Protection and Resource Control in Distributed Operating Systems

Sape J. MULLENDER
Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

and

Andrew S. TANENBAUM
Department of Mathematics and Computer Science, Vrije Universiteit, Postbus 7161, 1007 MC Amsterdam, The Netherlands

Local networks often consist of a cable snaking through a building with sockets in each room into which users can plug their personal computers. Using such a network for building a coherent distributed or network operating system is difficult because the system administrators have no control over the users' machines – not the applications programs, not the system kernel, not even the choice of hardware. In this paper we discuss a general method to protect such systems against malicious users without placing any restrictions on the kinds of operating systems that can be constructed. Depending on the details of the hardware, either one-way functions or public key cryptography forms the basis for the protection. As an example of our method, we show how a traditional object-oriented capability system can be implemented on top of the basic protection mechanism, and how a service for accounting and resource control can be constructed.

Keywords: Capability, local network, operating system, protection.

Sape J. Mullender was born in Amsterdam. He has been at the Vrije Universiteit in Amsterdam from 1970 to 1983, first as a student of mathematics and computer science, later as a staff member, teaching programming and researching distributed operating systems. With prof. Tanenbaum he designed the *Amoeba* distributed operating system, which is now being implemented. Since 1984, Sape Mullender works at the Centre for Mathematics and Computer Science (CWI) in Amsterdam to lead a research group on computer networks and distributed operating systems.

North-Holland
Computer Networks 8 (1984) 421–432

1. Introduction

Local networks often consist of a cable snaking through a building with sockets in each room into which users can plug their personal computers, intelligent terminals, file servers, tape drivers, etc. Some of these machines may provide services for others, whereas others may simply be "consumers" rather than "producers" of service. Building a coherent, secure operating system in an environment in which the system administrators have no way to prevent malicious users from plugging uncontrollable personal spies into the network presents great difficulties. In this paper we discuss a general method for constructing secure operating systems under these circumstances without putting any undue restrictions on the kind of systems that can be built. We do not attempt in this paper to provide protection against forms of attack other than through the regular network interfaces. Other methods of protection, such as end-to-end encryption, are necessary to ward off passive or active wire tappers. This paper discusses a method whereby any person can plug any computer into the network, even if it is running a completely untrustworthy (or malicious) operating system, so it can use the resources offered by the system and offer services to the system, yet not harm the system's security.

We explicitly assume that a malicious user has complete control over his own machine, including

Andrew S. Tanenbaum was born in New York. He received an S.B. degree from M.I.T. and a Ph.D. from the University of California at Berkeley. His current research interests are distributed systems and compiler-writing technology. Dr. Tanenbaum is the author of two books, *Structed Computer Organization* and *Computer Networks*, as well as numerous scientific papers. He has been a consultant for Bell Laboratories and IBM, and has lectured in 10 countries. He is also the project leader for the Amsterdam Compiler Kit, which is used by universities and corporations around the world.

its operating system. Consequently, we reject any solution to the problem that requires users to only run machines that have user/kernel mode hardware, with an "officially approved" operating system running in kernel mode. It is too easy for anyone to just plug an arbitrary machine into the socket. However, we will consider solutions that require special (but very simple) hardware facilities on the host's interface chip, or in the interface card between the network cable and the users' machines, because the interfaces can be physically locked up in the cable duct within the wall, if need be, to prevent tampering. Finally, we will also consider solution not requiring any special hardware at all.

The key problem in a network of anonymous machines is authentication. Any user should be able to offer a service in such a way that no other (malicious) user can impersonate it. In effect, if three machines claim to be Johnson's New Improved File Server, how can a user machine tell which is the real one and which are imposters? Identifying services by having a directory server that maps service names onto physical socket numbers is not always feasible, for example, because some services may move around or be accessed via another network or via dial-up telephone lines.

In our proposal, authentication does not rely on any "physical socket numbers", nor does it introduce any vulnerable and undesirable centralised "authentication server". Instead, access to services is controlled by knowledge (or lack thereof) of certain critical information. The mechanism described below is incorporated in the Amoeba distributed operating system [13] being developed at the Vrije Universiteit.

2. Services, Ports, Signatures, and Capabilities

The basic paradigm of the proposed distributed system is the *service*. Every object within a certain service can only be accessed through one of the *servers*, the processes that constitute the service. Each server accepts requests in the form of *messages* from a *port* on a client process to a *port* on the server process. The server then sends a reply from its port to the client port. A service is analogous to an abstract data type in that the only way the client can manipulate the objects is via the set of allowed messages; he has no way of getting at the representation directly.

Every port has two names, one for clients to send to it, and one for servers to read from it: the *put-port*, and the *get-port*. The relation between them will be described later. Port names are (long) bit strings, chosen randomly from a sparse address space, so that without knowledge of its number, a client cannot send messages to a port; nor is it possible to "make up" port numbers and come up with a legal (existing) port number with any reasonable chance of success.

