COMP 242 Class Notes

Section 8: Protection

1. Protection

Reading: Chapter 14 and 15, Singhal and Shivratari.

An operating system needs to provide **protection** mechanisms to prevent undesirable operations such as:

(1) A process being able to manipulate hardware translation tables, access arbitrary main memory locations, or perform arbitrary operations on files on disk.

(2) An external agent being able to read or modify the information being communicated between two machines on a network.

(3) An arbitrary user being able to log on to a computer.

These mechanisms can then be used to make a system **secure**. Note that the presence of protection mechanisms are not enough to make a system secure, since these mechanisms may be improperly used or may not be sufficient.

1.1. A Model of Access Control

Access control concerns itself with the first question: how should we control the operations performed by a process on files and other resources supported by the OS? We study below a model of access control that provides such protection. The model we describe is quite general; it may be used protect not only files, but also directories, devices, segments and other shared entities.

Each shared entity will be called an **object**. An object provides certain **operations**, which a process may invoke to manipulate the object. A process can invoke an operation on an object only if it has appropriate **access right** or **privilege** to do so.

We can represent the state of all access rights by constructing an **access matrix** of the following form:

	/dev/console	~fred/prog.c	~fred/letter	/usr/ucb/vi
fred's P	getc, putc, read, write	getc, putc, read, write	getc, putc, read, write	execute
fred's Q	getc, putc, read, write	getc, putc, read, write	getc, putc, read, write	execute
jane's R	getc, putc, read, write	getc, read		execute

In this table, there are four objects: the console, the text file "prog.c" in user fred's directory, the file ‘‘~fred/letter’’, and the object file for an editor; and three processes: P, Q and R. Processes *P* and *Q* are running on behalf of user fred, whereas process *R* is running on behalf of user jane.

Each entry in the table indicates the access rights of a process on an object. Thus all three processes can invoke the execute operation on the editor but cannot read or modify it. Processes P and Q, running on behalf of user fred can both read and modify fred's prog, while process R, running on behalf of user jane, can only read the file.

In the above table, we have assumed that the access rights are defined in terms of the specific operations of an object. Thus the ‘getc’ operation on an object is associated with the ‘getc’ access right. It is often useful to define an object-independent set of access rights associated with a general set of operations to which the specific operations on an object are mapped. For instance the ‘Read’ access right may be associated with the ‘getc’ and ‘read’ operations on a terminal device.

The following is a possible set of general access rights, when the objects are restricted to files, devices, and directories:

Read
Write
Append

Insert
Execute
Delete
Lock
Modify Rights
Set Owner
Create Group
Add Member

The last two rights allow creation of user groups (discussed later) and addition of members to it, respectively. The mapping from specific operations on devices and files to this general set is obvious. The mapping from specific operations on directories to the general set is not obvious, and will be discussed later.

The following is the definition of the access matrix for our example in terms of the general set of access rights:

	/dev/console	~fred/prog.c	~fred/letter	/usr/ucb/vi
fred's P	RW	RW	RW	X
fred's Q	RW	RW	RW	X
jane's R	RW	R		X

In several systems, all processes running on behalf of the same user have the same access rights defined by the access rights of the user. In such systems, the rows of the table describe the access rights of users instead of processes. Thus the access matrix changes to:

	/dev/console	~fred/prog.c	~fred/letter	/usr/ucb/vi
fred	RW	RW	RW	X
jane	RW	R		X

In the rest of the discussion we shall assume this form of an access matrix.

Whenever a process attempts to access an object, the operating system can refer to the access matrix to validate the access. Checking for privileges on each access might be an expensive operation. For files and other objects associated with 'open' and 'close' calls, the system may ask the process to indicate which operations it will invoke when it opens the object. At this point, the system checks if the process has the appropriate access rights. Later, when the process actually makes an access, the system needs to only check if it is one of the accesses indicated by the 'open' call.

It is important to note that the access matrix is purely a logical construct and is seldom stored in the form described (why?). It is often divided into smaller portions. Here are two popular methods of doing so.

1.2. Capability Lists

One way to partition the matrix is by rows. Thus we have all access rights of one user together. These are stored in a data structure called a **capability list**, which lists all the access rights or capabilities that a user has. The following are the capability lists for our example:

Fred --> /dev/console(RW)--> ~fred/prog.c(RW)--> ~fred/letter(RW) --> /usr/ucb/vi(X)
Jane --> /dev/console(RW)--> ~fred/prog.c(R)--> ~fred/letter() --> /usr/ucb/vi(X)

When a process tries to gain access to an object, the operating system can check the appropriate capability list.

This arrangement has several drawbacks:

- If each capability list has an entry for all objects, many entries will indicate that no access is allowed. For instance, most of Fred's personal files may be protected from Jane. This waste of space may be eliminated by only listing those objects for which a user has some access.
- The set of objects accessible by one user, specially privileged users, may be very large. It can be expensive to search capability lists for such users, unless we use capability-based addressing, presented later when we discuss capabilities in more detail.

- An initial capability list must be generated for a new user. It is not clear what the initial list should look like.

1.3. Access Lists

The dual of capability lists is **access lists**, which divide the access matrix by columns. An access list is associated with each object, and lists all users and their privileges over the object. Thus the access list for our example is:

/dev/console --> fred(RW) --> jane(RW)
~fred/prog.c --> fred(RW) --> jane(R)
~fred/letter --> fred(RW)
/usr/ucb/vi --> fred(X) --> jane(X)

This arrangement also has some disadvantages. The set of all possible users is likely to be very large, and many processes will have identical access rights over an object. Therefore it is useful to group users into classes. All members of the same class can be given the same privileges for the object. Unix follows this approach. It partitions users into three classes:

(1) The owner of the file

(2) Users in the same group as the owner

(3) Other users

Thus "~fred/prog.c" can be given the access rights:

self RW
group R
others no access

Multics and AFS allow the access list to contain individual user names as well as larger grouping. Thus if Fred wants Jane to be able to read and write "~fred/prog.c" he can define the access list as follows:

self RW
group R
others no access
Jane R

1.4. Inheritance

Essentially, Multics supports inheritance in the subject dimension, that is, it allows access specifications made for groups of subjects to be inherited by members of the group. One may want to also support inheritance in the object and right dimensions.

