Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Security Services

John Bull, Andrew Herbert

Abstract

This report derives architectural principles that offer a foundation for ANSA security services. It describes a general philosophy for achieving security in open distributed systems, and describes the services needed in the infrastructure to realise this philosophy. The report avoids system design choices, except as examples to demonstrate practical application of concept. It is oriented towards engineering, but gives some attention to computational issues. It does not address Enterprise and Information issues of security, these being discussed in a companion Architectural Report [APM1006].

APM.1007.00.02 **Draft** 21 October 1993

Architecture Report

Distribution:
Supersedes:
Superseded by:

Security Services

Architecture Report

Security Services

John Bull, Andrew Herbert

APM.1007.00.02

21 October 1993

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This report derives architectural principles that offer a foundation for ANSA security services. It describes a general philosophy for achieving security in open distributed systems, and describes the services needed in the infrastructure to realise this philosophy. The report avoids system design choices, except as examples to demonstrate practical application of concept. It is oriented towards engineering, but gives some attention to computational issues. It does not address Enterprise and Information issues of security, these being discussed in a companion Architectural Report [APM1006]

The report is aimed at architects concerned with interactions between systems that cooperate within a federation. It deals with the prerequisites needed to build security in this scenario. It will also be of interest to designers of nucleus interfaces, and to designers of individual nuclei for individual platforms. Sufficient background work has been done, e.g. [BULL92], to be confident that current cryptographic mechanisms can be used to create systems sympathetic with the proposed principles.

This report is **not** a general treatise on security: it is focused towards ANSA. It argues that general security services can be implemented in ANSA conformant systems once engineering mechanisms are in place to support some fundamental architectural principles, notably:

- encapsulation, and
- the distribution and use of shared secrets

Consequently, the report does not address all possible security services that one might expect to find in a system. It concentrates on those services that are necessary and sufficient for building secure systems. For example, the report considers issues of integrity, but it does not deal with confidentiality, except where this is needed to protect integrity information, such as secret keys and access certificates. Examples of general security services that might be built are listed in §3.16.

The style of the report is to discuss some issue in a narrative, and then list the architectural principles derived (the conclusions of the discussion) as bulleted items. Thus, architectural principles will be found dispersed throughout the document in close proximity to supporting text.

2 General Philosophy

Distributed computer networks of unlimited extensibility and scale will evolve over the next decade. A huge variety of computers will join international communities, to offer, request, and trade services for their users. Computer systems will wish to co-operate freely within this immense open trading enterprise, yet stay secure.

An obvious reaction to this challenge is to agree standards, create a central authority, invent a global naming scheme, issue certificates, impose an infrastructure, and lay down a policy hierarchy. This approach requires a ubiquitous infrastructure and global control. It is likely to run into problems of a huge bureaucratic superstructure, similar to that of a command economy. However, the analogy gives a clue to a better approach: that is, to come closer towards human behaviour in a market economy. People trade information and services in open society, much like a federation of computer systems, and do it well. Distributed computing could do the same.

Individuals set their own standards of good behaviour. People maintain and enforce personal security policies, but with advice and guidance from, and assessment by, higher authority. They form federations based on negotiated agreements about control and ownership. Individuals form corporations which set their own security standards and, ultimately, defend themselves. Corporations similarly negotiate to form federations. At all levels of granularity, federations restructure, come, and go, as an enterprise evolves.

These observations suggest that distributed system security could benefit from a shift of focus away from administrator imposed hierarchies of control, towards what individual service providers might need to trade their services, yet maintain and enforce their own security policies. For example, service providers might consult authentication and authorisation services before taking security decisions. This inverts the hierarchy: rather than have the administrators on top, controlling the individual subjects and objects, the individuals sit on top, with services below to support them. This theme leads to a different view of the primary security services, mechanisms and protocols needed for large scale distributed processing, and provides the basis for the principles derived in this document.

Computer systems compare with human corporations. Individuals compare with finer grain objects. Within computer systems, autonomous objects with secure boundaries are needed that are responsible for their own actions. Objects may then control their own security policies and protect their own integrity, but may seek advice and guidance from objects they trust. Security policies can be imposed, but this would be done by constraining interactions to be only though authorised individuals: for example, a corporation where external interactions were only allowed through a managing director.

This view can be realised if hardware and operating system platforms allow encapsulation boundaries to be set and enforced around software objects. Each object is then protected from all others. Current technology trends are towards

this object-oriented future, and most current hardware designs allow for object-orientation. However, this is not always done conveniently and, in practice, it is usually only possible to encapsulate like groups of objects.

Many object-based implementations enable a principle of local control of security policy where each object can control all offers and provision of service beyond its boundary. Objects may consult shared services provided by other trusted objects, but ultimately they defend themselves. A community of such objects can behave as a higher level system object which defends itself. This is analogous to individuals submitting to security policies within a company, and companies co-operating within a trading community. Since each object is forced to be accountable for itself, security weaknesses and breaches can be localised. There is no global authority on whom to rely and lay blame.

