A Secure Identity-Based Capability System

Li Gong
University of Cambridge Computer Laboratory
Cambridge CB2 3QG, England

January 1989

Abstract

We present the design of an Identity-based CAPability protection system ICAP, which is aimed at a distributed system in a network environment. The semantics of traditional capabilities are modified to incorporate subject identities. This enables the monitoring, mediating, and recording of capability propagations to enforce security policies including the $\star$-property in the Bell-LaPadula model. It also supports administrative activities such as traceability. We have developed an exception list approach to achieve rapid revocation and the idea of capability propagation trees for complete revocation. A separate access control list is to represent and interpret security policy. Compared with existing capability system designs, ICAP requires much less storage and has the potential of lower cost and better real-time performance. We propose to expand Kain and Landwehr's design taxonomy of capability-based systems to cover a wider range of designs.

Introduction

Access control is a fundamental mechanism to maintain security in computer systems. It is the process that decides *who* is authorized to have *what* access rights on *which* objects with respect to some security models and policies. A security model and a security policy are different concepts although they are closely related. In this paper however, it will suffice to use them interchangeably. Security models are guidelines to direct system designs and also the standards by which a system's security properties can be evaluated. For example, in the popular Bell-LaPadula model [8], each subject (process or user) is assigned a clearance level and each object (file, data segment) is attached a security level like top secret or classified. A $\star$-property requires that a subject can only write to an object at a level at least as high as itself and it can only read from an object at a level at most as high as itself. This ensures that information only flows upwards. It is referred to as write-up and read-down.

Many systems use Lampson's access matrix [7] to represent and interpret the particular security policy. In the matrix, the rows represent subjects and columns objects. The access rights that a subject holds for an object can be found at the intersection of the row and the column belonging to the subject and the object. It is complex to manipulate the matrix directly because the number of objects can be very large. Also a matrix for a real system tends to be very sparse, so most systems do not store the access rights in a matrix form. Rather, they use either an access control list approach or a capability approach. In the access control list approach, the matrix is viewed by column. Each object is associated with an access control list which stores the subjects and their access rights for the object. The list is checked to see whether to grant an access. In the capability approach, the matrix is viewed by row. Each subject is associated with a capability list which stores its access rights to all concerned objects. Possessing a capability is the proof of possessing the corresponding access rights.

In a distributed system in a network environment, both approaches have their merits. An access control list approach implements some centralized control and supports administrative activities better. For example, it can easily answer a question such as which subjects have what access to a particular object, which is a commonly asked question when something goes wrong. The ability to answer this is called traceability. However it cannot express specific denial of access rights, which might be useful in some cases.

On the other hand, checking the validity of a capability is cheaper because it can be done locally, whereas in the access control list approach either an expensive replication or a slow centralized check has to be done. Another important feature of a capability system is that it supports better both the least-privilege principle and protected subsystems. These limit the damage when protection is partially compromised. Moreover, a capability can be easily timestamped or forwarded to a sub-server as an authorization to carry out a task. Also, the failure behavior may be better. For instance, in a scheme where capabilities can be precomputed and distributed as certificates, the protocol works even when the authentication server is not available. However, an unmodified capability system cannot solve the confinement problem, the problem of confining unauthorized information flow [6].

It seems clear that merging the two approaches can yield better systems than using either one in isolation. Karger and Herbert [4,5] designed and implemented an augmented capability architecture to support lattice security and traceability of access. The idea was to use capability-based protection at the lowest level for implementing confined domains, in support of access control lists for expressing security policies outside the security kernel. Their design was for a hardware-supported centralized system.

We aim at a distributed network environment and merge the two approaches another way. In the Identity-based CAPability system ICAP, access control lists are in support of the capability protection mechanism and nicely solve the revocation problem. The design can enforce security policies and solve the confinement problem. It also works for a centralized system. It is expected to have satisfactory performance. In the sections, we first examine in detail the weakness of traditional capability systems. Then we show how to incorporate identities and access control lists to solve the problems of confinement and capability revocation. We also discuss related issues including performance. After that, we recall the taxonomy for capability-based systems [3] and examine ICAP in its context. We propose expansions which may allow new designs.

