Protection[1]

Butler W. Lampson
Xerox Corporation
Palo Alto, California

Abstract

Abstract models are given which reflect the properties of most existing mechanisms for enforcing protection or access control, together with some possible implementations. The properties of existing systems are explicated in terms of the model and implementations.

Introduction

'Protection' is a general term for all the mechanisms that control the access of a program to other things in the system. There is an immense variety of such mechanisms. Indeed, it is normal for a single system to have a number of different and unrelated ones: for example, a supervisor/user mode, memory relocation and bounds registers, some kind of file numbers for open files, access control by user to file directories, and a password scheme for identification at logon. It therefore seems desirable to take a rather abstract approach to the subject.

Matters are somewhat confused by the fact that a system can be complete from the point of view of a community of friendly and infallible users, without any protection at all. This observation should not, however, be taken to imply that protection is a minor consideration. On the contrary, real users are far from infallible, not to mention their programs, and privacy is a crucial issue in most real systems. As a result, protection considerations are pervasive in any system design.

The original motivation for putting protection mechanisms into computer systems was to keep one user's malice or error from harming other users. Harm can be inflicted in several ways:

a) By destroying or modifying another user's data.

b) By reading or copying another user's data without permission.

c) By degrading the service another user gets, for example, using up all the disk space or getting more than a fair share of the processing time. An extreme case is a malicious act or accident that crashes the system—this might be considered the ultimate degradation.

[1] This paper first appeared in *Proc. 5th Princeton Conf. on Information Sciences and Systems,* Princeton, 1971, p 437. It was reprinted in *ACM Operating Systems Rev.* **8**, 1 (Jan. 1974), p 18. This version was created from the Princeton version of the paper by scanning and OCR; it may have some errors.

More recently it has been realized that all of the above reasons for wanting protection are just as strong if the word ‘user’ is replaced by ‘program’. This line of thinking leads in two main directions:

a) Towards enforcing the rules of modular programming so that it is possible, using the protection system, to guarantee that errors in one module will not affect another one. With this kind of control it is much easier to gain confidence in the reliability of a large system, since the protection provides fire walls which prevent the spread of trouble [4, 9].

b) Towards the support of proprietary programs, so that a user can buy a service in the form of a program which he can only call, but not read [9]. A simple example might be a proprietary compiler whose service is sold by number of statements compiled. A more complex case is a program that runs test cases against a proprietary database as well.

Another example may suggest that some generality is really required to handle those problems, rather than a few ad hoc mechanisms. This is the construction of a routine to help in the debugging of other programs. Such a debugger needs to be able to do all the things which the program being debugged can do, but must protect itself and its data (breakpoints, symbol tables, etc.) from destruction by malfunctions in the program being debugged [8, 9].

The models for protection systems which occupy the body of this paper are arranged in order of decreasing generality and increasing complexity. It is not clear whether to take this correlation as a cause for satisfaction or for despair.

Protection domains

The foundation of any protection system is the idea of different protection environments or contexts. Depending on the context in which a process finds itself, it has certain powers; different contexts have different powers. A simple example of a two context system implemented in hardware is any computer with supervisor and user or problem states. A program in supervisor state can execute I/O instructions, set the memory protection registers, halt the machine, etc. In user state these powers are denied to it. A somewhat more elaborate example is OS/360 MVT, where there is one supervisor context and up to 15 user contexts; the limit of 15 is enforced by the use of 4-bit keys in the 360’s memory protection system. Still another example is individual users of a multi-access system—each one has his own protected program and files, so that there are at least as many protection contexts as there are users [15].

The variety of these examples is intended to suggest the generality and subtlety of the idea. Many words have attempted to capture it: protection context, environment, state or sphere [3], capability list [10], ring [4], domain [9]. The relatively neutral word ‘domain’ will be used here, since it has fewer misleading associations than any alternative. An idealized system called the *message system* is described below in order to clarify the meaning of this term and provide a framework for the description of real systems.

The message system consists of processes that share nothing and communicate with each other only by means of *messages*. A message consists of an identification of the sending process followed by an arbitrary amount of data. The identification is supplied by the system and therefore cannot be forged. Processes are assigned integer names by the system in such a way that a given integer always refers to the same process; aside from this property the names have no significance. The sending process supplies the data. Any process may send messages (at its expense) to any other process. Messages are received one at a time in the order in which they were sent. See [5] for an actual system very similar to this one, but described from a somewhat different viewpoint.

