The Digital Distributed System Security Architecture

Morrie Gasser, Andy Goldstein, Charlie Kaufman, Butler Lampson
Digital Equipment Corp. 85 Swanson Rd., Boxborough, Mass. 01719
Proc. 12th National Computer Security Conf., NIST/NCSC, Baltimore, 1989, pp 305-319.

Abstract

The Digital Distributed System Security Architecture is a comprehensive specification for security in a distributed system that employs state-of-the-art concepts to address the needs of both commercial and government environments. The architecture covers user and system authentication, mandatory and discretionary security, secure initialization and loading, and delegation in a general-purpose computing environment of heterogeneous systems where there are no central authorities, no global trust, and no central controls. The architecture prescribes a framework for all applications and operating systems currently available or to be developed. Because the distributed system is an open OSI environment, where functional interoperability only requires compliance with selected protocols needed by a given application, the architecture must be designed to securely support systems that do not implement or use any of the security services, while providing extensive additional security capabilities for those systems that choose to implement the architecture.

1. Overview

The state of the art of computer security today is such that reasonably secure standalone operating systems can be built, and reasonably secure connections between the systems can be implemented. The purpose of the Digital Distributed System Security Architecture is to permit otherwise secure standalone systems to interoperate in a distributed environment without reducing the level of security and assurance of those systems. By "interoperate" we mean the ability to use, in a distributed fashion, all of the security capabilities inherent in standalone systems. Users "login" just once to the distributed system, users and objects have unique global names, and mandatory and discretionary access will be enforced regardless of the relative locations of the subjects and objects.

This architecture primarily addresses features for "commercial-grade" security and lower TCSEC [DOD85] classes up through Bl. It addresses many security needs outside the scope of the TCSEC, and does not cover assurance requirements required by TCSEC classes B2 through Al. However, nothing precludes a system from implementing this architecture with a level of assurance beyond Bl.

The architecture makes extensive use of encryption. Confidentiality and integrity for communication using symmetric (secret) key cryptography is presumed to be inexpensive and pervasive. Asymmetric (public) key cryptography is used for key distribution, authentication and certification. Users authenticate themselves with smart cards containing private keys and mechanisms to calculate cryptographic algorithms, and all systems possess their own private keys to authenticate themselves to other systems.

Authentication is assisted by the use of certificates, digitally signed by certifying authorities and stored in a distributed naming service that provides a hierarchical name space. A certification hierarchy tied to the naming hierarchy, along with the use of certain naming conventions, eliminates the need for global trust or trust of the naming service. Systems that need to act on behalf of other systems or users are explicitly given the right to do so through certificates signed by the delegating parties.

In this paper key terms defined here are in *italics*. While most of these terms are well-known, the definitions here may be unconventional, different from past usage in similar contexts.

2. Security policy and reference monitors

The traditional concept of a single security policy and reference monitor [Ames83] for the entire computer system is not practical for a distributed system. While there are certain distributed environments where security management responsibility is centralized, in most cases the individual systems comprising the distributed system must be considered to be independently managed and potentially hostile toward each other. Mutually suspicious systems should be required to cooperate only to the extent that they need each other's services, and no more. Moreover, even if we could assume that a large distributed system were centrally managed under a single security administrator, building a distributed reference monitor to provide all the security capabilities of a single system presents unsolved research challenges.

Rather than a common security policy and reference monitor, each system implements its own reference monitor enforcing its own policy. Each reference monitor is responsible for controlling access to the objects it maintains. In the most general case the reference monitor on one system receives a request to access one of its objects from a subject controlled by a reference monitor on another system. Access is permitted only if the reference monitor for the object can verify that proper subject authentication has taken place, that the system from which the request is received has been duly authorized by the subject to make that request, and that there is compatibility between the security policy of the requester's reference monitor and that of the object's so that access rights can be evaluated. Implicit in this compatibility is some level of mutual trust of the reference monitors.

In today's systems, reference monitors are usually operating systems and large subsystems or servers that manage their own objects directly. In the future distributed system any application may become the reference monitor for its own set of subject and objects. The subjects and objects controlled by such a reference monitor may be implemented out of other subjects and objects controlled by another underlying reference monitor. Also, in certain limited cases, several systems may "team up" to comprise a larger system implementing a single distributed reference monitor, all implementing exactly the same policy and fully trusting each other. At this time the security architecture does not explicitly address the mechanisms needed to construct composite objects or multiple reference monitors in a computer system, and does not impose any structure on the relationship between reference monitors. The architecture simply allows all reference monitors who are able to identify their own components to securely manage their globally accessible resources in a uniform manner.

For the most part, the architecture defines interoperable security mechanisms and does not address degrees of assurance of reference monitors as addressed in the TCSEC. Regardless of their assurance, it is expected that all systems conforming to the architecture will implement interoperable mechanisms. Assurance, where important, will impose design constraints and methodologies on individual systems but should not influence the security-related external behavior of those systems. For example, a security kernel architecture might permit a reference monitor to be contained within a subset of a whole operating system, allowing that system as a component of the distributed system to be granted an Al rating. Such a subset must implement all of the relevant security mechanisms that might be implemented by other (e.g., C2) systems on the distributed system where the entire operating system acts as a reference monitor. The architecture also permits different reference monitors to have a mutual understanding of their respective degrees of assurance and accreditation ranges so that they can determine whether their security policies are compatible.

3. Computing model

The world is made up of interconnected systems and users. A system is comprised of hardware (e.g., a computer) and software (e.g., an operating system), and a system can support one or more software systems running on it. Systems implement other systems, so, for example, a computer implements an operating system which implements a database management system which implements a user query process. In this manner, a system whose reference monitor controls one set of objects might implement another system with a reference monitor for another set of objects. For purposes of the security architecture, we rarely distinguish between the different types of software systems such as hosts, operating systems, database management systems, servers, and applications, and we rarely need to get involved in the possible hierarchical relationship between systems built out of underlying systems.