In the real world, *signatures* are used to uniquely identify who signed what. A signature must therefore have the following properties: only the "owner" of the signature must be able to sign documents with his signature, and others must be able to verify that the signature is genuine, and that it belong to its purported owner. We shall show (in Section 5) that our ports can also be used as signatures. As with ports, there are also pairs of signatures, *public signatures* and *private signatures*. The private signature is needed to sign a message, and the public signature can be used to check if it is genuine.

A *capability* in our proposed protection scheme is a bit string that gives the bearer permission to use some services, or more specifically, to perform some set of operations on some object (e.g. a file) managed by a certain server. Such a capability consists generally of four parts, as shown in Fig. 1. The *Server* field of a capability is the put-port for the service, the *Object* field, serves as an index in a table of objects, maintained by the server, and serves to identify the object, the *Rights* field indicates which operations may be requested with this capability, and the *Random* field makes the capability sparse. We refer to Section 7 for a further discussion of capabilities.

In a centralised system with a protected operating system kernel, capabilities can be kept in the kernel; in a distributed system of the kind discussed earlier, the kernels in the user machines cannot be trusted, so another protection mechanism is needed. We have chosen to use the sparseness of the port space and capability space as our protection mechanism. The port name space has been made sparse, and port names are chosen randomly, so that without knowledge of a port name a process cannot access the service behind that port.

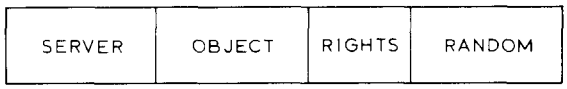

Fig. 1. Lay out of a typical capability.

Given the name of a service (an ASCII string, say) *directory servers* can be used to look up the ports for that service, just like a telephone directory is used to find a person's phone number. The directory service can have a tree structure, so that every user can have a private directory to keep capabilities. If a user does not divulge the capability for his private directory, he can keep everything in it secret. For added protection, clients can encrypt ports before trusting them to the directory service. Note that directory service need not be a service offered by "The System". The directory service does not need any privileges, nor need it be special in any sense.

Upon login, a *login server* can authenticate a user (check the password), and subsequently give the capability of the *home directory* to the command interpreter. From this directory a user can easily obtain the necessary capabilities.

Public services can use a well-known public directory to store the ports that give access to them. A private service can make itself known to selected clients by putting its put-port in their directories (using a public write-only capability).

3. The User Interface

The International Standards Organisation (ISO) has proposed a model for the communication between computers. This Open Systems Interconnection model (OSI) [1] consists of seven layers of protocol, each layer performing a well-defined function.

Most high-speed local networks are broadcast networks; that is, each packet passes every host and is taken off the broadcast medium by the intended recipient. The *network layer*, which is responsible for routing and congestion control is therefore often not present in these local area networks.

Our proposal – to use capabilities (ports) as addresses – can be viewed as a proposal for another kind of network layer. Our new network layer is responsible, by matching up get-ports and put-ports, for getting messages to their destination. We intend to show that this network layer can be made secure under two different assumptions. First, we shall assume that the network layer functions reside in a small hardware interface which is inserted between the host and the network in such a way that the interface cannot be tampered with. Later,we shall drop this assumption and show that a solution is still possible using public key encryption.

The message passing mechanism is implemented by the *port layer*, which, in turn, uses lower protocol layers (data link layer and physical layer as in the ISO OSI proposal). The port layer may be implemented as a set of system calls, or as a set of subroutines in user space, possibly in read-only memory. User programs can directly interface to the port layer, or via a set of library routines, implementing higher layers of the OSI protocols, or simulating an operating system environment. The interface is simple, two calls suffice for sending and receiving messages, and these calls are the same for clients and servers:

put (**var** putport, srcport, signature: PORT;
var buffer: MESSAGE);
get (**var** getport, srcport, signature: PORT;
var buffer: MESSAGE);

The first three parameters are a local, and a remote port, and a signature. The last parameter is either a buffer of data to be sent, or a buffer to put received data in. There is another call for servers, for the creation of ports:

makeport (**var** getport, putport: PORT);

On top of this layer, a library layer can be built with little effort, which combines the ports in requests and replies with the *object*, *rights*, and *random* fields of Fig. 1 to form capabilities for objects. To a client, interfacing to this set of library routines, sending requests to a server is no different conceptually, from sending requests directly to the associated object.

4. Protection Using One-Way Functions

In this section we shall discuss protection in networks in which a (small) hardware interface has been inserted between the user machines and the cable; in the following one we will extend the idea

to networks lacking such interfaces. Figure 2 gives an example of the kind of system we are referring to here. Between each user machine and the shared cable is a simple hardware device, the *interface*, to be described below. In many local networks, each user machine has a logic board inserted into its backplane to provide contention control and other functions. Our interface could easily be included on such a board. Alternatively, it could be inside the wall, between the socket and the cable, as mentioned earlier. In any case, we assume the interface is tamperproof. Furthermore, we assume that each interface can monitor all the traffic on the cable, selecting out messages intended for it.