Within this system everything belongs to some process and cannot be accessed by any process other than its owner. Each process is therefore a single domain. It can also be viewed as a separate machine, complete with memory, file storage, tape units, etc., and isolated by hardware from all other processes except for the message transmission system described above. This scheme provides a logically complete (though in general rather inefficient) protection system (except for two points that are discussed below). Let us examine what can be done with it in order to bring out its essential features.

The first point (which has nothing to do with protection) is that we can simulate a subroutine mechanism in this system, viewing one process (A) as the calling routine and another (B) as the routine being called. To call B, A sends B a message specifying the parameters and then waits for B to reply. To return, B replies with another message containing the value, if any, and then waits for another call.

We now observe that unlike an ordinary subroutine call, this works even if B must be protected from A, for example, if B is the supervisor and A a user program. It works because B determines where it is ‘entered’, namely at the point where it waits for A’s message. Random transfers of control to an arbitrary point in B are not possible. This is not to say that multiple entry points are not possible, since B can decode one of the parameters to select an entry point.

Furthermore, the ‘return’ is also protected. Thus, if A mistrusts B (for example, in the case of a command processor calling a user program) the same argument shows that B will not be able to return to A except in the manner intended by A. Spurious additional ‘returns’ (extra messages) from B are equally impossible, since A knows when it is expecting a return message from B and can ignore messages at other times. The scheme clearly works even if each domain mistrusts the other, as in the case of calling a proprietary program [10].

What if A calls B by this mechanism and B never returns, either because B has a bug or because it is malicious? If A wishes to guard against this possibility, it need only arrange, before calling B, to receive a message from some reliable system process C after a certain amount of time has elapsed which is longer than B is expected to run. If the message A receives next is from C rather than from B, then A knows something has gone wrong and can proceed to take corrective action.

Finally, suppose that some unauthorized domain Y attempts to call B. Recall that as part of each message the system supplies the identity (name) of the caller. This identification may be thought of as a signature, a seal, a badge, etc., which B can use to check the validity of the call. The essential point is that the identification is supplied by the system, which guarantees that it cannot be forged. This point is so simple, and yet so subtle, that we will illustrate it with an example. Suppose that A, whose name is 6389, sends to B a message consisting of three numbers: 31, 45, 9. What B will receive is four numbers: 6389, 31, 45, 9. The first number is supplied by the system from its knowledge of the identity of the sender. There is no way for A to affect the value of the first number in the message. From B’s point of view then, the message starts with a single identifying integer. If B is expecting a message from A, all it needs to do for each message is to check that it starts with A’s name. How B gets to know A’s name is an interesting question which will be pursued below, but for the moment perhaps the following simple scheme will suffice: A’s user at his terminal asks A for the number and shouts it across the room to B’s user, who types it into B. Remember that this number is *not* a password. Knowing it allows B to give A access, but does not help anyone else (including B) to impersonate A, as the description of message handling given above should make perfectly clear.

The kind of protection or access control that can be enforced with this system is extremely flexible and general, since arbitrary programs can be written by users to make the protection decisions. Suggested exercise: show how an instructor could implement a grading program which gives student programs at most three tries at obtaining a correct answer.

As was mentioned earlier, the system we have been describing has two major flaws:

a) It is impossible to retain control over a runaway process, since there is no way to force a process to do anything, or to destroy it. Although such a process cannot do any damage, it can waste resources. This makes debugging difficult.

b) An elaborate system of conventions is required to get processes to cooperate. Suppose, for example, that process A has some data that it wants to share with several other processes belonging to friends of the user who owns process A. A’s owner (whom we think of as another process communicating with A via a terminal) must find out the names of his friends’ processes and put some program into A which knows these names and responds appropriately to messages carrying these names as identification. In addition, A and its correspondents must agree on the interpretation of messages.

The message system is as bare of convenience as a central processor is for executing programs. The processor needs an elaborate system of conventions in the form of loaders, binary formats, assemblers, etc., to make it usable, and these appurtenances quickly take on a life of their own. The same thing must happen to our primitive protection system. Some of the issues raised by these two points are discussed in the next section.

Objects and access matrices

In order to provide facilities for controlling processes from outside, it is necessary to have a systematic way of controlling access to one process from others. In order to provide useful conventions for sharing among processes, it is necessary to have a systematic way of describing what is to be shared and of controlling access to shared things from various processes. Access to processes can be controlled by a simple tree structure [5, 8], but it can also be handled more generally by the same machinery that we will establish to solve the second problem. (It is not at all clear that the scheme described below is the only, or even the best, set of conventions to impose, but it does have the property that almost all the schemes used in existing systems are subsets of this one.)

