Chapter 14 System Protection

Goals of Protection

- Principles of Protection
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix
- Access Control
- Revocation of Access Rights
- Capability-Based Systems

Objectives

 Discuss the goals and principles of protection in a modern computer system

 Explain how protection domains combined with an access matrix are used to specify the resources a process may access

 Examine capability and language-based protection systems

Goals of Protection

Reason of protection :
Need to prevent the mischievous, intentional violation of an access restriction by a user.
Need to ensure that each program component active in a system uses system resources only in ways consistent with stated policies.

- Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem.
- An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.
- The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use.
- These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system.
- A protection system must have the flexibility to enforce a variety of policies. Policies for resource use may vary by application and they may change over time.
- For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse.
- Policies are likely to change from place to place or time to time.

Principles of Protection

- A key time-tested guiding principle for protection is the **principle of least privilege**.
- It dictates that programs, users and even systems be given just enough privileges to perform their tasks.

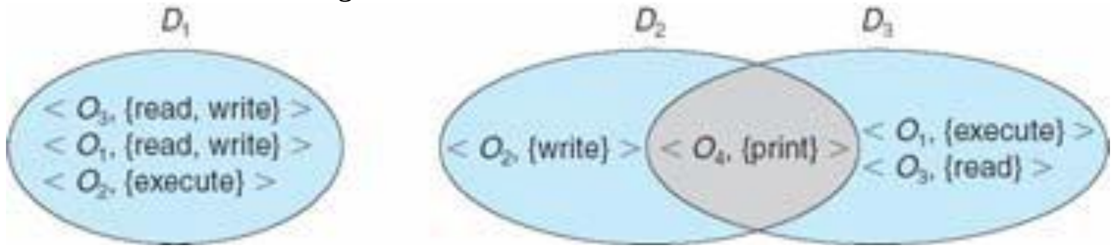
- An operating system also provides system calls and services that allow applications to be written with fine-grained access controls.
- It provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.
- Managing users with the principle of least privilege entails creating a separate account for each user, with just the privileges that the user needs.
- Computers implemented in a computing facility under the principle of least privilege can be limited to running specific services, accessing specific remote hosts via specific services, and doing so during specific times.
- The principle of least privilege can help produce a more secure computing environment.

Domain of Protection

- A computer system is a collection of processes and objects.
- Hardware objects - CPU, memory segments, printers, disks, and tape drives
- Software objects - files, programs, and semaphores).
- Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations.
- A process should be allowed to access only those resources for which it has authorization.
- Furthermore, at any time, a process should be able to access only those reso1Jrces that it currently requires to complete its task.
- This second requirement, cmmonly referred to as the *need-to-know* principle, is useful in
- limiting the amount of damage a faulty process can cause in the system.

Domain Structure

Access-right = <*object-name*, *rights-set*>
where *rights-set* is a subset of all valid operations that can be performed on the object.
Domain = set of access-rights



- For example, if domain D has the access right <*file F*, {read, write}>, then a process executing in domain D can both read and write file *F;* it cannot perform any other operation on that object.
- Domains do not need to be disjoint; they may share access rights.
- For example, in the above Figure, we have three domains: D1, D2, and D3 .
- The access right < 0 4, {print}> is shared by D2 and D3, implying that a process executing in either of these two domains can print object 0 4 .
- A process must be executing in domain D1 to read and write object 0 1, while only processes in domain D3 may execute object 0 1.
- The association between a process and a domain may be :
  Static - if the set of resources available to the process is fixed throughout the process's lifetime.

Dynamic - if the set of resources available to the process changes throughout the process's lifetime until it's execution.

Domain Switching : If the association is dynamic, a mechanism is available to allow enabling the process to switch from one domain to another.
Allows changes to the content of a domain. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

A domain can be realized in a variety of ways:

- Each *user* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed -generally when one user logs out and another user logs in.
- Each *process* may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each *procedure* may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

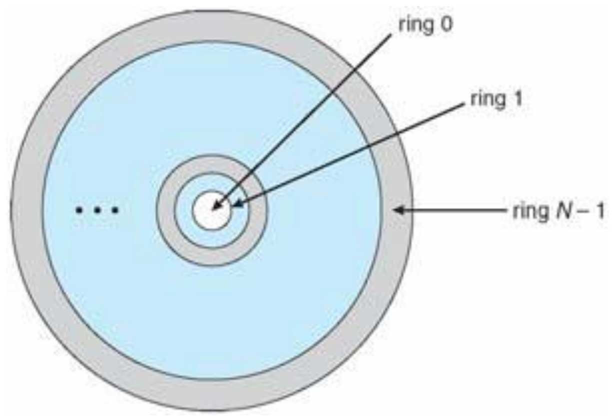
Domain Implementation (UNIX)