When people negotiate, they rarely exchange things directly. Written or verbally, they pass references to things, particularly references to services offered by others. Initially, a service reference is created by the owner of the thing or service. It is passed to others, who may negotiate its further transfer before finally it is used. The service provider may then apply whatever criteria he chooses to decide whether to grant access, which may well involve checking credentials with trusted authority. For example, person A might pass a reference to his accountant to person B, but when B invokes this reference, the accountant will wish to check B's authority over information concerning A. To do this, the accountant might check with A's lawyer, who holds A's delegated authority.

Computer system objects may similarly send references to objects, including references to themselves. Information would be retained within objects for manipulation by those objects in response to interface requests. Only results, the final answers, need be handed over. Rights that attach to references can be delegated and revoked easily, since the service provider always retains control over the information and is able to enforce a revocation request. Handing over a copy of the raw data relinquishes control over it forever.

Access certificates may be used to confer on clients rights to invoke services. In principle, all such rights can be granted only by the service provider. In practice, access certificates could be created by a parent, and delegated by agents, provided the server can trace authority back to itself. Ultimately, the server always has the last word on whether to meet a request.

Assuming systems can support some degree of engineering encapsulation, an object-based approach does not imply a need to discard existing systems. Initially, an object may be large: for example, it may encompass a whole computer system or database. Intermediate objects (introduced later as guard objects) may be inserted and, over time, rights to negotiate may devolve to smaller units: ultimately to individual objects. Whatever the granularity, objects may always consult services of other authorities they trust, such as an authentication service. Secrets shared between objects allow objects to sign and seal access certificates to confirm source identity and message content, much as people use signatures on unforgeable cheques (which are references to banking services). Object identity, like a person's name, is used in a security context as a reference to an agreed, or registered, secret key, or signature.

Distributed computing offers an opportunity for physical separation and containment. It also implies a need for locating, trading, sharing and accessing distributed services. This doesn't make the provision of security easier or harder: it changes the focus. It implies using existing security

technology and mechanisms in a different way. Overall, the solution proposed requires computer systems to align more closely with the established principles of human interaction.

3 Architectural Principles

3.1 Background

In the past, discussion of computer system security has been mostly in the context of closed homogeneous networks. ANSA is concerned with a fundamentally different environment: that of physically separate systems integrated into heterogeneous networks of arbitrary scale. In such an environment, systems are likely to be subject to different security policy regimes. Each system will wish to co-operate within a federation, yet remain locally autonomous. Each system will supply services to other systems, consume services from them, yet defend against them, since they might be hostile. The need for cooperation is particularly evident as systems begin to participate in service trading, with dynamic binding between supplier and consumer. Since each system is autonomous, security can be managed in such environments only by each system defining its own security policy and controlling invocations of itself.

3.2 Encapsulation requirement

A fundamental prerequisite of any security system is a boundary. Whatever is inside the boundary is protected from hostile action outside the boundary. In traditional computer systems, boundaries have been drawn around the operating systems, with hierarchical rings of protection established from a hardware core outwards. Operating systems facilities control access between *subjects* (e.g. users) who might wish to access *objects*[1] (e.g. files). Both the administration and monitoring of access control has therefore been vested in the system infrastructure, which aims to impose security.

The obvious way to extend this view to homogeneous networks is to join infrastructures, so that all *subjects* accessing *objects* appear to see one infrastructure. This is the view that distributed operating systems aim to provide. However, this approach does not provide the autonomy required for federation, and a fundamentally different (or additional) approach is needed.

The alternative is to have *objects* protect themselves; not have the operating system dictate to them. Thus a security boundary is placed around each object, and each object is responsible for deciding what may cross its boundary; that is, for deciding and enforcing its own security policy. This is wholly consistent with the principle of object encapsulation. Encapsulating an object ensures that it is possible to separate it from all other objects. If encapsulation is enforced for all objects, they cannot maliciously attack each other. Thus ANSA requires:

- object encapsulation to be rigorously enforced - no back doors;
- each object to maintain and enforce its own security policy;

1. *Object* in italics here conveys its traditional meaning of file or record, etc.

- transfer of privilege out of or into an object to be controlled by the object. This principle is elaborated in [MINSKY84], albeit in a database context.

3.3 Encapsulation provision

The physical means to achieve object encapsulation can only come from the objects environment. Ultimately, this must come from the hardware and operating system infrastructure; most probably[1] from the same mechanisms that the operating system uses to protect itself. Thus rather than the operating system acting on behalf of each object, it must allow each object to protect itself as if each were a miniature operating system. The operating system then becomes a service provider, offering protection services, like any other (encapsulated) object. Thus ANSA requires:

- each platform infrastructure (in ANSA terms - the nucleus) to provide services that achieve physical encapsulation of objects;
- encapsulated objects to be protected physically from direct interference by other objects (including operating system objects). Modification of internal object state is done by the object, at its discretion, in response to invocations of interfaces that it provides.

