Capability-based Access Control Mechanisms

Lecturer: Professor Fred B. Schneider

Lecture notes by Lynette I. Millett

We have been discussing access control lists (ACLs) as a mechanism to do authorization. The mechanism is like a guard sitting at a guard house guarding access to an object. When a subject comes along and wants access, the guard checks the subject's name against a list. The subject is authorized to have access if its name is on the list.

Several issues arise with ACLs. First, it is necessary to encode the lists tersely in order to save space and lookup times. Second, in order to support small protection domains, a user should be able to have multiple identities. Third, it is generally a good idea to have conservative defaults, since new objects will be created often. Tools like 'chmod' that change file permissions can be bothersome, so it's important to have good defaults in place.

A related concern is that of conveying security information to users. We need good user metaphors for security. There are many examples of physical experiences that provide good metaphors for computing including the desktop metaphor in user interfaces. We seek such a metaphor for access control. One possibility is to describe access control in terms of locks and keys. However, this metaphor isn't completely helpful, because people do not regularly create new rooms and new locks, whereas new objects are created in computer systems all the time. The notion of nested locks is not intuitive, either. In the physical world, it is not likely that one would use a key to go through a door, not look at anything in the room and use another to key to go through another door. Yet that is what happens when a file is accessed by way of a path name.

Capability-based Mechanisms

One approach to storing an access control matrix, discussed previously, is to store columns with objects (an ACL). We will now discuss another approach: storing rows with subjects (capabilities).

A *capability* can be thought of as a pair (x, r) where x is the name of an object and r is a set of privileges or rights. With each subject we can store that subject's capabilities. And, the subject presents to the guard a capability in order to get access to an object. Note that a capability is completely transferable; it doesn't matter *who* presents the capability. This framework completely eliminates the need for authentication. However, with ACLs we were assuming that authentication was unforgeable. With capabilities, we now need a way to make capabilities unforgeable. The success of a capability-based mechanism depends on it.

The term "capability" originated in a 1966 paper by Dennis and Van Horn. An operating system based on these ideas was then built for a PDP-1 by Ackerman and Plummer. There were earlier systems that implement capabilities directly in the hardware, however. All of the 'action' in systems that use capabilities is in implementing and achieving their unforgeability.

A first question is how to represent object name x in capability (x,r). (We require different objects to have different names.) One solution would be to use the address of the object as its name. This is not quite sufficient, since a pointer to an array and a pointer to the first element in an array are the same address, yet clearly different objects. Thus, we must also encode the length of the object as well. The start address plus object-length uniquely describes the object. We have seen this before in segmented memory, and in systems with segmented memory we can use the addressing hardware to support the naming convention. An obvious problem with this naming scheme, though, is that if the address changes, then the name changes. Therefore, this scheme is most useful in situations where objects don't move around a lot.

Instead of using an object's address, we could use a random bit string as a name for an object. Suppose we have 50 bits. If a new object is named each microsecond, then it will take about 35 years to run out of names. (The namespace size issue is not just a security issue. We see it also in telephony with new area codes being created and, of course, on the Internet.) With unique bitstrings for names, we need a way to translate from the name to the object; a hashtable, which might also include protection rights, is a reasonable solution.

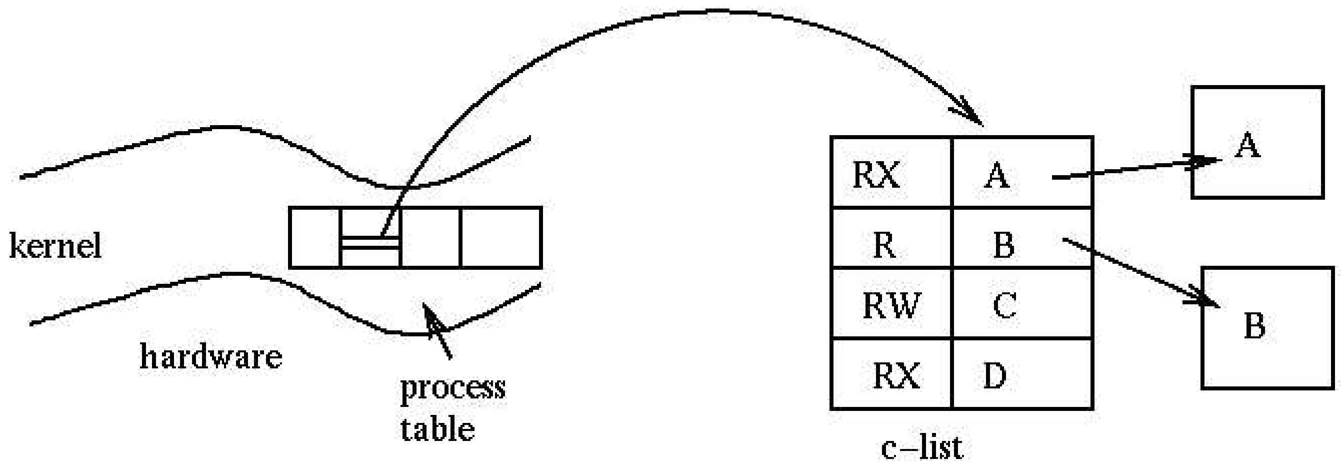
We still have to find a way to make capabilities unforgeable. There are a number of possibilities.