A user interacts physically through a keyboard and screen that are electrically (or securely) connected to a system: usually a workstation, timesharing system, or terminal server. The user invokes an operating system and applications processes on that system which he trusts to perform work on his behalf. The work may involve only data local to the workstation, or may involve data on and interaction with remote services on other systems.

All interactions, direct or indirect, between a user and a remote system pass through the user's local system. Therefore the local system must be trusted to accurately convey the user's commands to the remote system, and the remote system must be trusted to implement the commands. Because the local system has access to any remote information that the user can access on that remote system, the user has no choice but to trust his local system to be faithful to his wishes.

The remote system, in order to satisfy a user's command, may need to forward the command, or make an additional request, to a second remote system. In such a case the first remote system must also be trusted to accurately reflect the user's wishes. In general, the user may interact through a chain of systems, where the user must trust each system in the chain, and where communications between the systems in the chain is assumed to be secure so that the commands and responses are safe from alteration, forgery or disclosure.

4. Message authentication and secure channels

The architecture depends extensively on the use of a *message hash* function that yields a message authentication code (MAC), a short "digest" of a message that is much more efficient to communicate and store than the original message. A good hash function has the property that, given the MAC of one message, it is computationally infeasible to create another message having the same MAC. While cryptographic MACs are frequently used where two parties already have established a cryptographic association, message hashes of greatest interest to the architecture are those whose security does not depend on knowledge of shared keys, so that anyone can check the MAC of a message but nobody can forge another message with the same MAC. This permits MACs of widely used messages to be freely distributed without prior negotiation of keys. An example of such a hash function is provided in Annex D of the CCITT recommendation X.509 [CCITT88b].

In this architecture, *communicating securely* means satisfying one or both of the properties: (1) knowing who originally created a message you are reading, which we call *authentication,* and (2) knowing who can read a message you create, which we call *confidentiality*. The ISO (International Standards Organization) term "data origin authentication" [ISO88b] is equivalent to property (1). Our concept of authentication also implies "data integrity": assurance that the message you are reading is exactly the same as the one that was created (if the message is altered then it's not a message from the originator).

The term "peer entity authentication", used by ISO to describe the property that you know with whom you are communicating, is subsumed in our architecture by both properties (1) and (2). In the security architecture it is meaningless to have "peer entity authentication" by itself: without either confidentiality or data origin authentication (with integrity) you cannot tell whether your message is protected or whether you are actually receiving what was sent and so communication is not secure in any practical sense.

ISO's definition of "confidentiality" is also not strictly the same as ours, as we assume that the recipient is known and must therefore have been authenticated at some time in the past.

The concept of a *secure channel*, introduced by Birrell, et al. [Birrell86], is an abstract way of viewing how we accomplish properties (1) and (2). A channel is a path by which two or more entities communicate. A secure channel may be a protected physical path (e.g., a physical wire, a disk drive and medium) or an encrypted logical path. A channel need not be real time: a message written on a channel may not be read until sometime later. A secure channel provides either authentication or confidentiality, or both, while an insecure channel provides neither. Communication via insecure channels is permitted but is not addressed by the architecture.

Secure channels have identifiers known to the senders and the receivers. A secure physical channel is identified by a hardware address such as an I/O port number on a computer or a disk drive and block number. An encryption channel is identified by an encryption key. Any message encrypted under a given key is said to be written on the channel identified by that key, regardless of whether that message is "sent" anywhere. When the message is decrypted it is said to be read from the channel. The ciphertext of an encrypted message may be written on another channel before being decrypted: typically the cipher-text is written on an insecure channel for transmission, read from the insecure channel, and finally read from the secure channel by decryption.

For a secure channel that provides authentication, the senders are known to the receivers and are thus authenticated. Specifically, a receiver of a message on a secure channel can determine that the message was written by someone in a known set of senders. If there is more than one possible sender then, in order to determine the actual sender, the receiver must trust the senders to cooperate by properly identifying themselves within the text of the message or by not sending unless requested.

For a secure channel that provides confidentiality, the receivers are known to the senders and are authorized by the senders to receive the information. In most cases there is usually only one possible receiver. If there is more than one, and the sender wants to limit the message to a specific receiver, then the sender must trust the other receivers not to read messages unintended for them.

A symmetric key channel (identified by a secret encryption key) provides confidentiality and, can provide authentication with the use of a MAC for integrity. For a symmetric key channel all authorized senders and receivers must share the same key, and therefore all senders and receivers are in the set authorized to read or write information on the channel.

An asymmetric key channel (identified by either its private or public key) provides authentication if a message is encrypted with the private key, or confidentiality if a message is encrypted with the public key. A single encryption operation cannot provide both properties (even though a single public/private key pair can provide both). Typically there is a unique pair of keys for each principal. The principal keeps its private key confidential and the public key is made generally available (online or through some directory service). This and the following description of asymmetric key chan-

nels primarily applies to the RSA public key algorithm [Rivest78].

In an asymmetric key channel used for authentication, the sender creates a "digital signature" of the message by encrypting the MAC of the message using the sender's private key, and sends the signature along with the original (plaintext) message. Any recipient who knows the sender's public key can verify the signature by recalculating the MAC and comparing it to the decrypted signature, to determine whether the original message was signed by the sender. The sender is authenticated to the receiver because only the sender knows the private key used to sign the MAC.

It is impractical for all entities in the distributed system to know the correct public keys of all other entities with which they want to communicate. Entities are typically identified using network addresses or names expressed as character strings. A special kind of signed message, termed a *certificate*, is used to unforgeably associate the name of an entity with its public key. Certificates also have a number of related functions as described below.