3.4 Object access to physical resources

Encapsulation implies that no object should be able to interfere with the internal state of any other, whether by way of reading, writing, execution or otherwise, wherever the object may be in system memory. Where platform interfaces could be used to breach encapsulation, such as to read or copy physical media, object access to such interfaces must be controlled by the nucleus. Either a system must be configured to force all application calls, in the first instance, to the nucleus, or a platform must trap offending calls and pass control of them to the nucleus. There is then a distinction between nucleus functionality which is present to create an environment and supply ANSA services, and platform enforcement to assist the nucleus to prevent breach of encapsulation and direct application access to resources. Thus:

- the infrastructure must provide for interception of all system calls for physical resources, and allow monitoring of them by the nucleus. In effect, the nucleus must encapsulate all physical resources within its encapsulation boundary.

Encapsulation also implies that when an object is created, extended, relocated, or is given new or additional resources in any way, any associated resource, particularly virtual memory, should be provided in an initial restart (clean or cleared) state. Thus the object should start with no residual components of any other objects. Conversely, when an object is deleted (at its request), it would require all traces of itself to be removed, so preventing breach of encapsulation even after death. In practice, a given system design is likely to choose either resource allocation time or object delete time to satisfy both needs. Objects should neither be able to acquire *trojan horses* during resource allocation (leading to *Time Bomb, Logic Bomb, Virus* or *Worm* attacks), nor have their confidentiality breached post deletion.

1. It could come from language type protection mechanisms (e.g., through compilers, loaders) although eventually this still comes down to physical hardware protection.

- When an object is granted or releases a physical resource, that resource must not contain any remnants of past ownership.

3.5 Engineering encapsulation

On platforms which offer ring, or hierarchical protection systems, it would be very difficult to encapsulate every object. In such cases, objects with similar security characteristics would have to occupy the same hierarchical protection level. Objects within a level would then not benefit from direct hardware mutual protection. An engineering unit of encapsulation is needed which is less fine-grained; the **capsule**. The object may remain as a unit of encapsulation as seen from a computational point of view, whereas a unit of encapsulation seen from an engineering point of view will apply to a chosen set of computational objects.

A capsule may contain a number of objects sharing similar security characteristics, with all having access to the same set of nucleus services, with all sharing a resource set, and with all being attributed to the same principal. Within a capsule, some attempt could be made to engineer mutual protection through software tools, such as compilers, loaders and databases, but this relies on the integrity of the tool set, which in turn relies on hardware protection. In practice, validating a tool chain would be difficult, and any claims of mutual protection engineered in this way might inspire little confidence.

- For practical engineering of object-based security on existing platforms, the physical unit of encapsulation would be a capsule, where capsules contain sets of objects with like security requirements.

Capsules are already used to contain the effects of failure. One might then ask if it is reasonable to assume that capsules used for both this and security purposes, and for other, as yet undefined, engineering purposes, will contain the same object sets? Since a security breach or physical failure of one object would compromise all objects of a capsule, and since security and physical failure have similar implications for recovery, it is reasonable to assume that capsules needed for these two purposes will co-incide. The need for capsules in security, failure, or other contexts, appears to arise from similar physical limitations, so it is likely that capsules will be built around the same object sets whatever the original reason for their existence.

3.6 Implementation of object-based security

If encapsulation is properly enforced, access to an object will be **only** through interfaces. Access is then achieved by the nucleus initiating an activity (process) to run in the object, and by this activity responding to an invocation of an operation on one of its interfaces. If the object regards the invocation as authentic and authorised, it will respond positively to the implied service request, otherwise it will reject the request. Ultimately the object is responsible for deciding whether to allow or deny access and, if allowed, it is responsible for any change to its internal state. The principles are:

- that objects are primarily responsible for their own security policy and for enforcement of that policy (as already discussed in §3.2). System designers may choose to have objects consult common security servers, but primary responsibility still rests with the object.

- an object address cannot be secret. Transparency mechanisms may wish to manipulate addresses, so it is undesirable to include them in an envelope that prevents them being modified.

Security policy and enforcement may be written explicitly into the objects' application code. Security parameters then become part of the object state which the object may update in response to any invocation of any of its interfaces. This allows object security to operate at a granularity of interface arguments, and to be sensitive to history and the concurrency of sharing clients. An object may use whatever criteria it chooses to decide access. It may make the decision entirely unaided, using a policy represented in its own internal state. It may use criteria based on time, past experience, co-incidence of requests, etc. It may seek assistance from other servers; in particular, from authentication and authorisation servers. In general, the object may vary its own policy, and make access decisions however it chooses.

This approach allows ultimate versatility, but the use of different decision procedures in each object may be impractical in a real system construction. Where there are many objects with similar security policy or enforcement requirements, a more realistic approach would be to arrange for a *guard object* to act for, and maintain security information on behalf of many objects.

An object construction mechanism may arrange for the constructed object to invoke a guard whenever chosen object interfaces are invoked, but without this being written explicitly into the object code by the application designer. Alternatively, the system engineering may arrange for a guard to intercept and monitor invocations of other object interfaces. Both object construction and engineering schemes could be used, in part, in the same system at the same time. There are trade-offs in benefits and constraints between the various options.