Capability Systems

For simplicity, it is sufficient to examine the case where a capability describes a set of access rights for an object. An object is a data segment which may also contain security attributes such as access rights or other access control information. The abstract model of the environment is a distributed system in an open network where an *object server* provides access to data segments for clients. Any other servers will be introduced explicitly.

A classic capability is represented as

$$(Object, Rights, Random)$$

in which the first item is the name of the object and the second is the set of access rights. The third is a random number to prevent forgery and is usually the result of a one-way function f,

$$Random = f(Object, Rights)$$

Here f can be a publicly-known algorithm. It should not be based on other secret keys because key distribution introduces other difficulties. Its requirements are that it is computationally infeasible to inverse f and, given a pair of input and matching output it is infeasible to find a second input which gets the same output. There have been many practical designs for such functions. When an access request arrives at the server together with a capability, the one-way function f is run to check the result against the random number to detect tampering. If the capability is valid, the access is granted to the client.

Boebert made clear in [1] that an unmodified or classic capability system can not enforce the $\star$-property or solve the confinement problem. The main pitfall of a classic capability system is that "the right to exercise access carries with it the right to grant access". Since a capability is just a bit string, it can propagate in many ways without the detection of the kernel or the server. Thus although the *grant* and *take* capabilities in the Take-Grant model [10] specify the possible capability propagations they can not limit propagations because they are capabilities themselves. Scrambling capabilities as in Amoeba [9] prevents forgery but does not limit propagation because a principle in capability systems has been that *whoever* holds a capability has the right to use it. In other words, capabilities are identity independent. One of the difficulties in implementing Clark and Wilson's commercial model [2] is the difficulty of certifying that a *transformation procedure* must not pass its access rights for a *constrained data item* to the other non-certified procedures.

To support security policies, classic systems have to be modified to control capability propagations. Some kind of check against security policy has to be done somewhere, if not everywhere, in the lifetime of the capabilities. This is reflected in a taxonomy for capability systems [3]. Karger and Herbert [4,5] took an approach where subjects can pass their capabilities freely as usual, but when a capability is used to request an access, the security kernel must check whether the access should be granted according to the security policy. In other words, holding a capability is no longer both necessary and sufficient to access an object as in classic systems. It is now only necessary. The policy is represented by an access control list at a higher level.

A different approach is taken in the ICAP design. When a capability is to be propagated, the kernel or an *access control server*, which may or may not be the object server, checks to see whether to allow the propagation, according to the security policy. The object server does not check against the security policy when a capability is later used for access. By monitoring capability propagations, solving the certification difficulty in [2] is trivial. The intuitive motive of this scheme is the observation that the number of capability propagations is usually much less than the number of their uses so that it seems more economic to check the security policy at propagation time than at access time; moreover, the real time response will be better if the security policy, which may be complex and expensive to check, is checked at propagation time. In some situations this can be done well in advance of access time.

Note that in hardware-supported centralized systems, very tight control can be implemented to the extent that capabilities have to be prepared by hardware, so that malicious subjects cannot supply false ones. In a network capability system however, capabilities are in user space. When nodes can be compromised and lines can be corrupted, false capabilities can be presented to servers. In such a case, a technique such as the *unconfined right* in Hydra, which is meant to disable a capability propagation, has no real effect. Some *soft* protection measures have to be implemented.

The ICAP Architecture

We make the assumption that proper authentication is done, in the belief that in an open environment identifying subjects is necessary and the first step in enforcing any security measures or any administrative activities. We use *object server* to refer to the manager who mediates the access to objects. It can be a kernel or a file server.

Basic Structure

Generally a capability is a bit string and can propagate in many ways without detection, or in other words, the server normally cannot monitor and mediate capability propagations. In ICAP, a capability is created in a way that it will fail the validity check when used by processes of users other than the owner. Only the server can propagate a valid capability. An analog is a ticket embedded with a photo of its legitimate holder. People other than the ticket holder cannot use the ticket if the photo is attached in an unforgeable way. Only an appropriate authority can produce valid tickets.