This machinery can be described in terms of another idealized system called the *object system*. It has three major components: a set of *objects* which we will call *X*, a set of domains which we will call *D*, and an *access matrix* or access function which we will call *A*. Objects are the things in the system which have to be protected. Typical objects in existing systems are processes, domains, files, segments, and terminals. Like everything we are describing, the choice of objects is a matter of convention, to be determined by the protection requirements of each system. Objects have names with global validity, which we will think of as 64-bit integers. Object names are handed out by the protection system on demand, and their interpretation is up to the programs that operate on the objects. This point is supposed to be clarified by the file handler example below.

Domains were introduced in the previous section. They are the entities that have access to objects. The essential property of a domain is that it has potentially different access than other domains. In the message system, each domain was also a process and had exclusive access to its own objects and none to any others. This idea is now being generalized so that objects can be shared between domains. There are actually two ways to look at this generalization.

a) Each domain owning objects in the message system agrees by convention that it will do certain things with these objects upon demand from some other domain, provided the other domain has access according to the rules below.

b) At least for certain 'built-in' objects the access rules below will be enforced by the underlying machinery which implements the system (whether this is hardware, as in the case of memory protection, or software, as in the case of file directories, is not important). This viewpoint may lead to greater efficiency (memory protection is an extreme example) but it is clearly not as general and must be supplemented by (a) if the system is to be extensible. As far as the protection system is concerned, it makes no difference which view is taken.

Note that domains are objects, and that objects do not 'live in", or 'belong to' domains.

The access of domains to objects is determined by the access matrix *A*. Its rows are labeled by domain names and its columns by object names. Element $A_{i,j}$ specifies the access which domain *i* has to object *j*. Each element consists of a set of strings called *access attributes*; typical attributes are 'read', 'write', 'wakeup'. We say that a domain has 'x' access to an object if 'x' is one of the attributes in that element of *A*. Attached to each attribute is a bit called the *copy* flag which controls the transfer of access in a way described below. With the *A* of figure 1, for example, domain 1 has 'owner' access to file 1 as well as explicit 'read' and 'write' access. It has given 'read' access to this file to domains 2 and 3.

Entries in the access matrix are made and deleted according to certain rules. A domain d_1 can modify the list of access attributes for domain d_2 and object *x* as follows (examples assume the access matrix of figure 1):

	Domain 1	Domain 2	Domain 3	File 1	File 2	Process 1
Domain 1	*owner control	*owner control	*call	*owner *read *write		
Domain 2			call	*read	write	wakeup
Domain 3			owner control	read	*owner	

*copy flag set

Figure 1: Portion of an access matrix

a) d_1 can remove access attributes from $A_{d_2,x}$ if it has 'control' access to d_2. Example: domain 1 can remove attributes from rows 1 and 2.

b) d_1 can copy to $A_{d_2,x}$ any access attributes it has for x which have the copy flag set, and can say whether the copied attribute shall have the copy flag set or not. Example: domain 1 can copy 'write' to $A_{2,file\ 1}$.

c) d_1 can add any access attributes to $A_{d_2,x}$, with or without the copy flag, if it has 'owner' access to x. Example: domain 2 can add 'write' to $A_{2,file\ 2}$.

The reason for the copy flag is that without it a domain cannot prevent an undebugged subordinate domain from wantonly giving away access to objects.

The rules above do not permit the 'owner' of an object to take away access to that object. Whether this should be permitted is an unresolved issue. It is permitted by most systems; see [13] for a contrary view. If it is to be permitted, the following rule is appropriate.

d) d_1 can remove access attributes from $A_{d_2,x}$ if d_1 has 'owner' access to x, provided d_2 does not have 'protected' access to x.

The 'protected' restriction allows one owner to defend his access from other owners. Its most important application is to prevent a program being debugged from taking away the debugger's access, it may be very inconvenient to do this by denying the program being debugged 'owner' access to itself.

One peculiarity of rules (b) and (c) is that a domain cannot prevent a subordinate domain from cooperating with a third, independent domain, for example, from cheating on a test. This could be remedied by requiring d_1 to have 'augment' access to d_2 in those two rules.