Transformer technology, see [APM1004], could be used during object construction to insert an invocations of a guard. A guard, together with declarations of security policy, could apply to a specific instance of an interface, to all interfaces of a specific type, or to all interfaces.

- An instance specific guard could be fine grain (i.e. could vary policy with interface operations and arguments) and it could be history sensitive.
- A type specific guard would be preferable where policy need not be varied between interactions, where it is not sensitive to history (only the current state matters), and where there is less need for fine granularity.
- A fully generic guard could be used where the same policy could apply to all interfaces of all types; the guard could be then be fully bound to the object.

The guard may be located anywhere, but if it is outside the objects' capsule then the path between object and guard must be protected by mutual authentication of both message source and content. Also a message framing system must be used to tie objects' requests to guards' responses. In all cases, regardless of whether the guard is constrained to a capsule, the guard may invoke other objects, local or remote, for further advice and guidance. For example, a guard may invoke a remote authentication service.

An object is created by the nucleus instantiating an object template. This may consist of both application and guard, in which case the combination is simply loaded. Alternatively, the template may consist of an application, plus a policy statement, in which case guards are added as part of application load.

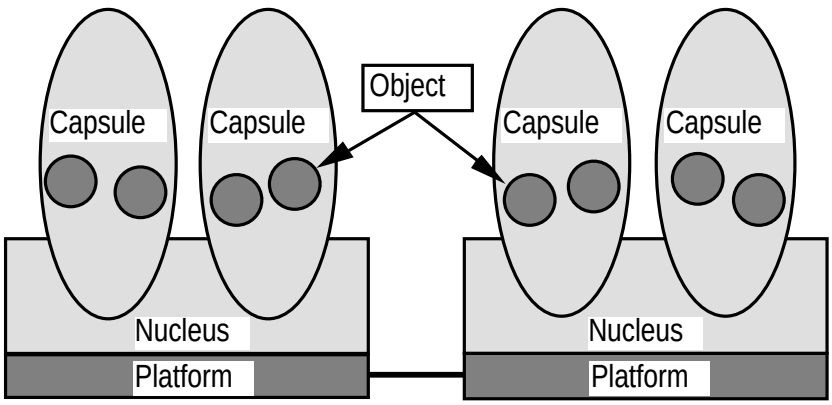
Where the same security policy applies to all interfaces of all objects within a capsule, a guard may be treated as a security manager for the capsule, with all object interfaces bound to the guard. The guard then acts on behalf of the nucleus. This allows an involuntary imposition of a security policy on a capsule by the engineering. This is the least versatile of all the options, but it may be useful when wishing to impose mandatory system security policies.

Guards are concerned with objects protecting themselves from other objects. A converse issue, confinement, is concerned with the system protecting itself and other objects from an imported untrusted *trojan horse* which could leak information, such as authentication keys, from a capsule address space. To constrain the *trojan horse* object, the nucleus must encapsulate it in its own capsule, and give it a nucleus interface which monitors its attempts to create other bindings. The engineering of this requires capsule creation to operate on both a capsule template and an associated nucleus template to construct both a capsule and nucleus interface specifically for the imported object.

3.7 Nucleus responsibilities

A small self-contained nucleus has access, through the platform operating system, to hardware resources. The nucleus is, in effect, contained in every capsule, and every capsule may call it directly, subject to the capsule having acquired rights to access the individual nucleus interfaces. If the applications are to be *protected*, the platform must offer some minimum set of engineering facilities, and the nucleus must control application access to platform resources. Applications must be constrained to obtain resources only through nucleus interfaces, and the nucleus must control access to each of its interfaces. The structure is illustrated in figure 3.1.

Figure 3.1: Structure of Platforms, Nuclei, Capsules, and Objects



The most fundamental resource needed by a capsule from the platform is a private, physically isolated, address space. The nucleus provides this through an associated *create capsule* interface. The nucleus will take an object template as an interface parameter, instantiate the template in a protected (capsule) address space, initiate its associated process (activity), assign it various privileges, and provide it with means to communicate with its environment through nucleus interfaces.

3.8 Access Control to the platform

The nucleus cannot offer any security guarantees; it can only offer application to platform mapping services. It has responsibility to provide a uniform and consistent environment for the objects, manufactured from what the platform has to offer. In particular, the nucleus cannot prevent objects from calling the platform directly should the platform choose to permit this. However, on reasonably sophisticated hardware architectures, it would be possible to request a platform to trap direct requests and redirect them to privileged nucleus services. For security, it is important that the nucleus and platform facilities, which engineer this protection, should be small, self contained, and easy to inspect.

- The nucleus must rely on the platform engineering for hardware monitoring of access to system resources and, particularly, for hardware enforcement of security boundaries.
- The security of the nucleus can be no better than the platform engineering will allow.
- For purposes of accreditation[1], security services offered by the nucleus must be limited to only that which is essential, and must be small, self contained, and easy to inspect.

