CHAPTER 14

System Protection

The various processes in an operating system must be protected from one another's activities. For that purpose, various mechanisms exist that can be used to ensure that the files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

In this chapter, we examine the problem of protection in great detail and develop a unifying model for implementing protection.

It is important that the student learn the concepts of the access matrix, access lists, and capabilities. Capabilities have been used in several modern systems and can be tied in with abstract data types. The paper by Lampson [1971] is the classic reference on protection.

14.1 Set up a dynamic protection structure that changes the set of resources available with respect to the time allotted to the three categories of users. As time changes, so does the domain of users eligible to play the computer games. When the time comes that a user's eligibility is over, a revocation process must occur. Revocation could be immediate, selective (since the computer staff may access it at any hour), total, and temporary (since rights to access will be given back later in the day).

14.2 $A(x,y)$ is a subset of $A(z,y)$.

14.3 The roles in a role-based access control facility are similar to the domain in the access-matrix facility. Just as a domain is granted access to certain resources, a role is also granted access to the appropriate resources. The two approaches differ in the amount of flexibility and the kind of access privileges that are granted to the entities. Certain access-control facilities allow modules to perform a *switch* operation that allows them to assume the privileges of a different module, and this operation can be performed in a transparent manner. Such switches are less transparent in role-based systems where the ability to switch roles is not a privilege that is granted through a mechanism that is part of the access-control system, but instead requires the explicit use of passwords.

14.4 Rights amplification is required to deal with cross-domain calls where code in the calling domain does not have the access privileges to perform certain operations on an object but the called procedure has an expanded set of access privileges on the same object. Typically, when an object is created by a module, if the module wants to export the object to other modules without granting the other modules privileges to modify the object, it could export the object with those kinds of access privileges disabled. When the object is passed back to the module that created it in order to perform some mutations on it, the rights associated with the object need to be expanded. A more coarse-grained approach to rights amplification is employed in Multics. When a cross-ring call occurs, a set of checks are made to ensure that the calling code has sufficient rights to invoke the target code. Assuming that the checks are satisfied, the target code is invoked and the ring number associated with the process is modified to be ring number associated with the target code, thereby expanding the access rights associated with the process.

14.5 The ring-based protection scheme requires the modules to be ordered in a strictly hierarchical fashion. It also enforces the restriction that system code in internal rings cannot invoke operations in the external rings. This restriction limits the flexibility in structuring the code and is unnecessarily restrictive. The capability system provided by Hydra not only allows unstructured interactions between different modules, but also enables the dynamic instantiation of new modules as the need arises.

14.6 The ring should be associated with an access bracket (b1, b2), a limit value b3, and a set of designated entry points. The processes that are allowed to invoke any code stored in segment 0 in an unconstrained manner are those processes that are currently executing in ring i where $b1 \leq i \leq b2$. Any other process executing within ring $b2 < i \leq b3$ is allowed to invoke only those functions that are designated entry points. This implies that we should have $b1 = 0$ and set $b2$ to be the highest ring number that comprises of system code that is allowed to invoke the code in segment 0 in an unconstrained fashion. We should also store only the system call functions as designated entry points and we should set $b3$ to be the ring number associated with user code so that user code can invoke the system calls.

14.7 Capabilities associated with domains provide substantial flexibility and faster access to objects. When a domain presents a capability, the system just needs to check the authenticity of the capability and that could be performed efficiently. Capabilities could also be passed around from one domain to another domain with great ease, allowing a system with a great amount of flexibility. However, the flexibility comes at the cost of a lack of control: revoking capabilities and restricting the flow of capabilities is a difficult task.

14.8 The strength of storing an access list with each object is the control that comes from storing the access privileges along with each object, thereby allowing the object to revoke or expand the access privileges in a localized manner. The weakness with associating access lists is the overhead

of checking whether the requesting domain appears on the access list. This check would be expensive and needs to be performed every time the object is accessed.

14.9 Yes, this approach is equivalent to including the access privileges of domain B in those of domain A as long as the switch privileges associated with domain B are also copied over to domain A.

14.10 The principle of least privileges only limits the damage but does not prevent the misuse of access privileges associated with a module if the module were to be compromised. For instance, if a system code is given the access privileges to deal with the task of managing tertiary storage, a security loophole in the code would not cause any damage to other parts of the system, but it could still cause protection failures in accessing the tertiary storage.

14.11 The principle of least privilege allows users to be given just enough privileges to perform their tasks. A system implemented within the framework of this principle has the property that a failure or compromise of a component does the minimum damage to the system since the failed or compromised component has the least set of privileges required to support its normal mode of operation.

14.12 The contents of the stack could be compromised by other process(es) sharing the stack.

14.13 Reference counts.

14.14 The ring protection scheme in MULTICS does not necessarily enforce the need-to-know principle. If an object must be accessible in a domain at ring level j but not accessible in a domain at ring level i, then we must have $j < i$. But this requirement means that every object accessible in level i must also be accessible in level j.

A similar problem arises in JVM s stack inspection scheme. When a sequence of calls is made within a doPrivileged code block, then all of the code fragments in the called procedure have the same access privileges as the original code block that performed the doPrivileged operation, thereby violating the need-to-know principle.

In Hydra, the rights-amplification mechanism ensures that only the privileged code has access privileges to protected objects, and if this code were to invoke code in other modules, the objects could be exported to the other modules after lowering the access privileges to the exported objects. This mechanism provides fine-grained control over access rights and helps to guarantee that the need-to-know principle is satisfied.

14.15 Hierarchical structure.

14.16 A process may access at any time those resources that it has been authorized to access *and* are required currently to complete its task. It is important in that it limits the amount of damage a faulty process can cause in a system.

14.17 A hardware feature is needed allowing a capability object to be identified as either a capability of accessible object. Typically, several bits are

necessary to distinguish between different types of capability objects. For example, 4 bits could be used to uniquely identify 2^4 or 16 different types of capability objects.

These could not be used for routine memory protection as they offer little else for protection apart from a binary value indicating whether they are a capability object or not. Memory protection requires full support from virtual memory features discussed in Chapter 9.

14.18 When a Java thread issues an access request in a `doPrivileged()` block, the stack frame of the calling thread is *annotated* according to the calling thread's protection domain. A thread with an annotated stack frame can make subsequent method calls that require certain privileges. Thus, the annotation serves to mark a calling thread as being privileged. By allowing a Java program to directly alter the annotations of a stack frame, a program could potentially perform an operation for which it does not have the necessary permissions, thus violating the security model of Java.

14.19 This would be useful as an extra security measure so that the old content of memory cannot be accessed, either intentionally or by accident, by another program. This is especially useful for any highly classified information.