System consists of 2 domains:

- User
- Supervisor
- Domain = user-id
- Domain switch accomplished via file system.
- Each file has associated with it a domain bit (setuid bit).
- When file is executed and setuid = on, then user-id is set to owner of the file being executed. When execution completes user-id is reset.

Domain Implementation (MULTICS)

Let *Di* and *Dj* be any two domain rings.
If $j < i \Rightarrow Di \subseteq Dj$



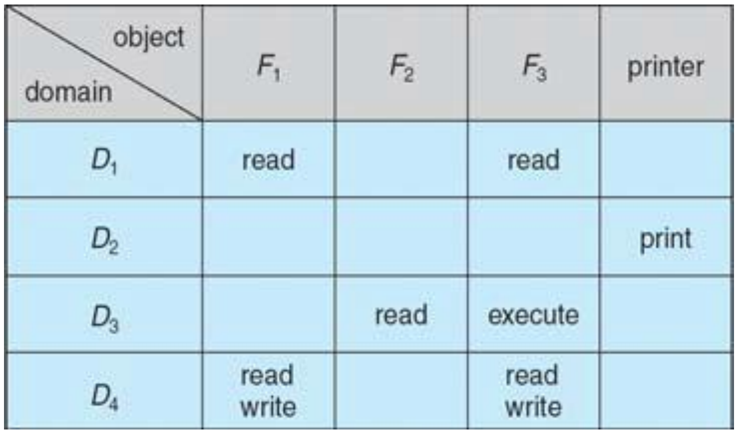
In the MULTICS system, the protection domains are organized hierarchically into a ring structure. Each ring corresponds to a single domain (Figure). The rings are numbered from 0 to 7. Let D; and *Dj* be any two domain rings. If *j* < *i*, then D; is a subset of *Dj*- That is, a process executing in domain *Dj* has more privileges than does a process executing in domain D;. A

process executing in domain Do has the most privileges. If only two rings exist, this scheme is equivalent to the monitor-user n1ode of execution, where monitor mode corresponds to Do and user mode corresponds to D1.
Domain switching in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring. This switch must be done in a controlled manner; otherwise, a process could start executing in ring 0, and no protection would be provided. To allow controlled domain switching, we modify the ring field of the segment descriptor to include the following:
Access bracket. A pair of integers, *bl* and *b2,* such that *bl* ::=: *b2.*
Limit. An integer *b3* such that *b3* > *b2.*
List of gates. Identifies the entry points at which the segments may be called.
If a process executing in ring *i* calls a procedure (or segment) with access bracket *(bl,b2),* then the call is allowed if *bl* <=*i* <= *b2,* and the current ring number of the process remains *i.*

Access Matrix

Model of protection can be viewed abstractly as a matrix, called an access matrix.
The rows of the access matrix represent domains and the columns represent objects.
Each entry in the matrix consists of a set of access rights. Because the column defines objects explicitly, we can omit the object name from the access right.
The entry access(i,j) defines the set of operations that a process executing in domain O_i can invoke on object O_j.
To illustrate these concepts, we consider the access matrix shown in Figure. There are four domains and four objects-three files (F1, F2, F3) and one laser printer. A process executing in domain 0, 1 can read files F1 and *F3* . A process executing in domain 0 , 4 has the same privileges as one executing in domain 0 1; but in addition, it can also write onto files F1 and *F3* . Note that the laser printer can be accessed only by a process executing in domain 0 2.

The users normally decide the contents of the access-matrix entries. When a user creates a new object *Oi,* the column *Oi* is added to the access matrix with the appropriate initialization entries, as dictated by the creator.
View protection as a matrix (*access matrix*)
Rows represent domains
Columns represent objects
Access(i, j) is the set of operations that a process executing in Domain i can invoke on Object j.