Other platform resources relevant to security, which are provided by a combination of hardware and system software, are those of secure persistent storage, processing time, communications, time measurement, encipherment and key generation. Applications must access these resources through controlled nucleus interfaces. Where they are not provided by the platform, nucleus facilities may offer substitutes, but such compromises introduce weaknesses. This may be illustrated by noting that the security of a node may be maximised by isolating it. Sharing of services within distributed systems only makes prospects worse. Where weaknesses do arise, they need to be contained to affect only the local environment, and this must be a crucial aim of any distributed system security architecture. Any local weakness must not be allowed to propagate.

- Security concerns arise when services or resources are shared. Consequently, security is improved by minimising sharing, or by limiting access to shared resources.

3.9 Access Control to the nucleus

A protected record must exist that specifies which objects may call which nucleus interfaces, and it must be used by the nucleus to control interface access to itself. There are various ways to engineer this. The nucleus could maintain an access list, or it could pass capabilities (access tickets) to objects. Each of these engineering devices has various merits and defects and different choices will suit different platforms. In all cases, the mechanism used would be invisible to the application object which calls, or is prevented from calling, a nucleus interface. The protected record held by the nucleus is most likely to be maintained on a per capsule, rather than per object basis since the nucleus,

1. Inspection by auditors, since it is not feasible to use conventional tests to measure the effectiveness of security features.

though operating system and hardware assistance can only enforce encapsulation boundaries around capsules.

An object with rights to access a nucleus interface may pass on those rights to any new object it creates, when and only when the new object is created. There is no need to pass rights to objects other than children, nor other than at the time of their birth (as explained below). This facility is considered necessary and sufficient for propagation of nucleus access privileges.

The facility is considered necessary because without an object being able to transfer and relinquish its rights to nucleus services it would behave for all time as a nucleus substitute. It would never be able to devolve the task to a successor or an auxiliary.

The facility is also considered sufficient. It might be argued that an administrator should set policy concerning the transfer of nucleus access rights but, since an object possessing a right could always act on behalf of another to circumvent the policy, introducing administrator interference can only add complication without imposing authority. It might also be argued that an object should have the option to transfer its nucleus access rights to objects other than those it creates. This assumes: that the possessing object could identify such a target object; that it acquires a reason for making such a transfer; and that it has a means to decide the legitimacy of the transfer. Other than when the object is created, circumstances which meet these criteria should never arise. Any need for an object to access a nucleus resource must be inherent from the time of the objects' creation, and delaying transfer of access rights can only add complication without providing real security benefits.

- All objects are instantiated by the nucleus, and the nucleus, when it instantiates an object, must decide on and record the objects rights of access to its interfaces.
- Object access rights to nucleus interfaces need only be decided and recorded at the time of object creation.
- The nucleus must control access to its interfaces using whatever facilities may be engineered from the platform operating system and hardware facilities. The chosen mechanism for access control may be distinct from access control provided by non-nucleus servers.

3.10 Possible access control models

When an interface reference is invoked, the server must check the client's authenticity and its authority to obtain service. These will be decided, respectively, by the server, or a trusted agent of the server, checking a shared secret presented by the client, and by checking a record that authority had been granted to the client. The client may have acquired authority by a transfer, in which case the same checks must apply to the source of the transfer. Following various checks, the server must decide, based on some security policy, whether to grant or deny access. In all cases, although the server ultimately has control, it may seek advice and guidance on any aspect of authenticity, authority, or security policy, from trusted third parties.

A possible model to support this requirement is for a common ancestor of the client and server in a bootstrapping situation, or a shared authority in inter-domain co-operation, to retain a table of client identities against rights to

services available; known as an access control list (ACL). In addition, the ancestor would distribute secrets (usually keys) to the offspring to allow a server to verify the identity of a client. Either this would be done by secret keys shared directly between client and server, or by the server consulting a trusted authentication service which registers keys against identities.

- **ACL model:** given a service request by way of a client invocation of an interface reference, the server action must be to check authenticity using a key passed by the source and check authority by reference to an ACL. Where no ACL entry exists, the server may need to check the identity and authority of any source of authority transfer and apply a security policy partially based on the transfer rights of the source.
- Since access rights are indexed by identities, which must be unambiguous, an ACL model requires that the client and server are in a common name space.

An alternative model requires that the server create a sealed and signed access certificate (AC) for whatever it wishes to apply access control (e.g., an interface or operation) and return this with an interface reference. Each transfer of the access certificate, which would probably, but not necessarily, accompany transfer of the interface reference, would similarly be signed by an object transferring authority. When the interface reference is invoked, the certificate, with any accumulated signatures, would be presented as an authority to obtain service.

- **AC model:** an access certificate behaves like a capability created and issued by the server but with some essential differences:

1. on presentation, the server could check that it created the certificate;
2. in the first instance, the access certificate is authorised to a specified recipient whose identity is encoded in the AC;
3. nested signatures can be used subsequently to trace any transfers of authority. The server can then check accumulated transfers of authority against an authority transfer policy.