- Hardware tags: Each word will have a 1-bit tag associated with it. The tag "on" means that programs cannot change or copy that particular word (because it stores a capability). Hardware to enforce such tags was implemented in a few systems, none of which were particularly successful commercially.
- Protected address space: Store capabilities in parts of memory that are inaccessible to programs. (Recall that we have done this before with ACLs.)
- Language-based security: We can use a programming language to enforce restrictions on access and modification to capabilities. Java takes this approach, for example.
- Cryptography: Take the capability and use encrypting to render it unintelligible to programs that should not be accessing it.

Protected Address Space Implementation of Capabilities

In order to store capabilities using a protected address space, we associate with each process a c-list (capability list). Recall that the operating system stores and loads processor state information for each process. We include the c-list in that state information.

The c-list is stored in the kernel's memory and is a table with rights and pointers to objects.



Instead of giving capabilities directly to processes, we only provide processes with indices to the table. A process can utter only these indices, and thus a process can only talk about capabilities that it *has*, making it impossible to create new capabilities. We are using indirection to prevent processes from forging capabilities.

Associated with a c-list are meta-instructions that allow capabilities to be changed. These meta-instructions include: create a new c-list, copy a capability into the c-list, and delete a capability from a c-list. These operations allow processes to instruct the kernel to move capabilities around. For information on how the kernel protects its own memory see an operating systems text or course.

Sparse Name-Space Implemention of Capabilities

An alternative approach to ensuring unforgeability the capability is represented with a bit string having the following structure:

object name	rights	"random" bits

The "random" bits, when interpreted correctly, will provide unforgeability. Specifically, the kernel, when creating a capability does the following: create a random bit string r, store the bit string that encodes the object name, the rights, and r in its own memory, and then give a copy of the capability to the user. Users can pass capabilities around as much as they like; the kernel is not involved in rights propagation. However, users can't *change* rights, because then the capability bit string would not match the kernel's local copy (which is checked whenever the kernel is presented with a capability.) If the random string is long enough, it would take an attacker a long time to guess. A good random number generator is required or else guessing random numbers becomes easy.

Cryptography for Implementing Capabilities

Encryption can be used to scramble an input and produce enciphered text that is hard to decipher without a key. The scrambling depends on the key. Suppose that the kernel has a key *k* that is a secret. Assume that the kernel can compute

$$[\text{object} \,||\, \text{rights}]_k,$$

the object name concatenated with access rights and encrypted using the key *k*. The kernel can then return this encrypted string as the capability. However, we also need to ensure that random processing by a subject won't also produce something meaningful. To do this, a capability is constructed by prefixing the unencrypted object name to the above encrypted string:

$$(\text{object}: [\text{object} \,||\, \text{rights}]_k).$$

Once capabilities are implemented in this way, it is only necessary for the kernel to maintain the secret key k in storage. A brute force attempt to guess the key remains possible, but empirical results suggest that a 64 bit key is usually sufficient.

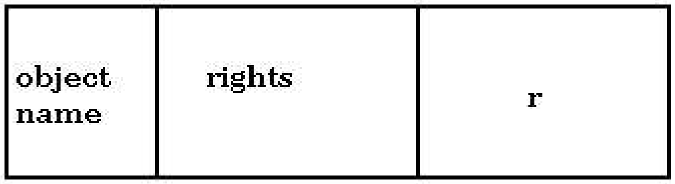
It should be noted that the above capability is not a completely random string. A user knows that the object is encoded, and it also knows the object's name (since that is an unencrypted part of the capability). With enough such capabilities and knowledge of the encryption algorithm, it might be possible for users to learn something about the secret key k. Attacks based on such knowledge are referred to as "partially known plain text attacks." To prevent such attacks, we can add random bits R (referred to as "salt") to the beginning of the encrypted portion :

$$(\text{object}: [R: \text{object} \,||\, \text{rights}]_k).$$

While this scheme does not use a lot of kernel storage, it is still expensive for a process to give-away capabilities with only a subset of the access rights. A call to the kernel, passing it the process's original capability is needed. (The kernel decrypts this capability and passes back a modified capability.)

