Security Management and Protection—A Personal Approach

Maurice V. Wilkes
Digital Equipment Corporation

In all security systems, a distinction must be drawn between security management and protection. Security management can be described by the well-known lattice model and implemented by means of an access control algorithm in the file manager. The effect is to erect a security fence around each user's file directory. For protection the simpler capability model suffices. In addition to the user file directories there is a system-wide capability index. This is not protected by a security fence and procedures that can make direct use of it may only be used in that part of the operating system—the security kernel—which is certified by a security inspector. At the present time the efficient implementation of a capability-based protection system presents difficulties and further research is called for.

INTRODUCTION

Computer security has a number of distinct aspects. This paper is concerned with the relatively narrow problem of safeguarding information held in a time-sharing system from being communicated to unauthorized persons through the action of a user working at a terminal. It will not be concerned with the safeguarding of information against accidental corruption or destruction, nor with the protection of the communication channels by encryption or otherwise. Similarly, outside the scope of this paper is discussion of ways of monitoring the actions of system programmers and others who have access to the computer room.

In all security systems, manual or mechanized, a distinction must be drawn between security management and protection. In the discussion of manual systems the emphasis is usually on the former. Documents are assigned security classifications and each user is accorded a security clearance. The well known lattice model of Bell and La Padula[1] provides a convenient means of formalizing such a security management system. A person wishing to obtain a copy of a classified document must show that his security clearance entitles him to do so. Having obtained possession of the document, he must then ensure that the information in it is properly protected. In manual systems the onus of protection rests entirely on the individual. It has two aspects that may be termed physical and moral, respectively. The recipient of the document must attend to its physical security, for example, by keeping it locked up when he is not using it and he has the moral duty of not communicating its contents, verbally or otherwise, to unauthorized parties. In a computer system, access to files can be controlled by an access control algorithm within the file manager. There is no difficulty in designing the algorithm so as to implement the lattice model. It is only necessary to associate with each file a list of the persons or classes of persons who are permitted to access it, together with a statement of the type of access to which they are entitled, namely read, write, or execute. When a user needs access to a file, the program operating on his behalf calls the file manager and quotes his security clearance.* If this is in order the file manager opens the file for him.

Once a file has been opened for a user, the physical responsibility for its protection rests with the operating system. This enforcement of protection by the operating system is by no means a trivial matter, and discussions of computer security are largely concerned with it.

Theoretical discussion of protection can be based on the capability model; this is a simpler model than the lattice model needed to discuss security management. A capability names a resource and confers certain access rights in respect of it. A non-computer example of a capability is a pass to a building that allows the user to enter certain specified rooms. Such a pass will only be issued after the security status of the applicant has been examined, but thereafter no further reference to his status is necessary; the mere possession of the pass is sufficient to confer the access rights. The same is true of a capability. For example, when a file is opened on behalf of a user, the process operating for him receives a capability for the file, and can then access it without further formality. It is important to observe that the capability is held by the user's process and that no representation of it is ever handled by the user himself.

CONFINEMENT

There is nothing to prevent an ill-disposed user from making a copy on paper of information in the computer and passing that copy to some third party, that is, committing a breach of security outside the computer. However, a secure computer system will prevent him from using the computer itself to commit such a breach of security, either deliberately or accidentally. In particular, the system will prevent a user from copying information from a file at one level of security classification into a file at a lower level of classification, either directly or after modification. This is known as *confinement*.[2, 3]

* Note that it is only people who have a security clearance. There is no provision, in the system being described here, for attaching security clearances to system programs or to physical devices.

CCC–0010–4620/84/0027–0003$02.50

A straightforward way of enforcing confinement is to design the access control algorithm so that a user is never given write access to files at a level of security different from that at which he is currently working. This implies that when logging in, a user must specify the level at which he proposes to work. If he logs in at the secret level, he may expect to have write access to certain files at that level, but no more than read-only access to files at the confidential or unclassified level. He will, of course, have no access at all to files at the top secret level. A user may log in at any level up to the highest that his security clearance will permit. If provision for downgrading information is required, then an exception to the above rule must be made in favour of a small class of users who are specifically authorized to perform the downgrading operation; these users will be allowed write access to files at a level below that at which they are logged in.