- Since access rights are represented **by** the AC (identity is not used as an index), ACs are independent of the name spaces they traverse.

These models are not mutually exclusive. Both may exist together within the system structure:

- An access certificate need not originate from the server, provided the server has the means to check the authority of the originator of an access certificate using an ACL entry.
- Where an access certificate is transferred to a recipient that has no ACL entry, that transfer of authority would have to be confirmed with a signed transfer. Where a certificate is transferred to a recipient known to have an ACL entry, there would be no need to authorise the transfer.

In all circumstances, the server controls access, even though reference to higher authority might be needed to make a decision. In effect, access certificates allow ACL entries, which might otherwise be maintained by the server, to migrate through the distributed system.

3.11 Threats against ACLs and Access Certificates

Each object will, in principle, be protected from all other objects by engineering encapsulation (enforced between capsules by hardware). It is then only possible for a client object activity to invoke a server object interface via nucleus services. The interface from an object into its nucleus is protected by encapsulation, but the path from one object to another will not necessarily be secure. Interfaces provide gateways through the encapsulation boundary into the server, and the nucleus can dispatch an activity in the server to respond to a client request, but it is then the responsibility of the server to decide the authenticity and authority of the request.

In the ACL model, ACLs can be located within a server and are thus protected by encapsulation. In the AC model, access certificates have to be integrity protected during migration. This may be achieved using cryptographic sealing with a key of the originator. In effect, an element of originators' encapsulation has to accompany the access certificate.

For authentication purposes, both models require authentication keys to accompany requests for access. In the ACL model, authentication has to be achieved though authentication certificates issued by a separate service. In the AC model, authentication and access control may be overlaid into the same access certificates.

3.12 Use of secrets

Provision of security integrity requires that the server protect itself from:

- a non-legitimate client posing as the legitimate one - source integrity;
- interception and modification of a communication - content integrity;
- an intercepted communication, delayed, and replayed - time integrity.

An additional threat is that the original message might be intercepted and destroyed completely, or that communication channels might be swamped, or obstacles inserted to delay reception. This is a *denial of service* attack. If the server is invoked, but denies it, this is *non-repudiation* problem. These are particularly difficult situations to handle outside of the context of specific applications, and they will not be dealt with as general architectural services.

A server can protect itself against the above threats only by using a secret that only it and a legitimate client could have known. The secret can be used to sign a request, thus confirming the identity of the sender, and to seal the message, thus confirming the integrity of the message. Although it would be possible to protect against the threats separately, for example by having one secret for signing (to guarantee source integrity), and a different one for sealing (to guarantee content integrity) there appears to be no plausible justification for handling them separately.

Time integrity requires the embedding of additional information, such as timestamps or sequence numbers. This will require clock synchronisation. This can be done, but further detail is outside the scope of this document.

A client might also wish to know that its outgoing communication reaches a legitimate destination in an unmodified form. A protocol to achieve this is possible, involving a signed and sealed confirmation returned by the server.

A further need in client/server communication may be to prevent disclosure of information (i.e. to provide confidentiality). This may best be engineered within the basic communications electronics once keys are distributed. Various ways to provide confidentiality are possible, such as by symmetric or asymmetric encryption, by hiding signals in noise, or by subliminal signals embedded in messages, but basically any technique requires a secret (in the form of keys and/or agreed algorithm) to be shared by the communicating parties. Provided a system can distribute new secrets (using existing ones) with integrity, confidentiality can be offered as an option.

3.13 Key distribution

Two capsules A and B taken in complete isolation would need the confidentiality provided by an existing shared secret to negotiate a new one. Hence initial shared keys can be installed only by some external third party that is superior to both. Because of encapsulation, this authority must be the nucleus acting on behalf of a parent during object creation. Hence shared key connections between capsules are gradually propagated down the family tree of capsules from the initial bootstrap process.

When interaction is needed between capsules of hitherto disjoint systems, shared keys must be placed into those systems by some external common authority, such as by co-operating system administrators. Once such keys are placed, they can then be used in much the same way as they are in the bootstrap process to propagate other connections between systems. Key connections that share common nodes can always be used to create new key connections. For example, if connections exist between W and X, X and Y, and Y and Z, then a new connection can be created between W and Z.

- Key connections between objects for purposes of authentication must be propagated by the nucleus during the process of object creation.
- Given shared secrets for purposes of authentication between X and Y, and between Y and Z, it is possible, with the co-operation of Y, to set up new shared secrets between X and Z which may be unknown to Y.