The protection scheme envisioned here is based on one-way ciphers [4,9,14]. The designers of the interface logic must choose a function, F, such that given $F(x)$ it is very difficult to deduce x. The function must be carefully chosen, because the function itself will be publicly known, thus enabling potential intruders to experiment with it at great length.

To achieve the necessary protection, the get-port and put-port associated with each service will be related by the formula:

$$put\text{-}port = F(get\text{-}port).$$

When a process wants to create a new service, it calls *makeport*, which picks a number chosen randomly from a large (i.e. sparse) address space, and uses it as the get-port. It then computes the put-port using the one-way function. The put-port can be safely given to (selected) clients to use for sending messages to the server.

To receive messages, the server does a *get* on the get-port. In effect, it passes the get-port to the interface, which immediately applies F to it to compute the corresponding put-port. This put-port is then stored in a little table inside the interface. As each message passes by on the cable, the interface compares the destination port contained in it to the put-ports stored in its internal table. If a match is found, the message is copied to the interface and then passed to the user machine.

To see that this system works, note that the only way for the intruder in Fig. 2 to receive a message intended for the server is to do a *get* on the server's get-port, a number that the server keeps secret, and which cannot be derived from the publicly known put-port. Furthermore, due to the sparseness of the port space, picking get-ports

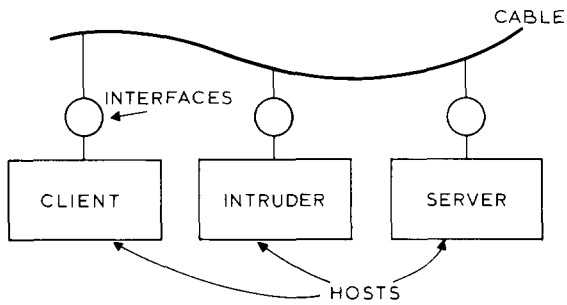

Fig. 2. The network model, using a network interface.

at random and hoping to catch an occasional message is unlikely to be productive, especially if the interface only has space to store a limited number of ports to scan for. The port to be used by the server to reply to the client can be safely included in the client-to-server message, because the intruder has no way of obtaining this message.

As an aside, it is noteworthy that servers can migrate around the network easily in this system, since messages are directed to logical ports, not physical machine or socket numbers.

The one-way function mechanism is also used to send *signed* messages. When a *put* is done, the message passes through one of the interfaces, which applies F to the *srcport* and the *signature* fields of the message. A process can thus select a *private signature* (e.g. by a call to *makeport*). The *public signature*, the encrypted form of the private signature, can be given to other processes which can use it to check the identity of the sender of messages. The process in possession of the secret private signature is the only process capable of sending messages with the encrypted public signature. Note that signatures are just ports, although they are not necessarily used as such.

Not only is the signature field encrypted on transmitted messages, but the srcport field also. This puts no undue restriction on communicating processes, because they need the get-port version of the srcport anyway in order to receive replies, and it prevents sequences of messages such as the following: process X sends a message to server A, with server B's put-port as the srcport (both ports are public and the srcport is not encrypted); server A may or may not make sense of the message, but in any case it will send a reply to B, because A assumes B is the sender; B will receive the message, assume it is a (probably illegal) request and

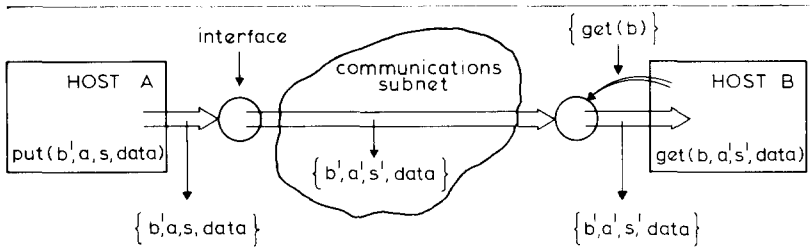

Fig. 3. A message, containing a destination port b', a source port, a, and a signature port s, as it passes from host A to host B. Get-ports are indicated by a, b and s, put-ports by a', b' and s'.

react by sending a reply back to A, which, in turn, treats it as a request, and so on. Encryption of the srcport prevents such and other pranks.

The whole process of sending messages is illustrated in Fig. 3, where we see a process on host A transmitting a message to a process on host B. We assume that B does a *get* on port b, and A does a *put* on port b', where $b' = F(b)$. B's *get* is passed to the interface, which applies F to b, and waits until an appropriate message comes by. A uses srcport a, and signature s in its *put* call. The message from A, after it passes through the interface into the communication subnet, contains b', a', s', and, when it is delivered to B after having passed B's interface, it contains b, a', and s'.

Our basic idea is also applicable to networks other than that of Fig. 2. As an example, consider a network in which the routing is done using physical machine numbers rather than the port matching with associative memory system just described. When a *put* is done, the interface broadcasts a "locate" message to all the other interfaces, asking "Do you know where this put-port is located?" After the replies have come back, the sending interface knows which physical address to use. For efficiency, the interface could maintain a cache of recently used ports and their locations, to avoid repeated lookups. An elegant solution to the problem of locating ports in store-and-forward networks is described in [7].