If the course outlined above is followed, the problem of confinement is solved at the security management level; it does not impact in any way on the problem of enforcing protection.

THE CERTIFICATION OF AN OPERATING SYSTEM

Certification implies that a competent person has inspected the code of the operating system in detail, and has felt able to issue a certificate to the effect that, in his opinion, the system is secure. In much the same way an auditor issues a certificate confirming the correctness of a set of accounts. Ideally, certification should be based on a formal proof of correctness, but significant theoretical and practical advances in the art of proving programs correct will be needed before this can be done for a complex system. For the foreseeable future, reliance must be placed on a systematic search for security loopholes. It is in this sense, rather than in that of formal proof, that the term security inspection is used in this paper. The security inspector concentrates on security, and only concerns himself with functional correctness to the extent that is necessary for him to confirm the security of the system. Thus, a system may have functional defects of a serious kind—for example, it may be incapable of displaying information on a user's terminal—and yet pass a security inspection with flying colours.

Operating systems are ordinarily written without any regard to certification and in such cases the task of certification, if attempted, would prove to be an impossible one. Efforts have, therefore, been directed towards structuring an operating system in such a way that certification for security is possible. What follows is essentially a discussion of this problem from the point of view of the capability model.

One approach to the design of a secure system is to identify a part of the operating system known as the *security kernel*. Provided that the security kernel is properly written, software bugs in those parts of the operating system that lie outside it, together with software bugs in users' programs, will be incapable of causing breaches of security. The advantage gained is that it is only the security kernel and not the entire operating system that has to be certified. Clearly, this advantage is only realized in practice if the security kernel is a small part of the operating system and this is a requirement that it is, in practice, not easy to satisfy.

OBJECT-BASED SYSTEMS

What are referred to as *objects* have various roots in computer development. One important root is formed by the classes that appeared in the SIMULA language. Another is formed by the protected procedures introduced by Dennis and Van Horn. Later the term abstract data type came into use in programming language circles. These developments led to the packages in ADA and the modules in MODULA 2. The above are all software developments and the protection associated with the objects—by whatever terms they are known—is enforced at compile time. Concurrently work has proceeded on unconventional processors in which capabilities are implemented directly in the hardware. These developments stem from the work of Fabry at Chicago in the 1960s. They include the Plessey System 250,[4] the CAP computer at Cambridge University,[5, 6, 10] and the Intel iAPX 432.[7] In all these systems, protection is enforced at run time. The Hydra operating system[8] is capability based, but the protection is enforced by software instead of by hardware. An unimplemented capability-based system has been described by Feiertag and Neumann.[5]

Objects fit well into the capability model. At the abstract level it is sufficient to say that a capability confers access rights to a named object, for example, to a segment, to a procedure, to a file directory, or possibly to a physical device. At a less abstract level it may be helpful to observe that a capability is, in the first instance, a capability for a segment. This segment may contain arbitrary data, and such a segment constitutes the simplest form of object. On the other hand, the data in the segment may have a special format and be designed to be interpreted in a particular way. If so there will be a type associated with the capability that specifies how the data are to be interpreted.

Among the items embedded in an object may be capabilities for other objects, and thus we have the concept of compound objects. An important form of compound object is a procedural object, or protected procedure to adopt one of the terms introduced above. A diagrammatic representation of simple and compound objects is given in Fig. 1.

