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, [1-3] and thus is only summarized here. The influence of HDM on the security of PSOS I also discussed elsewhere.[4] In addition, Linden[5] 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.[6] and subsequently that the actual programs for the system and its applications are consistent with those specifications.[2]

The design of PSOS has been formally specified using a SPECIfication and Assertion Language called SPECIAL.[7] These specifications1 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 contract 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 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 dictionary contains entries, each of which is a mapping from a symbolic object name to a capability. Each directory is access 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_acess(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 usr but not permit that user to pas 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 hither-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 re 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 ever instruction). The directory type manager uses a less speedy algorithm because fast access is not 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

obtain the representation capabilities, and then only upon presentation of the abstract object capability. The abstract object manager performs the mapping from abstract object capabilities to representation object capabilities, some of the bookkeeping functions necessary to implement abstract objects, and some storage allocation. Although the abstract object manager is intended to be useful and appropriate for a wide variety of type managers and does make the programming of a type manager much easier, it is only a service and is not essential to the construction of type managers.

The capability mechanism itself could have been constructed with many of the facilities of the abstract object manager included. This would have resulted in a capability mechanism that would be more elaborate and—for some applications—more efficient and easier to use. This is the approach taken by other capability systems cited above. On the other hand, such a capability mechanism would have required a more complex implementation. More significantly, the capability mechanism could then not have been placed at the lowest level of abstraction in the system design, and some of the physical and virtual resources of the system could not have been implemented using capabilities—requiring a different means for reference. Although having several different naming schemes is possible (and common in most systems), it destroys the uniformity, conceptual simplicity, elegance, ease of use, and possibly the efficiency of the system. It is for this reason that PSOS has a very simple, but fully general, capability mechanism, and that programs enhancing the use of the capability mechanism can be introduced as extensions at higher levels of the design.

As noted above, there is no clearly delineated system boundary in PSOS. One would normally draw the system boundary at the interface to the community abstractions. However, all the programs that implement the community abstractions (such as directories or user processes) could be provided by users as user programs. The community programs have no special privilege other than claming resources at initialization by taking possession of certain capabilities. For example, the user-process type manager takes possession of the capabilities for certain system processes which it then multiplexes to create many user processes. If the system's user-process type manager did not claim all the available system processes, then it would be possible for a user to provide a different user-process type manager with the same or different facilities. Similarly, the abstract object manager has no special privilege at all. A user might program his own abstract object manager if he so desired.

The abstractions at or below any level in the design of Table I form a consistent and useful system. Clearly, the lower the level chosen as the "top level" of the system, the more primitive that system will be. If all of the physical resources (levels 1 through 5) are present, then the full PSOS could be reconstructed on the restricted system, but more likely, one would construct a somewhat different system. Thus, the PSOS design represents a family of systems. One can choose the level that provides the best set of resources to fulfill the needs of the desired system without having to include unnecessary facilities. Then one can augment this level with the new type managers to create abstract objects appropriate to the desired applications. Writing such "system" type managers requires no additional skill or privilege other than that required to write ordinary user programs. The distinction between a "system" program and a "user" program is thus indeed blurred.

PSOS RELATIONSHIP TO KERNELS

Several recent operating systems have been constructed using a "kernel" architecture. Such systems include the Kernelized Secure Operating System (KSOS) [14,15] and tow precursor systems developed at MITRE [16, 17] and UCLA.[18] The term *kernel* is used loosely in the literature, but for the purpose of this discussion a kernel is that part of the operating system that is both necessary and sufficient to satisfy certain requirements of the system. For example, if the essential requirement of a system is that it enforces a certain security policy, then that part of the system that enforces the security policy constitutes the kernel. By this definition, a kernel is meaningful only with respect to some requirement or some set of requirements. The kernel must contain all those parts of the system that pertain to meeting the requirements, i.e., there is no part of the system outside the kernel that can cause the system not to meet its requirements. Also, the kernel can contain only those parts of the system that are necessary to meet the requirements, i.e., the kernel should not contain anything that does not pertain to the meeting of the requirements. The reasoning behind kernel-based architectures is that since a kernel contains only that part of the system essential to meeting requirements, it can be small, compared to the system as a whole, and therefore has a better chance of being correct. One of the main advantages of the kernel approach is the clear statement of purpose of the system. Since a kernel is meaningful only with respect to some explicit requirements, these requirements serve as the statement of purpose of the system. The other main advantage is the enhanced probability of correct operation. Since the programs that are critical to the correct operation of the system are isolated in the kernel, a great deal of attention can be paid to getting this code right, and less attention can be paid to other system code that

may be important but is not critical. The relatively small size of the kernel significantly improves the chances of applying formal verification techniques to the programs in the kernel in a cost-effective manner, where applying these techniques to the entire system would be unwieldy.