When a parent object creates a child, the parent must provide the nucleus with a key to pass to the child for parent-child communication[1]. Where the child is created on the same nucleus, there is no strict need for a key since the path between parent and child must, de-facto, be secure, but it is necessary to allow for the general case where the child could be created on, or could migrate to a different nucleus. Where a child is created on a remote nucleus, a secure communication path must exist from the parent to the remote nucleus via various other servers and this must have evolved from keys set up manually by co-operating administrators. A parent could delegate resources to a child, or a child could obtain resources from a parent, although on a remote platform the parent will be a surrogate. A parent, or surrogate parent could retain control of all resources delegated to a child, so the child is not protected from the parent, other than for any protection afforded by encapsulation. However, the parent is (subject to the distribution of objects in capsules) fully protected from the child.

1. Strictly this should be presented as parent to child **capsule** communication, with keys maintained on a per capsule basis, but since parent objects, in capsules, create child objects, in capsules, it is more convenient to discuss procreation in terms of objects.

- Authentication between objects that are not associated by common ancestry, such as between systems, require some common authority, such as co-operating administrators, physically to place keys in those systems.
- Certification servers may be used to distribute keys to systems (perhaps using public key encryption). Ultimately their authority to do this derives from the agreement and co-operation of administrators.

3.14 The bootstrapping process

On initial start-up, the nucleus establishes a Node Manager (NM) from a primeval template and initiates a process to run in it. The nucleus allows the NM rights to access all nucleus interfaces. The NM creates other servers using the *create capsule* interface and, as necessary, propagates its nucleus access rights to other servers. Eventually, when the NM has created a full repertoire of basic servers and delegated nucleus interface rights to them, it may delete itself.

The NM will create a number of other basic servers to manage resources such as volatile store, persistent store, communications, peripherals, and other hardware facilities. The detail will be subject to a particular system design, but a likely scenario may include creation of a login server which owns all keyboards. On the first physical access, the login server would create a principal server, password server, and administrators server, and would designate the first user as the administrator. The administrator would be invited to assign himself a principal identity and password. He could then create further services. Alternatively, the principal and password servers could be created by the NM with pre-assigned identifiers and passwords based on licence conditions.

Where all services are based on the same nucleus, the system could be pre-configured. The administrator would introduce all other users and delegate services to them directly. For access to and from remote services, the administrator could create trading, authentication and authorisation services. In turn, these servers might create encryption and key update services (to replace keys extant for a long time with new ones).

On start-up, the cold start NM might create a warm start NM. This would reside in persistent store, and on a later start-up, the cold start NM could then retrieve and activate it. In turn, the warm start NM would retrieve and activate other existing servers. However, macro/utility programs that bypass normal restart or recovery procedures must be avoided, since they may be used to penetrate an encapsulation boundary (known as *superzapping*).

3.15 Human interfaces to the system

Human users are also logically remote systems, so when an administrator introduces a user he must set up a shared secret between the user and system. This is usually done by a password which corresponds to the shared secret key. In this case the password is used in an initial exchange to establish a physically secure connection between the user and a surrogate user object that acts on behalf of the user in the system. Other methods of user authentication are possible, such as using user-held smart cards, although a mutual secret between user and system is still needed, such as a key planted in the smart card by an external authority.

- For architectural purposes, human users may be regarded as remote distributed systems. In particular, for authentication purposes, they must share secrets with system objects.

Authentication may be one-way, or mutual. One-way authentication generally involves a server authenticating a client to check for an authentic source and an uncontaminated request. Mutual authentication involves also a client authenticating a server to check for an authentic destination and uncontaminated receipt. Mutual authentication is often forgotten in human password protocols, yet it may be just as important for the user to check that he is connected to a legitimate system, as for the system to check for a registered user. For a human to authenticate mutually with the system and to transfer authority to system objects, the user would need to have functionality equivalent to a system object. If a smart card were to assist it must behave as for any other platform, but in this case it would be a human hand-held one.

- When a user connects to a system, the user may wish to check the identity of the system, as much as the system may wish to check the identity of the user. This is referred to as mutual authentication.

The identities and passwords of human users would be introduced by the system administrator. Initially, the administrator would be the only registered user set up by the cold start process. He would then have the authority to introduce other users and delegate rights to them. This would involve introducing new local user identities and initial user passwords. The process relates closely to key distribution described earlier: the administrator acts as a parent to the real uses and the system object representing him in the system, introducing them through password keys.

Once keys are planted to connect remote systems, rights to use remote services can be delegated. This process of delegation operates in much the same way as for the bootstrap process which propagates rights to use local services. The local administrator holds the means to access remote services via the remote authentication service, and he can delegate this ability to other objects, including user objects. An administrator may control the extent to which local users are permitted access to any services, local or remote, and the extent to which remote users can access local services. This is a matter of security policy, but it should be noted that policy is determined locally; each administrator determines his own policy for his own local environment.

- Because users are, in effect, remote objects, transfers of authority though shared secrets apply to them, just as for other system objects.

Basic key sharing to allow secure connection of client and server objects is relatively easily perceived as an end result, but it may be achieved in many ways. Most good security books offer examples of public and private key encryption, zero knowledge proof, subliminal channels, and one-way hash functions. In all cases, there is a shared secret, which may be used to achieve message integrity and, if needed, confidentiality.