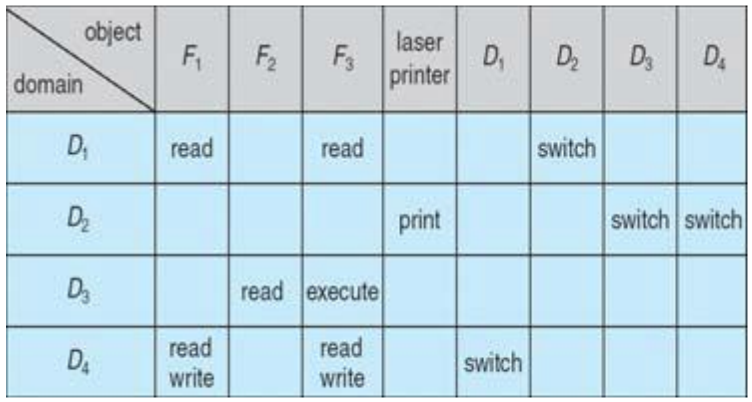


object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Fig A
Processes should be able to switch from one domain to another. Switching from domain Dj to domain *Di* is allowed if and only if the access right switch E access(i,j). Thus, in Figure , process

executing in domain D2 can switch to domain D3 or to domain D4 . *A* process in domain D4 can switch to D1, and one in domain D1 can switch to D2.

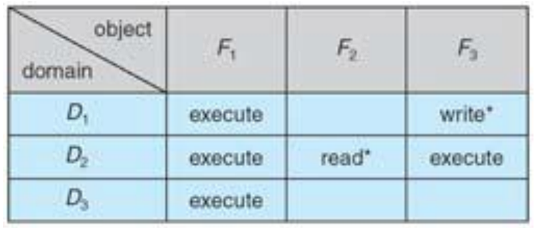
Access Matrix of above Figure With Domains as Objects



object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

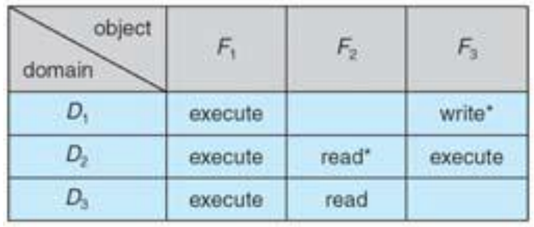
Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.

Access Matrix with *Copy* Rights



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

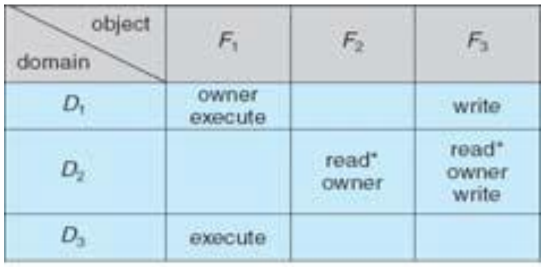
(b)

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right.
The *copy* right allows the access right to be copied only within the column ,for which the right is defined.
For example, in Figure (a), a process executing in domain D2 can copy the read operation into any entry associated with file *F2* . Hence, the access matrix of Figure (a) can be modified to the access matrix shown in Figure (b).
This scheme has two variants:

- A right is copied from access(i, *j)* to access(Jc, *j);* it is then removed from access(i, *j).* This action is a *transfer* of a right, rather than a copy.
- Propagation of the *copy* right may be limited. That is, when the right R* is copied from access(i,j) to access(lc,j), only the right *R* (not R*) is created. A process executing in domain D_k cannot further copy the right *R.*

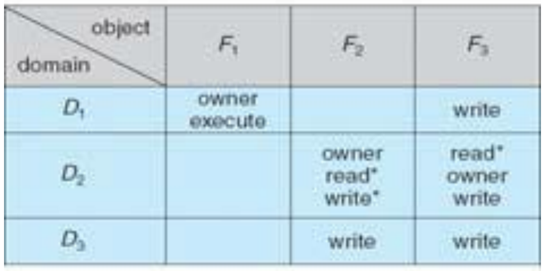
A system may select only one of these three *copy* rights, or it may provide all three by identifying them as separate rights: *copy, transfer,* and *limited copy.*
We also need a mechanism to allow addition of new rights and removal of some rights. The *owner* right controls these operations.
If access(i, *j)* includes the *owner* right, then a process executing in domain *Di* can add and remove any right in any entry in column *j.*
For example, in Figure (a), domain D1 is the owner of F1 and thus can add and delete any valid right in column F1.
Similarly, domain D2 is the owner of *F2* and *F3* and thus can add and remove any valid right within these two columns. Thus, the access matrix of Figure (a) shown below can be modified to the access matrix shown in Figure (b).