There is a fundamental difference between a classic system and ICAP in the structure of capabilities. In the former case, for each object one capability is created for each different set of access rights that is required, and the capabilities that are kept by the server and other subjects for the same set of rights are the same. For example, if subject S1 possesses a *read only* right and S2 possesses a *read and write* right for an object, the server has to have two different capabilities C1 and C2. S1 holds C1 and S2 holds C2. In ICAP, only one capability for each object is stored at the server and different subjects' capabilities for the same object are distinct. This is achieved by changing the semantics of those items in traditional capabilities to incorporate identities, maybe the owner-id's, into the capabilities.

When the server creates a new object on behalf of client C1, an *internal* capability is created as

$$(Object, Random0)$$

and stored in the server's internal table. As usual this table is protected against tampering and leakage. C1 is sent an *external* capability

$$(Object, Rights, Random1)$$

which looks exactly the same as a classic capability but

$$Random1 = f(C1, Object, Rights, Random0)$$

When C1 presents the capability later, the server runs the one-way function f to check its validity. Note the number $Random0$ should also possess a kind of freshness to counter a playback attack. For example it could have a timestamp. We do not further discuss this side issue.

Because the internal random number is kept secret, the external capabilities are protected against forgery. Moreover, since proper authentication is done, subjects cannot masquerade as others. Another client C2 cannot use this capability even if it possesses a copy because the identities are different, hence the results of applying f will be different. Any valid propagation has to be completed by the server rather than by the clients. In other words, the server can monitor and mediate any capability propagations.

Propagation

When C1 wants to pass

$$(Object, Rights, Random1)$$

to a process owned by C2, it must explicitly present the request to the server. If the request complies with the security policy, the server retrieves the secret $Random0$ from its internal table, creates

$$(Object, Rights, Random2)$$

where

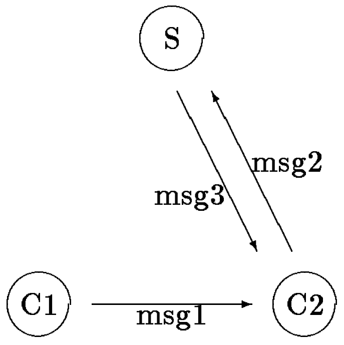
$$Random2 = f(C2, Object, Rights, Random0)$$

and passes it to C2. It is important to point out that the storage at the server is much less than in other schemes. The reasons are that only one copy of an internal capability is stored for each object rather than for each different capability which corresponds to a unique set of access rights for that object, and an internal capability is shorter than a normal capability since it does not store any information like access rights. Moreover, fewer copies of capabilities are stored in user space because in ICAP at propagation time a client's capabilities for an object can be easily combined into one.

An alternative for the propagation mechanism is when C1 wants to pass a capability to C2, it signs such a request and sends it to C2 instead of the server. When C2 wants to use the capability for the first time, it presents the request to the server. The server checks the signature and the policy to see whether to grant the access. If the answer is positive, it allows the access, creates and returns a new capability to C2. This saves at least one message in normal situations and invokes the authentication mechanism less, but causes a little overhead and delay when access is requested the first time. These two alternatives can be combined in such a way that an appropriate one is chosen for each particular propagation. Note that signing messages has to be done in a real implementation anyway unless the links are absolutely secure. For signature schemes please refer to any standard literature.

The cascaded authentication [11] using a mechanism called "passport" can be an underlying technique to support our propagation mechanism. However, we suggest that what can be done to the passport at each transit point be in accordance with some policy rather than simply further restricted. Please see ICAP' answer to the taxonomy question 4 in the next section for more discussion on this issue.

The following diagram is an illustration of the different actions taken by the discussed systems at the propagation times and access times. S is the server, C1 and C2 are clients, msg1 to msg3 are messages.



1. A classic capability system.

   msg1 : C1 propagates a capability cap1 to C2.

   msg2 : C2 requests the access by presenting cap1 to S.

   msg3 : S checks the validity of cap1 and grants access.

2. Karger's SCAP.

   msg1 : C1 propagates cap1 to C2.

   msg2 : C2 requests the access by presenting cap1 to S.

   msg3 : S checks both cap1's validity and whether the access complies with the security policy. If so it grants the access.

