Network Working Group
Internet Draft

Charles Kaufman
Digital Equipment Corporation

October 1991

DASS

Distributed Authentication Security Service

DRAFT

STATUS OF THIS MEMO

This DRAFT document specifies the Services, Interfaces, Operation,and Protocols of the DASS Authentication Service. The DASS Authentication Service is used by applications to strongly authenticate and establish shared keys with peer applications. Distribution of this memo is unlimited.

Contents

Figures

1 Introduction

1.1 What is DASS?

Authentication is a security service. The goal of authentication is to reliably learn the name of the originator of a message or request. The classic way by which people authenticate to computers (and by which computers authenticate to one another) is by supplying a password. There are a number of problems with existing password based schemes which DASS attempts to solve. The goal of DASS is to provide authentication services in a distributed environment which are both more secure (more difficult for a bad guy to impersonate a good guy) and easier to use than existing mechanisms.

In a distributed environment, authentication is particularly challenging. Users do not simply log on to one machine and use resources there. Users start processes on one machine which may request services on another. In some cases, the second system must request services from a third system on behalf of the user. Further, given current network technology, it is fairly easy to eavesdrop on conversations between computers and pick up any passwords that might be going by.

DASS uses cryptographic mechanisms to provide "strong, mutual" authentication. Mutual authentication means that the two parties communicating each reliably learn the name of the other. Strong authentication means that in the exchange neither obtains any information that it could use to impersonate the other to a third party. This can't be done with passwords alone. Mutual authentication can be done with passwords by having a "sign" and a "countersign" which the two parties must utter to assure one another of their identities. But whichever party speaks first reveals information which can be used by the second (unauthenticated) party to impersonate it. Longer sequences (often seen in spy movies) cannot solve the problem in general. Further, anyone who can eavesdrop on the conversation can impersonate either party in a subsequent conversation (unless passwords are only good once). Cryptography provides a means whereby one can prove knowledge of a secret without revealing it.

People cannot execute cryptographic algorithms in their heads, and thus cannot strongly authenticate to computers directly. DASS lays the groundwork for "smart cards": microcomputers sealed in credit cards which when activated by a PIN will strongly authenticate to a computer. Until smart cards are available, the first link from a user to a DASS node remains vulnerable to eavesdropping. DASS mechanisms are constructed so that after the initial authentication, smart card or password based authentication looks the same.

Today, systems are constructed to think of user identities in terms of accounts on individual computers. If I have accounts on ten machines, there is no way a priori to see that those ten accounts all belong to the same individual. If I want to be able to access a resource through any of the ten machines, I must tell the resource about all ten accounts. I must also tell the resource when I get an eleventh account.

DASS supports the concept of global identity and network login. A user is assigned a name from a global namespace and that name will be recognized by any node in the network. (In some cases, a resource may be configured as accessible only by a particular user acting through a particular node. That is an access control decision, and it is supported by DASS, but the user is still known by his global identity). From a practical point of view, this means that a user can have a single password (or smart card) which can be used on all systems which allow him access and access control mechanisms can conveniently give access to a user through any computer the user happens to be logged into. Because a single user secret is good on all systems, it should never be necessary for a user to enter a password other than at initial login. Because cryptographic mechanisms are used, the password should never appear on the network beyond the initial login node.

DASS was designed as a component of the Distributed System Security Architecture (DSSA) (see "The Digital Distributed System Security Architecture" by M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson, 1989 National Computer Security Conference). It is a goal of DSSA that access control on all systems be based on users' global names and the concept of "accounts" on computers eventually be replaced with unnamed rights to execute processes on those computers. Until this happens, computers will continue to support the concept of "local accounts" and access controls on resources on those systems will still be based on those accounts. There is today within the Berkeley rtools running over the Internet Protocol suite the concept of a ".rhosts database" which gives access to local accounts from remote accounts. We envision that those databases will be extended to support granting access to local accounts based on DASS global names as a bridge between the past and the future. DASS should greatly simplify the administration of those databases for the (presumably common) case where a user should be granted access to an account ignoring his choice of intermediate systems.

1.2 Central Concepts

1.2.1 Strong Authentication with Public Keys

DASS makes heavy use of the RSA Public Key cryptosystem. The important properties of the RSA algorithms for purposes of this discussion are:

- It supports the creation of a public/private key pair, where operations with one key of the pair reverse the operations of the other, but it is computationally infeasible to derive the private key from the public key.

- It supports the "signing" of a message with the private key, after which anyone knowing the public key can "verify" the signature and know that it was constructed with knowledge of the private key and that the message was not subsequently altered.

- It supports the "enciphering" of a message by anyone knowing the public key such that only someone with knowledge of the private key can recover the message.

With access to the RSA algorithms, it is easy to see how one could construct a "strong" authentication mechanism. Each "principal" (user or computer) would construct a public/private key pair, publish the public key, and keep secret the private key. To authenticate to you, I would write a message, sign it with my private key, and send it to you. You would verify the message using my public key and know the message came from me. If mutual authentication were desired, you could create an acknowledgment and sign it with your private key; I could verify it with your public key and I would know you received my message.

The authentication algorithms used by DASS are considerably more complex than those described in the paragraph above in order to deal with a large number of practical concerns including subtle security threats. Some of these are discussed below.

1.2.2 Timestamps vs. Challenge/Response

Cryptosystems give you the ability to sign messages so that the receiver has assurance that the signer of the message knew some cryptographic secret. Free-standing public key based authentication is sufficiently expensive that it is unlikely that anyone would want to sign every message of an interactive communication, and even if they did they would still face the threat of someone rearranging the messages or playing them multiple times. Authentication generally takes place in the context of establishing some sort of "connection," where a conversation will ensue under the auspices of the single peer-entity authentication. This connection might be cryptographically protected against modification or reordering of the messages, but any such protection would be largely independent of the authentication which occurred at the start of the connection. DASS provides as a side effect of authentication the provision of a shared key which may be used for this purpose.

If in our simple minded authentication above, I signed the message "It's really me!" with my private key and sent it to you, you could verify the signature and know the message came from me and give the connection in which this message arrived access to my resources. Anyone watching this message over the network, however, could replay it to any server (just like a password!) and impersonate me. It is important that the message I send you only be accepted by you and only once. I can prevent the message from being useful at any other server by including your name in the message. You will only accept the message if you see your name in it. Keeping you from accepting the message twice is harder.

There are two "standard" ways of providing this replay protection. One is called challenge/response and the other is called timestamp-based. In a challenge response type scheme, I tell you I want to authenticate, you generate a "challenge" (generally a number), and I include the challenge in the message I sign. You will only accept a message if it contains the recently generated challenge and you will make sure you never issue the same challenge to me twice (either by using a sequence number, a timestamp, or a random number big enough that the probability of a duplicate is negligible). In the timestamp-based scheme, I include the current time in my message. You have a rule that you will not accept messages more than - say - five minutes old and you keep track of all messages you've seen in the last five minutes. If someone replays the message within five minutes, you will reject it because you will remember you've seen it before; if someone replays it after five minutes, you will reject it as timed out.

The disadvantage of the challenge/response based scheme is that it requires extra messages. While one-way authentication could otherwise be done with a single message and mutual authentication with one message in each direction, the challenge/response scheme always requires at least three messages.