Another kind of network to which the one-way function idea is easily adapted is a star-shaped network, where each socket has a dedicated wire to a central switching machine, for example, the company PABX (Private Automatic Branch Exchange), which many office buildings use both for their in-house telephone system and for data. In this configuration, the functions previously performed in the interfaces are now performed by the switch, and the tables of put-ports are also located there. In this case we must assume that each host has a private and secure communication channel to the switch, and that the switch itself can be completely trusted. Similarly, in a store-and-forward packet switching network, the packet switches can process the *puts* and *gets* and locate ports by exchanging information among themselves, as described above.

Finally, if all the machines in a network do have protected operating system kernels, the interface function can be located in the kernel. The *get* and *put* commands then become system calls. If port locations is needed, the kernels can exchange suitable messages.

It is crucial for the one-way cipher protection mechanism that the interfaces cannot be circumvented. If an intruder can get around the interface he can receive all the messages that pass through the network, and generate messages purporting to be sent by any service. Ideally the interface is built into the network interface chip. The only way to tamper with it then, is not to use it and to build one's own interface. In many environments (office, university) this is not likely to happen because of the cost and effort involved. In a building where the interfaces are locked up in the cable duct in the wall our protection method is reduced to physically protecting the interfaces. Intruders communicating through an interface cannot circumvent the protection mechanism unless they succeed in stealing secret ports.

5. Protection Using Public-Key Cryptography

Now we will present the most general solution to the protection problem, one that requires neither a protected kernel nor any special interface hardware.

In an insecure network environment encryption of messages and ports is needed to send informa-

tion securely. We assume intruders can listen to other process' messages passing by, and even modify, intercept or replay messages. All this may affect the performance of the network, but it must not affect the security.

The method we shall describe here enables processes to communicate securely. It uses a public key cryptosystem [2] for the transmission of messages between client and server. Every service uses a key pair, the public key to be used for the encryption of messages to it, and the private key for decryption. The *makeport* primitive in the port layer is used to create port pairs, each port consisting of a port-identifier (which is used solely for identification of the port, and need not be kept secret), and an encryption key (put-port) or a decryption key (get-port). Messages to the server are encrypted and decrypted with these keys, and headed by the port-id in plaintext, so a process can find out which key to use.

The port-id need only be large enough that no two servers accidentally choose the same port name; however, even if they do, no harm is done – only the "real" server can decrypt messages addressed to it.

It is not necessary to encrypt the entire message, nor is it necessary to encrypt every message using public key encryption. If the information travelling between client and server is not secret, it is sufficient to encrypt only the header (including the reply port), and – to prevent intruders from modifying the plaintext contents – a checksum of the complete message. This checksum is also needed when whole messages are encrypted, because it guards against modification. The checksum algorithm must be carefully chosen, or it will be possible to generate modifications of a message that yield the same checksum. Protection against replay can be obtained by using sequence numbers in the traffic between client and server, a necessity in an unreliable network anyway. Public key encryption is still computationally expensive. There may be public key hardware that can encrypt and decrypt at network speeds in the future, but while fast public key is not available, it can be used to set up a secure channel using conventional cryptography.

Requests from client to server are encrypted, using the key in the put-port for the services. The replies from the server to the client are encrypted in exactly the same way, using the client's put-port, thus preventing other processes from decrypting replies, even if they have the put-port for the service.

6. Object-Oriented Protection

Using our protection mechanism as a basis, it is easy to build a traditional object-oriented capability-based protection system on top of it. Whenever a client asks a server to create a new object, the server returns the object's capability (see Fig. 1) in the return message. The *Server* field is the put-port used to send messages to the server managing the object. The *Object* field is just an index into the server's tables, to allow it to locate the object quickly. The *Rights* field, which may or may not be needed, is a bit map telling which operations are allowed on the object. The *Random* field provides the protection.

Before looking at the general case, consider an interesting special case – no *Rights* field, that is, the possessor of a capability can always perform all operations. To perform any operation, the client sends a message to the server's put-port, with the capability and the desired operation specified in the message. The server uses the *Object* field to index into its tables to find the entry for the object in question. If the random number stored there matches the one in the capability, the operation is allowed, otherwise it is not.

To provide for capabilities with partial rights, a slight variation is used. When the object is created, a random number is chosen and stored in the server's object table. A capability with the correct rights and a known constant (e.g. 0) in the *Random* field is then constructed. Finally, the *Rights* and *Random* fields are treated as a unit and encrypted, using the newly chosen random number as key. The resulting capability is sent back to the client.

Later, when the client tries to use the capability, the server uses the *Object* field to find the key and decrypt the *Rights* and *Random* fields. If the known constant appears in the plaintext *Random* field, the capability is apparently genuine, and the *Rights* bits can be believed. For this method to work, the encryption function must mix the two fields thoroughly, so changing any bit in either one will cause the decryption to fail.