3. Our ICAP.

   msg1 : C1 passes to C2 a signed message which requests the server to pass to C2 a set of access rights for an object.

   msg2 : When C2 wants to access the object for the first time it requests the access by presenting msg1 to S.

   msg3 : S checks the signature and the security policy. If the access is granted, S allows the access, creates and returns a new capability cap2 to C2.

   later access: C2 only needs to present cap2 to S (msg2) and its validity is checked (msg3).

Revocation

One major problem with capability systems is revocation, i.e., withdrawing capabilities granted earlier. When an access right is revoked, in an access control list scheme the only job is to update the corresponding entries in the lists. However in a classic capability system, revocation is difficult because capabilities can be copied and stored freely and there is no way to record these activities and it is impractical to search the entire system and invalidate all those copies. Existing revocation schemes include the back pointers in Multics, Redell's indirection, and Karger's chaining method and eventcounts [5].

ICAP uses an exception list and propagation tree scheme. An internal capability has an associated exception list which specifies the current policy decision like "client C's capability for this object has been revoked." When C1 wants to revoke a capability it gave C2 earlier, it presents the request to the server. The server then updates the corresponding list. When an access is required, both the exception list and the capability's validity are checked. These can be done in parallel.

To make sure that only the ancestors, maybe plus a few specially assigned security officers, can revoke, other identities can be embedded in an external capability which record from where it is inherited. This mechanism can be nested with a depth which is implementation specific. In this case, a capability passed from C1 to C2 and then to C3 would look like

$$(Object, Rights, C1, C2, C3, Random3)$$

where

$$Random3 = f(C1, C2, C3, Object, Rights, Random0)$$

This is a tree structure which records the path of capability propagations. We call it a propagation tree. When something goes wrong, it is straightforward to know from the access control lists who have what access to an object. In addition, it is easy to know from the propagation trees *how* the access rights were propagated. Note other information like access rights are not stored in the tree. And, if only parents can revoke, the depth will be a fixed length of 2 and computations are still simple.

In normal situations, fixed length capabilities are more convenient. An alternative scheme is to store the tree at a server instead. This is in fact a better approach because it saves storage on the whole. Now each branch of the tree is stored only once, instead of once in every descendent external capability. It also speeds up the validity checking procedure since the computation of the one-way function f will be simpler. It is interesting to point out that if access rights are also stored in the tree, it is in theory unnecessary to physically pass around copies of capabilities because the server can check the tree every time an access is required. This consumes a little more storage at the server, slows down the service a little only if the tree is large, but saves the transmission of the capabilities and the storage in client space. It is not surprising that this particular version is a modified access control list scheme which also records access rights inheritance. Combining the above ideas, our final scheme is as follows.

Summary of ICAP

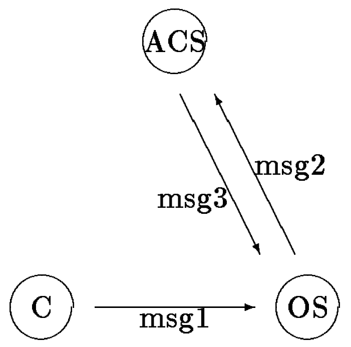
In ICAP, we have a server called the *access control server* ACS, which may or may not be the same as the *object server* OS. In the ICAP design, one ACS is supposed to support more than one OS. This also makes possible to distribute and replicate OS while maintaining ACS centralized. This makes it easier to provide adequate physical protection. However, it is sufficient to consider only one OS. OS stores the internal capabilities and their associated exception lists. ACS stores a complete access control list to represent and interpret the security policy. It also stores the propagation trees.

When a new object is created, OS creates a new internal capability and reports this to ACS. ACS then creates a new entry in the access control list. It can create a root for the propagation tree as well, although this can be done later at the first propagation time.

When a capability propagation is requested, including the case when a client requires more rights for itself, OS asks ACS whether the propagation is in accordance with the policy. If so, ACS records this in the corresponding propagation tree. If OS gets a positive reply, it grants access and creates a new capability. OS can do the creation while waiting for the reply if it would be idle otherwise. Note that if the concerned client has already got a capability for that object, it can choose to request a new capability with all the access rights granted both earlier and at this time. Note that the client does not have to search its capability list and supply the servers with the access rights it already has. ACS knows about them.