DOMAINS OF PROTECTION

The set of capabilities available to a process at any time constitutes the *domain of protection* in which the process is running. A process may enter any protected procedure for which it has a capability in its domain of protection, and it may pass any such capability as an argument to the protected procedure. Thus at any time a process has accessible to it (1) capabilities that it has taken with it when it entered the protected procedure and (2) capabilities that are permanently associated with the protected procedure. There are no capabilities—of the kind sometimes called global capabilities—that are permanently associated with the process. Thus there is no automatic inheritance of rights by a procedure in virtue of the fact that a particular process happens to be running in it. The reader's attention is directed especially to this point, since it constitutes an important difference between the protection system described here and conventional

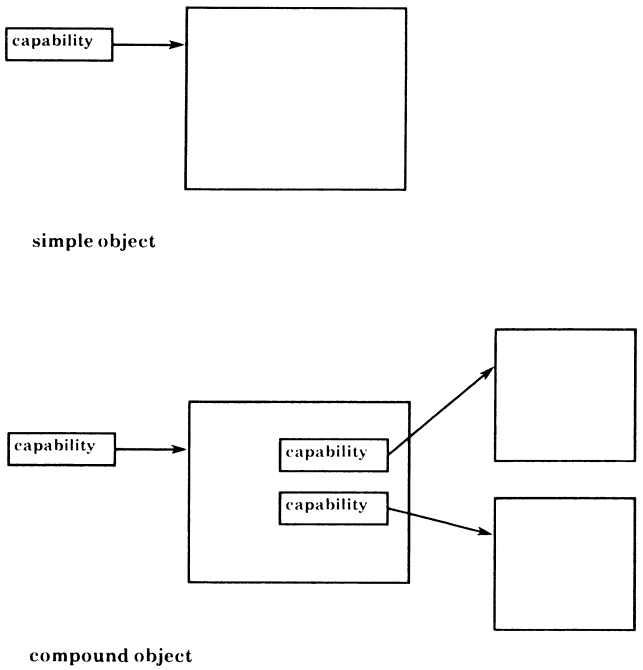

Figure 1

systems based on hierarchical scope rules. Systems which allow the automatic inheritance of rights are open to attack by *Trojan Horses*, that is, by sections of code deliberately embedded in proprietary compilers and similar software with the object of bringing about a breach of security.

The point just made may be illustrated by considering the way in which a proprietary and, therefore, suspect compiler could be safely run. As with all imported software, the compiler must be interfaced to the system under which it is to be run. In the present case, this would be done by encapsulating it in a protected procedure. The protected procedure would have no capabilities of its own, but would on being called receive capabilities for the segments it needed to do its work. At the minimum, it would receive two capabilities, namely one (read-only) for a segment containing the source code to be compiled and one (write-only) for a segment into which to place the compiled code. Since global capabilities are unknown in the system being discussed, there is no way in which the protected procedure in which the compiler is encapsulated can pass information to the outside world other than by putting it into a segment for which a capability is explicitly passed to it. The only form of Trojan Horse that could be effective would be one that had been inserted into the kernel by a dishonest system programmer and had escaped the detection of the security inspector.

There does, however, remain the danger that the compiler may store sensitive information within itself and incorporate this information in some way into the results of later compilations. The security inspector must, therefore, verify that a fresh copy of the compiler is read from the filing system whenever a new compilation is to be performed.

The avoidance of global capabilities, while giving the protection just mentioned against Trojan horses, does not prevent the exploitation of timing channels. In this respect, the system discussed here is no different from other systems. In the present state of knowledge, all that can be done is to exercise vigilance and to reduce to a minimum the bandwidth of any timing channels that can be identified.

THE SECURITY FENCE

In a capability object-based operating system it becomes necessary to distinguish between (1) objects which are known to a user (under alphanumeric names) and (2) objects which are embedded in other objects but which are not known to any user and for which no alphanumeric names exist. One is thus led to have indexes for objects at two levels. At one of these levels the indexes may be identified with User File Directories (UFDs). In terms of the capability model a UFD will, on being presented with the alphanumeric name of an object, deliver a capability for that object. At the other level there is a single index which I will refer to as the *capability index* and which contains an entry for all objects known to the system including those which appear in one or more UFDs. The capability index will, on being presented with a capability, deliver an indication of where in the system the corresponding object is to be found.

