IMPLEMENTING CAPABILITIES

Why single out capabilities for special attention ? It isn't that they're uniquely important, or overwhelmingly popular; both capabilities and access control lists are quite widely used. It's more because, though there is a strong family resemblance between passwords and capabilities, the upward step brings in new ideas in a way that the corresponding steps to access control lists and protection rings don't. The requirement to implement unforgeable tokens is difficult to satisfy adequately with traditional resources, so new techniques must be introduced.

WHAT DO WE NEED ?

Suppose we use a password to control access to a file – say, to give read access to a file called ABC. How does it work ?

- First, we have a password, which could be *GRXXXXXX25*. Or, of course, it could be practically anything else, except something sensible and obvious like *{ permission to read ABC }*.
- Second, we have a file, attached to which somewhere there must be a record of the password – and there must be similar records of any other passwords which stand for different sets of access privileges.
- Third, we make a request, which must say something like *{ read, ABC, GRXXXXXX25 }*.
- Fourth, the file access software must receive the request, go to the file attributes, retrieve the actual passwords (or something equivalent), and compare them one by one until it finds *GRXXXXXX25*; then it can open the file with the permitted access mode, provided that includes *read*.

That's a lot of work – and most of it could have been avoided if we had been able to use a password more like *{ permission to read ABC }*, directly including a lot of the information which the system had to store in the file directory and check when it opened the file. Why can't we use such a simple password ? – obviously, because it can too easily be forged. What we want, then, is something which includes the information of *{ permission to read ABC }*, but which can't be forged. That's the capability. Once we have it, it includes both the file name and the password, so we can just use an instruction like *{ read, capability }*; that's why capabilities are sometimes called unforgeable object names.

CAPABILITIES IN ACTION.

A capability is a token of permission to use some resource with some defined level of access. Several operations on capabilities must be available to subjects. It must be possible to *copy* a capability, to *reduce* its privileges, and to *transfer* it from one subject to another without restriction. It should also be possible to *withdraw* a capability which has been issued. It must not be possible to *forge* a capability : only the operating system may make new capabilities, and any other attempt to make a capability must start from an existing capability. Neither must it be possible to *increase* a capability's privileges, or it would be impossible to give different levels of access to different subjects. Finally, the system – though not necessarily individual subjects – must be able to *test* a capability in order to determine the level of access to a specified object which it permits.

It is by no means easy to implement something which behaves according to those specifications using the resources of a conventional computer. In this chapter we describe three approaches, all of which have been used in practice.

Everything hinges on the requirement that we must eliminate the possibility of forgery. What does this requirement of unforgeability imply ? It certainly means that the capability must be something which we can't write down for ourselves, because if we

could write it then so could someone else. This in turn means that we must make it physically impossible for any subject except the operating system to construct, or otherwise to tamper with, the capabilities, or we must make sure that we can detect that tampering has occurred. Therefore, either subjects must never be able to get hold of the capabilities themselves, or the system must give them to the subjects in such a way that they can't make their own instead. The methods we describe below use different means to achieve these ends. Briefly, capabilities might "float free", like other variables (hard), or be kept in a "subject table", accessible only to the system (easier), or they might be protected by cryptographic means and expressed in a form in which they can safely be transmitted from computer to computer (different).

THE VARIABLE MODEL.

In this type of implementation, the system manufactures a capability for an object when the object is first made, and provides appropriate operations by which capabilities can be changed on request. Capabilities are handled as variables of capability type. To show you are permitted access to some protected operation or data, you must submit an appropriate capability – for example, as an additional parameter to the request for access. It must be possible to pass these variables around like any other variables, but they must not be forgeable or modifiable in unauthorised ways. (Because capabilities are intended to be transferable, there's not a lot you can do about it if someone manages to steal one.) The system must therefore either be able to prevent illegal operations on capabilities, or identify the results of such illegal operations. Two ways of providing the required level of security are the use of tagged architecture and application of cryptographic methods.