When a revocation is requested, OS adds it onto the exception list and marks it as temporary, pending the concerned access. It then consults ACS as to whether the revocation is legal. ACS checks the corresponding propagation tree and replies. If it is illegal, OS simply resolves the pending access. If legal, OS marks it permanent and at the same time ACS can choose to do the full revocation in background as described below in detail. When all the capabilities are revoked or recomputed, OS replaces the old internal capability by the new one and deletes the entry from the exception list. Revocation requests are logged and the log will be continuously reviewed by security officers.

Because it stores the propagation trees, ACS has enough information to take any revocation measures. For example, suppose a capability is associated with a count which stores the number of current valid capabilities for the same object. When revoking, if the count is small, ACS can advise OS to create a new capability which effectively means that all capabilities for the same object have to be recomputed. It then goes to all subjects that hold capabilities for the object and replaces the old ones. When the count is large and the exception list is short, it can choose to just add an entry to the list for the meantime and postpone the full revocation. The relationships between the two servers are illustrated in the diagram below.



Creation 1. C requests to create a new object (msgl).

2. OS creates the object together with a new internal capability and reports this to ACS (msg2). It also creates and returns an external capability to C.

3. ACS updates the access control list. It can create a root for the propagation tree of this object.

Access 1. C presents the external capability (msgl). OS retrieves the corresponding internal capability and runs the one-way function to do a validity check and decide whether to grant access.

Propagation 1. C requests a propagation (msgl).

2. OS asks ACS whether the request is allowed by the policy (msg2).

3. ACS checks the policy and replies accordingly (msg3). If it is allowed, ACS records the propagation in the corresponding tree.

4. Upon getting a positive reply OS creates and returns a new external capability.

Revocation 1. C requests a revocation (msgl).

2. OS updates the exception list pending the access temporarily. Then it consults ACS as to whether the revocation is legal (msg2).

3. ACS checks the propagation tree and replies (msg3). If it is legal, ACS decides whether to arrange for a full revocation.

4. Upon receiving msg3, if the revocation is legal, OS marks it permanent and notifies C if necessary; otherwise, it resolves the pending access.

5. When a full revocation is done, ACS notifies OS. OS replaces the old capability in its internal table and deletes the entry from the exception list.

Our scheme has several advantages. First, security policy checking is done at propagation time. When an access is requested, only the validity of the capability is checked. This check only runs a one-way function which can be very fast even when done in software. This is more economic than checking the policy at access time because the number of access requests normally will be much greater than that of propagation requests. It is obvious from the notes of the above diagram that the most common request, access, needs the least number of messages.

Second, the exception list supports rapid revocation which has been difficult in classic capability systems. The exception lists are expected to be short. The exception lists can also support specific denial of access, which is impossible in a classic capability system.

Third, the access control server can easily answer administrative questions like who have what access to an object and how the access was propagated. It also completes the revocation job nicely in the background.

Finally, when a revocation is marked in the exception list, this revocation can be withdrawn. This is definitely an advantage because a false alarm or an error can be resolved without invoking the expensive full revocation mechanism.

Discussion

Performance Performance is always a major concern with capability-based systems. Two of the major tasks are checking and revoking. In a secure capability system, these two kinds of checks are essential. One is to check the validity of the capabilities against forgery and the other is to check whether the use of the capabilities complies with the security policy. The validity check must be done every time a capability is used. However, the policy check can be done either at the access time as in Karger's SCAP, or at the propagation time as in ICAP; the capability designs have to be different of course. We believe that checking the security policy at each propagation time is more economic than checking it at every access time because one single propagation is likely to correspond to more than one access. Since the policy can be complex and thus expensive to check, our scheme saves a lot. It has to be pointed out that SCAP uses a cache to reduce policy checks. Our scheme can be fast without a cache. Moreover, it is only when capabilities propagate across security domains that the security policy is checked. This further reduces the number of policy checks.

