Chapter 2
Access Control Matrix

GRANDPRÉ: Description cannot suit itself in words
To demonstrate the life of such a battle
In life so lifeless as it shows itself.
— *The Life of Henry the Fifth*, IV, ii, 53–55.

A *protection system* describes the conditions under which a system is secure. This chapter presents a classical formulation of a protection system. The *access control matrix model* arose both in operating systems research and in database research; it describes allowed accesses using a matrix.

2.1 Protection State

The *state* of a system is the collection of the current values of all memory locations, all secondary storage, and all registers and other components of the system. The subset of this collection that deals with protection is the *protection state* of the system. An *access control matrix* is one tool that can describe the current protection state.

Consider the set of possible protection states P. Some subset Q of P consists of exactly those states in which the system is authorized to reside. So, whenever the system state is in Q, the system is secure. When the current state is in $P - Q$,[1] the system is not secure. Our interest in representing the state is to characterize those states in Q, and our interest in enforcing security is to ensure that the system state is always an element of Q. Characterizing the states in Q is the function of a *security policy*; preventing the system from entering a state in $P - Q$ is the function of a *security mechanism*. Recall from Definition 1–3 that a mechanism that enforces this restriction is *secure*, and if $P = Q$, the mechanism is precise.

The *access control matrix model* is the most precise model used to describe a protection state. It characterizes the rights of each *subject* (active entity, such as a user or a process) with respect to every other entity. The description of elements

[1] The notation $P - Q$ means all elements of set P not in set Q.

of the access control matrix A form a *specification* against which the current state can be compared. Specifications take many forms, and different specification languages have been created to describe the characteristics of allowable states.

As the system changes, the protection state changes. When a command changes the state of the system, a *state transition* occurs. Very often, constraints on the set of allowed states use these transitions inductively; a set of authorized states is defined, and then a set of operations is allowed on the elements of that set. The result of transforming an authorized state with an operation allowed in that state is an authorized state. By induction, the system will always be in an authorized state. Hence, both states and state transitions are often constrained.

In practice, *any* operation on a real system causes multiple state transitions; the reading, loading, altering, and execution of any datum or instruction causes a transition. We are concerned only with those state transitions that affect the protection state of the system, so only transitions that alter the actions a subject is authorized to take are relevant. For example, a program that resets the value of a local counter variable in a loop to 0 (usually) does not alter the protection state. However, if changing the value of a variable causes the privileges of a process to change, then the process does alter the protection state and needs to be accounted for in the set of transitions.

2.2 Access Control Matrix Model

The simplest framework for describing a protection system is the *access control matrix model*, which describes the rights of subjects over all entities in a matrix. Butler Lampson first proposed this model in 1971 [1132]; Graham and Denning [547, 805] refined it, and we will use their version.

The set of all protected entities (that is, entities that are relevant to the protection state of the system) is called the set of *objects* O. The set of *subjects* S is the set of active objects, such as processes and users. In the access control matrix model, the relationship between these entities is captured by a matrix A with *rights* drawn from a set of rights R in each entry $A[s, o]$, where $s \in S$, $o \in O$, and $A[s, o] \in R$. The subject s has the set of rights $A[s, o]$ over the object o. The set of protection states of the system is represented by the triple (S, O, A). For example, Figure 2–1 shows the protection state of a system. Here, process 1, which owns file 1, can read or write file 1 and can read file 2; process 2 can append to file 1 and read file 2, which it owns. Process 1 can communicate with process 2 by writing to it, and process 2 can read from process 1. Each process owns itself and has read, write, and execute rights over itself. Note that the processes themselves are treated as both subjects (rows) and objects (columns). This enables a process to be the target of operations as well as the operator.