The system itself attaches no significance to any access attributes except 'owner', or to object names. Thus the relationship between, say, the file handling module and the system is something like this. A user calls on the file handler to create a file. The file handler asks the system for a new object name n, which the system delivers from its stock of object names (for example, by incrementing a 64-bit counter). The system gives the file handler 'owner' access to object n. The file handler enters *n* in its own private tables, together with other information about the file which may be relevant (for example, its disk address). It also gives its caller 'owner' access to *n* and returns *n* to the caller as the name of the created file. Later, when some domain *d* tries to read from file *n*, the file handler will examine $A_{d,n}$ to see if 'read' is one of the attributes, and refuse to do the read if it is not.

Some implementation techniques

Since *A* is sparse, it is not practical to store it in the obvious way. The simplest alternative is a global table *T* of triples $<d, x, A_{d,x}>$ which is searched whenever the value of $A_{d,x}$ is required. Unfortunately, this is usually impractical for a number of reasons:

a) Memory protection is almost certainly provided by hardware which does not use *T*. This is the major area in which the operating system designer has little control; it is discussed in the next section.

b) It may be inconvenient to keep all of *T* in fast-access memory, since at any given tine most objects and perhaps most domains will be inactive. An implementation is therefore needed which helps to keep the currently relevant parts of *A* readily available.

c) Objects and/or domains may be grouped in such a way that *T* is very wasteful of storage. A simple example is a public file, which would require a table entry for every domain.

d) It may be necessary to be able to obtain a list of the objects which a given domain *d* can access, or at least the objects for which *d* is responsible or is paying for.

An implementation which attacks (b) and (d) directly is to attach the *d*'th column of *A* to the domain *d* in the form of a table of pairs $<x, A_{d,x}>$. One of these pairs is usually called a *capability* [3, 7, 9, 10, 15].

If the hardware provides for read-only arrays which can only be generated by the supervisor, then each capability can be implemented as such an array, containing:

- The name of the object (a 64-bit integer),
- A suitable representation of the access attributes (perhaps as a bit string).

Most hardware does not provide the kind of protected arrays we have been assuming, but they can easily be simulated by the supervisor, at some cost in convenience, on any machine with memory protection of any kind. When this is done, it is usually convenient to group capabilities together into capability lists or *C-lists*. A domain is then defined by a C-list (and its memory, if that requires special handling; see the next section).

With this kind of implementation it may be convenient to allow additional information to be stored in a capability, for example, the disk address in the file handling example above, or a pointer to some table entry to save the cost of looking up the object name [10]. It is an interesting exercise to devise a mechanism for controlling who gets to alter this additional information.

Capabilities can also be used to attack problem (c) above. All we have to do is observe that it is possible to build an arbitrary graph of domains, each with a set of capabi1ities or C-list [3, 7, 13].

Everything we know about tree structured naming schemes can then be applied to economize on storage for shared capabilities.

A completely different approach is to attach the protection information to the object rather than the domain. In its most general form, the idea is to provide a procedure $A_x(d)$ for each object. The procedure is provided by the owner of the object and can keep its own data structures to decide who should get access. Note that at least some of these procedures will have to refrain from accessing any other objects in order to prevent infinite recursion.

The essential point is that the procedure gets the domain's name as argument, and this cannot be forged. However, unique names may not be very convenient for the procedure to remember—access is more likely to be associated with a person or group of people, or perhaps with a program. The ideas of the last few paragraphs can be applied here—capabilities can be used as identification, since they have the essential property that they cannot be forged. We will call a capability used for identification an *access key*; it is a generalization of a domain name as that was used in the message system [9].

All the access control procedure needs to know is what access keys to recognize. Each user, indeed each entity which needs to be identified by an access control procedure, gets hold of a unique access key by asking the supervisor for it, reads the value, and transmits the value to the people who wish to grant him access. They then program their access control procedures to return the desired attributes when that key is presented as an argument.

It is often inconvenient to call arbitrary procedures. A less general scheme is to attach to each object a list which consists of <key value, access attributes> pairs. It seems reasonable to call this an *access control list* for the object. It works in the obvious way: if the value of the key presented matches the value in one of the locks, the corresponding attribute is returned.

Another way to look at this scheme is as a generalization of one of the first protection systems, that of CTSS, which instead of the key value employed the name of the user as identified at login [1, 9, 15].

