Capability-Based Protection in Distributed Operating Systems

Tanenbaum, A.S.; van Renesse, R.; Mullender, S.J.

published in
Symposium Certificering van Software
1984

document version
Publisher's PDF, also known as Version of record

Capability-Based Protection in Distributed Operating Systems

Andrew S. Tanenbaum *
Robbert van Renesse *
Sape J. Mullender †

ABSTRACT

Capability-based operating systems have traditionally required large, complex kernels to manage the use of capabilities. In our proposal, capability management is done entirely by user programs without giving up any of the protection aspects normally associated with capabilities. The basic idea is to use one-way functions and encryption to protect sensitive information.

1. Introduction

Soon, most office buildings will have a cable snaking through the cable ducts, with an outlet in each room into which users can plug their personal computers. The traditional approach to protection, a secure operating system in every machine to check permissions before carrying out a command, is not suitable for such an environment. It is too easy for a malicious user to replace the operating system in one of the network machines, or to replace a machine altogether by one without a secure operating system, to obtain confidential information illicitly.

New methods for protection must be devised, methods that do not require secure, trustworthy operating systems. This paper presents mechanisms, based on encryption. We shall show that they are equally powerful, and, in some cases, more versatile than existing protection schemes, implemented by a secure operating system. We propose to base the software design on a different conceptual model – the object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more "capabilities" [Dennis66] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of "abstract data type." This model is especially well-suited to a distributed system because in many cases an abstract data type can be implemented on one

* Dept. of Mathematics and Computer Science Vrije Universiteit Amsterdam, The Netherlands

† Centre for Mathematics & Computer Science Amsterdam

of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to directly inspect the representation of an abstract data type by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is frequently convenient to implement the object model in terms of clients (users) who send messages to services. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth generation computers.

As an example of the problems that must be solved, consider a file server. Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

2. Ports and Capabilities

2.1. Ports

Every service has one or more *ports* [Mullender82] to which client processes can send packets to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the public file service may refuse to read or write files for clients lacking account numbers, appropriate authorization, etc.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. The basic primitive operations offered by the system are PUT(PORT, MESSAGE) and GET(PORT, MESSAGE). Since everyone knows the port of the file server, as an example, how does one insure that malicious users do not execute GETs on the file server's port, in effect impersonating the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may GET from which port. We reject this strategy because some machines, *e.g.*, personal computers connected to larger multimodule systems may not be trustworthy, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software. First we will describe the hardware solution; later we will describe the software solution.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, P, and G, related by: $P = F(G)$, where F is a (publicly-known) one-way function [Wilkes68, Purdy74, Evans74] performed by the F-box. The one-way function has the property that given G it is a straightforward computation to find P, but that given P, finding G is so difficult that the only approach is to try every possible G to see which one produces P. If P and G contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find G given only P. Note that a one-way function differs from a cryptographic transformation in the sense that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

Using the one-way F-box, the server authentication can be handled in a simple way. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients, or, in the case of public servers, is published. When the server is ready to accept client requests, it does a GET(G). The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the process that did GET(G). To send a packet to the server, the client merely does PUT(P), which sends a packet containing P in a header field to the server. The F-box on the sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do GET(G). However, G is a well-kept secret, and is never transmitted on the network, Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request packet.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, S, and publishes $F(S)$. The F-box must be designed to work as follows. Each packet presented to the F-box contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only packets for which the corresponding GET has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-

known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple mechanism for implementing security and protection, but gives operating system designers considerable latitude for choosing various policies. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware with precluding many as-yet-unthought-of operating systems to be designed in the future.

2.2. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralized in a single monolithic "capability manager." In our proposed scheme, each object is managed by some service, which is a user (as opposed to kernel) program, and which understands the capabilities for its objects.

A capability typically consists of four fields:

1. The put-port of the service that manages the object
2. An Object Number meaningful only to the service managing the object
3. A Rights Field, which contains a 1 bit for each permitted operation
4. A Random Number for protecting each object

The basic model of how capabilities are used can be illustrated by a simple example: a client wishes to create a file using the file service, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a packet to the file service's put-port specifying that a file is to be created. The request might contain a file name, account number and similar attributes, depending on the exact nature of the file service. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a succession of data packets, each one containing the capability and some data. When each WRITE request arrived at the file server process, the server would normally use the object number contained in the capability as as index into its tables to locate the file.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between READ, WRITE, DELETE, and other operations that may be performed on objects.

However, it can easily be modified to provide that distinction. In the modified version, when a file (object) is created, the random number chosen and stored in the file table is used as an encryption/decryption key. The capability is built up by taking the Rights Field (*e.g.*, 8 bits), which is initially all 1s indicating that all operations are legal, and the Random Number Field (*e.g.*, 56 bits), which contains a known constant, say, 0, and treating them as a single number. This number is then encrypted by the key just stored in the file table, and the result put into the newly minted capability in the combined Rights-Random Field. When the capability is returned for use, the server uses the object number (not encrypted) to find the file table and hence the encryption/decryption key. If the result of decrypting the capability leads to the known constant in the Random Number Field, the capability is almost assuredly

valid, and the Rights Field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the Rights Field also affects the known constant. Exclusive or'ing a constant with the concatenated Rights and Random fields will not do.