Tagged architecture.

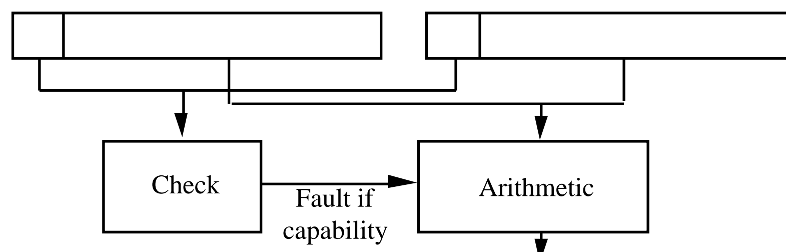
Hardware assistance is necessary to prevent changes to capability variables by ordinary programming methods; the processor must be able to recognise a capability variable, and reject any attempt to operate upon it except in carefully guarded ways. Here is a description of a possible implementation.

- To identify the capability variables, we provide every word in the computer system (including processor, memory, disc, anything else) with a *capability bit* which is on for capability variables but otherwise off. The variable might look like this :

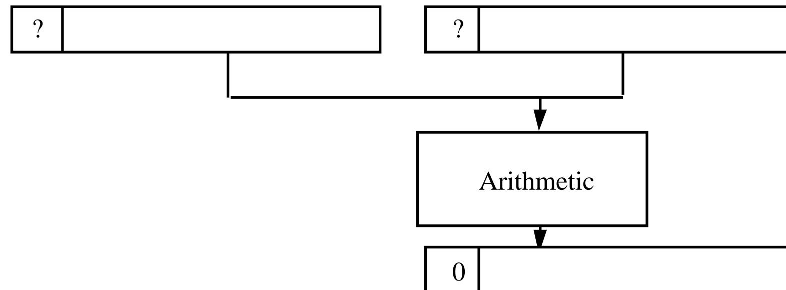
  Apart from the capability bit, that's just an ordinary machine word, which can be manipulated just like any other.

- We provide a single *hardware operator to construct a capability variable*, and ensure that it cannot be used in any processor state accessible to unprivileged software. This guarantees that capabilities can only be generated by appropriate operating system procedures. This operator is used by the system whenever any new object is constructed or registered; a master capability with full access privileges must be constructed for the new object, and returned to the subject which requested the construction or registration. It is then up to the subject to administer the capability in whatever way is appropriate.

- We add to the processor *hardware which checks the capability bits* of all operands of normal data manipulations, and causes a processor fault if it finds a capability bit on. This ensures that people can't try to upgrade capabilities, or direct them towards different resources.



That's a rather fierce restriction; for example, it also means that people can't look at the internal structure of capabilities, though that wouldn't matter in itself. A less restrictive, and rather simpler, alternative is simply to build the hardware so that the results of all ordinary arithmetic operations have their capability bits turned off – then you can do as much arithmetic as you want on capabilities, but you won't be able to make new ones. This system cannot detect attempts to perform arithmetic on capabilities, but, as nothing dangerous can result, perhaps that's all right.



- We provide further *hardware operators to implement the permitted capability operations* – to reduce and compare privileges. These need not be specially protected.

To use the capabilities, each operating system component which grants access to a resource must be able to accept a capability variable as well as its ordinary parameters. (You might not want to do this all the time – many resources don't need special protection.) For example, there might be a procedure for opening a file of the form **open(filename, access mode required, capability)**. The procedure must then use the access mode parameter to select a system-defined test which it then applies to the capability, using the result of the test to decide whether to permit or deny access.

Provided that you buy a computer with the requisite hardware, that's simple, straightforward, effective, and fast. It also makes sense : if you want a computer to handle integers or floating point numbers efficiently, you equip it with hardware to do the job. This is the same approach applied to capabilities.