One access list per object is likely to be cumbersome. Most systems group objects into *directories*, which in turn are objects, so that a tree structure can be built up. This adds nothing new, except that it introduces another kind of tree structured naming [15].

We observe that a directory is not too much different from a domain in structure. The access key method of obtaining access is, however, quite different in spirit from the capability method. It is also likely to be more expensive, and many systems have a hybrid implementation in which an object can be accessed once by access key to obtain a capability, which is then used for subsequent accesses. This process when applied to files is usually called *opening* a file [8, 9].

Memory protection

Memory protection hardware is usually closely related to mapping or relocation hardware. There are two aspects to this:

a) Memory that is not in the range of the map cannot be named and is therefore protected.

b) In paged or segmented systems (even 2-segment ones like the PDP-10) each page or segment in the map has protection information associated with it.

The essential point is that each domain must have its own address space, since otherwise there can be no protection [9,10]. It is also highly desirable for one domain to be able to reference the memory of another, subject to the control of the access function.

A segmented system in which any segment is potentially accessible from any domain fits well into the framework of the last two sections, usually with the C-list implementation [2, 4]. It may be a slight annoyance that a segment capability has a different form than other capabilities, but this is a minor problem. Some difficulties may arise, however, in connection with transfers of control from one domain to another, since it is unlikely that the hardware will allow path names in a tree of C-lists to be used as addresses [4].

In the absence of segmentation either pages or files may be treated as objects. With paging, data can be dynamically shared, and since the contents of the map can be changed when changing domains, there is a feasible, though far from elegant, means of passing memory around when necessary while preserving the security of each domain.

In the absence of paging, each domain will have to have private memory in general, memory that is not accessible to any other domain except through some ugly circumlocution. The naming problems that result are beyond the scope of this paper.

There is one exception to this observation, for the case of nested domains $d_1, \ldots, d_n$ (that is, $A_{d_1,x} \supset \ldots \supset A_{d_n,x}$ for all x) on machines with base and bound relocation: simply arrange the memory for the domains contiguously, with d_1 nearest to 0, and set the bound for d_1 to the total length of all the memory, for d_2 to the total length excluding d_1, etc. Now only a simple addition is required for d_i to interpret addresses in d_j, $i < j$ [6,10].

References

1. P. A. Crisman, ed., *The Compatible Time-Sharing System: A Programmer's Guide*, second edition. MIT. Press, Cambridge, MA, 1965.

2. Dennis, J. B., Segmentation and the design of multiprogrammed computer systems. *J. ACM* **12**, 4 (Oct. 1965), p. 589.

3. Dennis, J. B. and van Horn, E., Programming semantics for multiprogrammed computation. *Comm. ACM* **9**, 3 (March 1966), p. 143.

4.. Graham, R. M., Protection in an information processing utility. *Comm. ACM* **11**, 5 (May 1968), p. 365.

5. Hansen, P. B., The nucleus of a multiprogramming system. *Comm. ACM* **13**, 4 (April 1970), p. 238.

6. Harrison, M. C., Implementation of the SHARER2 time-sharing system. *Comm. ACM* **11**, 12 (Dec. 1968), p. 845.

7. Illffe, J. K., *Basic Machine Principles*, American Elsevier, New York, 1968.

8. Lampson, B. W. et al, A user machine in a time-sharing system. *Proc. IEEE* **54**, 12 (Dec. 1966), p. 1766.

9. Lampson, B. W., Dynamic protection structures. *Proc. AFIPS Conf.* **35** (1969 FJCC), p. 27.

10. Lampson, B. W., On reliable and extendible operating systems. *Proc. 2nd NATO Conf. on Techniques in Software Engineering*, Rome, 1969. Reprinted in *The Fourth Generation*, Infotech State of the Art Report 1, 1971, p 421.

11. Linde, R. R. et al, The ADEPT-50 time-sharing system. *Proc. AFIPS Conf.* **35** (1969 FJCC), P. 39.

12. Molho, L., Hardware aspects of secure computing. *Proc. AFIPS Conf.* **36** (1970 FJCC), p. 135.

13. Vanderbilt, D. H., *Controlled Information Sharing in a Computer Utility*. MAC TR-67, MIT, Cambridge, Mass., Oct. 1969.

14. Weissman, C., Security controls in the ADEPT-50 time-sharing system. *Proc. AFIPS Conf.* **35** (1969 FJCC) , p. 39.

15. Wilkes, N. V., *Time-Sharing Computer Systems*. American Elsevier, New York, 1968.