In an asymmetric key channel used for confidentiality, a sender encrypts a message with a receiver's public key which only the single receiver can decrypt with the private key. The sender's message is thus confidential. Since anyone can encrypt a message with someone's public key this channel does not provide authentication of the sender. To provide both authentication and confidentiality, a message must be first signed with the sender's private key and the result encrypted with the receiver's public key. In practice, both steps are rarely applied to the same message, and in fact the architecture rarely needs to make use of asymmetric key cryptography for confidentiality.

The most popular algorithm for symmetric key encryption is the Data Encryption Standard (DES). However, the DES algorithm is not specified by the architecture and, for export reasons, ability to use other algorithms is a requirement. The preferred algorithm for asymmetric key cryptography, and the only known algorithm with the properties required by the architecture, is RSA. As with DES, the architecture does not specify and will not depend on the details of the RSA algorithm; another algorithm with similar properties, if invented in the future, is permitted.

Access control does not apply to secure encryption channels: a secure encryption channel as defined in the architecture is created when needed and is not a limited resource or object to be protected. Access to the channel is determined by those who possess the encryption keys. A physical channel (whether or not it is used for security) is a limited resource to which access may need to be controlled. In such a case the channel would be treated as an object, with an ACL (see section 7) and perhaps mandatory access controls.

When two systems interact through a secure encryption channel (e.g., two nodes on different LANs using end-to-end encryption across a wide area network), there may be many intermediate systems (gateways, bridges or routers, etc.) in the path between the end systems. These intermediate systems are needed to support communications for the applications in the end systems but need not be trusted to keep the channel secure. Intermediaries in a secure physical channel, on the other hand) must be trusted.

For some applications involving several systems there are a number of secure channels between pairs of systems participating in the application. For example, consider a user on a workstation who submits a query that gets forwarded to a remote DBMS which accesses a record in a file on a file server. In this example the DBMS system is an endpoint of one secure channel (from the workstation) and an originating point for a second secure channel (to the file server). Normally all three systems must be trusted by the user because the DBMS processes both the query and the results being returned and there is no secure channel directly from the user's workstation to the file server. On the other hand, if the file server encrypts (and integrity-protects) a record that it hands to the DBMS, and the DBMS simply forwards the record to the user's workstation for decryption, then there is a secure channel between the file server and workstation and the user does not need to trust the DBMS to protect that record from disclosure or undetected modification.

In the context of communications it is simplest t think of secure channels as secure transport layer connections providing confidentiality and integrity of the data, even though transport is not the only place where there may be secure communications. In the context of authentication a secure channel is usually something defined by a given encryption key that is used to pass signed messages.

At this time, the architecture is not tied to any specific protocol suite. The detailed specifications of protocols, to be prepared eventually, will describe how to set up secure channels using specific network protocols.

5. Computers and loading

A *computer* is a system made up of a particular physical set of hardware components running some boot code. All connections between the computer and the rest of the world must be through secure channels.

An *engine* is a hardware or software device created by a *system* that can be *loaded* with a program to produce another system. The computer running its boot code provides an engine into which an operating system can be loaded, thereby creating what we commonly refer to as a host or *node*. Another example of

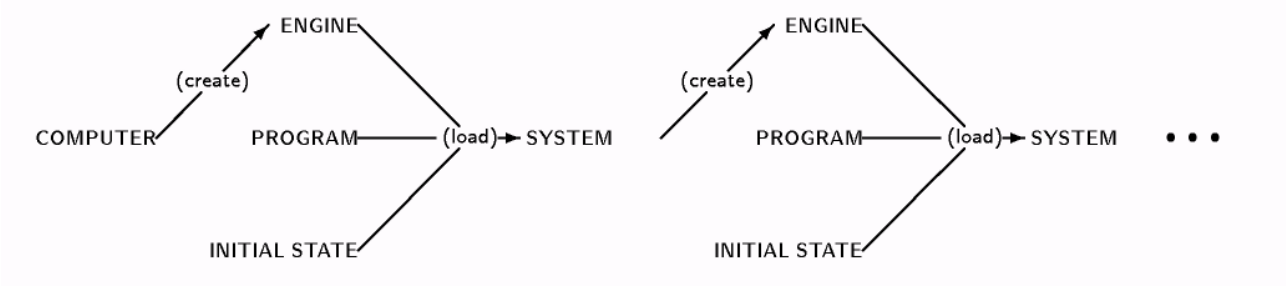

Figure 1: Computers, systems, programs and engines.

an engine is a process provided by an operating system. When loaded with an application program, the running process becomes a system. These relationships are illustrated in figure 1.

A *specification* is a description of a system's behavior (e.g., the specific behavior of a VAX 6250 computer or that of VMS 5.0, documented in some manual). While a specification is rarely written down precisely, users of (or systems interacting with) a system that is "certified" to meet a given specification can be assured that the system will behave as they expect. The architecture deals with the problems of certifying a system and determining whether that certification was done by someone you trust. Certifying a system does not have anything to do with software correctness—certifying that a system meets the "VMS 5.0 specification" simply means knowing that a specific program (the "VMS 5.0 boot image") was loaded into a specific type of system (a "VAX computer") using specific sysgen parameters. It is assumed that the particular boot image does what is intended—proving that the program in fact meets some written specification is outside the scope of the architecture.

In general, software is certified by the system loading the engine it has created, by verifying that the MAC of the software image is equal to the expected value for that software's specification. For example, if the MAC of an image you have just loaded is equal to the MAC you expect for "VMS 5.0 boot image" then you can be confident that you have just loaded a program that will behave according to the "VMS 5.0 specification." The MACs of various images that may be loaded into a given system are contained in certificates.

Each system, including the computer hardware itself, has a secret (the private portion of a private/public cryptographic key pair), generated randomly when the system is installed or created, which it uses to authenticate itself and to certify systems it creates. A system is responsible for protecting its secret from disclosure to the created systems. Through chains of reasoning beginning with the computer and ending with an application system (for example) it is possible to certify any desired aspect of a system or its behavior. In contrast to software systems' secrets which are created each time the system is rebooted, computer secrets are semi-permanent, stored in programmable read-only memory.