The disadvantage of the timestamp-based scheme is that it requires secure synchronized time. If our clocks drift apart by more than five minutes, you will reject all of my attempts to authenticate. If a network time service spoofer can convince you to turn back your clock and then subsequently replays an expired message, you will accept it again. The multicast nature of existing distributed time services and the likelihood of detection make this an unlikely threat, but it must be considered in any analysis of the security of the scheme. The timestamp scheme also requires the server to keep state about all messages seen in the clock skew interval. To be secure, this must be kept on stable storage (unless rebooting takes longer than the permitted clock skew interval).

DASS uses the timestamp-based scheme. The primary motivations behind this decision were so that authentication messages could be "piggybacked" on existing connection establishment messages and so that DASS would fit within the same "form factor" (number and direction of messages) as Kerberos.

1.2.3 Delegation

In a distributed environment, authentication alone is not enough. When I log onto a computer, not only do I want to prove my identity to that computer, I want to use that computer to access network resources (e.g. file systems, database systems) on my behalf. My files should (normally) be protected so that I can access them through any node I log in through. DASS allows them to be so protected without allowing all of the systems that I might ever use to access those files in my absence. In the process of logging in, my password gives my login node access to my RSA secret. It can use that secret to "impersonate" me on any requests it makes on my behalf. It should forget all secrets associated with me when I log off. This limits the trust placed in computer systems. If someone takes control of a computer, they can impersonate all people who use that computer after it is taken over but no others.

Normally when I access a network service, I want to strongly authenticate to it. That is, I want to prove my identity to that service, but I don't want to allow that service to learn anything that would allow it to impersonate me. This allows me to use a service without trusting it for more than the service it is delivering. When using some services, for example remote login services, I may want that service to act on my behalf in calling additional services. DASS provides a mechanism whereby I can pass secrets to such services that allow them to impersonate me.

Future versions of this architecture may allow "limited delegation" so that a user may delegate to a server only those rights the server needs to carry out the user's wishes. This version can limit delegation only in terms of time. The information a user gives a server (other than the initial login node) can be used to impersonate the user but only for a limited period of time. Smart cards will permit that time limitation to apply to the initial login node as well.

1.2.4 Certification Authorities

A flaw in the strong authentication mechanism described above is that it assumes that every "principal" (user and node) knows the public key of every other principal it wants to authenticate. If I can fool a server into thinking my public key is actually your public key, I can impersonate you by signing a message, saying it is from you, and having the server verify the message with what it thinks is your public key.

To avoid the need to securely install the public key of every principal in the database of every other principal, the concept of a "Certification Authority" was invented. A certification authority is a principal trusted to act as an introduction service. Each principal goes to the certification authority, presents its public key, and proves it has a particular name (the exact mechanisms for this vary with the type of principal and the level of security to be provided). The CA then creates a "certificate" which is a message containing the name and public key of the principal, an expiration date, and bookkeeping information signed by the CA's private key. All "subscribers" to a particular CA can then authenticated to one another by presenting their certificates and proving knowledge of the corresponding secret. CAs need only act when new principals are being named and new private keys created, so that can be maintained under tight physical security.

The two problems with the scheme as described so far are "revocation" and "scaleability".

1.2.4.1 Certificate Revocation

Revocation is the process of announcing that a key has (or may have) fallen into the wrong hands and should no longer be accepted as proof of some particular identity. With certificates as described above, someone who learns your secret and your certificate can impersonate you indefinitely - even after you have learned of the compromise. It lacks the ability corresponding to changing your password. DASS supports two independent mechanisms for revoking certificates. In the future, a third may be added.

One method for revocation is using timeouts and renewals of certificates. Part of the signed message which is a certificate may be a time after which the certificate should not be believed. Periodically, the CA would renew certificates by signing one with a later timeout. If a key were compromised, a new key would be generated and a new certificate signed. The old certificate would only be valid until its timeout. Timeouts are not perfect revocation mechanisms because they provide only slow revocation (timeouts are typically measured in months for the load on the CA and communication with users to be kept manageable) and they depend on servers having an accurate source of the current time. Someone who can trick a server into turning back its clock can use expired certificates.

The second method is by listing all non-revoked certificates in the naming service and believing only certificates found there. The advantage of this method is that it is almost immediate (the only delay is for name service "skulking" and caching delays). The disadvantages are: (1) the availability of authentication is only as good as the availability of the naming service and (2) the security of revocation is only as good as the security of the naming service.

A third method for revocation - not currently supported by DASS - is for certification authorities to periodically issue "revocation lists" which list certificates which should no longer be accepted.

1.2.4.2 Certification Authority Hierarchy

While using a certification authority as an introduction service scales much better than having every principal learn the public key of every other principal by some out of band means, it has the problem that it creates a central point of trust. The certification authority can impersonate any principal by inventing a new key and creating a certificate stating that the new key represents the principal. In a large organization, there may be no individual who is sufficiently trusted to operate the CA. Even if there were, in a large organization it would be impractical to have every individual authenticate to that single person. Replicating the CA solves the availability problem but makes the trust problem worse. When authentication is to be used in a global context - between companies - the concept of a single CA is untenable.

DASS addresses this problem by creating a hierarchy of CAs. The CA hierarchy is tied to the naming hierarchy. For each directory in the namespace, there is a single CA responsible for certifying the public keys of its members. That CA will also certify the public keys of the CAs of all child directories and of the CA of the parent directory. With this cross-certification, it is possible knowing the public key of any CA to verify the public keys of a series of intermediate CAs and finally to verify the public key of any principal.

Because the CA hierarchy is tied to the naming hierarchy, the trust placed in any individual CA is limited. If a CA is compromised, it can impersonate any of the principals listed in its directory, but it cannot impersonate arbitrary prinqcipals. DASS provides mechanisms for every principal to know the public key of its "parent" CA - the CA controlling the directory in which it is named. The result is the following rules for the implications of a compromised CA:

a) A CA can impersonate any principal named in its directory.

b) A CA can impersonate any principal to a server named in its directory.

c) A CA can impersonate any principal named in a subdirectory to any server not named in the same subdirectory.

d) A CA can impersonate to any server in a subdirectory any principal not named in the same subdirectory.

The implication is that a compromise low in the naming tree will compromise all principals below that directory while a compromise high in the naming tree will compromise only the authentication of principals far apart in the naming hierarchy. In particular, when multiple organizations share a namespace (as they do in the case of X.500), the compromise of a CA in one organization can not result in false authentication within another organization.

DASS uses the X.500 directory hierarchy for principal naming. At the top of the hierarchy are names of countries. National authorities are not expected to establish certification authorities (at least initially), so an alternative mechanism must be used to authenticate entities "distant" in the naming hierarchy. The mechanism for this in DASS is the "cross-certificate" where a CA certifies the public key for some CA or principal not its parent or child. By limiting the chains of certificates they will use to parent certificates followed by a single "cross certificate" followed by child certificates, a DASS implementation can avoid the need to have CAs near the root of the tree or can avoid the requirement to trust them even if they do exist. A special case can also be supported whereby a global authority whose name is not the root can certify the local roots of independent "islands".

1.2.5 User vs. Node Authentication

In concept, DASS mechanisms support the mutual authentication of two principals regardless of whether those principals are people, computers, or applications. Those mechanisms have been extended, however, to deal with a common case of a pair of principals acting together (a user and a node) authenticating to a single principal (a remote server). This is done by having optionally in each credentials structure two sets of secrets - one for the user and one for the node. When authentication is done using such credentials, both secrets sign the request so the receiving party can verify that both principals are present.

