

# The foundations of a provably secure operating system (PSOS)

By Richard J. Feiertag and Peter G. Neumann

*SRI International*

Menlo Park, California

*Proceedings of the National Computer Conference*

*AFIPS Press, 1979, pages 329-334.*

## INTRODUCTION

PSOS has been designed according to a set of formal techniques embodying the SRI Hierarchical Development Methodology (HDM). HDM has been described elsewhere,<sup>1-3</sup> and thus is only summarized here. The influence of HDM on the security of PSOS is also discussed elsewhere.<sup>4</sup> In addition, Linden<sup>5</sup> gives a general discussion of the impact of structured design techniques on the security of operating systems (including capability systems).

HDM employs formally stated requirements, formal specifications defining the design of each module in a hierarchical collection of modules, and formal statements of the module interconnections. In the case of PSOS, there is a formal model describing the requirements of the basic protection mechanism and additional formal models of the requirements of various applications (e.g., Reference 6). HDM provides the formalism and the structure that make the formal verification of the system design and implementation possible and conceptually straightforward. This formal verification consists of formal proofs that specifications satisfy the desired requirements,<sup>6</sup> and subsequently that the actual programs for the system and its applications are consistent with those specifications.<sup>2</sup>

The design of PSOS has been formally specified using a SPECIFICATION and Assertion Language called SPECIAL.<sup>7</sup> These specifications define PSOS as a collection of about 20 hierarchically-organized modules. Each module typically is responsible for objects of a particular type defined by that module. From the user point of view, the most important modules are those for capabilities, for virtual memory segments, directories, user processes, and for creating user defined abstract objects. Some modules are to be implemented in software, some in firmware, and some in hardware—as dictated by the efficiency required.

Capabilities provide the protection mechanism for all such objects in PSOS, and are discussed in the next section. The subsequent sections of this paper summarize the development methodology used in PSOS, present the protection mechanism provided by PSOS capabilities, exhibit its properties, show its applicability in developing data and procedure abstractions, and contrast the PSOS approach with the kernel approach to achieving secure systems. There are many important issues relating to the use of capabilities in PSOS (and other computer systems) that are not presented in this paper. Many of these issues are discussed in the references cited here.

## PSOS CAPABILITIES

The concept of the capability has appeared in several other operating systems (e.g., References 8-13). Although capabilities are a fundamental part of the design of each of these systems, they all differ in the way they use and interpret capabilities. PSOS differs from its predecessors in its uniform use of capabilities throughout the system and in the simplicity and primitive nature of the basic capability mechanism.

Each object in PSOS can be accessed only upon presentation of an appropriate capability to a module responsible for that object. Capabilities can be neither forged nor altered. As a consequence, capabilities provide a controllable basis for implementing the operating system and its applications, as there is no other way of accessing an object other than by presenting an appropriate capability designating that object.

Each PSOS capability consists of two parts; a *unique identifier* (uid) and a set of *access rights* (represented as a Boolean array). By definition neither part is modifiable, once a capability is created.

- *Unique Identifiers*—PSOS generates only one original capability for each uid. Any number of copies can be made of a given capability for which a copy is to be made. Therefore, a procedure or task that creates a new capability with some uid knows that the only capabilities that can have that uid must have been copied either directly or indirectly from the original. In other words, the creator of a capability with a given uid is able to retain control over the distribution of capabilities with that uid.
- *Access Rights*—The set of access rights in a capability for an object is interpreted by the module responsible for that object to define what operations may be performed by using that capability. The interpretation of the access rights is constrained by a *monotonicity rule*, namely that the presence of a right is always more powerful than its absence. The interpretation of the access rights may differ for different objects, but the monotonicity rule must always apply.

The access rights for a segment capability (as interpreted by the segment manager) indicate whether that capability may be used to write information into the designated segment, to read that information, to call that segment as a procedure, and to delete that segment. In PSOS, a directory contains entries, each of which is a mapping from a symbolic object name to a capability. Each directory is accessed via a capability for that directory. For directories, the interpretation of access rights is done by the directory manager. The access rights for a directory capability indicate whether that capability may be used to add entries to the designated directory, to remove entries, and to use the capability contained in that entry.