In the case of the object dimension, we can allow access rights to be associated with groups of objects, which apply to all members of the group that do not override them. Objects can be based on their type (the IS-A relationship) or their structure (IS-PART-OF). For instance, we can give the Append right to all objects that are of type Bibliography unless a specific instance overrides it. Similarly, we can give the Write right to all objects (files/directories) in a particular directory unless a particular object overrides it.

AFS partially supports such inheritance - in the context of access lists. It associates a directory with both file rights and directory rights. The file rights apply to all files in the directory. Moreover, when a new directory is created, it gets a copy of the (file and directory) rights of its parent. Thus, the file and directory rights specified for a directory apply to all of its descendents.

This approach is not true inheritance, however, since one cannot change the rights of a specific file. (AFS reduces this problem by looking at both the Unix file rights associated with a file and the AFS file rights associated with the containing directory. Access is allowed only if both sets of rights allow it. In the case of Unix rights, only the owner rights are examined. Again this is not true inheritance and is more a hack for backward compatibility with Unix.) Moreover, a directory does not truly inherit the rights of its parent in that if the rights of the parent of a directory are changed, the rights of the children directories are not changed, even if no explicit access specifications have been made for these directories.

What about the right dimension, does it make sense to apply one access definition to multiple rights? Consider the Write right. If a subject can write to an object, he can also insert into it. Thus, it should not be necessary to give insert rights to subjects once write rights have been given. We have used the *imply* relationship here to reduce the number of access definitions that have to be made and stored. Some potential imply relationships are:

Write => Insert => Read => Append
Create Group => Add Member

Note that implication and inheritance are similar in that one access definition applies to multiple cases. However, the latter does not support overriding. It is semantically inconsistent to give the Write right but not the Insert right to a subject.

We can also consider inheritance in the right dimension. We can group the rights into right groups such as Object Rights (Write, Read, Append, Insert, Execute) and Administration Rights (Modify Rights, Set Owner, Create Group, Add Member). We could then give a subject (group) as all rights in a right group to an object (group). For instance, we can give all members of the 242 group all Data Rights to all objects in the directory 242notes. An access definition specifying a more specific right (Write) can then override one specifying a more general right (Data Rights).

We now have the multiple inheritance problem - that is, one may inherit conflicting definitions from different sources. This problems is, in general, hard to resolve in an elegant way and we will not consider it any further.

All kinds of inheritances can be supported by both access lists and capability lists. In particular, object inheritance is a powerful way to reduce the problem of long capability lists. One can store only entries corresponding to groups of objects rather than individual objects. This is analogous to storing entries corresponding to groups of users in access lists.

1.5. Negative Rights

So far, we have assumed that an access matrix contains either positive rights or no rights. We have not assumed the existence of explicit negative rights. Without inheritance, negative rights are not needed: the absence of a right (in the access matrix, capability list, or access list) implies denial of access. With inheritance, however, the rule changes. If a right is given to a general group but not a specific member, the member inherits the right even though no right was explicitly given to it. For instance, if the Write right is given to 242 but not student joe in the class, then we assume that the right is given to joe (and all other students in the class). Absence of negative rights does not create any problem when a whole group is to be given some access right but becomes painful to use when all but a few members of a large group have to be given the right. For instance, what if the Write right is to be given to 242 but not student joe in the class? Without negative rights, I would have to explicitly give all but one member of the group the positive right, and thus not make use of the power of inheritance. Therefore, systems that support inheritance also allow explicit specification of negative rights denying access. Continuing with the example, I could give all but one member of the group 242 the write rights to an object by giving the group 242 the positive Write right and the particular member the negative Write right.

1.6. Directories and Access Control

The semantics of the abstract set of access rights in the context of files and devices is straightforward, but not for directories. The following is a possible definition of these access rights for directories:

- Read: determine the names of the files in the directory
- Write: modify local file names, add and delete files. However, the user is not allowed to open the directory for writing, for which no operation exists. The only way to modify a directory is by service calls.
- Append: add new files.
- Delete: remove this directory.
- Modify Rights: modify access rights to this directory
- Set Owner: set the owner of the directory
- Execute: open files in this directory. This right is used to protect the files in the directory. A file can be opened only if a user has the ‘execute’ access right on all directories in the absolute name of the file. (Should relative names be used instead?)

Not all systems define one set of rights for both directories and files. In particular, AFS defines two sets of rights - file rights (Read, Write, Lock) and directory rights (Lookup, Insert, Delete, Administer). This approach not only decreases uniformity in the system but also does not allow a file to inherit the rights of its directory. Thus, AFS does not allow a file to inherit the read (Lookup) right of its directory. A symptom of this non uniformity is that the Insert right is defined for directories but not files.

1.7. Aliases and Indirect Files

If a file has several aliases, a process might use different absolute names to open it. Thus it may be able to open it under one alias but not another. However, even if a process is able to 'open' a file, it must still have appropriate access over it.

If a process can open a file under more than one alias, is it possible that it may be granted access under one alias but another? The answer depends on whether privileges are associated with with a name or with a file. If they are associated with a name, then they are stored, along with the name, in the directory of the file. Otherwise, they are stored with the file. In the former case, the alias chosen for the file influences its accessibility while in the latter case it does not. Unix chooses the second alternative whereas AFS supports a hybrid of the two alternatives by looking at both rights stored in the file and the rights stored in the directory.

Let us now consider indirect files. How should we interpret access rights on indirect files? We could ignore them completely or we could require that a process have appropriate access rights on both. The latter alternative becomes awkward if indirect files can contain both file and directory names, since sometimes the access rights are interpreted as file rights and sometimes as directory rights. Berkeley Unix 4.2 chooses the former alternative. Any attempt to modify the access rights of an indirect file results in modification of the file to which it points.