When this modified protection system is used, the owner of the object can easily give an exact copy of the capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is harder. To accomplish this task, the process must send the capability back to the server along with a bit mask and a request to fabricate a new capability whose Rights Field is the Boolean-and of the Rights Field in the capability and the bit mask. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

This modified system works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We have devised yet another protection system that does not have this drawback. This third scheme requires the use of a set of N commutative one-way functions, $F_0, F_1, \cdots, F_{N-1}$, corresponding to the N rights present in the Rights Field. When an object is created, the server chooses a random number and puts it in both the file table and the Random Number Field, just as in the first scheme presented. It also sets all the Rights Field bits to 1.

A client can delete permission k from a capability by replacing the random number, R, with $F_k(R)$ and turning off the corresponding bit in the Rights Field. When a capability comes into the server to be used, the server fetches the original random number from the file table, looks at the Rights Field, and applies the functions corresponding to the deleted rights to it. If the result agrees with the number present in the capability, then the capability is accepted as genuine, otherwise it is rejected. Note that although the Rights Field is not encrypted, it is pointless for a client to tamper with it, since the server will detect than immediately. In theory at least, the Rights Field is not even needed, since the server could try all 2^N combinations of the functions to see if any worked. Its presence merely speeds up the checking. It should also be clear why the functions must be commutative – it does not matter in what order the bits in the Rights Field were turned off.

The organization of capabilities and objects discussed above has the interesting property that although no central record is kept of who has which capabilities, it is easy to retract existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in the file table. Obviously this operation must be protected with a bit in the Rights Field, but if it succeeds, all existing capabilities are instantly invalidated.

2.3. Protection without F-Boxes

Earlier we said that protection could also be achieved without F-boxes. It is slightly more complicated, since it uses both conventional and public-key encryption, but it is still quite usable. The basic idea underlying the method is the fact that in nearly all networks an intruder can forge nearly all parts of a packet being sent except the source address, which is supplied by the network interface hardware. To take advantage of this property, imagine a (possibly symmetric) conceptual matrix of conventional (*e.g.*, DES) encryption keys, with the rows being labeled by source machine and the columns by destination machine. Thus the matrix selects a unique key for encrypting the capabilities in any packet. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). With this arrangement, intruder I can easily capture packets from client C to server S, but attempts to "play them back" to the server will fail because the server will see the source machine as I

(assumed unforgeable) and use element M_{IS} as the decryption key instead of the correct M_{CS}. No matter what the intruder does, he cannot trick the server into using a decryption key that decrypts the capabilities to make sense, that is, to contain random numbers that agree with those stored in the file tables.

To avoid having to run the encryption/decryption algorithm frequently, all machines can maintain a hashed cache of capabilities that they have been using frequently. Clients will hash their caches on the unencrypted capabilities in the form of triples: (unencrypted capability, destination, encrypted capability), whereas servers will hash theirs in the form of triples: (encrypted capability, source, unencrypted capability).

To set up the matrix initially, the following procedure can be used. A public server, such as a file server, makes its put-port and a public encryption key known to the whole world. When a new machine joins the network (*e.g.*, after a crash or upon initial system boot), it sends a broadcast message announcing its presence. Suppose, for example, the file server has just come up, and must (1) prove that it is the file server to other processes, and (2) establish the conventional keys used for encrypting capabilities in both directions.

A client machine, C, which receives the broadcast from the alleged file server, F, picks a new conventional encryption key, K, for use in subsequent C to F traffic and sends it to F encrypted with F's public key. F then decrypts K and replies to C by sending a packet containing both K and a newly chosen conventional key to be used for reverse traffic. This packet is encrypted both with K itself and with the inverse of F's public key, so C can use K and F's public key to decrypt it. If the decrypted packet contains K, C can be sure that the other conventional key was indeed generated by owner of F's public key, thus convincing C that he is indeed talking to the file server. Both of the above-mentioned conditions have now been fulfilled, so normal communication can now take place. Note that the use of different conventional keys after each reboot make it impossible for an intruder to fool anyone by playing back old packets.

3. Summary

This paper has discussed a model for a fifth generation computer system architecture and its operating system. The operating system is based on the use of objects protected by sparse capabilities. Conclusions

The paper shows that it is possible and practical to build capability-based distributed operating systems, with capability management outside of the operating system kernel. Since the operating system itself is particularly vulnerable to attack in an office environment as we have described, our method is more secure than traditional protection schemes that must rely on the security of the operating system kernel.

Two methods have been presented for the implementation of authenticated communication between client and server processes, one using F-boxes, the other using a combination of public key encryption and conventional encryption techniques. Currently public key encryption is still expensive, both in terms of computational effort and storage requirements. The F-box mechanism is a good alternative until fast public key algorithms arrive. F-boxes can be put in the cable ducts, on the network interface cards, in integrated circuits that carry out the network protocol, or, if necessary, in the operating system kernel.

Capability management need not be carried out by a secure operating system: all operations on capabilities that are currently implemented in secure operating system kernels can also be carried out by choosing appropriate encryption techniques, with which client processes can be allowed to handle capabilities and carry out certain

(restricted) sets of operations on them.

References

[Dennis66]
Dennis, J. B. and Horn, E. C. van, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[Evans74]
Evans, A., Kantrowitz, W., and Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Mullender82]
Mullender, S.J. and Tanenbaum, A.S., "Protection and Resource Control in Distributed Operating Systems", IR-79 (to appear in Computer Networks), Vrije Universiteit, Amsterdam, August 1982.

[Purdy74]
Purdy, G. B., "A High Security Log-in Procedure," *Comm. ACM*, vol. 17, no. 8, pp.442-445, August 1974.

[Wilkes68]
Wilkes, M. V., *Time-Sharing Computer Systems*. New York:American Elsevier, 1968.