The exception lists support rapid revocation. It is convenient and can be very fast. With little extra cost of storing the short lists, it ensures security while full revocation is taking place in the background. Only a one-way function is employed rather than an reversible encryption algorithm thus a software implementation would be tolerable. Finally, a shorter internal capability table and shorter external tables mean smaller storage and faster searching and sorting. All these reduce the cost considerably and give a potential for better real-time response.

Subject Representations Simple uid's as identities may not be sufficient if users are allowed to work at different security levels. In this case, domain id's rather than uid's are incorporated in the capabilities. A domain is the Cartesian product of the set of user id's, and that of the working security clearance levels. The clearance levels are mainly for non-discretionary control purposes and the uid's are mainly for discretionary control purposes. A domain id is similar to a *role.* A subject can act as different roles when working at different clearance levels. Group-id can also be included for some widely used objects. For example, to use the news facility, an "any" uid can be set up even for future users.

Discretionary Control Some discretionary control is needed even in a military environment. For example, a colonel may choose to report to a particular general in a special situation. In ICAP, discretionary control is built on top of the non-discretional control mechanisms and is interpreted by the propagation constraints in the security policy.

Protected Subsystems A type-id can be employed to enforce security in abstract data types and object-oriented programming. In such cases, there is a manager for each type or software package which is given a unique type-id. Once the type-id is embedded into a capability, the capability is sealed. Only the appropriate type manager can unseal the capability and get access to the object. A similar kind of enter capability can be used to enforce protected subsystems. A software package can be entered only with its enter capability. This can implement such requirements as that some operations can only be done through a certified secure package.

Capability Expansions As said above, a capability can be stamped to specify its lifetime. A capability can also be passed to allow another party to carry out a task on behalf of the capability owner. This particular capability may have a lifetime as "only once". Some hints or addresses can also be associated with capabilities for locating purpose when objects can migrate. To do this, the capability format has to be expanded and signature schemes are involved. These techniques are not discussed here.

Network Partitions In the presence of network partitions, apart from other availability problems, a serious threat is that a revocation request may fail to reach the object server. A centralized access control list approach has this problem too. However, the damage in ICAP is limited to what the local object server can do. Some administrative measures can be taken when serious partitions happen. It will help if the access control server stores back-up copies of the internal tables and exception lists thus is able to reboot the object servers whenever necessary. In case the access control server is not available to an object server, the object server may take actions following some guidelines.

The Taxonomy and ICAP

Kain and Landwehr [3] developed a design taxonomy for capability systems. It is based on selections from the following questions and answers list. The answers for a particular system reflect how well or poorly the system can solve the confinement problem.

1. What happens when a capability is created ?

   a. No access rights inserted.

   b. Access rights inserted.

2. What happens to the prepared-for-access capabilities describing a segment if the security attributes of that segment are modified ?

   a. Access rights not changed upon attribute change.

   b. Capability flagged for future change upon attribute change.

   c. Access rights updated upon attribute change.

3. What happens to the located-in-segment capabilities describing a segment if the security attributes of that segment are modified ?

   a. Access rights not changed upon attribute change.

   b. Capability flagged for future change upon attribute change.

   c. Access rights updated upon attribute change.

4. What happens when a capability is copied ?

   a. Access rights not changed.

   b. Access rights further restricted by context rules.

   c. Access rights set to the maximum consistent with the access rules set by the policy.

   d. Access guaranteed to be updated properly by software.

5. What happens when a capability is prepared for access ?

   a. Access rights not changed.

   b. Access rights restricted by the access rights policy.

   c. Access rights set to the maximum consistent with the security policy in force.

6. What happens when the processor attempts to access an object ?

   a. No checks made.

   b. The access checked against the available access rights.

c. The maximum possible rights computed and the attempted access checked against these computed rights.

The Honeywell Secure Ada Target SAT is *aaaacb*. An unmodified capability machine, the Plessey System 250, is *baa(a* or *b)ab*. The SCAP architecture is *abbabb* [5]. Examining ICAP in their context, we find our answers to question 1 to 3 fail to match the given choices. It seems clear that ICAP is able to enforce security policies including the ⋆-property in the Bell-LaPadula model. We thus propose to expand the taxonomy to include new possible answers as given by ICAP in order to cover a wider range of designs.