1.8. The Access Matrix Again

Consider the final version of the access matrix we defined. It described for each user and object in the system a set of object-independent access rights. All processes started by a user inherited the access rights of the user.

There are several problems with this version of the access matrix:

(1) All processes started by a process have all the access rights of the user. This is undesirable since a process, typically, needs access to a small set of objects to do its job. For instance a process executing a calendar program needs to access, among the different files in the system, only the 'calendar file' that defines a set of appointments, a process executing the compiler needs to access the input source files, the object files, and a few temporary files, and a process that prints the contents of a file needs access only to that file.

(2) The access rights of a process are fixed while it is executing. This again is undesirable, since the set of objects a process needs to access, typically, changes dynamically. For instance, a compiler after reading a source file, no longer needs access to that file.

(3) A process cannot **amplify** its access rights to include those that are not available to the user on whose behalf it is executing. Such amplification is necessary to do certain kinds of tasks. For instance a mail program invoked in Unix needs to be able to create a mail file in the directory /usr/spool/mail, which is writable only by root.

(4) The access rights are object-independent and thus not of a fine enough granularity. For instance Unix supports the access rights: read, write, and execute. Often it is important to distinguish between different kinds of reads. As an example consider a bibliography created by some user who has also put, for each item, an annotation describing his view of the referenced paper. He may want to allow his colleagues to read the reference items, but not the annotations. (Some of bibliography items may reference papers written by his colleagues!) Unix, however, does not distinguish between these two kinds of object-defined 'reads'. Thus the author of the database has to either give no read access to the file or full read access.

1.9. Modes

Most architectures allow a process to execute in at least two modes: **kernel** and **user**. In the kernel mode a process can set up translation tables, change the processor state that determines the mode of the processor, use the kernel translation table and execute other privileged instructions, while in the user mode it uses the process' translation

table and instructions. Thus these architectures allow a process to switch between the access rights of the user and the kernel.

1.10. Multics Rings

Multics supported an extension of the privileged mode idea via an access control mechanism based on a *ring* structure. In that system, the hardware supported not two modes (kernel and user) per process, but up to 64. The access rights of a process changed when it executed not only kernel procedures, but also ordinary procedures stored in users' segments.

Each mode was called a **ring**, and was associated with a set of access rights. The rings in the system were hierarchically arranged: the innermost ring had the most power and the outermost ring the least. In general the access rights associated with ring *i* were a superset of the access rights of ring *i+1*.

In Multics, each object is accessed via a segment, and each segment was associated with one of the rings. A segment description contained an entry that defined the ring number. In addition, it contained three access bits to control reading, writing, and execution. With each process a *current-ring-number* counter was associated, which determined the current access rights of the process. A process executing in ring *i* could read and write a segment in ring *j*, *j* >= *i*, but could not read or write a segment in ring *j*, *j* < *i*. It could, however, execute a segment in any ring, when its *current-ring-number* was changed to that of the executed segment.

Here is a use of the rings facility, assuming there are four rings. Ring 0, the most powerful mode, would be devoted to the kernel. Ring 1, could contain the mail program, which needs to be able to create files in arbitrary directories. Ring 2, could contain a grading program used to evaluate student programs, and ring 3 could contain the student programs.

This scheme, as stated above, is unsatisfactory for two reasons: First, a process in an outer ring should not be able to execute any instruction in an inner ring. In particular a user process should not be able to execute any instruction in the kernel, which is kept in ring 0. Instead, it should be able to execute an inner segment at certain special entry points. Second, every segment in an inner ring should not be executable by all the outermost rings. For instance student programs, in our example, should not be able to call the grading program.

To overcome these drawbacks, Multics supports a modification of the scheme outlined above. It stores in a segment descriptor, instead of a single ring number, the the following fields:

(1) **Access bracket**: A pair of integers, *b1* and *b2*, such that *b1* <= *b2*.

(2) **Limit**: An integer *b3*, such that *b3* >= *b2*.

(3) **List of gates**: The entry points (gates) at which the segments may be called.

If a process executing in ring *i* tried to execute a segment with access bracket (*b1*, *b2*), then the call was allowed if *b1* <= *i* <= *b2*, and the current ring number of the process remained *i*. Otherwise, a trap to the kernel occurred and the situation was handled as follows:

(1) If *i* <= *b1*, then the call was allowed to occur and the *current-ring-no* of the process was changed to *b1*. Thus the access rights of the process are reduced. If parameters are passed that refer to segments in a ring lower than *b1*, then these segments were copied into an area accessible in ring *b1*.

(2) If *i* > *b2*, then the call was allowed to occur only if *i* <= *b3*, and the call had been directed to one of the designated entry points in the list-of-gates. If the call was successful, the *current-ring-no* of the process was changed to *b2*. This scheme allowed processes with limited access rights to call procedures in lower rings with more access rights, but only in a carefully controlled manner.

Regarding R/W, these rules imply that a process with ring *i* that tries to R/W a segment with access bracket (*b1*, *b2*), is allowed to do so if *i* <= *b2*.

Now, in our example, the kernel would be associated with the access bracket (0, 0), limit 3, and the list of gates containing entry points of the system calls and the grading program could be associated with the access bracket (1, 1), and the limit 1.

The Multics scheme still had the disadvantage that ring (hierarchical) structure did not allow enforcement of the need-to-know principle. If an object was to be accessible in ring *i*, but not in *j*, then it was necessary that *i* < *j*. But this meant that every object accessible in ring *j* was accessible in ring *i*.

1.11. Unix SETUID

Unix provides a simplified version of the Multics access control mechanism to change the access rights of a process dynamically. A process can switch between user and kernel mode. Moreover, in user mode, a process executing with the access rights of one user, when it executes a file owned by another user gets the access rights of the second user if a bit (called SETUID) in the object file is on. This feature supports rights amplification and easily handles the mail example. The mail program is owned by root and has its SETUID bit on. When the command interpreter subprocess forked to process the 'mail command' executes the mail program it acquires the access rights of root and can create/modify a file (owned by the receiver) in the directory '/usr/spool/mail'.

