SECURITY WITHIN THE SYSTEM

At this level of security, as with the protection methods, there are techniques based on all three participants in the common system operations – subject, object, and machinery – and cryptographic techniques can be of assistance.

CAPABILITIES.

Passwords work reasonably well as protection devices, but experience shows that they're not very reliable if you need strict security. We would like something which preserves the good features of passwords, but is more controlled. Capabilities are designed to meet this requirement.

We would like to preserve the password's link with the subject, so that we can easily pass on privileges to other people or processes; but they must somehow be made impossible to steal or to forge. For greater versatility, we would like subjects to be able to copy or reduce capabilities, and pass them to other subjects. This means that the original owner of an object must receive some sort of complete capability embodying all rights over the object when the object comes into being; clearly, only the system must be able to construct a capability. No other operations are permitted.

To satisfy these requirements, capabilities must be implemented in such a way that you can't manipulate them by ordinary means – which means that they have to be represented in hardware, or at least kept outside your addressing space – or they must be such that any attempt to tamper with them is detectable, perhaps by cryptographic techniques. We describe some implementation methods in the next chapter.

To use capabilities for tasks which require them, you do something amounting to presenting the capability as you would otherwise present a password. With some implementations, it's done automatically for you, while in others you might pass the capability as a parameter. As well as that, though, certain other operations are required for general administration. Here's a list, with notes. There are more details in the next chapter; not all the operations are required with some implementations of capabilities, and some of them are quite trivial to implement. Nevertheless, they have to be there, and the list gives you a set of conditions which must be satisfied if you dream up a new implementation for capabilities.

New capability : Every time you make something – a file, a programme, etc. – you will be given a capability with all rights over the object. You will have to produce the capability when you want to use the object, and you might want to pass it on to other people.

Copy : If you do want to pass on the capability, or to store it in a safe place in case you inadvertently lose the original (recall that even you can't use the object without a capability), you are likely to want a copy.

Transfer : This is the operation with which you pass a capability to someone else.

Reduction : When you give someone else a capability for one of your objects, you might not want to pass on all rights. For example, you might want to give read-only access to a file you've made. The reduction operator is used to change the rights for this purpose; it accepts a capability and a set of permissions, and produces a capability which includes the permissions which were allowed by the original capability. Notice that *only* reduction is ever provided; if you could increase the rights, then the protection mechanism wouldn't work any more.

Validate and Test : There's nothing secret about the meaning of a capability, so it's quite reasonable that you should be able to inspect a capability to find out what it allows you to do.

Withdraw : What do you do if you give someone a capability for an object, and then find that the person isn't trustworthy ? You then want to withdraw that capability.

As an example of an application of capabilities, consider this requirement. The software package SP performs a simple clerical operation on local files, for which it must read and write files in the local directory. Each subject using SP must be individually invoiced for each session, so it is decided to keep a secure log file containing start and stop times, and the identification of the subject concerned, for each use of SP. Unfortunately, SP is not well designed for this purpose; the software is supplied as a code file which cannot be changed, and which contains no provision for accounting procedures. The constraints which must be enforced are these :

People must not have direct access to SP	- or they could run it without a log entry.
People must not have access to the log file	- or they could fiddle their accounts.
People must have access to something	- or they can't get any work done.
Therefore there must be a programme (say) SPGUARD to act as intermediary.	
People must have access to SPGUARD	- so that they can run SP indirectly.
SPGUARD must survive throughout the execution of SP	so that it can record both start and stop times in the log
SPGUARD must be able to determine when SP stops running	ditto

If we assume an operating system resembling Unix, where several processes may run simultaneously, with the ability to wait until their children have died, then the sequence of events might be something like this :

1 : Person runs SPGUARD.
2 : SPGUARD makes a starting log entry.
3 : SPGUARD starts SP.
4 : SPGUARD gives Person a read and write capability for SP, permitting interaction.
5 : Person uses SP.
6 : Person finishes, and stops SP.
7 : SPGUARD, notified that SP has stopped, makes a finishing log entry.
8 : SPGUARD stops.

The access matrix is something like the diagram below. ("User" means that the subject – a programme – has the same rights as the subject executing it. This is the normal assumption, and we only need special provision in cases where some different protection rules are required.)