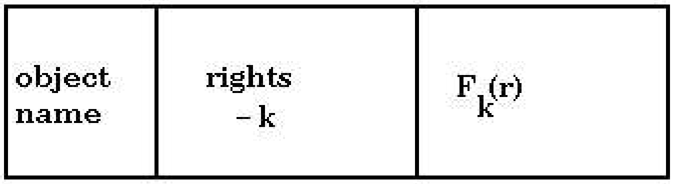
In order to allow processes to more easily pass around capabilities, we will take advantage of "one-way trapdoor functions." Such functions are difficult to invert, unless some trapdoor (secret) is known. Suppose we have a number of one-way trapdoor functions (F_1, F_2, F_3, ... F_n) that are also commutative (i.e. where $F_i(F_j(x)) = F_j(F_i(x))$). And, suppose that there are n access rights to some object of interest. Then to get the functionality of restricting access, we do the following:

- The kernel picks r a random string.
- A capability to allow all n access rights is created that looks like:



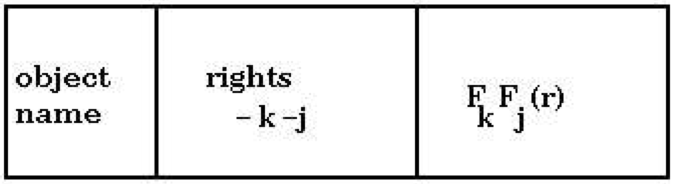
object name	rights	r

- We now need a way for a client to turn off the kth right before passing the capability along. This is done by having the client modify the capability to:



object name	rights - k	$F_k(r)$

At this point, the kernel knows r, so if the user changes the third field, then the kernel will note the tampering. The kernel uses the second field to determine which functions to apply to construct the third field from r. Doing so enables the kernel to determine which access rights are no longer present in the capability. For example, the user may now remove access right j by constructiong the new capability:



object name	rights - k -j	$F_k F_j(r)$

Elegant as these cryptography-based schemes are, protected memory space is the dominant implementation method for capabilities. The encryption techniques are used in distributed systems (where objects and object references actually leave a host.)

Implementing Protected Subsystems with Capabilities

We now discuss how capabilities can be used to build systems with small protection domains. Recall we desire small protection domains in order to practice the principle of least privilege.

We basically use a form of object orientation. We employ a type system with base types and extended (user-defined) types. Procedures are the only way to modify the state of an object. A domain change needs to be associated with invoking such a procedure (sometimes called "methods").

As an example, suppose we have a queue object and wish to invoke the append method. There are two possible ways this could work. One would be to pass an instance of a queue as an argument to the method. The other would be to pass a name of a queue to the method. The first allows the queue object to be memoryless, and the second requires static memory.

If we pursue the first option, what capabilities should be present at different points? Before calling append, the process should not be able to change the queue. (We want only the methods provided by the queue object to do so.) Thus, the call/invocation of append should actually increase the rights so the management object can modify the queue. This is referred to as *rights amplification*. On the other hand, the queue object should not have access beyond arguments it is being invoked with (such as what data to append), so *rights attenuation* is also required. Amplification and attenuation provide a complete set of techniques to go from one set of rights to another.

We present two specific solutions to the implementation of amplification and attenuation.

Sealing is applicable in the case where we pass to the method a representation of the object on which the operation should be performed. It is very programming language oriented. We posit two operations: seal and unseal. The seal operation produces an unintelligible bit string from an object representation. In our queue example, the object manager would pass back a representation of the queue in sealed form. All a user can do is invoke routines (which can then unseal) with that sealed representation. This scheme can be thought of as using an opaque envelope; it leverages the type system. Seal and unseal can be built in terms of encryption techniques discussed previously. A problem, however, is that we can't control static storage, and information could be stolen. Further, without static storage the functionality of objects is limited.

Amplification, as implemented in the Hydra system at CMU by Jones and Wulf, demonstrated a second solution. In Hydra, everything is an object, and capabilities are used to address all objects. The kernel is involved in all operation invocations. Every object has a name (64 bits), a type (64 bits), and a representation. The representation consists of two parts: data (memory words that can be read/written) and a c-list. The c-list is stored in the kernel's protected memory. In order to create instances of an object, there is an object of type "type" and its "create" method is invoked. This create method uses a creation template, and storage is allocated according to the template. The template also has operations to initialize the c-list for the new object and has code for the operations that transform instances of the type of object being created.

An operation in Hydra is an object of type "procedure." Associated with it are instructions, constants, capabilities needed for execution of the procedure, and a template for the arguments. When a procedure is activated by the kernel, a local name space (LNS) is created. The local name space is a list of capabilities taken from two places: the capabilities already in the procedure and the actual parameters provided by the caller. As the code runs, all the names it utters are relative to the local name space. That is, small integers are used as offsets into the LNS.

The "action" here is all in how the LNS is created, because that defines what the procedure is allowed to do. Upon a call, the kernel does the following:

- It checks the capability of each argument to be sure that it has the necessary type and access rights. It does this by comparing the actuals with the parameter template associated with the procedure.
- If necessary, it does amplification of rights using the template. The parameter template specifies both *necessary rights* and *amplified rights*. The latter are rights that are temporarily added to the parameter for the duration of the procedure activation. If the argument has all the necessary rights, only then are the rights augmented according to the amplified rights.

Given the above, it would appear that the author of a procedure (and its template) could conceivably take rights to everything. To disallow this, there are certain generic rights associated with all capabilities that cannot be gained through amplification. Generic rights are:

- **Modify**: Leave this off a parameter and the object becomes read-only during procedure execution.
- **Environment**: Without this, the called procedure cannot save the capability for later use. This applies transitively if the capability is then passed to another procedure.
- **Unconfined**: It is not possible to store data or the capability list without this right.

All of these rights are controlled by the caller.