Access matrix with owner rights



object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

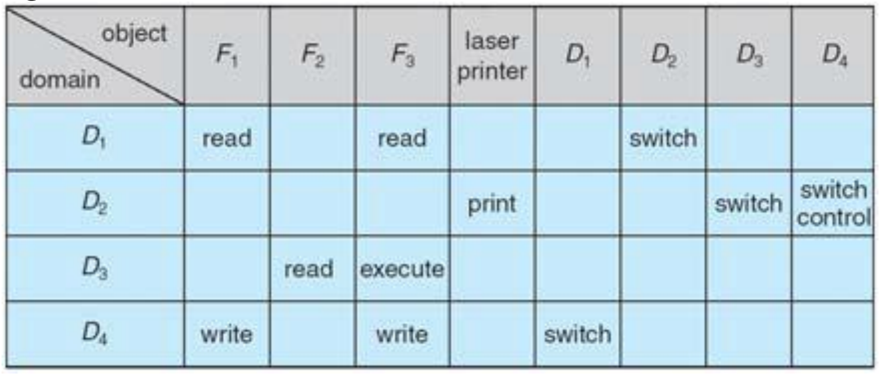
(a)



object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

The *copy* and *owner* rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The *control* right is applicable only to domain objects. If access(i, *j)* includes the *control* right, then a process executing in domain Di can remove any access right from row *j.* For example, suppose that, in Figure A, we include the *control* right in access(D2, D4). Then, a process executil1.g in domain D2 could modify domain D4, as shown in Figure below :



object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Confinement problem:

The *copy* and *owner* rights provide us with a mechanism to limit the propagation of access rights. However, they do not give us the appropriate tools for preventing the propagation (or disclosure) of information. This problem of guaranteeing that no information initially held in an object can migrate outside of its execution environment is called confinement.

Implementation of Access Matrix

Each column = Access-control list for one object
Defines who can perform what operation.
Domain 1 = Read, Write
Domain 2 = Read
Domain 3 = Read
Each Row = Capability List (like a key)
For each domain, what operations allowed on what objects.
Object 1 – Read
Object 4 – Read, Write, Execute
Object 5 – Read, Write, Delete, Copy

1.Global Table :
The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <*domain, object, rights-set*>. Whenever an operation *M* is executed on an object *Oj* within domain Di, the global table is searched for a triple <Di, *0j*, Rk>. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.
Drawback:
The table is usually large and thus cannot be kept in main memory, so additional I/0 is needed.
Virtual memory techniques are often used for managing this table.

2. Access Lists for Objects
Each column in the access matrix can be implemented as an access list for one object.
The empty entries can be discarded. The resulting list for each object consists of ordered pairs <*domain, rights-set*>, which define all domains with a nonempty set of access rights for that object.
When an operation *M* on an object Oi is attempted in domain Di, we search the access list for object 0 i, looking for an entry < Di, *Rk* > . If the entry is found, we allow the operation; if it is not, we check the default set. If *M* is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs.

3. Capability Lists for Domains
Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain.
For a domain capability is a list of objects together with the operations allowed on those objects.
An object is often represented by its physical name or address, called a capability.
The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain.
The capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process.
If all capabilities are secure, the object they protect is also secure against unauthorized access.
Capabilities are usually distinguished from other data in one of two ways:

- Each object has a tag to denote whether it is a capability or accessible data.
  The tags themselves must not be directly accessible by an application program.
  Hardware or firmware support may be used to enforce this restriction.

Although only one bit is necessary to distinguish between capabilities and other objects, more bits are often used.

.

- Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions.
  The other part, containing the capability list, is accessible only by the operating system.