This setup has a number of advantages. It permits access controls to be enforced based on both the identity of the user ***and*** the identity of the originating node. It also makes it possible to define users of systems who have no network wide identities who can access network resources on the basis of node credentials alone. The security of such a setup is less because a node can impersonate all of its users even when they are not logged in, but it offers an easier transition from existing ***.rhosts*** based mechanisms because it does not require creation of global identities for all users.

1.2.6 Protection of User Keys

DASS mechanisms generally deal with authentication between principals each knowing a private key. For principals who are people, special mechanisms are provided for maintaining that private key. In particular, it many cases it will be most convenient to keep passwords as secrets rather than private keys. This architecture specifies a means of storing private keys encrypted under passwords. This would provide security as good as hiding a private key were it not that people tend to choose passwords from a small space (like words in a dictionary) such that a password can be more easily guessed than a private key. To address this potential weakness, DASS specifies a protocol between a login node and a login agent whereby the login agent can audit and limit the rate of password guesses. Use of these features is optional. A user with a smart card could store a private key directly and bypass all of these mechanisms. If users can be forced to choose "good" passwords, the login agent could be eliminated and encrypted credentials could be stored directly in the naming service.

Another way in which user keys are protected is that the architecture does not require that they be available except briefly at login. This reduces the threat of a user walking away from a logged on workstation and having someone take over the workstation and extract his key. It also makes the use of RSA based smart cards practical; the card could keep the user's private key and execute one signature operation at login time to authenticate an entire session.

1.3 What This Document Won't Tell You

Architecture documents are by their nature difficult to read. This one is no exception. The reason is that an architecture document contains the details sufficient to build interoperable implementations, but it is not a design specification. It goes out of its way to leave out any details which an implementation could choose without affecting interoperability. It also does not specify all the uses of the services provided because these services are properly regarded as general purpose tools.

The remainder of this section includes information which is not properly part of the authentication architecture, but which may be useful in understanding why the architecture is the way it is.

1.3.1 How DASS is Embedded in an Operating System

While architecturally DASS does not require any operating system support in order to be used by an application (other than the services listed in Section 2), it is expected that actual implementations of DASS will be closely tied to the operating systems of host computers. This is done both for security and for convenience.

In particular, it is expected that when a user logs into a node, a set of credentials will be created for that user and then associated by the operating system with all processes initiated by or on behalf of the user. When a user delegates to a service, the remote operating system is expected to accept the delegation and start up the remote process with the delegated credentials. Most nodes are expected to have credentials of their own and support the concept of user accounts. When user credentials are created, the node is expected to verify them in its own context, determine the appropriate user account, and add node credentials to the created credentials set.

1.3.2 Forms of Credentials

In the DASS architecture, there is a single data structure called "Credentials" with a large number of optional parts. In an implementation, it is possible that not all of the architecturally allowed subsets will be supported and credentials structures with different subsets of the data may be implemented quite differently.

The major categories of credentials likely to be supported in an implementation are:

- **Claimant credentials** - these are the credentials which would normally be associated with a user process in order that it be able to create authentication tokens. It would contain the user's name, login ticket, session private key, and (at least logically) local node credentials and cached outgoing contexts.

- **Verifier credentials -** these are the credentials which would normally be associated with a server which must verify tokens and produce mutual authentication response tokens. Since servers may be started by a node on demand, some representation of verifier credentials must exist independent of a process. If an operating system wishes to authenticate a request before starting a server process, the credentials must exist in usable form. An implementation may choose to have all services on a "node" share a verifier credentials structure, or it may choose to have each service have its own.

- **Combined credentials** - architecturally, a server may have a structure which is both claimant credentials and verifier credentials combined so that the server may act in either role using a single structure. There is some overlap in the contents. There is no requirement, however, that an implementation support such a structure.

- **Stub credentials** - In the architecture, a credentials structure is created whenever a token is accepted. If delegation took place, these are **claimant credentials** usable by their possessor to create additional tokens. If no delegation took place, this structure exists as an architectural place holder against which an implementation may attempt to authenticate user and node names. An implementation might choose to implement **stub credentials** with a different mechanism than claimant or verifier credentials. In particular, it might do whatever user and node authentication is useful itself and not support this structure at all.

1.3.3 Support for Alternative Certification Authority Implementations

A motivating factor in much of the design of DASS is the need to protect certification authorities from compromise. CAs are only used to create certificates for new principals and to renew them on expiration (expiration intervals are likely to be measured in months). They therefore do not need to be highly available. For maximum security, CAs could be implemented on standalone PCs where the hardware, software, and keys can be locked in a safe when the CA is not in use. The certificates the CA generates must be delivered to the naming service to be registered, and a possible mechanism for this is for the CA to have an RS232 line to an on-line component which can pass certificates and related information but not login sessions. The intent would be to make it implausible to mount a network attack against the CA. Alternatively, certificates could be carried to the network on a floppy disk.

For CAs to be secure, a whole host of design details must be done right. The most important of these is the design of user and system manager interfaces that make it difficult to "trick" a user or system manager into doing the wrong thing and certifying an impostor or revealing a key. Mechanisms for generating keys must also be carefully protected to assure that the generated key cannot be guessed (because of lack of randomness) and is not recorded where a penetrator can get it. Because a certificate contains relatively little human intelligible information (its most important components are UIDs and public keys), it will be a challenge to design a user interface that assures the human operator only authorizes the signing of intented certificates. Such considerations are beyond the scope of the architecture (since they do not affect interoperability), but they did affect the design in subtle ways. In particular, it does not assume uniform security throughout the CA hierarchy and is designed to assure that the compromise of a CA in one part of the hierarchy does not have global implications.

The architecture does not require that CAs be off-line. The CA could be software that can run on any node when the proper secret is installed. Administrative convenience can be gained by integrating the CA with account registration utilities and naming service maintenance. As such, the CA would have to be on-line when in use in order to register certificates in the naming service. The CA key could be unlocked with a password and the password could be entered on each use both to authenticate the CA operator and to assure that compromise of the host node while the CA is not in use will not compromise the CA. This design would be subject to attacks based on planting Trojan horses in the CA software, but is entirely interoperable with a more secure implementation. Realistic tradeoffs must be made between security, cost, and administrative convenience bearing in mind that a system is only as secure as its weakest link and that there is no benefit in making the CA substantially more secure than the other components of the system.

1.3.4 Services Provided vs. Application Program Interface

Section 3 of this document specifies "abstract interfaces" to the services provided by DASS. This means it tells what services are provided, what parameters are supplied by the caller, and what data is returned. It does not specify the calling interfaces. Calling interfaces may be platform, operating system, and language dependent. They do not affect interoperability; different implementations which implement completely different calling interfaces can still interoperate over a network. They do, however, affect portability. A program which runs on one platform can only run on another which implements an identical API.

In order to support portability of applications - not just between implementations of DASS but between implementations of DASS and implementations of Kerberos - a "Generic Security Service API" has been designed and is outlined in Annex B. This API could be the only "published" interface to DASS services. This interface does not, however, give access to all the functions provided by DASS and it provides some non-DASS services. It does not give access to the "login" service, for example, so the login function cannot be implemented in a portable way. Clearly an implementation must provide some implementation of the login function, though perhaps only to one system program and the implementation need not be portable. Similarly, the Generic API provides no access to node authentication information, so applications which use these services may not be portable.

The Generic API provides services for encryption of user data for integrity and possibly privacy. These services are not specified as a part of the DASS architecture. This is because we envisioned that such services would be provided by the communications network and not in applications. These services are provided by the Generic API because these services are provided by Kerberos, there exist applications which use these services, and they are desired in the context of the IETF-CAT work. The DASS architecture includes a Key Distribution service so that the encryption functions of the Generic API can be supported and integrated. Annex B specifies how those services can be implemented using DASS services.