The system becomes less effective if used in a network, unless the communications lines are made very secure; once capabilities get outside the processor and the hardware directly under its control, they're just bits like everything else, and are subject to tampering by anyone who can manage a wiretap. If the network includes any computers which don't have hardware capabilities, you don't even need a wiretap. To use capability variables in such a system, different methods are needed.

A cryptographic technique.

If we can't rely on hardware, then we can no longer prevent attempts at undesirable operations; but we can hope to identify their results. We have to use software, but we can still implement capability variables adequately by relying on cryptographic techniques. The approach is typical of cryptographic methods : we can't strictly prevent people from cracking the system, but we make it too expensive to be worth the trouble.

The example described is used in the Amœba operating system[REQ15], specifically designed for use in distributed systems which link computers of any type.

The basic information in a capability is the identity of the object to be protected, and the rights over the object which can be exercised by whoever holds the capability. Amœba leans strongly towards object-oriented methods, so every object in the system is identified by its name, and the name of the server with which it is associated. (Compare the Macintosh system.) Each server has a 48-bit systemwide identifying number, and assigns a 24-bit number to each object under its control. Together, these give a unique 72-

bit name. The rights are encoded into 8 bits, which might be interpreted differently according to the object protected.

These 80 bits of basic information are protected by appending 48 bits of checking information. This is generated from the first 80 bits using some sort of "one-way algorithm" – a computation which is comparatively easy to perform if you know how, but which is exceedingly difficult to invert even if you can produce examples of the complete 128-bit capability variables at will. The complete capability variable looks like this :

```
SSSSSSSSSSSSSSSSSSSSSSSS    OOOOOOOOOOOO    RRRR    CCCCCCCCCCCCCCCCCCCCCCCC

        Server                  Object      Rights           Check
          48                      24           8               48
```

Now anybody can look at the capabilities, but only some agent with privileged access to the encoding algorithm can check them – so even though you know all about the structure, and you can manipulate the rights bits any way you like, you can't work out the correct check bits, and the checking algorithm will catch you if you try to use your forgery.

THE PRIVILEGE MODEL.

This approach is quite different. A capability is regarded as a privilege administered by the operating system. It is held as part of the information which the system has about the subject, and can never be directly manipulated by the subject. The capabilities owned by a person who is registered normally by the system are recorded with the userdata information, and are only accessible to operating system routines. There must be provision for associating capabilities with any active subject (programme, device, etc.) and for transferring capabilities when new operations are started. Capability operations – constructing new capabilities, and transferring them, perhaps with reduced rights, to other subjects – are handled by operating system procedures.

Using capabilities of this sort is fairly expensive, as system calls, and perhaps also userdata references, must be executed for every operation. The expense can be reduced by using the capability system only for objects which are specially sensitive in some way, leaving the rest to the mercies of the ordinary operating system protection devices. In this case, objects to be guarded by capabilities must be marked in some recognisable way. If there is any doubt about the security of the marks, the objects can also be registered with the operating system. On any attempt at access to a marked object, the system must inspect the subject's capability list in the obvious way, and take action accordingly. Notice that this is quite automatic : a subject might even be given a capability and use it without being aware of its existence. This system is very well suited to administering resources used by inexperienced people.

COMPARISON.

Apart from questions of efficiency, the big difference between the privilege and variable models (regarding the cryptographic model as a special implementation of the variable model) is the centralisation of control in the privilege model. If all the capability information is held by the operating system, it can all be found if required. With a variable model, on the other hand, a capability might be hidden away in archived files, quite inaccessible even if you wanted to look for it – and in any case, in the cryptographic model, quite unidentifiable even if you could search.

That doesn't matter much in itself, but becomes a problem if you want to withdraw access privileges for some resource. As capabilities can be handed on from one person to another without constraint, if you are using the variable model you don't know who has capabilities for any object; all you can do is to make a new capability for the object to be protected, and start distributing this, amended as required, to people who are entitled to it. This takes time to diffuse through the system, and until the diffusion is complete, some people who should have access to the resource might find themselves locked out. Using the privilege model, though, all capabilities are accessible to the system at all times, so

such modifications are much easier – and you could in principle check on just who was passing privileges to whom.