This system has some properties slightly different from conventional capability systems. To start

with, it is perfectly safe to allow users to manipulate capabilities directly within their own address spaces, because any attempts to tamper with the bits will easily be detected. Second, clients can give duplicates of their capabilities away at will. However, clients can also give away subsets of their rights by simply asking the server to generate a new capability with a subset of the *Rights* bits on. (This operation could itself be protected by a *Rights* bit.)

But the most striking difference from conventional capability systems is the ease with which an authorised client can "take back" all existing capabilities, without the need for indirect objects, back pointers and other complicated machinery (see e.g. [5]). All the client need do is ask the server to change the random number in the server's object table and return a capability encrypted with the new random number. This operation, which would normally be protected by a bit in the *Rights* field, instantly invalidates all outstanding capabilities to the object.

When this method of manipulating the *Rights* bits is used, a client who wants to pass a capability with a reduced set of rights to another process must ask the server to generate the new capability for him. We have discovered an alternative scheme that does not incur this overhead. It works as follows. Suppose a capability has n rights, numbered 1 to n. Choose n one-way functions, $G_1, \ldots, G_n$ that are commutative, that is,

$$G_i G_j(R) = G_j G_i(R) \quad \text{for all functions } G_i \text{ and } G_j.$$

Let R be the *Random* field of a capability generated by a server as before. The initial capability has all the rights on.

To create a new capability with right k off, replace R by $G_k(R)$. This new capability can be further restricted by applying another G, and so on. Since the G's commute, the order in which the functions are applied is irrelevant. When the capability has been sufficiently restricted, the *Rights* field is loaded with the unencrypted bit mask.

When the capability is presented to the server, the server applies the G's indicated by the *Rights* field to the random number stored in the object. If the result agrees with the *Random* field in the capability, the capability is valid and the *Rights* field is genuine. Note that the presence of the mask in the *Rights* field is merely for efficiency; in theory, the server could try all 2^n combinations of the functions to see if any matched. If a client modifies the *Rights* field, the server will simply reject the capability as invalid.

7. Comparison with Other Protection Schemes

We have now shown how protection can be achieved in a distributed or network operating system using one-way functions or, if need be, public-key cryptography. We have also demonstrated how this mechanism can be used as basis for building a distributed capability-based operating system. In this section we compare our approach to protection in computer networks to other proposals.

Among the ideas that have been suggested are centralised authentication servers, passwords, access control lists, and public-key cryptography. Although a centralised authentication server [8] has some merit, we feel that it is not desirable to introduce a powerful, centralised, and therefore vulnerable component, into a distributed system if it can possibly be avoided. Our interface scheme shows how this goal can be achieved without any centralised components at all.

Another approach frequently suggested for networks of highly autonomous, multiprogrammed mainframes or large minicomputers is to have each remote request accompanied by the password of the requesting user [11]. In effect, the user logs onto the remote machine, and thereafter is treated as a local user. In a controlled environment, in which all machines on the network are assumed to be completely trustworthy, this method may work, but in the case we have been considering – anybody can plug an arbitrary machine into the network at will – sending passwords in plaintext obviously will not work. Encrypting them helps, but introduces a host of problems involving key distribution and authentication.

Donnelley and Fletcher [3] suggest using access control lists as a protection mechanism. To do so, however, assumes that a foolproof way exists to authenticate users. They assume that the network takes care of this somehow, perhaps by permanently associating each physical socket number on the network with a specific user. If this assumption is valid, their method works, but if the whole building is wired and people can carry portable computers from office to office at will, as we have assumed, another method is needed.

Donnelley and Fletcher have also proposed a protection mechanism using public-key cryptography [3]. In this proposal, a server encrypts capabilities sent to clients using its own private key. Since only it knows the private key, intruders cannot forge valid capabilities. Capabilities are distributed to clients by encrypting them with the clients' public keys.

Donnelley and Fletcher are concerned about the possibility of intruders illicitly obtaining data from inside clients' memories. For this reason they ensure that capabilities are always encrypted with the client's public key, even inside the client's own memory. Even if one believes that such measures are necessary (which we do not), there remains the danger that an intruder clever neough to acquire an encrypted capability from inside a process' address space will also be clever enough to acquire the decryption key, which must be stored somewhere nearby, because it is needed every time a capability is used. For some reason, they assume servers are immune to theft, since inside them capabilities are stored in plaintext.

The problem of capability theft aside, this method has several other drawbacks. To start with, public-key encryption is more expensive than a one-way function, which is why we use encryption only if the interface method (in hardware of software) is not applicable. Another advantage of the interface method is that, when implemented in hardware, at least, it integrates addressing and protection in a simple way by allowing messages to be sent to ports rather than to machines. Our capability scheme also provides mechanisms for associating individual access rights with capabilities, as well as a way to retract outstanding capabilities.

Another point is that one-way functions are easier to find than cryptographic systems, because the former are a superset of the latter. This property is inherent, because one-way functions do not require the existence of an inverse function, whereas encryption functions do. With a larger "space" to choose functions from, it should be simpler to find a suitable one-way function than a suitable cryptographic transformation.