The Services Provided also differ from the GSSAPI because there are important extensions envisioned to the API for future applications and it was important to assure that architecturally those services were available. In particular, DASS provides the ability for a principal to have multiple aliases and for the receiver of an authentication token to verify any one of them. We want DASS to support the case where a server only learns the name it is trying to validate in the course of evaluating an ACL. This may be long after a connection is accepted. The Services Provided section therefore separates the Accept_token function from the Verify Principal Name. The other motivation behind a different interface is that DASS provides node authentication - the ability to authenticate the node from which a request originates as well as the user. Because Kerberos provides no such mechanism, the capability is missing from the GSSAPI, but we expect some applications will want to make use of it.

1.3.5 Use of a Naming Service

With the exception of the syntactical representation of names, which is tied to X.500, the DASS architecture is designed to be independent of the particular underlying naming service. While the intention is that certificates be stored in an X.500 naming service in the fields architecturally reserved for this purpose in the standard, this specification allows for the possibility of different forms of certificate stores. The SPX implementation of DASS implements its own certificate distribution service because we did not want to introduce a dependency on an X.500 naming service.

1.3.6 Key Hiding - Credentials

The abstract interfaces described in section 3 specify that "credentials" and "keys" are the inputs and outputs of various routines. Credentials structures in particular contain secret information which should not be made available to the calling application. In most cases, keeping this information from applications is simply a matter of prudence - a misbehaving application can do nearly as much damage using the credentials as it can by using the secrets directly. Having access to the keys themselves may allow an application to bypass auditing or leak a key to an accomplice who can use it on another node where a large amount of activity is less likely to be noticed. In some cases, most dramatically where a "node key" is present in user credentials, it is vital that the contents of the credentials be kept out of the hands of applications.

To accomplish this, a concrete interface is expected to create "credentials handles" that are passed in and out of DASS routines. The credentials themselves would be kept in some portion of memory where unprivileged code can't get at them.

There is another aspect of the way credentials are used which is important to the design of real implementations. In normal use, a user will create a set of credentials in the process of logging on to a system and then use them from many processes or jobs. When many processes share a set of credentials, it is important for the sake of performance that they share one set of credentials rather than having a copy of the credentials made for each. This is because information is cached in credentials as a side effect of some requests and for good performance those caches should be shared.

As an example, consider a system executing a series of copy commands moving files from one system to another. The credentials of the user will have been established when the user logged on. The first time a copy is requested, a new process will start up, open a connection to the destination system, and create a token to authenticate itself. Creating that token will be an expensive operation, but information will be computed and "cached" in the credentials structure which will allow any subsequent tokens on behalf of that user to that server to be computed cheaply. After the copy completes, the connection is closed and the process terminates. In response to a second copy request, another new process will be created and a new token computed. For this operation to get a performance benefit from the caching, the information computed by the first process must somehow make it to the second.

A model for how this caching might work can be seen in the way Kerberos caches credentials. Kerberos keeps credentials in a file whose name can be computed from the name of the local user. This file is initialized as part of the login process and its protection is set so that only processes running under the UID of the user may read and write the file. Processes cache information there; all processes running on behalf of the user share the file.

There are two problems with this scheme: first, on a diskless node putting information in a file exposes it to eavesdroppers on the network; second, it does not accomplish the "key hiding" function described earlier in this section. In a more secure implementation, the kernel or a privileged process would manage some "pool" of credentials for all processes on a node and would grant access to them only through the DASS calls. Credentials structures are complex and varying length; DASS may organize them as a set of pools rather than as contiguous blocks of data. All such design issues are "beyond the scope of the architecture".

Implementations must decide how to control access to credentials. They could copy the Kerberos scheme of having credentials available to processes with the UID of the login session which created them and to privileged processes or there may be a more elaborate mechanism for "passing" credentials handles from process to process. This design should probably follow the operating system mechanisms for passing around local privileges.

1.3.7 Key Hiding - Contexts

The "GSSAPI" has a concept of a security context which has some of the same key hiding problems as a credentials structure. Security contexts are used in calls to cryptographically protect user data (from modification or from disclosure and modification) using keys established during authentication. The "services provided" specification says that create_ and accept_token return a "shared key" and "instance identifier". The GSSAPI says that a context handle is returned which is an integer. A secure implementation would keep the key and instance identifier in protected memory and only allow access to them through provided interfaces.

Unlike credentials, there is probably no need to provide mechanisms for contexts to be shared between processes. Contexts will normally be associated with some notion of a communications "connection" and ends of a connection are not normally shared. If an implementation chooses to provide additional services to applications like message sequencing or duplicate detection, contexts will have to contain additional fields. These can be created and maintained without any additional authentication services.

1.4 The Relationship between DASS and ISO Standards

This section provides an introduction to DASS authentication in terms of the ISO Authentication Framework (DP10181-2). The purpose of this introduction is to give the reader an intuitive understanding of the way DASS works and how its mechanisms and terminology relate to standards. Important details have been omitted here but are spelled out in section 3.

1.4.1 Concepts

The primary goal of authentication is to prevent impersonation, that is, the pretense to a false identity. Authentication always involves identification in some form. Without authentication, anyone could claim to be whomever they wished and get away with it.

If it didn't matter with whom one was communicating, elaborate procedures for authentication would be unnecessary. However, in most systems, and in timesharing and distributed processing environments in particular, the rights of individuals are often circumscribed by security policy. In particular, authorization (identity based access control) and accountability (audit) provisions could be circumvented if masquerading attempts were impossible to prevent or detect.

Almost all practical authentication mechanisms suitable for use in distributed environments rely on knowledge of some secret information. Most differences lie in how one presents evidence that they know the secret. Some schemes, in particular the familiar simple use of passwords, are quite susceptible to attack. Generally, the threats to authentication may be classified as:

– **forgery,** attempting to guess or otherwise fabricate evidence;

– **replay,** where one can eavesdrop upon another's authentication exchange and learn enough to impersonate them; and

– **interception,** where one slips between the communicants and is able to modify the communications channel unnoticed.

Most such attacks can be countered by using what is known as strong authentication. Strong authentication refers to techniques that permit one to provide evidence that they know a particular secret without revealing even a hint about the secret. Thus neither the entity to whom one is authenticating, nor an eavesdropper on the conversation can further their ability to impersonate the authenticating principal at some future time as the result of an authentication exchange.

Strong authentication mechanisms, in particular those used here, rely on cryptographic techniques. In particular, DASS uses public key cryptography. Note that interception attacks cannot be countered by strong authentication alone, but generally need additional security mechanisms to secure the communication channel, such as data encryption.

1.4.2 Principals and Their Roles

All authentication is on behalf of principals. In DASS the following types of principals are recognized:

– **user principals**, normally people with accounts who are responsible for performing particular tasks. Generally it is users that are authorized to do things by virtue of having been granted access rights, or who are to be held accountable for specific actions subject to being audited.

– **server principals**, which are accessed by users.

– **node principals,** corresponding to locations where users and servers, or more accurately, processes acting on behalf of principals can reside.

Principals can act in one of two capacities:

– the **claimant** is the active entity seeking to authenticate itself, and

– the **verifier** is the passive entity to whom the claimant is authenticating.

Users normally are claimants, whereas servers are usually verifiers, although sometimes servers can also be claimants.