1.12. Bell LaPadula Model

This discussion is taken from HongHai Shen's thesis.

The Bell-LaPadula Model (BLM), also called the multi-level model, was proposed by Bell and LaPadula for enforcing access control in government and military applications. In such applications, subjects and objects are often partitioned into different security levels. A subject can only access objects at certain levels determined by his security level. For instance, the following are two typical access specifications: ''Unclassified personnel cannot read data at confidential levels'' and ''Top-Secret data cannot be written into the files at unclassified levels''. This kind of access control is also called *mandatory access control*, which, according to the United States Department of Defense Trusted Computer System Evaluation Criteria is ''a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (e.g., clearance) of subjects to access information of such sensitivity". The converse of mandatory access control is *discretionary access control*, which is defined as ''a means of restricting access to objects based on the identity of subject and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) to any other subject".

The Bell-LaPadula model supports mandatory access control by determining the access rights from the security levels associated with subjects and objects. It also supports discretionary access control by checking access rights from an access matrix.

Each object is associated with a security level of the form (classification level, set of categories). Each subject is also associated with a maximum and current security level, which can be changed dynamically. The set of classification levels is ordered by a < relationship. For instance, it can be the set `top-secret, secret, confidential, unclassified,` where

```
unclassified < confidential < secret < top-secret
```

A category is a set of names such as `Nuclear` and `NATO.` Security level A *dominates* B if and only if A's classification level is greater than or equal to B's classification level, and A's category set is a superset of B's. For instance,

```
top-secret, {Nuclear, NATO}
```

dominates

```
secret, {NATO}
```

because

```
top-secret > secret
```

and the set

```
{Nuclear, NATO}
```

contains

```
{NATO}
```

In the model, an access request (subj, obj, acc) is granted if and only if all of the following properties are satisfied:

(1) simple security property (no read up): if acc is read, then level(subj) should dominate level(obj).

(2) *-property - called the star property (no write down): if acc = append, then level(obj) should dominate level(subj); if acc = write, then level(obj) should be equal to level(subj).

(3) discretionary security property: the (subj, obj) cell in the matrix contains acc.

Like Multics, this model has the problems of hierarchical access control and does not always support the need to know principle except in rigid military situations.

1.13. Capability-Based Systems

The systems we have studied so far support access lists. We now study in detail systems that support capability lists (e.g: Hydra, CAP, iMAX for INTEL 432, IBM System 38, Amoeba). These systems, often called **capability-based systems** since the central concept they support is the notion of a **capability**, provide a very flexible protection mechanism.

Each process is associated with a list of capabilities called a **C-list**. Each capability in that list indicates the **type** of the object (e.g: file, device, directory), a **rights** field, which is a bit map indicating which of the legal operations on this type of object are applicable, and an **object** field which is a pointer to the object itself (in Unix, the equivalent of an inode number). Thus the following is an example of a C-list:

Type	**Rights**	**Object**
File	R--	Pointer to File3
File	RWX	Pointer to File4
File	RW	Pointer to File5
Directory	--X	Pointer to Directory3

How does the system check that a process can perform an operation on an object? Here are some alternatives:

First, the process, when it performs an operation on an object, indicates both the object and the operation, and the system checks the C-list to see if the operation can take place. The disadvantage of this scheme, as we have seen earlier, is that on each operation on an object the kernel has to search through a possibly long list of capabilities.

Second, the list of capabilities of a process are stored in the process' address space, and thus can be named by the process. Now when a process asks the system to perform an operation on an object, it also presents the capability to the kernel, which can then check if the operation is valid. It does not have to search the C-list for the capability; since the process presents the appropriate capability for checking. A problem with this approach is that, since capabilities are stored in the process' address space, it can manufacture capabilities and thus perform arbitrary operations on arbitrary objects.

A better approach, used in Hydra, is to use a combination of these two approaches. The C-list of the process is stored in the kernel address space. A process refers to a capability via the index of the capability in the C-list. (Just as a Unix process names a file via an index in the file table). Now, instead of capabilities, descriptors for them are stored in the process' address space. When a process needs to perform an operation on an object, it names the capability for it via the descriptor for the capability. Thus the system does not have to search the C-list. Moreover, a process cannot manufacture capabilities.

Another approach is to keep the C-list in the process' space, but encrypt each capability with a key unknown to the process.

Yet another approach is to define a a hardware that distinguishes between memory words that store capabilities and those that do not. The hardware can then enforce that capabilities are manipulated only in the kernel mode. Two methods are used to distinguish between words that contain capabilities and those that do not. In a **tagged memory** these two kinds of words are distinguished by a **tag** bit, while in a **partitioned memory**, they are distinguished by the segments in which they reside (thus the segment descriptor contains a bit that indicates whether it contains capabilities). The use of either tagged or partitioned memory architectures forces an inconvenient tradeoff. If the addressable unit is large, one can afford a tag bit in each unit. If the byte or the bit is the unit of addressing, it is out of the question to associate a tag bit with each unit. Partitioned memory avoids this kind of storage overhead. However the need to separate capabilities and non-capabilities requires the use of many more small segments than in non-capability systems. As a result, many more units have to be transferred between memory and disk, leading to I/O congestion. The CAP system uses partitioned memory whereas intel 432, IBM system 38 and other capability-based architectures used tagged memory.

All approaches other than the first one use the notion of *capability-based addressing*, that is, objects are addressed through capabilities (rather than pointers or file descriptors, for instance.)

User-Defined Types

In the other systems we have seen so far, the type of the protected object was defined by the kernel. As a result an 'object-independent' set of access rights could be defined by the kernel, which supported the predefined operations applicable on these predefined types. Capability-based systems go a step further and allow definition of user-defined types, which have user-defined operations applicable on them. Objects of these types were associated with a set of type-independent rights called **kernel rights**, which are defined by the kernel for all objects (e.g: read, write) and a set of type-dependent rights called **auxiliary** rights, which control which of the user-defined operations can be invoked on the object. Thus a capability has fields for both these kinds of rights.