A copy of a capability may be made, but the resulting capability cannot have any access rights that the original capability did not—as is seen from the following list of possible operations upon capabilities. (There are other access rights, meaningful to capabilities of all types, to be discussed under storage permissions.)

## THE PSOS PROTECTION MECHANISM

Capabilities provide the basis for a flexible protection mechanism, as follows:

### *Tagging of capabilities*

In PSOS, capabilities can be distinguished from other data because they are tagged throughout the system (i.e., in the processor, and in both primary and secondary memory) by means of a tag bit inaccessible to programs. Consequently, the hardware can enforce the nonforgeability and unalterability of capabilities.

### *Operations upon capabilities*

There are only two basic operations that involve actions upon capabilities (as opposed to actions based on capabilities, which is the normal mode of accessing objects), as follows.

`c = create_capability` (i.e., with a previously unused uid) having all access rights.

`cl = restrict_access(c, mask)` creates a capability with the same uid as the given capability `c` and with access rights that are the intersection of those of the given capability `c` and the given maximum (`mask`); i.e., it creates a possibly restricted copy.

### *Store permissions*

The second capability operation described above appears to permit unrestricted copying of capabilities. For certain types of security policies this unrestricted copying is too liberal. For example, one may wish to give the ability to access some object to a particular user but not permit that user to pass that ability on to other users. Because simplicity of the basic capability mechanism is extremely important to achieve the goals of PSOS, any means for restricting the propagation of capabilities should not add complexity to the capability mechanism.

A few access rights (only one is currently used by PSOS itself) are reserved as *store permissions*. This is the only burden placed on the capability mechanism. The interpretation of the store permissions is performed by the basic storage object manager of PSOS, namely the segment manager. Each segment in the system is designated as to whether or not it is capability store limited for each store permission. If a segment is capability store limited for a

particular store permission, then it can contain only capabilities that have that store permission. This restriction can be enforced by a simple check on all segment-modifying operations.

By properly choosing the segments that are capability store limited, some very useful restrictions on the propagation of capabilities can be achieved. The restriction used in PSOS is not allowing a process to pass certain capabilities to other processes or to place these capabilities in storage locations (e.g., a directory or interprocess communication channel) accessible to other processes. (Other restrictions are also possible using store permissions, such as restricting a capability to a subsystem or a particular invocation of a subsystem. For example, see Reference 1, page II-25.) More general means for restricting propagation of capabilities and for revoking the privilege granted by a capability can be implemented as subsystems of PSOS. The store permission mechanism has been selected as primitive in the system because it achieves the desired result with negligible additional complexity or cost.

## DATA AND PROCEDURE ABSTRACTIONS

PSOS consists of a collection of data and procedure abstractions constructed in a hierarchical fashion as shown in Table I. Each level in the hierarchy represents a collection of abstractions introduced at that level. Abstractions at higher (numbered) levels are implemented using abstract objects introduced at lower levels in the design. It is unimportant whether an abstraction is implemented in hardware, firmware, or software. It is reasonable that abstractions introduced at lower levels be implemented largely in hardware or firmware and that abstractions introduced at higher levels be implemented largely in software. However the demarcation between hardware and software is not established by the design, and it is quite possible that abstractions occurring throughout the system be implemented as hybrids, i.e., partially in hardware and partially in software.