When a computer is asked to boot some software, the boot hardware in the computer (usually implemented as software in read-only memory) calculates a MAC of the operating system that it has loaded, and, before permitting execution, verifies (by checking certificates received with the boot image or provided to it by system management) that an operating system with the designated MAC is permitted to run on that computer. If verified, the boot hardware generates a private/public key for use by the loaded operating system, signs, using its boot secret, a certificate associating the MAC with the new public key, deletes the boot secret from any place that operating system can get to, and then begins execution of the loaded operating system. The operating system, in turn, uses its new private key as a secret to sign for other systems (applications) that it loads, and so on. When asked to authenticate itself to a remote system, the operating system presents as credentials its certificate signed by the computer. In this manner, with minimal new mechanisms in the hardware, the computer has protected itself from being loaded with malicious software, and other systems who trust the computer's boot hardware can verify the identity of the loaded operating system. Of course, if the operating system is compromised after it starts running nobody may find out. Techniques to insure that the operating system is able to protect itself and remain in secure state after it starts running are addressed by operating system security mechanisms and are outside of the Distributed System Security Architecture.

6. Naming

A *principal* is an entity that can be granted access to objects or can make statements affecting access control decisions. Principals are subjects in the TCSEC sense, but not all subjects are principals. For example, a principal may spawn multiple process within a system, each one identified as its own subject to the operating system, but the architecture treats each of these subjects as if they were the original principal and makes no attempt to isolate them from each other. When a principal accesses an object the reference

monitor for the principal in control of the object must have some way of identifying the requesting principal, and this identification is in the form of a unique global identifier. These global identifiers are Digital Naming Service (DNS) names.

Users and systems (nodes, servers, etc.) are named principals who have DNS names. There are also principals such as smart cards, processes, and sessions that do not have DNS names and that always act on behalf of other (named) principals. The use of DNS is pervasive in the architecture, but the primary reason for DNS names is so that users can identify principals and can enter their names on access control lists (see section 7). Without DNS names, users would have to identify principals with unwieldy cryptographic keys.

DNS has a hierarchical tree structure, with a single root at the top and directories at the branches. A principal's name lies within some directory and the principal always knows (or can determine) its place in the hierarchy from the root; the series of directory names from the root down to the principal is the principal's DNS name. In figure 2, for example, the full DNS name of principal P8 is TOP.MID-1.LOW.BOT.P8. While DNS names are human-readable, it is not expected that people will have to type a full DNS name very often. The DNS structure and the services provided by DNS are very similar to the directory proposed by CCITT and ISO [CCITT88a].

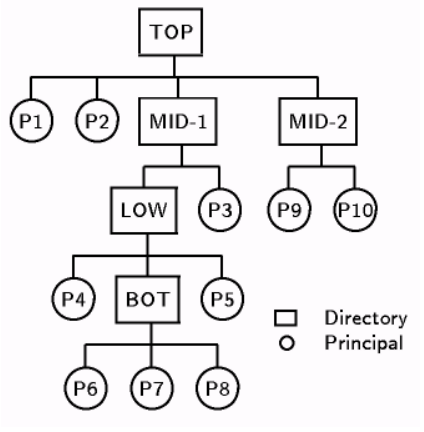

Figure 2: Example of DNS hierarchy.

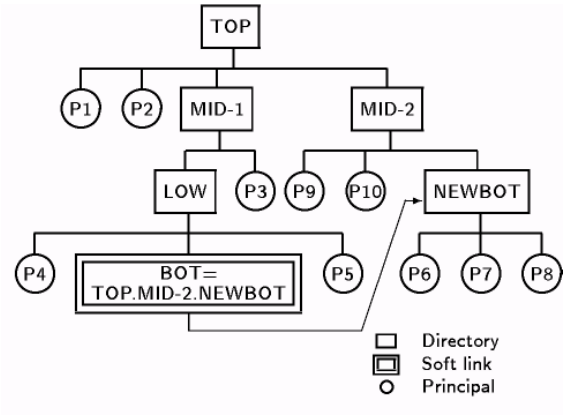

Figure 3: Symbolic link in DNS.

Principals, and even large sections of the hierarchy (subtrees), may be moved from one place in the tree to another as organizational and other associations change. This means that a principal's name (usually, just the directories in a principal's name) can change, perhaps without the principal's awareness. When a subtree is moved a *symbolic link* may be placed at the old location's parent directory that points to the new location of the subtree, thereby permitting principals to be found using their old names (see figure 3). Symbolic links serve a number of other purposes not related to security.

Because of symbolic links, a principal may be identified by several DNS names, only one of which is the true name. In figure 3, the principal originally known by the name TOP.MID-1.LOW.BOT.P8 in figure 2 is now located at TOP.MID-2.NEWBOT.P8, and may be referenced by either name due to the presence of the symbolic link at the old location of the BOT directory. To provide a fast way to determine whether two names refer to the same principal (something that the access control mechanism must be able to do) a principal also has a unique-identifier (UID) which doesn't change even if the DNS name of a principal changes. The UID is stored in DNS in the directory entry for the principal, and plays an additional role in the reassignment of names and definition of the directory hierarchy. With minor exceptions, the UID is used by the security architecture for performance rather than for security. Thus, the algorithm for enforcing uniqueness of UIDs is outside the architecture. In a few cases where security depends on uniqueness of UIDs, there are simple ways to enforce it.

Except for the names, UIDs and symbolic links, other aspects of the DNS architecture are not relevant to the security architecture and security (except certain types of revocation described in section 11) does not depend on correct functioning of the DNS servers. Of course, if DNS does not function correctly availability might suffer.

7. Access control

All information to which access is controlled is contained in objects. All objects have *access control lists* (ACLs): lists of principals (identified by DNS name) who may have access to the object, along with their access rights. There are a small number of architecturally defined access rights, such as "read," "write," etc., and some number of system-defined rights. It is the responsibility of the system (the reference monitor) controlling an object to enforce the ACL. An operating system, for example, enforces the ACLs for the files in its file system. The principal that controls an object is not listed on the ACL.