There are, as one might expect, some disadvantages to the use of kernels. Kernels cannot be casually modified because, by definition, all the code in the kernel is essential to meeting the requirements of the system, and any modification is likely to cause the system to deviate from that which is required. One must take extreme care to be sure that a change in the kernel will not compromise its correct operation.

In order to be able to construct a small kernel, the requirements limit the applications for which the kernel is useful. For, example, if the requirement of a kernel is to enforce a particular security policy, then only applications requiring that policy can be reasonably implemented using that kernel. It is not possible to implement another security policy that is inconsistent with the given security policy.

Yet another of the major problems with the kernel approach is the difficulty of designing a system in such a way that those programs essential to meeting the requirements are isolated from the nonessential programs. Finally, experience with the systems mentioned above indicates that kernels are still quite large. Clearly the size of a kernel depends upon the requirements it is supposed to meet, but reasonable requirements tend to require a large part of the system to be part of the kernel. Large kernels do not enhance one's confidence in the correct operation of the system.

Consider, for example, the kernels of KSOS and of the MITRE system. The requirement of these kernels is that they enforce a multilevel security policy. Upon close examination of these systems, it is seen that what is labeled the "kernel" is not really the kernel at all, but is only part of the kernel. These systems have so-called *trusted processes*, namely programs that are internally able to violate the requirements, but whose external interface is consistent with the requirements. These trusted processes include programs for file system backup and retrieval, I/O spooling, and network interfaces. These programs are not labeled as part of the kernel because their function is in some sense peripheral to the main task of the system. However, their correct operation is as essential to meeting the requirements as any kernel program. If the system is to be prove correct, the programs that are used by the trusted processes must be formally verified. Inclusion of the code for the trusted processes into the kernel makes the resulting kernel much larger. This illustrates a difficulty in designing a small kernel.

It is a matter of judgment as to whether the advantages of the kernel approach outweigh the disadvantages. For the situation in which one has clearly defined, specific-overriding requirements for which a small kernel can be constructed, then the kernel approach is ideal.

PSOS is well suited to situations in which one wants to support many applications with different or conflicting requirements. Because PSOS is highly extensible and easily supports different type managers with strong control over access to objects and type managers, it makes possible the support of many different sets of requirements on one PSOS implementation. For example, several subsystems have been designed for PSOS that enforce different security constraints. A particular task could be constrained to have access to only one of these subsystems, but several tasks may be executing different subsystems simultaneously. In a sense, each of these subsystems can be viewed as a "kernel" for the tasks having access to them, but PSOS can support any number of such subsystems. Of course, one still has the problem of assuring the correctness of these subsystems and those parts of PSOS which the subsystems use. However, assuring the correctness of these subsystems on PSOS should be significantly easier than assuring the correctness of a stand-alone kernel, because each subsystem will be much smaller and simpler than it would be if it had to be implemented as a stand-alone system. The UCLA system [18] is an interesting case in that its requirements for security are very broad and general. The UCLA kernel attempts to be like PSOS in its ability to support a wide range of security policies simultaneously. However, the resulting requirement does not permit as wide a range of policies to be implemented, and the system design is not as uniform or elegant as the PSOS design.

SUMMARY

The capabilities of PSOS provide a flexible naming and protection mechanism that can be used to implement arbitrarily complex subsystems efficiently fulfilling a wide variety of requirements. The properties of PSOS that make this possible are summarized as follows.

1. The capability mechanism is extremely simple, with only two operations involving the creation of capabilities and none permitting the alteration of capabilities. There is no policy embedded in the mechanism.

2. The operations on capabilities can be completely controlled at the most primitive conceptual level of the system design and implemented in hardware. Capabilities are tagged and nonforgable, and the protection they provide is not by passable.
3. Capabilities and other PSOS facilities encourage strong modularity via the creation of data and procedure abstractions. Such abstractions are the basis of the design of the PSOS system itself and can be used equally well in application programming.
4. The capability mechanism can be used equally well for user programs. There are no special protection mechanism necessary to protect system programs.
5. The capability mechanism is fully general and can simultaneously support subsystems that implement arbitrary policies. Mechanisms for initialization, backup and recovery, and auditing for both PSOS and its subsystems can be constructed without subverting the protection mechanism.
6. The operations that can be performed on an object of a particular type are precisely those defined by the type manager for that object. The operations permitted upon the particular object designated by a given capability are limited by the access rights of the given capability.
7. If a user is in possession of only one capability for an object, and he wishes to confer some or all of the access rights to another user (or to another program), he may create and pass a new capability whose access rights are a subset of those of the original capability. There is no way in which an additional access right can be introduced. (Note, however, that type managers must consistently enforce the monotonicity of access rights. That is, the presence of the right must be more powerful than the absence of that right. This is guaranteed for system-defined object types, and must be assured by the type managers for other types.)
8. Propagation of capabilities can be restricted by use of capability store permissions. The passage of a capability to other users can be prevented by not including process store permission in that capabilities access rights.