Here is an example to clarify the idea of user-defined types: Assume that a user would like to provide a new type that defines a bibliography database in which entries can be examined, modified, deleted, and inserted. Then he could define the database as a user-defined type:

```
protected_type Bibliography {
   contents: file;
   procedure Examine (obj: capability) { ... };
   procedure Insert (obj: capability) { ... };
   procedure Delete (obj: capability) { ... };
   procedure Modify (obj: capability) { ... };
   };
```

The declaration specifies the name of the type to be protected, the structure of the internal representation of data part of this type, and the operations associated with this type. The operating system uses this declaration to register a new type in its database of user-defined types.

Now a process can ask the operating system to create a new object of this type. In response, the operating system allocates space for the field 'contents', which is the internal representation of the object, creates a new capability

Type	**Rights**	**Object**
Bibliography	---|EIDM	Pointer to the Internal Repn

and returns the newly created capability to the caller. The process may then use this capability to perform any of the four operations on this object. Note that the capability does not contain the read (R) or (W) rights. As a result, a user of this capability cannot examine or modify directly the internal representation of the data of a protected object, and has to do so indirectly by invoking the type-specific operations.

How is a type named? A type is itself considered a protected object, of the predefined type TYPE, and is named by a capability for it. It is associated with the operation **create**, which may be invoked to create an instance of the type. This capability is named in the modified type declaration:

capToBibliography: **protected_type** Bibliography {...}

is of the form

Type	**Rights**	**Object**
TYPE	---Create	Pointer to Defn of Bibliography

and can be used to create a new instance of the type:

newBib := create (capToBibliography, arg1, arg2, ...)

If all types are objects, and each object has a type, then what is the type of TYPE? The type TYPETYPE? In that case what is the type of TYPETYPE? TYPETYPETYPE?... We can get rid of this infinite sequence by making TYPE a distinguished object that is an instance of itself. Like other instances of TYPE, TYPE is also a type, and is associated with the operation *create* which creates instances of TYPE. This operation is invoked implicitly by the declaration:

capToBibliography: **protected_type** Bibliography {...}

which gets translated into a call of the form

capToBibliography := create (capToTYPE, a1, a2, ...)

where the arguments specify the name of the type, the internal representation, and the operations.

Transferring Capabilities

Suppose a process has created a new instance of the type 'Bibliography':

capToBib := create (capToBibliography, ..)

and would like to share this object with other processes. How does it do so? The answer depends on the way capabilities are represented:

It can simply pass the value of the variable *capToBib* to the other process via IPC. This approach would work in a system in which *capToBib* is an encrypted word, if the key used to encrypt capabilities is common to all processes.

It can make a special kernel call:

send_capability (receiver, capToBib)

which sends the capability to to process *pid*, which receives it via the call:

receive_capability (copyOfCap)

In tagged or partioned memory systems, the kernel would use hardware instructions available in kernel mode to create the new capability. If C-Lists are stored in kernel space, then the kernel would create a new index in the process table of 'receiver'.

Selective Transfer of Access

It is often useful for a process to give to another process a subset of the access rights it has to an object. For instance a process *p* that creates a new instance of 'Bibliography' via the call:

capToBibliography: **protected_type** Bibliography {...}

has the right to examine, insert, modify, and delete bibliography items. It may want to give other processes the right to only examine the items.

How is selective transfer of access achieved? One approach is to provide a kernel call

Copy (origCap, newRights) : capability

that returns a copy of *origCap* that includes only those rights specified by *newRights*. Another approach is to make each procedure an object that can be executed, and require that a a process performing operation *p* on object *O* provide capabilities to both *O* and *p*. Thus a process, to invoke the *Examine* operation on instance *O* of 'Bibliography', would execute the following statement:

execute (capToExamine, Examine (capToO))

Hydra uses a combination of these two approaches. A process can execute operation *p* on object *O* only if it has a capability *c1* that allows execution of *p and* a capability *c2* that allows operation *p* on *O*.

Changing Domains of Protection

We saw earlier the need to change the set of access rights available to a process, often called its **domain**, as the set of objects it needs to access changes. Capability-based systems are particularly well-suited to support this feature. The example of Hydra shows how this may be done.

In Hydra, the domain of a process changes on each procedure call. Capabilities that are to be shared between the caller's domain and the callee's domain are passed as parameters. In addition the callee may have some static capabilities that are available on each invocation. As an example, consider the definition of the following procedure *q*:

```
procedure q (a1: capability) {
  var c3: capability = create (...))
  ....
```

Now assume that some process *A* with capabilities *c1* and *c2* makes the call:

q (c1)

Then after the call, the domain of protection of the process includes the capabilities *c1* and *c3*.

This feature can be used, for example, by a compiler to ensure that only the *ReadSource* procedure gets the capability to read a source file.

Amplification

Assume that a process calls procedure:

Examine (capToObject: capability)

passing it a capability to the object on which the operation is to be invoked. The procedure *Examine*, defined with the type declaration of 'Bibliography', needs to read the internal representation of the object to perform its duty. Thus it needs the kernel 'R' right on the object to do so. However, *capToObject* does not have this right, since the internal representation of an object is not visible from outside. Therefore, the procedure cannot perform the requested operation!

A solution, adopted in Hydra, is to associate each capability parameter of a procedure with a **capability template** of the following form:

Type	**Check-Rights**	**Rights**
Bibliography	---|E	R--|

When the procedure is called, the system checks if a capability argument has the same type as the type of the associated template, and has the rights specified in the *check-rights* field of the template. If both conditions are met, then the call is successful, and the system assigns to the corresponding capability parameter a new capability that is the same as the capability argument, except that rights are those specified in the template and not the argument. Thus if the capability argument corresponding to the above template was:

Type	**Rights**	**Object**
Bibliography	---E	Pointer to Object

then the corresponding parameter is assigned the capability:

Type	**Rights**	**Object**
Bibliography	R--	Pointer to Object

Thus the callee may have greater freedom to operate on an object than the caller. In particular, as illustrated by this example, the callee may be able to access the internal representation of an object while the caller cannot.