Interpretation of the meaning of these rights varies from system to system. Reading from, writing to, and appending to files is usually clear enough, but what does "reading from" a process mean? Depending on the instantiation of the

	file 1	*file 2*	*process 1*	*process 2*
process 1	read, write, own	read	read, write, execute, own	write
process 2	append	read, own	read	read, write, execute, own

Figure 2–1 An access control matrix. The system has two processes and two files. The set of rights is {read, write, execute, append, own}.

model, it could mean that the reader accepts messages from the process being read, or it could mean that the reader simply looks at the state of the process being read (as a debugger does, for example). The meaning of the right may vary depending on the object involved. The point is that the access control matrix model is an *abstract* model of the protection state, and when one talks about the meaning of some particular access control matrix, one must always talk with respect to a particular implementation or system.

The *own* right is a *distinguished* right, which is a right that is treated specially. In most systems, the owner of an object has special privileges: the ability to add and delete rights for other users (and for the owner). In the system shown in Figure 2–1, for example, process 1 could alter the contents of $A[x, \text{file1}]$, where x is any subject.

EXAMPLE: The UNIX system defines the rights *read*, *write*, and *execute*. When a process reads, writes, or executes a file, these terms mean what one would expect. With respect to a directory, however, *read* means to be able to list the contents of the directory; *write* means to be able to create, rename, or delete files or subdirectories in that directory; and *execute* means to be able to access files or subdirectories in that directory. When a process interacts with another process, *read* means to be able to receive signals, *write* means to be able to send signals, and *execute* means to be able to execute the process as a subprocess.

Moreover, the UNIX superuser can access any (local) file regardless of the permissions the owner has granted. In effect, the superuser *owns* all objects on the system. Even in this case however, the interpretation of the rights is constrained. For example, the superuser cannot alter a directory using the system calls and commands that alter files. The superuser must use specific system calls and commands to alter the directory, for example by creating, renaming, and deleting files.

Although the "objects" involved in the access control matrix are normally thought of as files, devices, and processes, they could just as easily be messages sent between processes, or indeed systems themselves. Figure 2–2 shows an example access control matrix for three systems on a local area network (LAN). The rights correspond to various network protocols: *own* (the ability to add servers), *ftp* (the ability to access the system using the File Transfer Protocol, or FTP [1539]),

host names	*telegraph*	*nob*	*toadflax*
telegraph	own	ftp	ftp
nob		ftp, nfs, mail, own	ftp, nfs, mail
toadflax		ftp, mail	ftp, nfs, mail, own

Figure 2–2 Rights on a LAN. The set of rights is {ftp, mail, nfs, own}.

functions	*counter*	*inc_ctr*	*dec_ctr*	*manager*
inc_ctr	+			
dec_ctr	–			
manager		call	call	call

Figure 2–3 Rights in a program. The set of rights is {+, –, call}.

nfs (the ability to access file systems using the Network File System, or NFS, protocol [142]), and *mail* (the ability to send and receive mail using the Simple Mail Transfer Protocol, or SMTP [1068,1538]). The subject *telegraph* is a personal computer with an FTP client but no servers, so neither of the other systems can access it, but it can FTP to them. The subject *nob* can access *toadflax* and *nob* itself using an NFS client, and both systems will exchange mail with one another and can *ftp* to each other.

At the micro level, access control matrices can model programming language accesses; in this case, the objects are the variables and the subjects are the procedures (or modules). Consider a program in which events must be synchronized. A module provides functions for incrementing (*inc_ctr*) and decrementing (*dec_ctr*) a counter private to that module. The routine *manager* calls these functions. The access control matrix is shown in Figure 2–3. Note that "+" and "–" are the rights, representing the ability to add and subtract, respectively, and *call* is the ability to invoke a procedure. The routine *manager* can call itself; presumably, it is recursive.

In the examples above, entries in the access control matrix are rights. However, they could as easily have been functions that determined the set of rights at any particular state based on other data, such as a history of prior accesses, the time of day, the rights another subject has over the object, and so forth. A common form of such a function is a locking function used to enforce the Bernstein conditions,[2] so when a process is writing to a file, other processes cannot access the file; but once the writing is done, the processes can access the file once again.

[2]The Bernstein conditions ensure that data is consistent. They state that any number of readers may access a datum simultaneously, but if a writer is accessing the datum, no other writers or any reader can access the datum until the current writing is complete [177].