Finally, our method is more robust than that of Donnelley and Fletcher. In theirs, if a server's secret key is ever compromised, it must be changed immediately, thus invalidating all existing capabilities to objects managed by that server. It is not clear how one can recover from this situation. The server cannot just issue a new capability to anyone presenting an old one, because it cannot distinguish genuine ones from those generated with the compromised key. In contrast, if the get-port of one of our servers is compromised, it too must choose a new one (and publish the new put-port), but the objects it controls are still protected by their individual random numbers. The only potential harm is that prior to the discovery of the compromise, an intruder may have improperly intercepted and processed some requests from clients. Fortunately, even with his knowledge of the server's get-port, the intruder cannot access existing objects. In the event that the random number in an object is inadvertently disclosed, only that one object is comprmised, thus greatly limiting the potential dmage.

8. Efficiency Considerations

Building a network interface as we have described is straightforward. Each interface needs a processor (or hardwired logic) to compute the one-way function. In addition it needs a table to store the put-ports corresponding to the gets that have been issued locally. If a put-port occupies, say, 6 bytes, 1000 of the, for example, will require 6K of memory. To allow both rapid lookup and deletion of a port after a successful *get*, the put-port should be hashed into a, say, 10-bit number giving the index into a 1K hash table. A table entry has 6 bytes for the port and 2 bytes for a pointer to an overflow chain in a second table. The main table has 8K bytes, and the overflow table needs another 8K bytes in the worst case (all ports hash to the same 10-bit number). This amount of memory (16K bytes) costs less than $100, and the price is falling rapidly.

When the start of a message is detected on the cable, the interface should start reading it into an internal buffer. As soon as the header is available, the hash code of the message's destination port can be computed. While the data part of the message is pouring in, the interface can look up the hash code in the port table. If a hit is found, the message is kept and passed on to the computer, otherwise it is just ignored.

The efficiency of one-way ciphers as the method to obtain secure communication between processes

far exceeds the efficiency of public-key encrypted messages. In contrast to public-key schemes, generating keys (ports) in the one-way cipher method is cheap: the cost of generating a port-size random number. Any good random number generator will do. Computing a one-way cipher function on a 64-bit get-port typically takes 1 to 10 milliseconds, depending on the hardware, and the algorithm used. Encrypting the header and the checksum of a packet, say a 512-bit quantity, with existing public key algoritms will take from 100 milliseconds up to perhaps 10 seconds. Additionally, public key encryption must be done twice per message, once to encrypt the message, and once to decrypt it again. When one-way ciphers can be used, the encryption need only be done once per *get* operation; if the interface is clever and keeps a cache of recently encountered get-port/put-port combinations, it need not even be done that often.

The size of a port, when one-way ciphers are used, would typically be between 48 and 128 bits. With a 48-bit port, and 1000 different services, only 1 in 10^{11} ports are used. Assuming a one-way ciphers is used with a degeneracy of less than 200 [9], one try in 10^9 will yield a legal get-port for one of the put-ports. A brute force effort to crack the one-way cipher can take as few as 10^6 seconds, assuming an enciphering time of 1 msec. This is barely acceptable. If the degeneracy can be lowered, or more bits are used, the cracking time becomes much longer. This issue is discussed in [9].

The keys in existing public key cryptosystems must be large [6,10]. Rivest, Shamir and Adleman's algorithm, for example, needs keys of about 512 bits to withstand prolonged attacks. In contrast to the one-way cipher scheme, and existing capability systems, these capabilities require considerable space as well as long encryption and decryption times. We must note, however, that any system without any protection from an operating system and with an insecure communication system, needs encryption to transmit messages securely anyway.

Fast local networks can transmit packets at a rate of one every few milliseconds. A good protection mechanism should work at the same rate, at least. The one-way cipher mechanism meets this requirement, especially if it becomes possible to ue "one-way cipher hardware" in the future. The public key scheme gets nowhere near this requirement yet; at the moment fast public key algorithms do not yet exist. With current public key algorithms, software encryption times of 100 msec to 10 sec per message should be expected. In the forseeable future we shall probably see reasonable fast "public key chips", but it is doubtful that public-key encryption will ever be as fast as applying a one-way function, because the latter is inherently less stringent.

9. Resource Control

In this section a service is discussed that can be used for several forms of accounting and resource control. For each user and service, the *Bank Server* maintains *Bank Accounts*, containing amounts of *Virtual Money* (because of the resemblance to "real" money) in one or more *currencies*. Service is measured in Virtual Money: Disk blocks, cpu seconds, phototypesetter pages, and database queries are examples of services that can be offered by the system, in return for an amount of Virtual Money in one or more currencies.

The Bank Service, as the manager of Virtual Money, handles requests from clients and servers to transfer amounts of Virtual Money from one account to another. Clients pay for services rendered by servers in Virtual Money, which is used to maintain resource quotas, to do accounting, to schedule resources among a group of users, etc. Before we go into the possible policies for resource control we shall look at the structure and mechanisms of the Bank Server.