ACLs may contain names of *groups* of principals. Groups are objects with DNS names and may be created and modified by ordinary users, not just by system managers. All groups must exist as an explicit list of principals—there is no architectural support for "implicit" groups identified through some kind of naming convention (for example, "all principals contained in a given directory") but implementing such a capability is not precluded. However, large groups may be constructed out of smaller groups: groups may be nested (may name other groups) to an arbitrary depth. The ability to efficiently support both very small and very large groups, with tens of thousands of members, is essential for practical use of some of the security mechanisms specified by the architecture, and schemes have been developed that permit DNS to support them.

ACLs may list specific principals that are denied access, even if those principals are contained in groups that are permitted access. It is also possible to deny access to groups that are subgroups of other groups on the ACL. Certain other restricted forms of group denial are possible, but it is impractical, in a distributed environment where group nonmembership cannot be certified, to implement denial to arbitrary groups.

In addition to listing the principals that may access an object, the ACL may list the systems to which access may be delegated (see the discussion of delegation in section 10). This capability means that an object might not be accessible from "untrusted" workstations even if the user has delegated to that workstation.

ACLs may be implemented in a number of ways on different systems, but, because of their user visibility, it is important that ACLs have similar semantics on all systems. The VMS system-owner-group-world mask, or Unix owner/group/other bits, are primitive forms of ACLs, but such forms must be augmented (not necessarily replaced with something else) to provide the necessary semantics outlined above.

ACLs are objects themselves and have ACLs that specify who can read or modify them. An ACL may be its own ACL, or there may be other ACLs dedicated to ACL access. Figure 4 illustrates one way a file's ACL and an ACL's ACL may be related. In this figure the ACL for the ACL's ACL is itself.

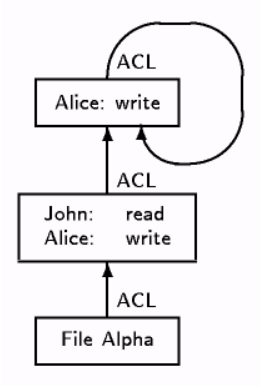

Figure 4: A file's ACL and an ACL's ACL.

8. Authentication

(In the following discussions we use as an example a principal sending a request to a system or service. In fact, the terms "system", "server" or "service" are just different names for principals—the model does not distinguish between a server and any other type of principal.)

In order to mediate access to an object that it controls, a server must *authenticate* that the identity of the requester is as claimed. Secure channels provide this "strong authentication." The password is the most common type of authentication mechanism used in systems today but the password does not provide a secure channel. At the beginning of a conversation, a set of messages are exchanged between a principal and a server, where the server establishes that it is in fact receiving messages from a secure asymmetric key encryption channel whose only possible sender is a given principal. Similarly, the principal may wish to mutually authenticate the server, and this is possible because the server is also a principal.

In order for a server to know that it is currently communicating with a given principal, a server must be sure that the signed messages it is receiving are not replays of old messages from a previous conversation (possibly sent by a third party). To deal with timeliness, a challenge/response scheme is used at the beginning of each conversation, where the server sends a random number to the principal and the principal returns the number in a signed message. Replay of a response to an old (different) challenge is not accepted. Within this signed message is other information that permits the two parties to continue to communicate in a manner that is safe from replays of past conversations.

Once two principals have authenticated each other using asymmetric key cryptography, one of them typically will generate a random secret key and send it to

the other. This secret key will be used to communicate (using symmetric key cryptography) in a manner that provides continued authentication and confidentiality for future messages during the conversation. Symmetric key cryptography is usually used for data exchange because asymmetric key cryptography is too slow.

Authentication can also be initiated with symmetric key cryptography where a principal authenticates itself to a trusted online "key distribution center" and the key distribution center provides the information necessary for that principal to then authenticate itself to a server. The indirect authentication through a trusted third party is required because otherwise the server would have to be told the secret key of the principal, leaving the principal exposed to masquerading by the server.

Nodes and other systems that need to authenticate themselves have secret or private keys stored in non-volatile memory within them, and they implement the RSA and DES algorithms using hardware or software. It is expected that software implementations of RSA or DES (without specialized hardware) will perform adequately for authentication at the beginnings of conversations, but specialized hardware will be needed to calculate DES at a speed adequate for data exchange. Before such specialized hardware becomes widely available, the authentication functions can be implemented in software without protecting the data exchange. This "authenticate at session initiation only" function provides some measure of security in certain applications even though the architecture does not recognize the subsequent unprotected data exchange as a security capability.[1]

Since users cannot remember RSA keys hundreds of bits long, and cannot calculate algorithms in their heads, user authentication requires a computer for the calculations and a portable means of storing the user's private key. Technology is just emerging that will provide both in the form of a "smart card". Each user possesses a smart card containing that user's private key, the user's secret personal identification number (PIN), and a microprocessor that can compute the RSA algorithm.[2] The user authenticates himself to the workstation by inserting the smart card into a reader, and entering the PIN into the reader (if the reader is trusted) or into the card (if the card has a keypad). The smart card refuses to operate if the correct PIN is not entered. The smart card then responds to a challenge from the workstation so that the workstation can authenticate the identity of the smart card. The workstation assumes that the user is in control of the smart card and thereby assumes it is communicating with the user through the keyboard and screen.

[1] In some international applications data exchange can be authenticated but by law must not be encrypted. Authenticated of data exchange requires the same high performance cryptographic hardware as does confidential data exchange.

[2] There are smart cards that can do simple calculations and can store RSA private keys, but if the card cannot do the complete RSA calculation then the private key must be disclosed to some external device for the calculation. A smart card is much more secure if there is no function enabling the key to be read out.