The view being put forward in this paper is that security management should be implemented at the level of the UFDs by means of an access control algorithm in the file manager. As explained above this algorithm takes account of the security level at which the user is logged in. It also takes account of any access restrictions that the owner of a file may impose on it. In other words, the algorithm implements both non-discretionary and discretionary controls. The effect is to put a security fence around each UFD.

Some of the objects whose names appear in a UFD will contain names of other objects in the UFD. These latter objects will be protected by the security fence around the UFD. Some of the objects whose names appear in the UFD may, however, have embedded in them actual capabilities as distinct from names; these may or may not be capabilities for objects that are represented in a UFD. Since a process using the object may refer to the capability index and find the location of the objects concerned, the security fences around the UFDs provide no protection. It follows that any object that has capabilities embedded in it must be regarded as being part of the security kernel and, as such, subject to the scrutiny of the security inspector. This leads to a clear definition of what is meant by the security kernel, namely, the set of objects that have capabilities embedded in them.

By way of illustration, Fig. 2 shows a UFD belonging to a particular user and containing entries for four objects. Also shown is an extract from the capability index, which is, of course, system wide. Names have an alphabetic form and capabilities are denoted by numbers. The capability index contains a pointer to the location of the object concerned—as explained above this is, in the first case a pointer to a segment either in high speed memory or on the disc. Some of the objects have references to other objects built into them; these references may take the form either of names or of capabilities. For the object ALPHA, the UFD gives the capability 1973 and the capability index indicates where the object is to be found

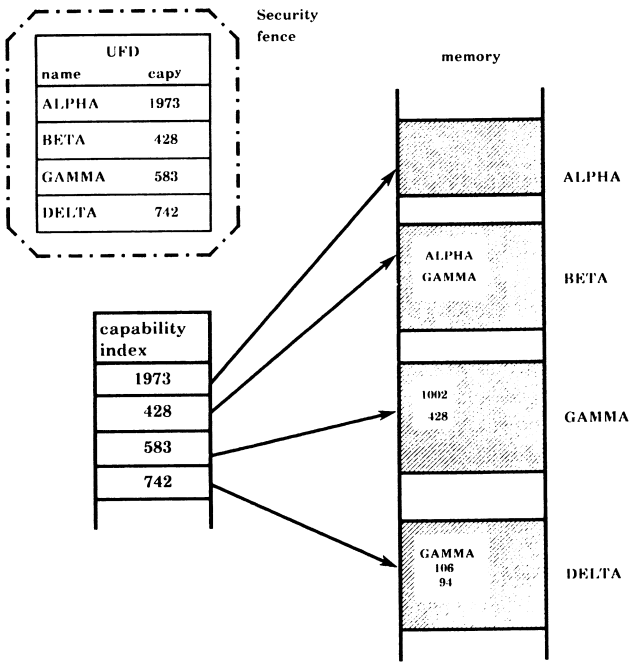

Figure 2

in high speed memory. This object has no references to other objects built into it, and may be used outside the security kernel. Object BETA contains the names of other objects but no actual capabilities; this may also be used outside the security kernel, since access to the objects whose names it contains can only be obtained by going through the security fence, and this will not be allowed unless the user on whose behalf the process is running has the necessary security clearance. Object GAMMA contains actual capabilities and, since access to the objects to which these refer can be obtained by going direct to the capability index—thus avoiding the security fence—the object GAMMA can only be used within the security kernel. The same applies to DELTA. There may be other entries in the capability index for objects that have nothing to do with the particular UFD shown. Some of the entries may relate to objects that are on the disc, rather than in high-speed memory.

LEAST PRIVILEGE

With the system described above it is possible to implement the *principle of least privilege* or a close approximation to it. This principle says, in capability terms, that at any time a process should have access only to those capabilities that it needs for its immediate purposes and no more; in other words the domain of protection should at all times be as small as possible. The advantage gained by following the principle of least privilege is that security inspection is made easier; indeed, it may be rendered practicable when otherwise it would be impracticable. The inspector is trying to prove a negative, namely that no breach of security can occur. This implies a search through the various possibilities that present themselves. Reducing the size of the domains of protection can reduce the scope of this search to one of manageable proportions.