Although no single commercially available computer has the facilities necessary to implement PSOS efficiently, each of the required facilities does exist on some computer. Therefore, the proper hardware support for PSOS can be implemented using established techniques. The formal techniques used to design PSOS make implementation straightforward 19 and make formal verification of correct operation possible. All of the advantages summarized here can make PSOS and subsystems implemented on PSOS far more secure and reliable than contemporary operating systems.

ACKNOWLEDGMENTS

The design of PSOS was accomplished by the close cooperation of several people. Outstanding among these are Larry Robinson who is primarily responsible for the development of HDM and who played a major role in the early design of the system, Karl Levitt who designed security related subsystems, and Bob Boyer who is responsible for the formal mathematical theory.

REFERENCES

1. Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt and L. Robinson, “A Provably Secure Operating System; the System, its applications, and Proofs.” SRI International, Menlo Park, California, February 1977.
2. Robinson, L., and K. N. Levitt, “Proof Techniques for Hierarchically Structured Programs,” *Communications of the ACM*, Vol. 20 No. 4, April 1977.
3. Robinson, L., and K. N. Levitt, P. G. Neumann and A. R. Saxena, “A Formal Methodology for the Design of Operating System Software,” in *Current Tends in Programming Methodology*, R. T. Yeh ed., Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersery, April 1977.
4. Neumann, P. G., “Computer system Security Evaluation,” *AFIPS Conf. Proc*., NCC 1978, Anaheim, California, January 1978, pp. 1087-1095.
5. Linden, T. A., “Operating System Structures to Support Security and reliable Software,” *Computing Surveys*, Vol. 8, No. 4, December 1976, pp. 409-445.
6. Feiertag, R. J., K. N. Levitt and L. Robinson, “Proving Multilevel Security of a System Design,” *Proc. ACM Sixth Symposium on Operating Systems Principles*, November 1977, pp. 57-65.

7. Roubine, O., and L. Robinson, *SPECIAL Reference manual*, SRI International, Menlo Park, California, January 1977.
8. Lampson, B. W., and H. E. Sturgis, "Reflections on an Operating System Design," *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp. 251-266.
9. Lampson, B. W., "Dynamic Protection Structures," Proc. 1969 *AFIPS Fall Joint Computer Conference*, Vol. 35, AFIPS Press, Montvale, New Jersey, 1969, pp. 27-38.
10. Sevcik, K. C., "Project SUE as a Learning Experience," Proc. AFIPS 1972 Fall Joint Computer Conference, Vol. 40, AFIPS Press, Montvale, New Jersey, 1972, pp. 571-578.
11. Wulf, W. A., et. al., "HYDRA: the Kernel of a Multiprocessor Operating System," *Communications of the ACM*, Vol. 17, No. 6, June 1974, pp. 337-345.
12. Needham, R., "Protection Systems and Protection Implementations," Proc. *1972 AFIPS Fall Joint Computer Conference*, Vol. 41, AFIPS Press, Montvale, New Jersey, 1972, pp. 571-578.
13. England, D. M., "Capability concept Mechanism and Structure in System 250," *Proc. IRIA International workshop on Protection in Operating Systems*, Institut de Recherche d'Informatique et de Automatique, France, 1974, pp. 63-82
14. McCauley, E. J., and P. Drongowski, "KSOS: Design of a Secure Operating System," NCC '79, New York, New York, June 1979.
15. Berson, T., and J. Barksdale, "KSOS: Development Methodology for a Secure Operating System," NCC '79, New York, New York, June 1979.
16. Millen, J. K., "Security Kernel Validation in Practice," *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp. 243-250.
17. Schiller, W. L., "The Design and Specification of a Security Kernel for the PDP-11/45," ESD-TR-75-69, The MITRE Corporation, Bedford, Massachusetts, March 1975.
18. Popek, G. J., and D. A. Farber, "A Model for Verification of Data Security in Operating Systems," *Communications of the ACM*, Vol. 21. No. 9, September 1978, pp. 737-749.
19. DeLashmutt, L. F., Jr., "Steps Toward a Provably Secure Operating System," Spring '79 COMPCOM, *Digest of Papers*, February-March 1979, pp. 40-43.