We must make some assumptions about who trusts whom in our system. Ideally, of course, one process need not trust any other process. Systems where this is the case, have been discussed in the literature [10,8], and require public-key encryption and storage of encrypted messages as proof of having been received. Although the methods presented here are sound, we believe that they are not yet practical. Public-key encryption requires very large keys, it is slow, and requires signed messages to be retained, because they may later be required as proof of having received them. It is therefore hard to realise public-key encryption without a trusted file system [12].

We assume the Bank Server is trusted by its clients; that is, the Bank Server's clients believe the Bank Server will stick to the rules on accounting matters. As a trusted third party, the Bank Server

can mediate between a server process and a client process, and see to it that the client has sufficient funds to pay for requests to be carried out by the server. The Bank Server does not trust its clients. Clients must provide a *capability* along with requests to the Bank Server if they want to withdraw funds or examine their accounts. Signatures are used by client processes to identify themselves to other processes.

Let us suppose a server and its client mutually distrust each other, but both trust the Bank Server. The server carries out requests in return for payment by the client. The Bank Server must check if the client has sufficient funds to pay for the request before it is carried out, and to pay after it has satisfactorily been carried out by the server. The problem lies in checking if a request has been carried out correctly. Generally, this means that the Bank Server must duplicate the work of the server. Another possibility seems to let clients check if the server's services are up to standard, but it is unlikely that clients will always honestly admit that requests were carried out correctly if payment can be avoided by lying about it.

We have assumed that servers distrust their clients, but that clients must have some, but not much, trust in the servers they use, namely only to the extent that a server, after having been paid in advance, will carry out the next request correctly. The procedure for making a request could then be something like this: Preparing to make a request, the client deposits the amount required on one of the server's bank accounts. Then the request is sent to the server. The server checks the account to see if there are sufficient funds and carries out the request. The client can check if the request was carried out to its satisfaction before sending the next request.

The structure of the Bank Server has been chosen to make this sequence of operations efficient. Usually servers will build up a reputation of being reliable and trustworthy, so that clients will deposit larger amounts on the server's accounts to pay for many requests at a time. Servers may implement caches of clients' credits so they do not have to enquire into the status of a client's account for every request that is made.

The Bank Server maintains accounts containing amounts of Virtual Money in different currencies. New currencies can be created dynamically. When a new resource is installed, for instance, a new currency could be created to represent the amount, or quota of the new resource that each user may consume. The Bank Server accepts requests to create a new currency from any process. (Since the Bank Server is a regular kind of service, it could of course be a policy that the creation of a new currency will cost a certain amount of Virtual Money in another currency). The Bank Server returns a *mint-master capability* for the new currency, plus the new currency's *name*, a small integer. The holder of the mint-master capability specifies the amount of Virtual Money to be coined in the new currency, and to which account it should be deposited. He can take Virtual Money out of circulation, mint some more, remove a currency altogether, or even devalue or revalue a currency.

The Bank Server also maintains *private accounts.* Each user may create one or more private accounts as follows. First, the user selects a private signature, which produces a public signature, by applying F to it. When the user's processes send subsequent messages with the same signature, the receivers of these messages can use the public signature as proof they came from the same user. After choosing this signature, the user sends a request, signed with the signature, to the Bank Server to create a private account. The Bank Server then creates the account, which is made to contain amounts of Virtual Money in any currency, and returns a capability for it. Initially, the account is empty. The public signature can be used to identify the account for purposes of depositing into it. The capability, or a capability with restricted rights, is necessary for other operations on the account, such as withdrawing from, examining, or closing down the account.

Besides private accounts there are *server accounts.* Server accounts can be used by services to hold the amounts, paid in advance by the clients, for services still to be rendered. Each server (i.e. each process) may create server accounts, which, depending on the system's policy, may cost a certain amount in some currency or other. As private accounts, server accounts are also created by sending a signed request to the Bank Server. The public signature given to the Bank Server, can be used subsequently to identify the server account in order to make deposits. The Bank Server returns a capability for the server account to the creator.

Server accounts are somewhat more complicated than private accounts. Each server account consists of zero or more *client subaccounts* that, in turn, contain amounts in one or more currencies. A client subaccount is identified by the client's public signature. Client subaccounts within a server account are created and removed dynamically: created when an amount is first deposited in it, and removed when empty. A request, depositiong into a client subaccount must contain the public signatures of the server account and the client subaccount within it. All other operations must be done using the capability, or a capability derived from it with fewer rights, combined with the client's signature.

With these structures for maintaining accounts, the Bank Server accepts commands to transfer amounts in one or mcre currencies from one account to another, such as

pay (*PrivAccnt*: **capability**; *ServSig*: **port**;
List: **curlist**);

which is used to transfer an amount from a private account to a server account. *PrivAccnt* is a capability for withdrawing from the private account. From the private account the Bank Server obtains the client's signature, which, together with the srver's signature, *ServSig*, identifies the server account and the client subaccount that the amount should go to. *List* is a list of one or more currencies and amounts to be transferred. Another command is

cash (*ServAccnt*: **capability**;
ClientSig, PrivSig: **port**; *List*: **curlist**);

which is used by servers to collect the Virtual Money, earned by the excution of commands on behalf of a client. *ServAccnt* is a capability for the server account, *ClientSig* identifies the client that the command was executed for. The server obtains the client's signature fRom the signature field of received commands. *PrivSig* identifies the private account where the amounts specified in *List* should be deposited. There are other commands, for example to transfer between private accounts, to create currencies, to examine accounts, etc.