There is another kind of principal:

– **certification authorities** (CA's) issue certificates which attest to another principal's public key.

1.4.3 Representation, Delegation and Representation Transfer

Of course, although it is users that are responsible for what the computer does, human beings are physically unable to directly do anything within a computer system. In point of fact, it is a *process* executing on behalf of a user that actually performs useful work. From the point of view of performing security controlled functions, the process is the agent, or representative, of the user, and is authorized by that user to do things on his behalf. In the terms used in the ISO Authentication Framework, the user is said to have a *representation* in the process.

The representation has to come into existence somehow. **Delegation** refers to the act of creating a representation. A user is said to create a representation for themselves by delegating to a process. If the user creates another process, say by doing an rlogin on a different computer, a representation may be needed there as well. This may be accomplished automatically by a process known as *representation transfer.* DASS uses the term delegation to also mean the act of creating additional representations on a remote systems.

A representation is instantiated in DASS as **credentials.** Credentials include the identity of the principal as well as the cryptographic "state" needed to engage in strong authentication procedures. Claimant information in credentials enable principals to authenticate themselves to others, whereas verifier information in credentials permit principals to verify the claims of others. Credentials intended primarily for use by a claimant will be referred to as *claimant credentials* in the text which follows. Credentials intended primarily for use in verification will be referred to as *verifier credentials*. A particular set of credentials may or may not contain all of the data necessary to be used in both roles. That will depend on the mechanisms by which the credentials were created.

In some contexts, but not here, the concept of representation and/or delegation is sometimes referred to as proxy. This term is used in ECMA TR/46. We avoid use of the term because of possible confusion with an unrelated use of the term in the context of DECnet.

1.4.4 Key Distribution, Replay, Mutual Authentication and Trust

Strong authentication uses cryptographic techniques. The particular mechanisms used in DASS result in the distribution of cryptographic keys as a side effect. These keys are suitable for use for providing a data origin authentication service and/or a data confidentiality service between a pair of authenticated principals.

Replay detection is provided using timestamps on relevant authentication messages, combined with remembering previously accepted messages until they become "stale". This is in contrast to other techniques, such as challenge and response exchanges.

Authentication can be one-way or **mutual.** One-way authentication is when only one party, in DASS the claimant, authenticates to the other. Mutual authentication provides, in addition, authentication of the verifier back to the claimant. In certain communications schemes, for example connectionless transfer, only one-way authentication is meaningful. DASS supports mutual authentication as a simple extension of one-way authentication for use in environments where it makes sense.

DASS potentially can allow many different "trust relationships" to exist. All principals trust one or more CA's to safeguard the certification process. Principals use certificates as the basis for authenticating identities, and trust that CA's which issue certificates act responsibly. Users expect CA's to make sure that certificates (and related secrets) are only made for principals that the CA knows or has properly authenticated on its own.

1.5 An Authentication Walkthrough

The OSI Authentication Framework characterizes authentication as occurring in six phases. This section attempts to describe DASS in these terms.

1.5.1 Installation

In this phase, principal certificates are created, as is the additional information needed to create claimant and verifier credentials. OSI defines three sub-phases:

- **Enrollment.** In DASS, this is the definition of a principal in terms of a key, name and UID.

- **Validation,** confirmation of identity to the satisfaction of the CA, after which the CA generates a certificate.

- **Confirmation.** In DASS, this is the act of providing the user with the certificate and with the CA's own name, key and UID, followed up by the user creating a *trusted authority* for that CA. A trusted authority is a certificate for the CA signed by the user.

Included in this step in DASS is the posting of the certificate so as to be available to principals wishing to verify the principal's identity. In addition, the user principal saves the trusted authority so as to be available when it creates credentials.

1.5.2 Distribution

DASS distributes certificates by placing them in the name service.

1.5.3 Acquisition

Whenever principals wish to authenticate to one another, they access the Name Service to obtain whatever public key certificates they need and create the necessary credentials. In DASS, acquisition means obtaining credentials.

Claimant credentials implement the representation of a principal in a process, or, more accurately, provide a representation of the principal for use by a process. In making this representation, the principal delegates to a temporary delegation key. In this fashion the claimant's long term principal key need not remain in the system.

Claimant credentials are made by invoking the get credentials primitive. Claimant credentials are a DASS specific data structure containing:

- a **name**

- a **ticket,** a data structure containing

  - a validity interval,

  - UID, and

  - (temporary) delegation public key, along with a

  - digital signature on the above made with the principal private key

- the **delegation private key**

Optionally in addition, there may be credential information relating to the node on which the user is logged in and the account on that node. A detailed description of all the information found in credentials can be found in section 3.

Verifier credentials are made with initialize_server. Verifier credentials consist of a principal (long term) private key. The rationale is that these credentials are usually needed by servers that must be able to run indefinitely without reentry of any long term key.

In addition, claimants and verifiers have **a trusted authority**, which consists of information about a trusted CA. That information is its:

- name (this will appear in the "issuer" field in principal certificates),

- public key (to use in verifying certificates issued by that CA), and

- UID.

Trusted authorities are used by principals to verify certificates for other principals' public keys. CAs are arranged in a hierarchy corresponding to the naming hierarchy, where each directory in the naming hierarchy is controlled by a single CA. Each CA certifies the CA of its parent directory, the CAs of each of its child directories, and optionally CAs elsewhere in the naming hierarchy (mainly to deal with the case where the directories up to a common ancestor lack CAs). Even though a principal has only a single CA as a trusted authority, it can securely obtain the public key of any principal in the namespace by "walking the CA hierarchy".

1.5.4 Transfer

The DASS exchange of authentication information is illustrated in Figure 1-1. During the transfer phase, the DASS claimant sends an **authentication token** to the verifier. Authentication tokens are made by invoking the create_token primitive. The authentication token is cryptographically protected and specified as a DASS data structure in ASN.1. The authentication token includes:

- a ticket,
- a DES authenticating key encrypted using the intended verifier's public key
- one of the following:
  - if delegation is not being performed, a digital signature on the encrypted DES key using the delegation private key, or
  - if delegation is being performed, sending the delegation private key, DES encrypted using the DES authenticating key
- an **authenticator,** which is a cryptographic checksum made using the DES authenticating key over a buffer containing
  - a timestamp
  - any application supplied "channel bindings". For example, addresses or other context information. The purpose of this field is to thwart substitution and replay attacks.
- additional optional information concerning node authentication and context.

As a side effect, after init_authentication_context, the caller receives a **local authentication context,** a data structure containing:

- the DES key, and
- if mutual authentication is being requested, the expected response.

In order to construct an authentication token, the claimant needs to access the verifier's public key certificate from the Name Service (labeled CDC, for Certificate Distribution Center, in the figure).