Level	Abstractions
16	user request interpretation
15	user environments and name spaces
14	user input-output
13	procedure records
12	user processes and visible input-output
11	creation and deletion of user objects
10	directories
9	abstract object manager
8	segments and windows
7	pages
6	system processes and system input-output
5	primitive input-output
4	arithmetic and other basic procedures
3	clocks
2	interrupts
1	registers and other storage
0	capabilities

**TABLE I--PSOS Abstraction Hierarchy**

It is convenient to group the levels of Table I into generic categories as shown in Table II. The generic categories collect abstractions satisfying similar goals. At the base of the hierarchy is the capability mechanism, from which all other abstractions in the system are constructed. Above the basic capability mechanisms are all the physical resources of the system, e.g., primary and secondary storage, processors and input/output devices. From the physical resources are constructed the virtual resources. These virtual resources present a more convenient interface to the programmer than the physical resources, permit multiplexing of the physical resources in a manner largely invisible to the user, and allow the system to allocate the physical resources so as to maximize their efficient use. Next in the PSOS hierarchy comes the abstract object manager, providing the mechanism by which higher-level abstractions may be created. As will be discussed in detail, it is possible to construct higher-level abstractions based solely on the capability mechanism; however, the abstract object manager provides services that make construction of such abstractions easier. The top two categories in the generic hierarchy include community abstractions and

user-created abstractions. The community abstractions are intended to be used by a large group of users, e.g., by all the users at a particular site. Such abstractions may be simple utility routines such as a compiler, or may actually create and control access to new virtual resources such as directories. The user abstractions are those intended for use by a limited group of individuals.

Level	Abstractions	PSOS Levels
F	User abstractions	14-16
E	Community abstractions	10-13
D	Abstract object manager	9
C	Virtual resources	6-8
B	Physical resources	1-5
A	Capabilities	0

**TABLE II-- PSOS Generic Hierarchy**

Of the properties stated previously, there are two important ones that make PSOS capability particularly useful in the construction of abstract objects.

1. The capability serves as a *unique* name for an abstract object.
2. The capability is unforgeable.

This means that a capability can be used as a name (guaranteed to be unique) by which an abstract object can be referenced, and access to the object can be controlled by limiting the distribution of the capability.

In addition, there are several important pragmatic reasons why PSOS capabilities are useful as a naming and protection mechanism for supporting abstract objects.

1. The capability mechanism has a very simple implementation. This allows capabilities to be built into the system at the lowest level of abstraction, thus making capabilities available for the most primitive objects.
2. Capabilities are uniform in size, making them easy to manage.
3. The inclusion of access rights in capabilities permits efficient fine-grained control of access to objects.
4. Capabilities can be written into storage (including secondary storage) and retrieved from storage in the same manner as other data, and therefore have many of the properties of other data.

Capabilities serve as names or tokens for all objects of PSOS. It is because the basic capability mechanism is so simple in concept and in implementation that construction of the most primitive objects (e.g., input/output channels, processors, and primary memory) as well as the most complex system objects (e.g., directories and user processes) and user application objects (e.g., a data management system) is possible using capabilities. This promotes a high degree of uniformity throughout the system and eliminates the need for many special-purpose facilities.

Objects that have many properties and operations in common and are managed by a single program are said to have a common *type*; that program is called a *type manager*. The type manager implements operations on an abstract object in terms of operations on the more primitive objects used to represent the abstract object. The type manager must be able to determine which objects are part of the representation used to implement an abstract object denoted by a given capability. In other words, a type manager must be able to map the unique identifier of a given capability into capabilities for its representation objects. The capability mechanism of PSOS does not predispose a type manager to any particular implementation of this mapping. Different type managers will require diverse mapping algorithms, depending upon the number of abstract objects and representation objects they must manage, the desired efficiency of operations on the abstract object, the desired simplicity of the mapping algorithm, and numerous other factors. For example, the segment type manager uses a mapping algorithm that is in almost all cases extremely fast; however, the algorithm is quite complex, requiring implementation in both hardware and software. Extreme speed is essential to the operations of the segment type manager because the segment operations are useful very frequently (at least once on every instruction). The directory type manager uses a less speedy algorithm because fast access is less essential.

Although the capability mechanism of PSOS does not prescribe a particular mapping algorithm, the system does provide some assistance in managing abstract objects. The abstract object manager provides a set of operations by which type managers can associate capabilities for abstract object with the capabilities for their representation objects. The type manager can then retrieve the representation capabilities by presenting to the abstract object manager the abstract object capability. This is done in such a way that only the type manager program itself can