1.13.1. Physical Analogy

For capability-based protection mechanisms a useful physical analogy is that each object is contained within a house having several doors that open into different rooms, and capabilities are keys to these doors. The object field in a capability is analogous to a pattern of notches on the key, and the access rights are a set of auxiliary 'bumps' that permit access to particular doors of the house. When a house (object) is constructed, the system gives the creating person (process) a key (capability) that will open all doors in the new house. What transpires after this is up to the person using the initial key.

To make this world of locks and keys a secure and useful one, one arrives at the following considerations:

(1) It must be impossible for a person to fabricate a key. Also, given a key, it must be impossible to alter the notches to open a different house or a different door.

(2) Given a key, it should be possible to make a copy of the key, either for oneself or for someone else.

(3) Given a key, it should be possible to remove (but not add) one or more of the auxiliary bumps to remove some of the key's access rights.

(4) For generality, one should be able to move and store keys in the same way as any other entity.

This analogy introduces some problems that need to be addressed by capability-based systems:

The 'do not copy' problem

In the locked-house world, person *A* might wish to give person *B* a key to a door in a house, but might want to preclude *B* from copying the key, for instance, to give it to a third party. Hence, one needs a mechanism to stamp a 'do not copy' on a key.

This can be achieved in capability-based system by associating each capability with **capability rights** in addition to the other rights. One of these rights determines if the capability can be copied.

The retraction problem

It is often important to be able to withdraw authority after it has been given. For instance, one might have given 10 people keys to a door, and then later decide that person *D* should no longer have have a key.

One solution is *indirection*: Rather than handing out keys to the house itself, one might hand out keys to a second house that contains a key to the first house. Hence one can withdraw the authority of a particular person or a class people by destroying one of the secondary 'key-holding' houses.

Thus capability-based systems often support *indirect capabilities* (which are like indirect files) that, from the holders point of view, can be used as direct capabilities. However, the creator of such a capability can invalidate the capability by destroying the intermediate object.

The 'what is this key' problem

The closest physical analogy to this problem is having no information about the properties of a key on a key chain. It represents a set of situation where the operating system has useful information that is not available to programs. Therefore some systems provide a *describe-capability* operation that given a capability, returns its type and access rights but not the internal representation of the object.

1.14. The Confinement Problem

Consider a client using the services of some server that it does not trust. It needs to, however, disclose some information to the server to get the job done. For instance, the server may help a client with tax computations, given some financial data from the client. The client is worried that the server may record this information and sell it to an interested party.

One step the system can take to protect the client is to ensure that there is no possible way for the server to record information. This may be achieved by ensuring that the server does not write information to any external file. However, the server may communicate the information to another process, called the collaborator. Now the object of the system is to make sure that there is no way for the server to leak information to the collaborator. Butler Lampson calls this the **confinement problem**.

The system may be able to ensure that the server cannot pass information to the collaborator by writing to shared memory, a shared file, or using IPC facilities. However, more subtle communication channels may exist between the server and the client. For instance, the server can try to communicate a binary bit stream as follows. To send a 1, it does computation for a fixed interval of time. To send a 0, it goes to sleep for the same interval of time. The collaborator can try to detect the bit stream by carefully monitoring the system load. The load will be lower when a 0 is being sent and higher when a 1 is being sent. The **covert communication channel** thus established is a noisy one, but enough redundant information can be sent to extract the information.

Modulating the CPU usage is not the only covert channel. The paging rate can also be modulated (many page faults for a 1, no page faults for a 0). Acquiring and releasing dedicated resources (tape drives, plotters, etc) can be used for signalling. (Acquiring a resource can mean a 1, and releasing a resource a 0). In Unix, the server could create a file to indicate a 1, and remove it to signal a 0. The collaborator can then use the *open* call to see if the file exists. This call can be used even if the collaborator has no permission to use the file.

Lampson also mentions a way for the server to leak the information to its human owner. Assume that the client needs to pay for the services of the server. Then the server process will need to send its owner a copy of the bill so that he knows how much to expect. The information can be encoded in the bill. For instance if the actual computing bill is 100 dollars and the client's income is 53K, then the server could report the bill as 100.53.

Just finding all the covert channels, let alone blocking them, is extremely difficult.

1.15. Cryptography

Consider two computers *A* and *B* communicating messages via a network that is also accessible to an intruder computer *I*. The following are some of the security violations that can occur:

(1) *Passive Wiretapping*: *I* reads a message.

(2) *Modification*: *I* modifies a message.

(3) *Site Impersonation*: *I* sends messages to *A* pretending it is *B*.

Cryptography is the only known solution to prevent security violations of this kind. It also has other applications, as we shall study later.

The basic concepts in cryptography are **encryption** and **decryption**. Encryption refers to the transformation of intelligible information into an unintelligible form for the express purpose of rendering it useless to an intruder. This transformation process is represented by a mathematical function (enciphering algorithm)

$$C = E(P, K_E)$$

where *P* is **plaintext** (or **cleartext**) to be encrypted, K_E is an **encryption key**, and *C* is the resulting **cyphertext**. Decryption is represented by a matching decryption function (decryption algorithm)

$$P = D(C, K_D)$$

where K_D is the **decryption key**.

In a cryptosystem a sender uses K_E to encrypts messages, which the receiver decrypts using K_D. An intruder's role in such a system is to guess K_D given the cyphertext, the encryption and decryption algorithms, and possibly some side information.

The strength of a cryptosystem is measured by its resistance to attack, that is, how difficult it is to determine K_D. Based on increasing amounts and types of available side information, attacks can be classified, in order of increasing intensity, as follows:

(1) *Cyphertext only*: The intruder has intercepted the cyphertext material and has general knowledge of an opponent's messages and statistical properties of the language (e.g., frequency of letters or words).

(2) *Known plaintext*: The intruder has, in addition, substantial amounts of plaintext and corresponding cyphertext.

(3) *Chosen plaintext*: The intruder can see, in addition, cyphertext for any plaintext.