Note that while an authenticator can only be used once, it is permissible to re-establish the same local authentication context multiple times. That is, the ticket and DES key establishment components of the authentication token may have a relatively long lifetime. This permits a performance improvement in that repeated applications of public key operations can be alleviated if one caches authentication contexts, along with other components from a successfully used authentication token and the associated verified principal public key value. It is a relatively inexpensive operation to create (and verify) "fresh" authenticators based on cached authentication context.

```
   Claimant Actions      | Communications |  Verifier Actions
                         |                |
        verifier name    |                |
                |        |                |
                |        |           +---+|
                \------------------- 55->|   ||
  trusted                |           |   ||
authorities              |           |CDC||
     |    +-----------+  |certificate|   ||
     |    |  Verify   |<-------------|   ||
     \--->|Certificate|  |           +---+|
          +-----------+  |                |
  Claimant        |      |                |
credentials    Verifier  |                |   Verifier
     |       Public Key  |                | Credentials
     |            |      |                |       |
     |            V      |                |       V
     |    +-----------+  | Authentication | +-----------+
     |    |   Make    |  |     Token      | |   Check   |  Replay
     \--->|  Token    |-----------------  5-->|  Token   |<-->Cache
          +-----------+  |                | +-----------+
   DES <---/      |      |                |  |   |    \----->DES
   key            |      |                | /Claimant        key
                  |      |                |/Public Key
                  |      |               /      |       trusted
                  |      |      Claimant /|      V      authorities
                  |      |+---+   Name  / | +-----------+     |
         authentication  ||   |<-------/  | |  Verify   |<----/
            context      ||   |certificate| |Certificate|
                  |      ||CDC|------------>|           |-->accept/
                  |      ||   |           | +-----------+   reject
                  |      ||   |           |       |      \
                  |      |+---+           |authentication\
                  V      |     mutual     |   context     V
          +-----------+  | authentication |       |      claimant
       /--|  Accept   |  |    response    | +----------+credentials
      V   |  Mutual   |<------------------- 55--|  Make    |(delegation)
  accept/ +-----------+  |                | | Response |
  reject                 |                | +----------+
                         |                |
```

Figure 1 - Authentication Exchange Overview

1.5.5 Verification

Upon receipt of an authentication token, the verifier extracts the DES key using its verifier credentials, accesses the Name Service (labeled CDC for Certificate Distribution Center) to obtain the certificates needed to perform cryptographic checks on the incoming information, and verifies all of the signatures on the received certificates and the authentication token. Verification can result in creation of new claimant credentials if delegation is performed.

As part of this process, verified authenticators are retained for a suitable timeout period.

1.5.6 Unenrolment

This is the removal of information from the Name Service. The only other form of revocation supported by DASS is certificate timeout. Every certificate contains an expiration time (expected in ordinary use to be about a year from its signing date). DASS does not currently support the revocation lists in X.509.

2 Services Used

Aside from operating system services needed to maintain its internal state, DASS relies on a global distributed database in which to store its certificates, a reliable source of time, and a source of random numbers for creating cryptographic keys.

2.1 Time Service

DASS requires access to the current time in several of its algorithms. Some of its uses of time are security critical. In others, network synchronization of clocks is required. DASS does ***not***, however, depend on having a single source of time which is both secure and tightly synchronized.

The requirements on system provided time are:

- For purposes of validating certificates and tickets, the system needs access to know the date and time accurate to within a few hours with no particular synchronization requirements. If this time is inaccurate, then valid requests may be rejected and expired messages may be accepted. Certificate expiration is a backup revocation mechanism, so this can only cause a security compromise in the event of multiple failures. In theory, this could be provided by having a local clock on every node accurate to within a few hours over the life of the product to provide this function. If an insecure network time service is used to provide this time, there are theoretical security threats, but they are expected to be logistically impractical to exploit.

- For purposes of detecting replay of authentication tokens, the system needs access to a ***strictly monotonic*** time source which is reasonably synchronized across the network (within a few minutes) for the system to work, but inaccuracy does not present a security threat except as noted below. It may constitute an availability threat because valid requests may be rejected. In order to get strict monotonicity in the presence of a rapid series of requests, time must be returned with high precision. There is no requirement for a high degree of accuracy. Inaccurate time could present a security threat in the following scenario: if a client's clock is made sufficiently fast that its tokens are rejected, someone harvesting those tokens from the wire could replay them later and impersonate the client. In some environments, this might be an easier threat than harvesting tokens and preventing their delivery.

- For purposes of aging stale entries from caches, DASS requires reasonably accurate timing of ***intervals***. To the extent that intervals are reported as shorter than the actually were, revocation of certificates from the naming service may not be as timely as it should be.

2.2 Random Numbers

In order to generate keys, DASS needs a source of "cryptographic quality" random numbers. Cryptographic quality means that knowing any of the "random numbers" returned from a series and knowing all state information which is not protected, an attacker cannot predict any of the other numbers in the series. Hardware sources are ideal, but there are alternative techniques which may also be acceptable. A 56 bit "truly random" seed (say from a series of coin tosses) could be used as a DES key to encrypt an infinite length known text block in CBC mode to produce a pseudo-random sequence provided the key and current point in the sequence were adequately protected. There is considerable controversy surrounding what constitutes cryptographic quality random numbers, and it is not a goal of this document to resolve it.

2.3 Naming Service

DASS stores creates and uses "certificates" associated with every principal in the system, and encrypted credentials associated with most. This information is stored in an on-line service associated with the principal being certified. The long term vision is for DASS to use an X.500 naming service, and DASS will from its inception authenticate X.500 names. To avoid a dependence on having an X.500 naming service available (and to gain the benefits of a "login agent" that controls password guessing), an alternative certificate distribution center protocol is also described.

The specific requirements DASS places on the naming service are:

- It must be highly available. A user’s naming service entry must be available to any node where the user is to obtain services (or service will be denied). A server’s naming service entry must be available from any node from which the service is to be invoked (or service will be denied).

- It must be timely. The presence of "stale" information in the naming service may cause some problems. When a password changes, the old password may remain valid (and the new password invalid) to the extent the naming service provides stale information. When a user or server is added to the network, it will not be able to participate in authentication until the information added to the naming service is available at the node doing the authentication. In the unusual circumstance that a key changes, the entity whose key has changed will not be able to use the new key until the new certificate is uniformly available.

- It must be secure with regard to certain specific properties. In general, the security of DASS protected applications does not depend on the security of the naming service. It is expected that the availability needs of the naming service will prevent it from being as secure as some applications need to be. There are two aspects of DASS security which do depend on the security of the naming service: timely revocation of certificates and protection of user secrets against dictionary based password guessing. DASS depends on the removal of certificates from the naming service in order to revoke them more quickly than waiting for them to time out. For this mechanism to provide any actual security, it must not be possible for a network entity to "impersonate" the naming service and the naming service must be able to enforce access controls which prevent a revoked certificate from being reinstated by an unauthorized entity. In the long run, it is expected that DASS itself will be used to secure the naming service, which presents certain potential recursion problems (to be addressed in the naming service design). If the naming service is not authenticated (as is expected in early versions) attacks where a revoked certificate is "reinstated" through impersonation of the naming service are possible.

The specific functions DASS requests of the naming service are simple:

- Given an X.500 name, store a set of certificates associated with that name.

- Given an X.500 name, retrieve the set of certificates associated with that name.

- Given an X.500 name, store a set of encrypted credentials associated with that name.

- Given and X.500 name, retrieve a set of encrypted credentials associated with that name.

Implementation over a particular naming service may implement more specialized functions for reasons of efficiency. For example, the certificates associated with a name may be separated into several sets (child, parent, cross, self) so that only the relevant ones may be retrieved. In order that access to the naming service itself be secure, the protocols should be authenticated. Certificates should generally be readable without authentication in order to avoid recursion problems. Requests to read encrypted credentials should be specialized and should include proof of knowledge of the password in order that the naming service can audit and slow down false password guesses.

The following sections describe the interfaces to specific naming services:

2.3.1 Interface to X.500

Certificates associated with a particular name are stored as attributes of the entry as specified in X.509. X.509 defines attributes appropriate for parent and cross certificates (CrossCertificatePair, CACertificate) for some principals; we will have to define a DASSUserPrincipal object class including these attributes in order to properly use them with ordinary users. Retrieval is via normal X.500 protocols. Certificates should be world readable and modifiable only by appropriate authorities.