9. Certification

When an access request arrives at a server on a secure channel, that channel is usually unambiguously associated with the public key of the principal making the request.[3] However, access to objects is specified in terms of DNS names on access control lists, not in terms of public keys, so just verifying the public key of the sender on a secure channel is insufficient for access control. To enforce the access control list the server must have some way to determine the DNS name that corresponds to that public key. To assist in this determination, the requesting principal provides its DNS name prior to the request, so the server's problem is to verify that the DNS name in fact belongs to that principal with the verified public key.

It is possible, but not practical, for each server to keep a table of DNS name-to-public key correspondence for all principals listed on its ACLs. A more general solution involves the use of certifying authorities (CAs) that are trusted by systems to provide this verification. A certifying authority is a principal that possesses its own private key, and its corresponding public key is made well known to the principals who choose to trust that CA. A CA willing to certify that a given public key belongs to a given DNS name signs a certificate stating that association. CAs perform other certifications as well (e.g., certifying that a given smart card's public key belongs to a user with a given DNS name, certifying that a given MAC identifies a given software image, and certifying that a given image may be loaded on a given computer), and CAs or other principals may also certify other things (such as group membership lists). In this section we are concerned only with the certification of a public key by a CA for use in authentication.

CAs do their certification as an offline process well in advance of the use of the certificates, usually when a principal's private and public key are first created. The mechanics of generating keys and becoming certified are details outside the scope of the architecture, but the process amounts to convincing a CA that the identity of a principal (e.g., its DNS name) corresponds to a given public key, in a manner similar to

[3] This explanation is greatly simplified; the association between a principal's public key and a given channel may be very indirect, involving many other secure channels and delegations.

convincing a notary public of the correspondence between your legal name and your signature. It is easy for a principal to prove, through a response to a challenge from a CA, that it possesses the private counterpart to an alleged public key, so the act of certification is one of verifying that the principal is in fact the one named.

Certification does not require that the CA either generate or know the private key of the principal being certified, so a principal does not expose itself to any threats if certified by an untrustworthy CA. A compromised CA only compromises those who trust its certificates.[4]

Any system that knows a CA's public key, and trusts the CA to vouch for the public key of the identified principal, can verify the signature on a certificate and can determine that the public key corresponds to the given DNS name. Certificates for authentication are usually stored in a DNS server, but a copy of the information (the name and public key, or perhaps the whole certificate), may be locally cached. While CAs may be online for convenience (e.g., to distribute newly signed certificates), CAs need not and in fact cannot work like online servers. Certification must involve an offline path to corroborate the identity of the principal.

By using signed certificates to determine public keys there need be no online "authentication server," and no centralized or replicated database of public keys is required (except to support revocation—see section 11). The certificates are distributed to the places where they are needed, and DNS provides a convenient mechanism for storing certificates locally.

There is no one CA that all principals are willing to trust for all authentications. Each directory in DNS has an associated CA (see figure 5), and several directories may share the same CA. Principals in a directory usually trust the directory's CA to certify other principals in that directory. The following lists the principals that the CAs in figure 5 are trusted to certify:

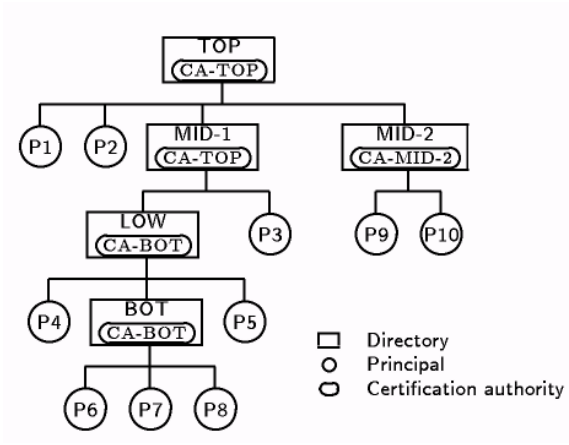

Figure 5: Certification authorities in directories of a DNS hierarchy.

CA-TOP certifies P1, P2, P3, CA-BOT, CA-MID-CA-BOT certifies P4, P5, P6, P7, P8, CA-TOP CA-MID-2 certifies P9, P10, CA-TOP

CAs are also trusted by those principals to certify the CAs of directories immediately above and below them (but of course it is unnecessary for a CA to certify itself if that CA is also associated with an adjacent directory.)

Typically, principals trust CAs close to them in the hierarchy. A principal is less likely to trust CAs farther from it in the hierarchy, whether those CAs are above, below, or in entirely different branches of the tree. If a server at one point in the hierarchy wants to authenticate a principal elsewhere, and there is no one CA that can certify both, then the server must establish a chain of trust through multiple CAs. This chain involves all the CAs in the path from the server, up through the hierarchy to the first directory that is common to both the server and the principal ("least common ancestor"), and then down to the principal. For example, in figure 5, P7 can authenticate P5 by trusting only CA-BOT. If P7 wants to authenticate P10, then all three CAs in the figure must be trusted because the least common ancestor is CA-TOP.

The authentication process assumes that the principal is identified to the server by a full DNS name, and that the server can determine the "least common ancestor" and correct CA path by a simple comparison of its own name with that of the principal. (For example, the least common ancestor CA common to TOP.MID-1.LOW.BOT.P7 and TOP.MID-1.LOW.P5 is CA-BOT in TOP.MID-1.LOW.) By use of a symbolic link on one of the intermediate directories it is possible to establish a shorter path by making it appear that the server and principal lie in a common subtree below their least common ancestor. A symbolic link alone is just a pointer for convenience of lookup, but when augmented with a "certification cross link", the certifi-

[4] When a server depending on a compromised CA manages the principal's resources or has been given the right to act on behalf of the certified principal (as when a file server manages a user's files or acts on behalf of a user) then the certified principal may be indirectly compromised.

cation path reflects the symbolic link path. A certification cross link permits a CA at one point in the hierarchy to directly certify any other CA or principal, thereby eliminating one or more higher level CAs from the default chain of trust. A cross link is a certificate signed by a CA that provides the public key of the CA for the target directory (or principal), and states that the name translation specified in the corresponding symbolic link is correct.