The requirement of most present-day commercial and government agencies is that the cryptosystem be able to withstand a chosen plaintext attack.

Unfortunately, all current encryption methods have problems. The only known provably secure method, called the 'one-time pad', involves the use of a *random key* that has the same length as the plaintext to be encrypted. The key is exclusive-or'd with the message to produce the cyphertext. Given the key and the cyphertext, the receiver uses the same method to reproduce the plaintext. After that the same key is never used again. The intruder is then faced with the possibility of inspecting 2^n messages, a great number of which will be valid.

This method also has its problems. Since both the originator and the recipient of the cyphertext must agree on the key, it must be transmitted as well, a procedure that is just as difficult as the original problem of secure transmission. One solution to this problem is to use computer-generated pseudo random numbers. These numbers, however, are too easy to guess.

A currently popular encryption method, developed by the National Bureau of Standards, is called DES, for 'Data Encryption Standard'. The algorithm (encryption or decryption) can be performed efficiently with a special-purpose chip, but far less efficiently with a program. The key is 56 bit (or bytes) long; critics argue that this length is too short and an exhaustive search will soon be possible using commercially available general-purpose computers.

The one-time pad and DES are considered **conventional**; they share the property that the same key is used for encryption and decryption. A non-conventional approach called **public-key** cryptography uses different keys for encryption and decryption, and is used as follows: Every user *U* has a pair of keys, one for encryption U_E and one for decryption U_D. It is impossible to guess U_E from U_D or vice versa. Cyphertext encrypted with U_E is decrypted with U_D. Everyone's *E* key is made public; it is saved in a public place where anyone may access it. Let's say *A* and *B* are two parties that want to send secure messages to each other. *A* encrypts messages for *B* using *B*'s public key B_E. *B*

can decrypt the messages using his private key B_D, but no one else can do so since no one else knows B_D. Similarly, *B* encrypts messages for *A* using *A*'s public key A_E. These messages are secret. However, *B* cannot be sure that *A* has sent the messages because anyone could have used B_E, *B*'s public key. Thus messages are not authenticated.

For authentication we need to assume that a user's decryption key can be used to encrypt cleartext that can be decrypted by his encryption key. Thus both of the following need to be true:

$D(E(P, U_E), U_D) = P$
$E(D(P, U_D), U_E) = P$

Now let *A* encrypt its messages to *B* using *A*'s secret key A_D. *B* can decrypt the messages by using *A*'s public key A_E. Now we have authentication: *B* is sure that *A* sent the message because only *A* knows A_D. However, we do not have secrecy: anyone else who gets this message can also apply *A*'s public key A_E and decrypt it!

These two ideas can be combined to create messages that are both secret and authenticated: *A* sends the following to B:

$C = E(D(P, A_D), B_E)$

which only *B* can read by computing:

$E(D(C, B_D), A_E)$

B is now sure that *A* has sent the message, since only *A* knows A_D. Thus we have both secrecy and authentication. (Would we have both if *A* sent instead the message $D(E(P, B_E), A_D)$?)

Encryption and decryption algorithms that can be used in public-key systems are too complex to be discussed here. Typically, a public-key system is based on a mathematical problem with the following two properties: 1) it is hard to find a solution to the problem, and 2) it is easy to recognize a solution to the problem. The encryption and decryption algorithms are such that the difficulty of finding a key is equal to solving the problem and the difficulty of encryption and decryption is equivalent to recognizing a solution to the problem. An example of such a problem is factoring the product of two very large prime numbers. A popular method called **RSA** (for Rivest-Shamir-Adleman) is based on this problem. One important class of such problems is the set of NP complete problems, which have the interesting property that solutions to them are recognizable in polynomial time whereas the best known methods for finding these solutions take exponential time, and a polynomial time solution to *one* of these problems would lead to polynomial time solutions to *all* of them (including, as it turns out the factoring problem even though this problem has not been shown to be NP-complete). An example of an NP complete problem is the knapsack problem: Given a set of items $a_1...a_n$, with sizes $s_1...s_n$ with values $v_1...v_n$, is there a set of these items such sum of the sizes is <= B (the capacity of the knapsack) and the sum of the values >= K (the total value required).

1.16. User-Authentication

The problem of identifying users when they log in is called **user authentication**. Most authentication methods are based on what the user knows (e.g: password), what the user has (a plastic card), or physical characteristics of the user (e.g: finger prints).

1.16.1. Passwords

The most widely used form of authentication is to require the user to type a password. Unix uses this approach. There are several ways to support passwords. We shall study some of these by studying the different schemes that have been used in Unix. We first study how Unix establishes contact with a user.

The system starts up executing the *init* process, whose process identifier is 1. On each terminal port available for interactive use, *init* forks a copy of itself, which attempts to open the port for reading and writing. (This new process has its own process identifier) The open succeeds when a directly connected terminal is turned on or a telephone call is accepted by a dial-up modem. Then the *init* process execs a program called *getty*.

Getty initializes terminal line parameters and prompts the user to type a *login name*, which *getty* collects. It then executes a program called *login*, passing the login name as an argument. The login process prompts a user for a password, and checks the input against an entry in the password file (/etc/passwd). If the password is valid, *login* sets the user identifier (*uid*) of the process to that of the user logging in and executes the shell specified in '/etc/passwd'.

We now study the different schemes used in Unix to check passwords.

The First Approach

Unix was first implemented with a password file that contained the actual passwords of all the users, and for that reason this file had to be heavily protected agains being read or written. This approach caused several problems:

(1) There was no way to prevent the making of copies by privileged users.

(2) Errors could cause the contents of the file to be accessed. In one case, one system administrator was editing the password file while another was simultaneously editing the daily message that is printed on everyone's terminal. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was displayed to every user when he logged on.

(3) The contents of the file saved on magnetic tape were available to anyone with physical access to these tapes.

(4) The contents of the file, which also contained other informations such as user name, had to be duplicated on other files so that non-privileged processes could access them. These files had to be updated whenever a user was added to or dropped from the system.