Encrypted credentials are stored with the entry of the principal under a yet to be defined attribute. The credentials should be encoded as specified in section 4. In the absence of extensions to the X.500 protocol to control password guessing, the encrypted credentials should be world readable and updatable only by the named principal and other appropriate authorities.

2.3.2 Interface to CDC

The CDC (Certificate Distribution Center) is a special purpose name server created to service DASS until an X.500 service is available in all of the environments where DASS needs to operate. The CDC uses a special purpose protocol to communicate with DASS clients. The protocol was designed for efficiency, simplicity, and security. CDCs use DASS as an authentication mechanism and to protect encrypted credentials from unaudited password guessing.

Each DASS client maintains a list of CDCs and the portion of the namespace served by that CDC. Each directory has a master replica which is the only one which will accept updates. The CDCs maintain consistency with one another using protocols beyond the scope of this document. When a DASS client wishes to make a request of a CDC, it opens a TCP or DECnet connection to the CDC and sends an ASN.1 (BER) encoded request and receives a corresponding ASN.1 (BER) encoded response. Clients are expected to learn the IP or DECnet address and port number of the CDC supporting a particular name from a local configuration file. To maximize performance, the requests bundle what would be several requests if made in terms of requests for individual certificates. It is intended that all certificates needed for an authentication operation be retrievable with at most two CDC requests/responses (one to the CDC of the client and one to the CDC of the server).

Documented here is the protocol a DASS client would use to retrieve certificates and credentials from a CDC and update a user password. This protocol does not provide for updates to the certificate and credential databases. Such updates must be supported for a practical system, but could be done either by extensions to this protocol or by local security mechanisms implemented on nodes supporting the CDC. Similarly, availability can be enhanced by replicating the CDC. Automating the replication of updates could be implemented by extensions to this protocol or by some other mechanism. This specification assumes that updates and replication are local matters solved by individual CA/CDC implementations.

Requests and responses are encoded as follows:

2.3.2.1 ReadPrinCertRequest

This request asks the CDC to return the child certificates and selected incoming cross certificates for the specified object. The format of the request is:

```
ReadPrinCertRequest ::= [4] IMPLICIT SEQUENCE {
      flags [0] BIT STRING DEFAULT {},
      index [1] IMPLICIT INTEGER DEFAULT 0,
```

```
        resolveFrom [2] Name OPTIONAL,
        principal Name,
        crossCertIssuers ListOfIssuers OPTIONAL
        }

ListOfIssuers ::= SEQUENCE OF Name
```

The first 24 bits of **flags,** if present, contain a protocol version number. Clients following this spec should place the value 2.0.0 in the three bytes. Servers following this spec should accept any value of the form 1.x.x or 2.x.x. **flags** bits beyond the first 24 are reserved for future use (should not be supplied by clients and should be ignored by servers).

index is only used if the response exceeds the size of a single message; in that case, the query is repeated with **index** set to the value that was returned by ReadPrinCertResponse.

resolveFrom and **principal** imply a set of entities for which certificates should be retrieved. **resolveFrom** (if present) must be an ancestor of **principal** and child certificates will be retrieved for **principal** and all names which are ancestors of **principal** but descendants of **resolveFrom.** The encoding of names is per X.500 and is specified in more detail in section 4. The CDC returns the certificates in order of the object they came from, parents before children.

crossCertIssuers is a list of cross certifiers that would be believed in the context of this authentication. If supplied, the CDC may return a chain of certificates starting with one of the named **crossCertIssuers** and ending with the named **principal.** One of **resolveFrom** or **crossCertIssuers** must be present in any request; if both are present, the CDC may return either chain.

2.3.2.2 ReadPrinCertResponse

This is the response a CDC sends to a ReadPrinCertRequest. Its syntax is:

```
ReadPrinCertResponse ::= [5] IMPLICIT SEQUENCE {
        status [0] IMPLICIT CDCstatus DEFAULT success,
        index [1] INTEGER OPTIONAL,
        resolveTo [2] Name OPTIONAL,
        certSequence [3] IMPLICIT CertSequence,
        indexInvalidator [4] OCTET STRING (SIZE(8))
                                    OPTIONAL,
        flags [5] BIT STRING OPTIONAL
        }

CertSequence ::= SEQUENCE OF Certificate
```

status indicates success or the cause of the failure.

index if present indicates that the request could not be fully satisfied in a single request because of size limitations. The request should be repeated with this index supplied in the request to get more.

resolveTo will be present if **index** is present and should be supplied in the request for more certificates.

certSequence contains certificates found matching the search criteria.

indexInvalidator may be present and indicates the version of the database being read. If a set of certificates is being read in multiple requests (because there were too many to return in a single message), the reader should check that the value for indexInvalidator is the same on each request. If it is not, the server may have skipped or duplicated some certificates. This field must not be present if the version number in the request was missing or version 1.x.x.

The first 24 bits of **flags,** if present, indicate the protocol version number. Implementers of this version of the spec should supply 2.0.0 and should accept any version number of the form 1.x.x or 2.x.x.

2.3.2.3 ReadOutgoingCertRequest

This requests from the CDC a list of all parent and outgoing cross certificates for a specified object. A CDC is capable of storing cross certificates either with the subject or the issuer of the cross certificate. In response to this request, the CDC will return all parent and cross certificates stored with the issuer for the named principal and all of its ancestors. Its syntax is:

```
ReadOutgoingCertRequest ::= [6] IMPLICIT SEQUENCE {
      flags [0] BIT STRING DEFAULT {},
      index [1] IMPLICIT INTEGER DEFAULT 0,
      principal Name
      }
```

The first 24 bits of **flags** is a protocol version number and should contain 2.0.0 for clients implementing this version of the spec. Servers implementing this version of the spec should accept any version number of the form 1.x.x or 2.x.x. The remaining bits are reserved for future use (they should not be supplied by clients and they should be ignored by servers).

index is used for continuation (see ReadPrinCertRequest).

principal is the name for which certificates are requested.

2.3.2.4 ReadOutgoingCertResponse

This is the response to a ReadOutgoingCertRequest. Its syntax is:

```
ReadOutgoingCertResponse::= [7] IMPLICIT SEQUENCE {
      status [0] IMPLICIT CDCStatus DEFAULT success,
      index [1] INTEGER OPTIONAL,
      certSequence [2] IMPLICIT CertSequence,
      indexInvalidator [3] OCTET STRING (SIZE(8))
                                       OPTIONAL,
      flags [4] BIT STRING OPTIONAL
      }

CertSequence ::= SEQUENCE OF Certificate
```

status indicates success of the cause of failure of the operation.

index is used for continuation; see ReadPrinCertRequest.

certSequence is the list of parent and outgoing cross certificates.

indexInvalidator is used for continuation; see ReadPrinCertResponse (the same rules apply with respect to version numbers).

The first 24 bits of **flags,** if present, contain the protocol version number. Clients implementing this version of the spec should supply the value 2.0.0. Servers should accept any values of the form 1.x.x or 2.x.x. The remaining bits are reserved for future use (they should not be supplied by clients and should be ignored by servers).