In figure 6, the cross link at the symbolic link MID in directory LOW permits P7 to avoid having to trust CA-TOP to certify P10. Instead, P7 authenticates P10 by trusting CA-BOT (to certify CA-MID-2), and CA-MID-2 (to certify P10). The least ancestor CA common to TOP.MID-1.LOW.BOT.P7 and TOP.MID-1.LOW.MID.P10 is CA-BOT in TOP.MID-1.LOW.

The hierarchical nature of the certification architecture described here is similar to that used in ISO's directory authentication framework [CCITT88b]. In ISO's architecture, however, users who have no a priori knowledge of the certification hierarchy must potentially trust all CAs because there is no explicit way to indicate the "least common ancestor" or other limitations to the chain of trust. The architecture used here is an outgrowth of work by Birrell, et al. [Birrell86].

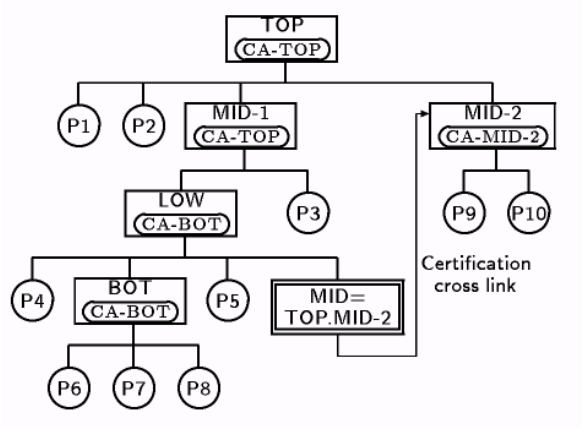

Figure 6: Symbolic link MID with certification cross link. CA-BOT certifies CA-MID-2

10. Delegation

When a user authenticates himself to a workstation, the user at the same time *delegates* to the workstation the right to speak on behalf of (act as a surrogate for) the user. This delegation is expressed in a certificate signed by the user's smart card at login. Delegation does not require any modification of ACLs. When the workstation accesses a remote service the workstation presents the delegation certificate to prove that the user authorized the surrogate. Note that remote access through a workstation does not require the remote system to reauthenticate the user. (The smart card does not play a role in any subsequent authentications or delegations.) Instead, the delegation certificate tells the remote system that the smart card trusts the workstation to accurately reflect the user's commands. The remote system may wish to also authenticate the local workstation, however, using a challenge/response. Where there is a cascade of systems involved, each system delegates to the next system the right to act on its behalf (or the right to issue statements on behalf of the user), thereby propagating the ability to act as a surrogate for the original user.

Once the user delegates rights to a system, that system can act on the user's behalf even after the user logs out. To limit the damage in the case of a subsequent malfunction or compromise of a system, a properly functioning system terminates the delegation when it is no longer needed (e.g., at the end of a session) by destroying its copy of any secret key generated for purposes of that delegation and by notifying the parties with which they were communicating to no longer honor the delegation. (We assume users trust their systems while they are using them, but not necessarily after they logout.) As a backup, in case of system malfunction, delegations also time out, the timeout being set when the delegation is made. It is the responsibility of the system enforcing access to honor the timeout and delegation termination.

A delegation to a system implies the system may make any statements at all on behalf of the delegator. While restricted delegation, where the user specifies only a subset of statements such as a list of specific objects that may be referenced, seems desirable, the types of restrictions that might be useful are highly application-dependent and cannot be specified by a security architecture. Instead, we use the concept of user *roles* for such restrictions. A user authenticates himself using a DNS name that is the name of one of several possible roles, and these roles are represented as one-member groups in DNS, all containing the actual user name in their membership list. By delegating the rights of a specific role the user delegates rights to access only those objects that list the role on their ACLs.

11. Revocation

The architecture provides for a high degree of assurance that access is only granted when authorized. But once granted, revocation of access is not provided with the same degree of assurance. Although revocation is required and supported, the revocation may not take place in a guaranteed amount of time or before any specific event, and there is no absolute assurance that it will ever take place (except that there is usually

some timeout or expiration that places an upper bound on the duration).

There are several things that one can imagine being revoked, all of which ultimately affect whether a principal has access to an object: access rights on ACLs, group membership, certificates for authentication, certificates for delegation, and authentication.

Immediate revocation is a difficult problem because it requires that either (1) systems not cache any information used to make access control decisions (public keys, group membership, ACL rights), or (2) there be a mechanism that reliably informs all systems using the access control information when a change has been made. Implementing (1) has an unacceptable effect on performance, and (2) is impractical since nobody can keep track of who is using the access control information.

Instead of immediate revocation, the architecture allows for "slow" revocation, where an application-by-application decision is made as to when, after a request to revoke, the revocation takes place. Most likely revocation will be determined by events: e.g., the next time a file is opened, the next time a user logs in, or when a delegation expires. Delayed revocation should be implemented in a way that causes users no surprises. Users maintaining ACLs, for example, might be informed that revocation has no effect on processes that currently have the file open.

A system is permitted to parse an ACL in advance, including expanding all groups named on an ACL, and to save that information for subsequent attempts by a principal to access the object. Removing a principal from a group or from an ACL will affect some subsequent access but is unlikely to affect accesses in progress. However, if (for example) the effect of this advance computation results in a user's access request being satisfied next time he logs in, even though he has since been removed from the group, then this implementation is not permissible unless a way can be found to convince users that such behavior is reasonable.