The Second Approach

An obvious solution was to encrypt each user's password P with some key K and store the encrypted version:

E (P, K)

in the password file. Now when the user tries to logs on to the system, the password he types is encrypted and compared with the encrypted version in the password file. If the two match, the login attempt succeeds. If the function E is hard to invert without a key, the password file could be read by everyone. If the function E is hard to invert even with a key, then the password and login programs could also be read by everyone. (Encryption functions that are hard to invert are called **trap-door** encryption algorithms)

A convenient and rather good encryption program available at that time was the M-209 program, which simulated the M-209 cipher machine used by the US Army during World War II. Unfortunately, the cyphertext produced by it was easily invertible, given the key. However, given the plaintext and the cyphertext, it was very hard to guess the key. Thus the new version of Unix used the password, not as the plaintext, but as the key, and a constant was encrypted using this key. The encrypted result was entered in the password file.

Attacks on the Second Approach

One approach to penetrating this scheme is to keep guessing the key until one succeeds. This method can work well since most people will use short passwords composed entirely of ASCII characters. In a collection of 3,289 passwords gathered from many users over a long period of time:

15 were a single ASCII character,
72 were strings of two ASCII characters,
464 were strings of three ASCII characters,
477 were strings of four alphanumerics,
706 were five letters, all upper-case or lower-case,
605 were six letters, all lower case.

On a PDP-11/70 the the amount of time needed to encrypt a potential password was 1.25 milliseconds. The time required to try all combinations of 5 lower case letters ((26^5) was 4 hours, of 6 lower case letters was 107 hours, of 4 ASCII characters was 93 hours, of 5 ASCII characters was 500 days, of 6 ASCII characters was 174 years. The search could be improved by trying first:

The app. 250,000 words in a dictionary spelled forwards and backwards

A list of first names, last names, street names, and city names (best obtained from some mailing list).

All valid license plates in the state.

Room numbers, social security numbers, telephone numbers etc.

Morris and Thompson, authors of the Unix password scheme, compiled a list of likely passwords using the above heuristics, encrypted each of these using the known password encryption algorithm, and checked to see if any of the encrypted password matched entries in their list. Over 86% of all passwords turned up in their list!

1.16.2. Improvements

Slower Encryption

Obviously the M-209 program was far too fast. Unix switched to the DES encryption algorithm, which is slow when implemented in software. The DES was implemented in the following way: The first eight characters of user's password are used a key to encrypt a constant (0). Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

Less Predictable Passwords

The password program was changed to urge the user to use harder passwords. If the user enters a alphabetic password shorter than 6 characters, or a password from a larger character set shorter than 5 characters, then he is asked to enter a longer password.

Salted Passwords

Consider an intruder trying to gain access to as many users of as many Unix systems as possible. For each password he tries, he can check the entries of all the users in the systems. Moreover, he can compile a list of likely passwords, encrypt them, and save the results in some sorted file, so that any encrypted password can be searched easily. As new passwords are added, he can check them against his compiled list at no encryption cost.

Unix uses a technique of **salted passwords** that renders attacks of the above kind useless. When a password is first entered, the password program obtains a 12 bit random number (by reading the real-time clock) and appends this to the password entered by the user. The concatenated string is used as the key for encryption, and both the 12 bit random number and the result of encryption are stored in the password file. When the user later logs on to the system, the 12 bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required as before to be the same as the 64-bit result of encryption stored in the password file.

Now an intruder cannot amortize the cost of one encryption over all the password entries to be searched. Moreover, a complied file of encrypted passwords has to contain 2^{12} entries for each guessed password. Thus if the intruder considers 'Purdue' a good guess, he would need to encrypts all of the strings 'Purdue0000'...'Purdue111'. Thus this method offers protection agains intruders who try to precompute a large number of encrypted password. However, it does not protect an individual user whose password is 'Purdue'. The intruder can read the random number from the password entry for the user, append the random number to the password 'Purdue', encrypt the result, and check it against the 64-bit cyphertext stored in the password file. He does not have to try all strings 'Purdue0000'...'Purdue111'.

One of the side effects of this modification is that it becomes impossible to find out whether a person with passwords on several machines has used the same password on all of them.

The Threat of the DES Chip

As mentioned earlier, chips are available to do the DES encryption fast (3 times as fast as software). To prevent the use of such chips, one of the internal tables of the DES algorithm is changed in a way that depends on the random number. This table is hardwired into the commercially available chip. Obviously, the intruder could design and build his own chip that takes the random number as input, but the cost would be very high.

User Names

Consider an intruder that is trying to guess both user names and passwords. He should not be able to tell, after an unsuccessful attempt, which of his guesses was bad: the user name or the password. Therefore Unix does the encryption of the password even if an invalid user name is typed.

1.16.3. Other Methods

The password method verifies a user's identity by checking on a piece of information only that user is supposed to know. A generalization of this idea is to have each user provide a long list of questions and answers, which are stored in the computer in an encrypted form. These questions should be chosen so that the user does not have to write their answers down. Examples (for Monty Python fans):

What is your quest?

What is your favourite colour?

How fast does a sea gull fly?

At login, the system asks one of these questions at random and checks to see if the answer is correct.

A completely different approach is to check is the user has some item, say a plastic card with a magnetic strip on it. The card is inserted into the terminal, which then checks to see whose card it is. This method can be combined with the password idea, as illustrated by automatic cash dispensing machines.

Another approach is to measure physical characteristics such as fingerprints or voiceprints. Yet another approach is signature analysis.

1.17. Computer Authentication

In the previous discussion, we saw how a user identifies himself to the system. The following example illustrates why it may be also important for the system to be able to identify itself to the user. A popular method for gathering passwords uses the following strategy. A user executes leaves running on a terminal a program that simulates the login behavior of the system: it prompts the user for a login name and a password. When a trusting user enters the two items, the program writes this information in a file accessible by the owner of the program, enters the error message "Invalid Login", and executes next the standard system user authentication program. The user, thinking he had mistyped his password, repeats his actions, which lead to a successful login. Thus he never realizes that his login name and password have been gathered by another user.