteed access during execution of a subsystem was discussed in the previous section, so we need only concentrate on revocation that affects "users". There are two instances - a user wants guaranteed access to an object he simply "uses" - reads or executes - or he wants guaranteed access to an object he modifies as well.

When two users share the right to modify an object, either they are cooperating or they are not. If they are cooperating, why are they doing something as bizarre as unexpected revocation? There are better ways to reach agreements about usage of the object.

If they are not cooperating, we must wonder why they are sharing modify access. It seems that the only reasonable answer is that one user has contracted with a second user to perform some function on the shared object. Revoking access indicates that the first user is revoking the contract. The only issue facing the second user is - how can the first user be forced to pay for resources expended by the second user (perhaps plus punitive costs) before revocation took place. Such problems can be dealt with by user subsystems in ways similar to that described in the following section.

The cases left to consider involve guarantees during reading and execution. Once a procedure has been called, a separate LNS is incarnated, so revocation of the procedure is not a problem. However, a user may have expended resources predicated on his continuing ability to read a file or execute a procedure. But a guarantee against revocation is in fact not sufficient. Another user may simply zero the file or change the procedure. Thus a guarantee against revocation is only useful coupled with a guarantee that no other user may modify or destroy the object.

All of this is accomplished in Hydra by freezing.[24] Once the creator of an object has placed within it those capabilities and data that it will permanently contain, she may freeze it. The capability for the object will have FRZRTS set and MDFYRTS removed. The object can be frozen only if all capabilities in the object's C-list are already frozen (have FRZRTS set), thus FRZRTS guarantees not only that the contents of the object remain permanently fixed, but those of all objects in its representation as well. Like MDFYRTS and ENVRTS, FRZRTS cannot be gained through amplification.

Furthermore, aliases cannot be frozen, and while an alias can be made from a frozen capability, the capability created for the alias will not contain FRZRTS. Thus a capability containing FRZRTS acts as a guarantee against revocation as well.

It is worth noting once again just how useful such guarantees are for publicly available procedures. Users of a computer system cringe so often when a compiler has been erroneously modified and no backup version has been made available. If users would demand frozen versions, new versions would necessarily have to be made available in a different (and hopefully more humane) way.[25]

3.8 The Accounting and Lost Object Problems

Let us consider a Track subsystem responsible for managing a disk and which makes available objects of type Track. Users having a Track capability may read or write the disk track it represents by making calls on procedures supplied by the Track subsystem.

The Hydra philosophy asserts that objects do not inherently have owners. All holders of a capability share responsibility for it - although there may be de-facto ownership in the sense that one user has rights to access the object more powerful than other users. In the case of a limited resource, such as a track, it does become relevant to ask who pays for it? This is important not only as a billing issue, but also in the sense of guaranteeing that some user will not hog the resource. Other systems, such as one developed at SRI [Neu74], force such objects to be held in accounting directories. Even though Hydra itself does not enforce such a policy, it is easy to construct a subsystem which does.

Let us suppose that whenever a user logs in, the Login routine places in the user's initial LNS a capability for an object uniquely identifying the user.[26] Before supplying a new track, the Track subsystem demands a capability for the User object and uses that to determine who will pay for the track. If the user has overused her disk allocation, then the Track subsystem could simply refuse to provide a new track. If mistakes are made, revocation or destruction of the Track object can allow the Track subsystem to reallocate the physical track.

Unfortunately, neither this (nor the SRI system) solves the Lost Object Problem. Suppose that even though a user has committed herself to paying for a track, she deletes all of her capabilities for the track without notifying the Track subsystem. Billing is not the issue - the problem is that the track has been lost. If the disk is oversubscribed, it should not be necessary to revoke some other user's track if the subsystem has discovered that some lost (unwanted) track is available.

Of course, the fact that a user has deleted all of her capabilities for the track does not necessarily indicate that the track was unwanted. In many systems, users often bemoan the fact that they have mistakenly deleted an irreplaceable file. In either case, it is clear that the ability to retrieve some types of objects when all capabilities for them have been deleted is quite useful.