In commercial computer centres, clients can use as many resources as they please, as long as they pay for them. Virtual Money can be used to represent real money. The execution of each request leads to the transfer of an amount representing the price of the operation that was carried out. The mechanisms provided by the Bank Server make a finer grained accounting mechanism possible. In most computer centres accounting and billing is only done by the centre itself, but our Bank Server allows the co-existence of many services, used by many different users, who are all paid separately for the services they offer. We can easily imagine, for instance, that a software house client of a computer centre writes a new compiler, offering it as a service, which, as it is used by other clients of that centre and the computer centre itself, pays part of the software house's bills.

In other types of computer centres, such as a university computer centre, Virtual Money can be used to represent the amount of resources a student may use up for a course; at the start of each course the system administrator transfers a certain amount of Virtual Money to each student who participates in the course. Different currencies can of course be used for different resources, effectively putting a quota on each resource. Re-allocable resources, such as disk blocks, can be given quotas by demanding payment for each allocated block, and giving a refund for each de-allocated block. The amount of Virtual Money given to each user then represents the maximum number of disk block he may use.

10. Conclusions

Many microcomputers and small office computers on the market today do not have built-in protection. Using off-the-shelf micros to build a ecure distributed operating system cannot be done using conventional operating systems, because these operating system exist at the mercy of protection hardware. We have presented other ways to achieve security in computer networks. Our mechanisms do not prevent different processes within one computer from interfering with each other, but they give clients and servers a way to mutually authenticate each other without any central authority.

Our interprocess communication model provides private communication channels between processes in one of the lower layers of the protocol hierarchy. This makes the design and implementation of the higher levels (retransmission, flow control, etc.) much simpler because there can be no interference from the outside.

Again, our model provides no protection against wire tapping; such protection can be achieved by using, for instance, end-to-end encryption on top of our protocol. The only threat we have tried to parry is the one that users will plug hosts with malicious operating systems into the network and try to use them to cheat the system's security. This threat is real; most operating systems have security flaws, and even if the operating system is perfect, halting a personal computer and switching to another (or modified) operating system is easily done.

One-way ciphers, or public key encryption form the basis of a secure communication mechanis, on top of which servers can implement services and objects, protected by the communication mechanism. The one-way cipher mechanism, and the public key mechanism can be built into existing operating systems with little effort. The one-way cipher scheme requires little or no special hardware, while the public-key encryption scheme can be used in any type of computer network.

References

[1] ISO open Systems Inteconnection Basic Reference Model, SIGCOMM 11 (April 1981) 15–65.

[2] W. Diffie and M.E. Hellman, Multiuser Cryptographic Techniques, Proceedings of the National Computer Conference (1967) 109–112.

[3] J.E. Donnelley and J.G. Fletcher, Resource Access Control in a Network Operating System, ACM Pacific '80 Conf. (Nov. 1980).

[4] A. Evans, W. Kantrowitz, and E. Weiss, A User Authentication Scheme Not Requiring Secrecy in the Computer, Comm. ACM 17 (Aug. 1974) 437–442.

[5] A. Jones and W.A. Wulf, Towards the Design of Secure Systems, Software P&E 5 (1975) 321–326.

[6] R.C. Merkle and M.E. Hellman, Hiding Information and Receipts in Trap-Door Knapsacks, IEEE Trans. Inf. Theory IT-24 (Sept. 1978) 525–530.

[7] S.J. Mullender and P.M.B. Vitányi, Locating Mobile Services in Computer Networks, CWI Computer Science Report CS-R84XX, 1984.

[8] R.M. Needham and M.D. Schroeder, Using Encryption for Authentication in Large Networks of Computes, Comm. ACM 21 (Dec. 1978) 993–999.

[9] G.B. Purdy, A High Security Log-in Procedure, Comm. ACM 17 (Aug. 1974) 442–445.

[10] R.L. Rivest, A. Shamir and L. Adleman, A Method for Obtaining Digital Signatures and Public Key Cryptosystems, Comm. ACM 21 (Feb. 1978) 120–126.

[11] L.A. Rowe and K.P. Birman, A Local Network Based on the UNIX Operating System, IEEE Trans. Soft. Eng. SE-8 (March 1982) 127–146.

[12] J.H. Saltzer, On Digital Signatures, Operating Systems Review 12 (Aprile 1978) 12–14.

[13] A.S. Tanenbaum and S.J. Mullender, An Overview of the Amoeba Distributed Operating System, Operating Syst. Rev. 15 (July 1981) 51–64.

[14] M.V. Wilkes, Time-Sharing Computer Systems, (American Elsevier, New York, 1968).