		Subject		
		Person	SPGUARD	SP
Object	SPGUARD	Execute	-	User
	SP	No	Execute	-
	Accounting files	No	Write	User
	Person's files	All	User	User

ACCESS CONTROL LISTS.

Just as capabilities are associated with subjects, we can construct security measures based on objects. The simplest such mechanism is the protection code, defined by an object's owner, and kept by the system with the object. This in some way gives information which the system can use to check each attempt to gain access to the object; access is denied unless the subject attempting access satisfies the condition. A simple example is the Unix file protection method, where each file is tagged with a set of protection codes which the system inspects whenever a subject tries to gain access to a file.

Like passwords, the simpler implementations of protection codes are acceptable for protection, but less than satisfactory for security. They cannot be stolen as passwords can, but they are not sufficiently specific to define conditions for access with sufficient precision. Access control lists remedy this defect. They work in much the same way as the protection codes, but instead of relying on standard, and fairly crude, tests of identity, the tests used can be made far more specific.

Being more specific, the requirements must be listed explicitly, and this is more complicated. In principle, the access control list for an object is a collection of entries of the form *{ subject, test[1 : modes] }*, each identifying a set of tests, one for each access mode, which must be satisfied if the subject is to be granted access in that mode. In practice, it is often sufficient to restrict the tests to simple records of whether or not access is permitted, and further abbreviation is possible by organising the subjects into groups.

To use access control lists, you must be able to **add entries** to the list for any object which you own. You must also be able to **alter** or **delete** entries as is appropriate. Details will depend on the system used, but the process need be no more elaborate than editing an ordinary file.

PROTECTION RINGS.

Capabilities developed from passwords, and access control lists from protection codes. Similarly, protection rings developed from supervisor calls. The effect of the supervisor call mechanism is to divide software into two sorts : an ordinary sort, which has limited access to the machine's functions, and a privileged sort, which can do what it wants. The privileged software can use unprivileged instructions as well as privileged, but unprivileged software cannot use privileged instructions.

In a protection ring system, this two-level class distinction system is expanded into a multilevel hierarchy. Imagine the system as a set of concentric circles, with the highest security in the middle. Everything in the system belongs to a protection ring. Every attempt by a subject to gain access to an object is checked. Operations by subjects on objects in outer rings are permitted; attempted operations on objects in inner rings are trapped and must be authenticated.

Just as with supervisor calls, protection rings must be implemented by hardware to make any sense at all. They work well in organisations where hierarchies are important.

MIXING METHODS.

The methods are not mutually exclusive, and many secure systems provide more than one. The mixture of capabilities and access control lists is fairly popular, because the two methods complement each other very effectively.

Capabilities excel in their ability to pass permission from one subject to another. Some way to do so is essential if (for example) a protected file must be used by a system utility to perform some task for a subject authorised to use the file. The subject can pass (a suitably reduced version of) the required capability to the system utility. Similarly, it is sensible in a large organisation to permit a subject to pass on an appropriate selection of privileges to an underling for certain purposes. Operations of this sort could certainly be managed by access control lists, but for such temporary purposes the overheads of the owners of the various objects requesting changes in their capability lists twice in each case would be considerable.

Access control lists can be made much more specific, and can be controlled by the object's owner at any time; changes are implemented instantly. Access control lists may be used to restrict access to certain members of groups even if all members of the groups have been given capabilities, or to impose short-term restrictions in special cases (such as a general restriction to read-only access while the owner require write access). This sort of specificity is very difficult to guarantee with capabilities, because by their very nature they pass out of the control of the owner of the object, and could be passed from a trusted agent to others which might be less trustworthy.

The use of mixed methods was the final recommendation from a study[REQ19] of possible ways to provide security systems for Java. The security techniques considered were capabilities, and two other software methods not discussed here, with each technique providing valuable features not present in the others. In an anticlimax, the authors - after recommending the combined system - end with a note that "understanding how to create such a hybrid system is a main area for future research".

COMPARE :

Lane and Mooney[INT3], Chapter 16; Silberschatz and Galvin[INT4], Chapter 13.

REFERENCE.

REQ19 : D.S. Wallach, D. Balfanz, D. Dean, E.W. Felten : "Extensible security architectures for Java", *Op. Sys. Rev.* **31#5**, 116-128 (December, 1997).

QUESTIONS.

What would you need to implement a system in which a person could belong to several "groups" ? What about the security problems ?