Another problem associated with capabilities is their permanence. A capability once given can lurk in someone's files for years, and then be exercised again when for some reason or other it's quite inappropriate. (The person has moved to a different job, the operation guarded by the capability has evolved into something rather different. etc.) The standard solution is to issue new capabilities as described above, but with the disadvantages also described above. An alternative possibility has been suggested[REQ16], in a rather more ambitious system including both capabilities and access control lists. Here, every object in the system has "money", and must pay rent. If the money runs out, the object is removed. Anyone who wishes to retain an object must therefore keep its bank balance topped up. That doesn't actually guarantee anything, but it does ensure that people don't just keep things hanging around because they've forgotten about them.

Here's a rather long table, summarising the three methods.

Operation	Implementation		
	Variable model, hardware implementation	Variable model, cryptographic implementation	Privilege model
New capability	Any system component which constructs or sets up an object of protected type (files, directories, devices, etc.) must, while running in supervisor mode, construct a capability variable including an identifier for the object and a complete set of privileges. This is converted into a capability using a "make capability" hardware operator, and can then be returned to the subject performing the action.	Any system component which constructs or sets up an object of protected type (files, directories, devices, etc.) must, while running in supervisor mode, construct a capability variable including an identifier for the object and a complete set of privileges. This is converted into a capability using a "make capability" function, and can then be returned to the subject performing the action.	Whenever a system component constructs or sets up an object of protected type (files, directories, devices, etc.), the operating system constructs a new entry in the capability list of the subject performing the action. The entry includes an identifier for the object and a complete set of privileges.
Copy	Conventional load and store operations may be used; the hardware must not change the capability bit.	Conventional load and store operations may be used; the capability is an ordinary variable.	No local copying operation is required.

Transfer	A capability may be transferred from subject to subject in a message, in a file, through shared memory, etc. Transfer through external communications facilities is unsafe, because there is unlikely to be any way to protect the special nature of the capability bit, and forgery is therefore easy.	A capability may be transferred from subject to subject in a message, in a file, through shared memory, etc. External communications raise no additional difficulties.	A system call is used. It is given the identity of the recipient, the identity of the object protected, and a list of the privileges which are to be transferred. Either it constructs a new capability in the recipient's capability list, or, if the recipient already held a capability for the object, adds any new privileges acquired.
Reduction	A hardware operator is required. It must accept the initial capability and a specification of the privileges to be removed, and construct a new capability representing the new set of privileges.	A supervisor call is required. It must accept the initial capability and a specification of the privileges to be removed, and construct a new capability representing the new set of privileges, which it returns to the caller.	This operation is included in the transfer operation.
Validate and Test	A hardware test operator is required for validation. Ordinary inspection is not a sensitive operation, and need not be protected : the capability is disentangled to give object identity and privileges, and obvious tests are conducted. If the encoding is complex, or the object identities are difficult to interpret, a system procedure can be provided	A supervisor call is required for validation. Ordinary inspection is not a sensitive operation, and need not be protected : the capability is disentangled to give object identity and privileges, and obvious tests are conducted. If the encoding is complex, or the object identities are difficult to interpret, a system procedure can be provided.	A supervisor call can be provided to control access to the capability list, but there is no reason why the operation itself should require any special protection.

Withdraw	There is no mechanism for withdrawal with capabilities implemented as variables.	There is no mechanism for withdrawal with capabilities implemented as variables.	A supervisor call can be used to request that the system withdraw specified privileges from identified subjects. The ability to withdraw privileges might itself be a capability, or it might be restricted to the original owner.

REFERENCES.

REQ15 : S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, H. van Staveren : *IEEE Computer* **23#5**, 44 (May 1990).)

REQ16 : M. Anderson, R.D. Pose, C.S. Wallace : *Computer Journal* **29**, 1 (1986).