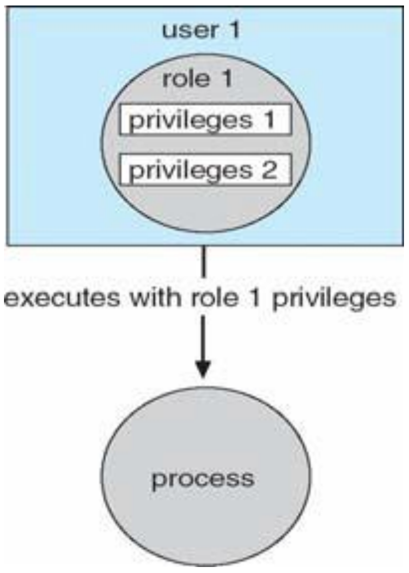
A Lock-Key Mechanism

This is a compromise between access lists and capability lists.
Each object has a list of unique bit patterns called locks.
Similarly, each domain has a list of unique bit patterns, called keys.
A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.
As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain.
Users are not allowed to examine or modify the list of keys (or locks) directly.

Access Control

- Protection can be applied to non-file resources
- Solaris 10 provides role-based access control to implement least privilege
- Privilege is right to execute system call or use an option within a system call
- Can be assigned to processes
- Users assigned roles granting access to privileges and programs
- Limiting processes to exactly the access they need to perform their work.
- Privileges and programs can also be assigned to roles.
- Users are assigned roles or can take roles based on passwords to the roles.

Role-based Access Control in Solaris 10



Revocation of Access Rights

The access list is searched for any access rights to be revoked, and they are deleted from the list. Revocation is immediate and can be general or selective, total or partial, and permanent or temporary.

Capabilities present a much more difficult revocation problem. the capabilities are distributed throughout the system, we must find them before we can revoke them.
Schemes that implement revocation for capabilities include the following:
Reacquisition : Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
Back-pointers : A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary. This scheme was adopted in the MULTICS system. It is quite general, but its implementation is costly.
Indirection : The capabilities point indirectly, not directly, to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it.
When an access is attempted, the capability is found to point to an illegal table entry. Table entries can be reused for other capabilities without difficulty, since both the capability and the table entry contain the unique name of the object.
The object for a capability and its table entry must match. It does not allow selective revocation.
Keys. A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process that owns the capability.
A master key is associated with each object; it can be defined or replaced with the set-key operation. When a capability is created, the current value of the master key is associated with the capability.
When the capability is exercised, its key is compared with the master key.
If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised.
Revocation replaces the master key with a new value via the set-key operation, invalidating all previous capabilities for this object.

Capability-Based Systems

Hydra

- Fixed set of access rights known to and interpreted by the system.
- Interpretation of user-defined rights performed solely by user's program; system provides access protection for use of these rights.

Hydra is a capability-based protection system that provides considerable flexibility. The system implements a fixed set of possible access rights, including such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights.
The interpretation of user-defined rights is performed solely by the user's program, but the system provides access protection for the use of these rights, as well as for the use of system-defined rights.
Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with objects of

the user defined type. When the definition of an object is made known to Hydra, the names of operations on the type become **auxiliary rights.**

Auxiliary rights - described in a capability for an instance of the type.

For a process to perform an operation on a typed object, the capability it holds for that object must contain the name of the operation being invoked among its auxiliary rights.

Rights Amplification - allows a procedure to be certified as **trustworthy** to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of and may exceed, the rights held by the calling process.

Procedure must not be regarded as universally trustworthy and the trustworthiness must not be extended to any other procedures or program segments that might be executed by a process.

Hydra provided a direct solution to the *problem of mutually suspicious subsystems.* This problem is defined as follows :

Suppose that a program is provided that can be invoked as a service by a number of different users (for example, game). When users invoke this service program, they take the risk that the program will malfunction and will either damage the given data or retain some access right to the data to be used (without authority) later.

Cambridge CAP System : Cambridge CAP System

CAP's capability system is simpler and superficially less powerful than that of Hydra.

Can be used to provide secure protection of user-defined objects.

CAP has two kinds of capabilities :

Data capability : It can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object. Data capabilities are interpreted by microcode in the CAP machine.

Software capability : The second kind of capability is the so-called which is protected, but not interpreted, by the CAP microcode. It is interpreted by a *protected* (that is, privileged) procedure, which may be written by an application programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure. When executing the code body of such a procedure, a process temporarily acquires the right to read or write the contents of a software capability itself.

Unlike Hydra, The interpretation of a software capability is left completely to the subsystem, through the protected procedures it contains.

Programmers can define their own protected procedures, but the security of the overall system cannot be compromised. The basic protection system will not allow an unverified, user-defined, protected procedure access to any storage segments (or capabilities) that do not belong to the protection environment in which it resides.