- Shared secrets are fundamental to secure interaction, and there are many ways to achieve them.

3.16 Using security services

The way that the basic facilities are used is a matter of system construction, which could be highly complex, but most system designs seek to reduce

complexity by separating concerns. This leads to a variety of servers which play some role in achieving system objectives. Servers which may be involved in security management are:

- an authentication server to hold verification information, such as keys, where each key is usually associated with access to a service. Each key would be identified with an owner;
- an authorisation server to hold a statement of policy. The authorisation server would be consulted about matters of delegation or rights of access;
- a login server to own input devices (particularly keyboards). The login mechanisms would use other services to authenticate users which appear at input channels and connect them to surrogate objects;
- a password server to hold passwords against user identities. This server could use encryption mechanisms to scramble passwords to protect their confidentiality;
- an principal server to register user identities and associate them with surrogate objects;
- a trader to act as a service broker and enable dynamic bindings of clients to servers;
- a factory server to create new objects. The factory server plays a significant role in placing keys to create secure parent child associations;
- *user objects* (internal system representations of users) including that of the systems administrator. User objects act as surrogate users within the system, and control the users rights and responsibilities.

The precise design of these servers, their responsibilities, the ways they interact, and even their existence, are all a matter of system design. The design will be driven by system requirements and technical merit. Systems will be different, and the extent to which they achieve similar objectives will lead to normal competitive pressures. Thus security aims should not be achieved by an attempt to create one homogeneous design. To allow distributed systems to interact, agreement is required about the placing of shared secrets, and the basic form of the integrity check field in remote communications. It may be noted that confidentiality is not fundamental to the architecture, except with respect to encapsulation and the protection of keys and access certificates. Confidentiality can be added if wanted, possibly using different keys, and possibly operating at an engineering (link) level without directly involving the applications.

4 Summary

4.1 Summary

This report suggests just two primary requirements for ANSA security. Everything else builds upon these basic themes:

1. **Encapsulation**

Encapsulation is fundamental, and requires support from the hardware and software services of the platform. It is difficult to engineer full object encapsulation using present technologies, but placing like objects in capsules permits a possible compromise.

Access lists can be protected by encapsulation.

2. **Private and shared secrets - their creation and distribution**

Secrets are used both for signing and sealing (and for encipherment when offering confidentiality) and may either be private or shared.

Shared secrets support authentication mechanisms which allows clients and servers mutually to check authenticity; that is to confirm source integrity. They can also be used for sealing to protect against tampering during transmission; that is, to confirm content integrity. Access certificates are protected by sealing, and transfers of them authorised by signing.

Shared secrets are required between communicating entities, or between them and a trusted third party. They must be distributed by a higher authority; either by the node manager, or by co-operating administrators.

Private secrets are needed to seal access certificates, which originate from and return to a server. A private secret is also needed to tie an invocation to its response - a special case that arises from the asymmetry of a client/server invocation.

These primary services may be used to produce a full security service. They provide building blocks for more sophisticated facilities, such as trading, cascaded authentication, and transfer and revocation of privilege. They allow the server, in the first instance, to control its own security policy. In a practical system design this authority may be delegated to other servers. Generic guard objects could allow a system design which offers security transparency to applications, with selective relaxation of this transparency.

The primitives do not immediately answer all secondary threats, such as denial of service and non-repudiation, nor do they cover timeliness, to protect against all forms of replay. However, these secondary threats can be covered within the primitives. For example, timeliness requires a timestamp on the binding reference (before sealing), although this raises issues of the integrity of time on each platform, and of synchronised clocks throughout the distributed environment.

4.2 Future work

The architectural principles outlined in this document may be realised through many possible system designs and could employ a variety of possible cryptographic mechanisms. Further work is required to analyse the advantages and disadvantages of particular system models, and analyse the trade-offs between the various mechanisms that could be employed to construct them.

References

Only a few references are included in this document because it is specifically directed towards ANSA: it is not a general treatise on security. Reference [APM1006] covers the general framework issues, and reference [BULL92] adds detail to some of the issues raised in this document. Both [APM1006] and [BULL92] contain large numbers of references to published papers which are cited when the text in require illustration.

[APM1004]

J.P. Warne, R.T.O Rees, “The ANSA Atomic Activity Model and Infrastructre”, Architecture Projects Management Ltd., Cambridge (UK), February 1993

[APM1006]

R.T.O. Rees, J.A. Bull, “A Framework for Federating Secure Systems”, Architecture Projects Management Ltd., Cambridge (UK), May 1993

[BULL92]

J.A. Bull, L. Gong, K.R Sollins., “Towards Security in an Open Systems Federation”. Computer Security - ESORICS 92. pp 3 - 20. Springer-Verlag LNCS Series, ISBN 3-540-56246-X & 0-387-56246-X, Nov. 1992

[MINSKY84]

N.H. Minsky, “Selective and Locally Controlled Transport of Privilege” ACM Transactions on Programming Languages and Systems, Vol. 6, No 4, October 84, pp 573 - 602