Answer to 1. c. No access rights are needed when an internal capability is created. Access rights are inserted for external capabilities.

Answer to 2. d. Access pending. Wait for new security policy interpretation to decide whether old capabilities are to be revoked.

Answer to 3. d. Same as the answer to question 2.

Answer to 4. d. Only when a capability is to propagate across a security domain, are access rights set to the intersection of the required transferred rights and the maximum consistent with the security policy.

This answer to 4 seems to be a special case of the given choice d. We would like to point out that it is inconvenient and unnecessary that only *further restricted* rights can be transferred as in choice *b*. For instance, it is perfectly legal that a lower security level user writes to an object that it has only a *write* capability and then transfers a *read* capability to a chosen user at a higher level to complete the information up-flow. An analogy is that when an employee wants to complain about his direct boss, he needs to be able to specify and complete the capability transfer to a chosen superior.

Answer to 5. a. Access rights not changed.

Answer to 6. b. Access checked against the available access rights.

Conclusion

A new design of an Identity-based CAPability protection system ICAP has been laid out. It incorporates subject identities in the capabilities by simply modifying the semantics of the items in a classic capability. This enables the kernel or server to monitor, mediate, and record capability propagations thus to enforce the ⋆-property in the Bell-LaPadula model or other security policies. This design requires a very simple and short internal capability table. For each object only one capability is stored at the server. The security policy is only checked once at each propagation time. The exception list makes rapid revocation convenient.

This list and the propagation trees allow full revocation to be done in background. The trees and the full access list can support administration activities such as traceability. This practical design potentially offers reduced cost and better real-time response. The system remains to be implemented to see how well it turns out in the real world. The ICAP's answers to the taxonomy questions fall out of the range supplied by Kain and Landwehr. We propose to expand their design taxonomy for capability-based systems to make new secure system designs possible.

Acknowledgement

We would like to thank Jean Bacon, Mike Burrows, Paul Karger, Mark Lomas, and David Wheeler for helpful comments on the technical contents and presentation.

References

[1] W.E. Boebert, "On the Inability of An Unmodified Capability Machine to Enforce the $\star$-Property", Proceedings of the 7th DoD/NBS Computer Security Conference, September, 1984.

[2] D.D. Clark and D.R. Wilson, "A Comparison of Commercial and Military Computer Security Policies", Proceedings of the 1987 IEEE Symposium on Security and Privacy, April, 1987.

[3] R.Y. Kain and C.E. Landwehr, "On Access Checking in Capability-Based Systems", *IEEE Transactions on Software Engineering*, Vol. SE13, No.2, February, 1987.

[4] P.A. Karger and A.J. Herbert, "An Augmented Capability Architecture to Support Lattice Security and Traceability of Access", Proceedings of the 1984 IEEE Symposium on Security and Privacy, April, 1984.

[5] P.A. Karger, "Improving Security and Performance for Capability Systems", Ph.D. thesis, also available as Technical Report No.149, University of Cambridge Computer Laboratory, October, 1988.

[6] B.W. Lampson, "A Note on the Confinement Problem", *CACM on Operating Systems*, Vol.16, No.10, October, 1973.

[7] B.W. Lampson, "Protection", Proceedings of the 5th Princeton Symposium on Information Sciences and Systems, Princeton University, March, 1971, reprinted in *Operating Systems Review*, Vol.8, No.1, January, 1974.

[8] C.E. Landwehr, "Formal Models for Computer Security", *ACM Computing Surveys*, Vol.13, No.3, September, 1981.

[9] S.J. Mullender, A.S. Tanenbaum, and R. van Renesse, "Using Sparse Capabilities in Distributed Operating System", Proceedings of the 6th International Conference on Distributed Computing Systems, May, 1986.

[10] L. Snyder, "Formal Models of Capability-Based Protection Systems", *IEEE Transactions on Computers*, Vol. C3, No.3, March, 1981.

[11] K.R. Sollins, "Cascaded Authentication", Proceedings of the 1988 IEEE Symposium on Security and Privacy, April, 1988.