In such a situation, Conservation or limiting propogation of capabilities is extremely important. If a user mistakenly deletes all of her capabilities for an object while a capability is still retained by the "demonic user", the object will not be retrieved.

In Hydra, a decision about retrievability may be made on a type by type basis when a new type is first created. A mechanism is provided which will retrieve a capability for an object of that type whose capabilities have all been deleted.[27] A subsystem can use this mechanism to provide a wide range of policies with respect to lost objects.

Through the construction of subsystems, procedures provide the mechanism necessary for the implementation of general policies involving future negotiated decisions. Subsystem procedures can contain the code that can decide, as situations arise, whether a user can be given access to a scarce resource like a track, an object previously lost, or some other object for which a prior unilateral access decision is inappropriate. It is subsystem procedures that can allow users to negotiate and make and break rules permitting access to information.

4. Conclusion

We have described how Hydra has solved a number of interesting protection problems by simply extending the interpretation of rights rather than providing a pastiche of unrelated mechanisms (except for Aliases and Retrievability).

24. Like the alias mechanism, freezing also is not yet implemented.

25. FRZRTS also can be used to solve a problem described by Rotenberg [Rot73] as the Blind Service Problem. Even frozen procedures can get hold of system information, such as time of day, which varies from call to call. Rotenberg describes situations where this information may be used to sabotage the execution of a particular caller. We expect to solve this problem in Hydra by preventing procedures called through capabilities containing FRZRTS from obtaining such system information.

26. The details are beyond the scope of this section - but it is important to note that the notion of a "user" is not required by Hydra and can be provided by a User subsystem which has a capability for the device on which the user logs in.

27. It is interesting to note that retrieving a lost object can be used to provide a channel of fairly high bandwidth. In order to prevent gross covert leakage out of confined LNSes, we do not allow objects of retrievable types to be created by confined LNSes. If we did, a confined LNS could leak the integer n by creating and deleting n objects of a retrievable type.

As noted by Peuto [Peu74] in his comparative study of Real Estate Law and Protection Systems, sophisticated protection desired by users of a protection system is, in principle, no different than that desired by parties to a legal contract. There are the attendant issues of remedies and adjudication to be considered if the contract is broken, either purposefully by one of the parties, or accidentally due to machine or program error.

It is certainly possible that mechanisms more sophisticated than that provided directly may be desirable, but we hope that the Track example has convinced the reader that subsystems can fill this need. More complex protection needs lead inevitably to tradeoffs between the desire to restrict or revoke access and the desire to guarantee certain kinds of behavior (not only guarantees against revocation). Our difficulty with revocation during subsystem calls was a simple example of this tradeoff and leads us to conclude in general that such decisions should be made by users (through subsystems constructed by them) rather than by the protection system directly.

We can make no promise that our list of protection problems is complete. It is a list of those that we have tried to solve in our design of Hydra and those discovered by other operating system builders. From that perspective, and from the limited experience of those already using HYDRA, we believe that the protection mechanisms we have provided adequately and reasonably meet the needs of Hydra users.

Of course, the approach taken by Hydra is based in the world of programming languages and file systems. The large Data Base systems of the future will likely have different protection needs. Access to information may be intimately tied to the values of the information itself as well as the history of previous access and the expectations of future access.

Nonetheless, the problems we have discussed here will remain. It is our hope that the essentially simple mechanisms we have provided will encourage builders of future systems to realize that they can do the same. Sadly, it seems that is the only hope for insuring the privacy of private information so madly collected and compulsively stored by this information hungry society.

5. Acknowledgements

The authors wish to express their appreciation to those who were active in the implementation and design of the Hydra system, notably Bill Wulf and Anita Jones. We also wish to thank numerous members of the MIT community whose comments on an early presentation of this material helped considerably in improving the contents of this paper.