AN OPEN QUESTION

It must not be concluded that, even if the principle of least privilege is followed, the task of the security inspector is an easy one. In particular, he may be faced with a problem in flow analysis. In capability terms, the problem may be explained as follows. The inspector must take account of the fact that, since capabilities can be passed to protected procedures as arguments, the domain of protection may differ on different occasions when the procedure is entered. This problem would not be a serious one if a protected procedure were unable to pass on to another protected procedure a capability that it had itself received as an argument. If the free passing of capabilities is allowed, then the security inspector may be presented with a difficult problem in flow analysis. For this reason, writers on capability systems have been led to propose the placing of restrictions on the passing of capabilities from one procedure to another.

It is clear that to prohibit entirely the passing on of capabilities received as arguments would put insuperable obstacles in the way of the writer of the operating system kernel. On the other hand, it is equally clear that some restrictions are desirable. For example, some capabilities might be marked to indicate that they could, in no circumstances, be passed on or that they could be passed on only a limited number of times.

CONCLUSION

The above discussion shows that the capability model provides a powerful conceptual tool with which to discuss protection as distinct from security management. Unfortunately, at the present time efficient implementation of a system described in terms of the capability model presents serious problems. Any direct implementation of capabilities in software terms leads to a low run-time efficiency. In particular, changing the domain of protection is a time-consuming operation. Such systems are, therefore, not in practice capable of supporting the frequent changes of protection domain that are demanded by the requirement that the domain should at all times be as small as possible.

Systems such as the Cambridge CAP in which capabilities are implemented in hardware are, in principle, capable of supporting without undue overhead very frequent changes in the domain of protection. However, such systems have, somewhat paradoxically, led to great software complexity and the reasons for this have been discussed by the author elsewhere.[9] It is possible that further research will lead to simpler systems of this type being developed. It is possible, also, that alternative forms of hardware support for rapid domain switching, that fall short of providing a general capability implementation, will be devised.

The scope rules found in current high-level languages do not lend themselves well to the implementation of the capability approach. This is unfortunate since the compiling of protection into the code once for all leads to a high run-time efficiency. Perhaps, as further developments in higher level languages take place, we may expect to see some convergence between the language approach to protection (compile-time enforcement) and the capability approach (run-time enforcement).

REFERENCES

1. D. E. Bell and L. J. La Padula, Secure computer systems: mathematical foundations and models. *Conference of the Society for General Systems Research,* Sacramento (1974). Mitre Corporation report M74-244.
2. B. W. Lampson, A note on the confinement problem. *Communications of the ACM* **16**, 613 (1973).
3. S. B. Lipner, A comment on the confinement problem. *Proceedings of the Fifth Symposium on Operating System Principles, ACM SIGOPS Review* **9** (5), 192 (1975).
4. D. M. England, Capability concept, mechanisms, and structure in System 250. *Protection in Operating Systems,* IRIA, Rocquencourt, France, p. 63 (1974).
5. R. J. Feiertag and P. G. Neumann, The foundations of a provably secure operating system. *AFIPS Conference Proceedings, NCC 79,* p. 329 (1979).
6. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its operating system,* North Holland, New York, 1979.
7. Intel Corporation. *Introduction to the iAPX 432 Architecture* Manual 171821-001 (1981).
8. W. A. Wulf, *et al. Hydra/C.mmp. An experimental computer system,* McGraw-Hill, New York (1981).
9. M. V. Wilkes, Hardware support for memory protection. *Proceedings of a Symposium on Architectural Support for Programming Languages and Operating Systems. Computer Architecture News* **10** (2), 107 (1982); also *SIGPLAN Notices* **17** (4), 107 (1982).
10. A. J. Herbert, A hardware supported protection architecture, In *Operating Systems* edited by D. Lanciaux. North Holland, Amsterdam (1979). Reprinted in Ref. 6.

Received May 1983