Certificates used for authentication expire, but on occasion a certificate needs to be revoked in advance because a principal's private key has been compromised, or because the person changes affiliation and can no longer be trusted to access objects on whose ACLs he is listed. Certificates for authentication are stored in a few well-known places (most likely, in DNS), and all services that use certificates will look for them in these well-known places. Revoking a certificate means deleting each copy of the certificate from these places. This deletion is somewhat unreliable because DNS directories are replicated, but if DNS is functioning normally the changes will propagate to the copies in a reasonable amount of time. The certification structure in ISO's directory authentication framework [CCITT88b] also depends on the directory for the "security" of certificate revocation.

A system may cache a certificate (or the information in a certificate) but should periodically check the well-known places to determine whether the cache is still valid. Other techniques, such as checking the time a directory was last modified, can be used to make this process more efficient. A properly functioning system will not accept a certificate from any source other than a DNS server whom it trusts for revocation. In particular, the authentication dialog does not include transmittal of authentication certificates in place of those that should be obtained from DNS. In the event of compromise of a DNS server, or inability for a system to contact a server, revocation will not work.

Authentication cannot be revoked. Once a certificate has been used to authenticate a principal, that authentication is valid for as long as the original certificate was valid, or until the system chooses to stop using the authentication. Since authentication tends to happen at the beginnings of sessions when secure channels are created, authentication is not useful beyond the end of a typical session, and properly functioning applications that expect sessions to last for days or weeks should probably reauthenticate at intervals commensurate with the interval at which they check DNS directories for changes in certificates.

Like authentication, delegation times out but cannot be revoked once granted. However, delegation timeouts, tied to the lifetime of most sessions, will be far shorter than the certificate timeouts on which authentication depends. Both authentications and delegations are erased when no longer needed (at the ends of sessions).

Because delegation timeouts are relatively short, it is possible that a delegation will have to be renewed during a session before it times out. A facility is provided whereby such a renewal can be initiated by the first system in the delegation chain and propagated to other systems in the chain, provided that the user's smart card is still in place to sign a new certificate.

12. Mandatory access controls

The goal of the architecture is to provide mandatory (non-discretionary) access controls in all systems that implement discretionary access controls, but it is realized that some systems will never be used in a mandatory control environment and so implementation of mandatory controls is optional. Even if not enforcing mandatory controls, systems should be compatible with those that do.

DoD-style mandatory security as specified in the TCSEC is supported through labeling mechanisms controlled by the individual reference monitors. Every object and subject under direct control of a reference

monitor has one or more *access class* labels, and mandatory access to local objects by local subjects is enforced in the usual manner.

A request originating from a remote system contains an access class label specified by the remote reference monitor, corresponding to the access class of the remote subject making the request. The local reference monitor uses this label, along with additional information about the remote reference monitor, to determine whether to allow the access. This additional information consists of certificates (obtained from DNS in a manner similar to the authentication certificates) that specify the *policy domain* and set of access classes for which the remote reference monitor is responsible. Access is granted only if the policy domain is appropriate (this domain may include information about the level of assurance of the remote system) and if the access class on the request is within the permitted set. The "cascading problem" discussed in the TNI [NCSC87] cannot be fully prevented except by system configuration, because none of the systems participating in the potential unauthorized write-down of information can be trusted to prevent it.

It is our intent to specify a commercial integrity architecture, perhaps based on the Clark and Wilson model [Wilson87], but work in that area remains to be done.

When both discretionary and mandatory access controls are applied to an access request, if either set of controls would disallow the request, then access is denied. In contrast to discretionary access controls, changes to mandatory access control attributes of principals and objects must take effect immediately. For example, security violations could occur if a request to "downgrade" or "upgrade" an access class does not immediately abort any accesses in progress that might no longer be allowed. The difficulty of implementing immediate revocation is mitigated by the fact that changes to mandatory attributes are rare, as noted above.

13. Problems not covered

The security architecture does not address all security concerns in computer systems. It concentrates on security problems that are unique to or exacerbated by distributed systems, such as authentication, secure communication, and global access control. Other problems in developing useful distributed systems, whether or not they have to do with security (such as global naming, synchronization, distributed databases, and assurance) are presumed to be addressed by other efforts, and a practical implementation of the security architecture may require solutions to problems in these other areas.

14. Status

The security architecture is intended for implementation across the entire Digital product line, including all operating systems, applications and hardware components. Any product acting on behalf of multiple users, or needing to take part in access control decisions, is affected by the architecture. When in place, the architecture will discourage the implementation of ad hoc, duplicative, and inconsistent security mechanisms in Digital software and hardware products. Of course, the security mechanisms will also be made available to customers for use by their own developers.

At this time of writing the details of the architecture (protocols, message formats, algorithms, etc.) are under development—little implementation has begun. Most of the groundwork and formal logic has been worked out, and functional specifications have been written.

References

[Ames83] Stanley R. Ames, Jr., Morrie Gasser, and Roger R. Schell, "Security Kernel Design and Implementation: An Introduction," *Computer*, Vol. 16, No. 7, July 1983.

[Birrell86] Andrew D. Birrell, Butler W. Lampson, Roger M. Needham, and Michael D. Schroeder, "A Global Authentication Service without Global Trust," Proceedings of the 1986 IEEE Symposium on Security and Privacy, IEEE Computer Society, 1986.

[CCITT88a] International Telegraph and Telephone Consultative

Committee (CCITT), X.500, The Directory - Overview of Concepts, Models and Services (same as ISO 9594).

[CCITT88b] CCITT, X.509, The Directory - Authentication Framework (same as ISO 9594-8).

[DOD85] Department of Defense, Trusted Computer System Evaluation Criteria, DOD 5200.28-STD, December 1985.

[ISO88b] International Standards Organization, ISO 7498-2, Security Architecture.

[NCSC87] National Computer Security Center, Trusted Network Interpretation, Ft. George G. Meade, MD, July 1987.

[Rivest78] R. L. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," Communications of the ACM, Vol. 21, No. 2, 1978.

[Wilson87] 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, IEEE Computer Society, 1987.