6. References

Coh75 Cohen, E., **Modelling Protection**, Ph. D. thesis, Carnegie-Mellon University (to appear).

Cos74 Cosserat, D. C., "A Data Model Based on the Capability Protection Mechanism", **International Workshop on Protection in Operating Systems**, IRIA, 1974.

Dah66 Dahl, O.-J., and Nygaard, K., "Simula - An Algol-Based Simulation Language", **Communications of the ACM 9**, 9 (September 1966).

Din73 Dingwall, T. J., **Communication within Structured Operating Systems, Cornell University Computer Science Dept.**, TR 73-167, May 1973.

Fab74 Fabry, R., "Capability-Based Addressing", **Communications of the ACM 17**, 7 (July 1974).

Gra72 Gray, J., Lampson, B., Lindsay, B., Sturgis, H., **The Control Structure of an Operating System, IBM Research** RC 3949, July 1972.

Jon73 Jones, A., **Protection in Programmed Systems**, Ph. D. thesis, Carnegie-Mellon University, June 1973.

JL75 Jones, A. and Lipton, R., "The Enforcement of Security Policies for Computation", **Proceedings of the 5th Symposium on Operating System Principles**, Austin, Texas, Nov. 1975.

Lam69 Lampson, B. W., "Dynamic Protection Structures", **AFIPS Conference Proceedings**, FJCC 1969.

Lam73 Lampson, B., "A Note on the Confinement Problem", **Communications of the ACM 16**, 10 (October 1973).

Lau74 Lauer, H. C., "Protection and Hierarchical Addressing Structures", **International Workshop on Protection in Operating Systems**, IRIA, 1974.

Lev75 Levin, R., Cohen, E., Corwin, W., Pollack, F., Wulf, W., "Policy/Mechanism Separation in HYDRA", **Proceedings of the 5th Symposium on Operating System Principles**, Austin, Texas, Nov. 1975.

Lis74 Liskov, B., **A Note on CLU**, Computation Structures Group Memo 112, MIT Project MAC, Nov. 1974.

Mor73 Morris, J., "Protection in Programming Languages", **Communications of the ACM 16**, 1 (January 1973).

Neu74 Neumann, P. et. al., "On the Design of a Provably Secure Operating System", **International Workshop on Protection in Operating Systems**, IRIA, August 1974.

Org73 Organick, E., **Computer System Organization: The B5700/B6700 Series**, ACM Monograph Series, Academic Press, 1973.

Peu74 Peuto, B., **Comparative Study of Real Estate Law and Protection Systems**, Ph. D. thesis, University of California at Berkeley, ERL-M439, May 1974.

Red74a Redell, D., **Naming and Protection in Extendible Operating Systems**, Massachusetts Institute of Technology, MAC TR-140, November, 1974.

Red74b Redell, D. and Fabry, R., "Selective Revocation of Capabilities", **International Workshop on Protection in Operating Systems**, IRIA, 1974.

Rot73 Rotenberg, L., **Making Computers Keep Secrets**, Ph. D. thesis, Massachusetts Institute of Technology, MAC TR-116, September 1973.

Sch72 Schroeder, M., **Cooperation of Mutually Suspicious Subsystems in a Computer Utility**, Ph. D. thesis, Massachusetts Institute of Technology, MAC TR-104, Sept. 1972.

Wal74 Walter, K., et. al., **Primitive Models for Computer Security**, Case Western Reserve University Technical Report ESD-TR-74-117, January 1974.

Wei69 Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System", **AFIPS Conference Proceedings 35**, FJCC 1969.

Wul74b Wulf, W., **Alphard: Toward a Language to Support Structured Programs**, Carnegie-Mellon University Technical Report, 1974.

WLP75 Wulf, W., Levin, R., Pierson, C., "An Overview of the HYDRA Operating System Development", **Proceedings of the 5th Symposium on Operating System Principles**, Austin, Texas, Nov. 1975.