2.3.2.5 ReadCredentialRequest

This request is made to retrieve an principal's encrypted credentials. To prevent unaudited password guessing, this structure includes an encrypted value that proves that the requester knows the password that will decrypt the structure. The syntax of the request is:

```
ReadCredentialRequest ::= [2] IMPLICIT SEQUENCE {
      flags [0] BIT STRING DEFAULT {}
      principal Name,
      logindata [2] BIT STRING DEFAULT {},
      token [3] BIT STRING OPTIONAL
      }
```

The first 24 bits of **flags** contains the version number of the protocol. The value 2.0.0 should be supplied. Any value of the form 1.x.x or 2.x.x should be accepted. Any additional bits are reserved for future use (should not be supplied by clients and should be ignored by servers).

principal is the name of the principal for whom encrypted credentials are desired.

logindata is an encrypted value. It may only be present if the version number is 2.0.0 or higher. It must be present to read credentials which are protected by the login agent functionality of the CDC. It is constructed as a single RSA block encrypted under the public key of the CDC. The public key of the CDC is learned by some local means. Possibilities include a local configuration file or by using DASS to read and verify a chain of certificates ending with the CDC [the CDC serving a directory should have its public key listed under a name consisting of the directory name with the RDN "CSS=X509"; the OID for the type CSS is 1.3.24.9.1]. The contents of the block are as follows:

- The low order eight bytes contain a randomly generated DES key with the last byte of the DES key placed in the last byte of the RSA block. This DES key will be used by the CDC to encrypt the response. Key parity bits are ignored.

- The next to last eight bytes contain a long Posix time with the integer time encoded as a byte string using big endian order.

- The next eight bytes (from the end) contain a hash of the password. The algorithm for computing this hash is listed in section 4.4.2. The CDC never computes this hash; it simply compares the value it receives with the value associated with the credentials.

- The next sixteen bytes (from the end) contain zero.

- The remainder of the RSA block (which should be the same size as the public modulus of the CDC) contains a random number. The first byte should be chosen to be non-zero but so the value in the block does not exceed the RSA modulus. Servers should ignore these bits. This random number need not be of cryptographic strength, but should not be the same value for all encryptions. Repeating the DES key would be adequate.

- The byte string thus constructed is encrypted using the RSA algorithm by treating the string of bytes as a "big endian" integer and treating the integer result as "big endian" to make a string of bytes.

token will not be present in the initial implementation but a space is reserved in case some future implementation wants to authenticate and audit the node from which a user is logging in.

2.3.2.6 ReadCredentialProtectedResponse

This is the second possible response to a ReadPrinLoginRequest. It is returned when the encrypted credentials are protected from password guessing by the CDC acting as a login agent. Its syntax is:

```
ReadCredentialProtectedResponse::=[16] IMPLICIT SEQUENCE {
        status [0] IMPLICIT CDCStatus DEFAULT success,
        encryptedCredential [1] BIT STRING,
        flags [2] BIT STRING OPTIONAL
        }
```

status indicates that the request succeeded or the cause of the failure.

encryptedCredential contains the DASSPrivateKey structure (defined in section 4.1) encrypted under a DES key computed from the user's name and password as specified in section 4.4.2 and then reencrypted under the DES key provided in the ReadPrinLoginRequest.

The first 24 bits of **flags**, if present, contains the version number of the protocol. Implementers of this version of the spec should supply 2.0.0 and should accept any version number of the form 2.x.x. Other bits are reserved for future use (they should not be supplied and they should be ignored).

2.3.2.7 WriteCredentialRequest

This is a request to update the encrypted credential structure. It is used when a user's key or password changes. The syntax of the request is:

```
WriteCredentialRequest ::= [17] IMPLICIT SEQUENCE {
      flags [0] BIT STRING DEFAULT {},
      authtoken [2] BIT STRING OPTIONAL,
      principal [3] Name,
      logindata [4] BIT STRING DEFAULT {},
      furtherSensitiveStuff [5] BIT STRING
      }
```

The first 24 bits of **flags** is a version number. Clients implementing this version of the spec should supply 2.0.0. Servers should accept any value of the form 2.x.x. Additional bits are reserved for future use (clients should not supply them and servers should ignore them).

token, if present, authenticates the entity making the request. A request will be accepted either from a principal proving knowledge of the password (see **logindata** below) or a principal presenting a token in this field and satisfying the authorization policy of the CDC. This field need not be present if logindata includes the hash2 of the password (anyone knowing the old password may set a new one).

principal is the name of the object for which encrypted credentials should be updated.

logindata is encrypted as in ReadPrinLoginRequest. It proves that the requester knows the old password of the principal to be updated (unless the token supplied is from the user's CA) and includes the key which encrypts furtherSensitiveStuff.

furtherSensitiveStuff is an encrypted field constructed as follows:

- The first eight bytes consist of the hash2 defined in section 4.4.2 with the last byte of the hash2 value stored first. The CDC stores this value and compares it with the values supplied in future requests of **ReadCredentialRequest** and **WriteCredentialRequest.**

- The next (variable number of) bytes contains a DASSPrivateKey structure (defined in section 4.1). This is the new credential structure that will be returned by the CDC on future **ReadCredentialRequest**s.

- The result is padded with zero bytes to a multiple of eight bytes.

- The entire padded string is encrypted using the key from **logindata** or **token** using DES in CBC mode with zero IV.

the new eight byte "hash2" defined in section 4.4.2 concatenated with the DASSPrivateKey structure encrypted under the new "hash1" all encrypted under the DES key included in logindata.

2.3.2.8 HereIsStatus

This is the response message to ill-formed requests and requests that only return a status and no data. It's syntax is:

```
HereIsStatus ::= [1] IMPLICIT SEQUENCE {
      status [0] IMPLICIT CDCStatus DEFAULT success
      }
```

status indicates success or the cause of the failure.

2.3.2.9 Status Codes

The following are the CDCStatus codes that can be returned by servers. Not all of these values are possible with all calls, and some of the status codes are not possible with any of the calls described in this document.

```
CDCStatus ::= INTEGER {

      success(0),

      accessDenied(1),

      wrongCDC(2),         --this CDC does not store the

                           --requested information

      unrecognizedCA(3),

      unrecognizedPrincipal(4),

      decodeRequestError(5),     --invalid BER

      illegalRequest(6),   --request not recognised

      objectDoesNotExist(7),

      illegalAttribute(8),

      notPrimaryCDC(9),    --write requests not accepted

                           --at this CDC replica

      authenticationFailure(11),

      incorrectPassword(12),

      objectAlreadyExists(13),

      objectWouldBeOrphan(15),

      objectIsPermanent(16),

      objectIsTentative(17),
```

```
parentIsTentative(18),
certificateNotFound(19),
attributeNotFound(20),
ioErrorOnCertifDatabase(100),
databaseFull(101),
serverInternalError(102),
serverFatalError(103),
insufficientResources(104)
}
```

3 Services Provided

This section specifies the services provided by DASS in terms of abstract interfaces and a model implementation. A particular implementation may support only a subset of these services and may provide them through interfaces which combine functions and supply some parameters implicitly. The specific calling interfaces are in some cases language and operating system specific. An actual implementation may choose, for example, to structure interfaces so that security contexts are established and then passed implicitly in calls rather than explicitly including them in every call. It might also bundle keys into opaque structures to be used with supplied encryption and decryption routines in order to enhance security and modularity and better comply with export regulations. Annex B describes a Portable API designed so that applications using a limited subset of the capabilities of DASS can be easily ported between operating systems and between DASS and Kerberos based environments. The model implementation describes data structures which include cached values to enhance performance. Implementations may choose different contents or different caching strategies so long as the same sequence of calls would produce the same output for some caching policy.

DASS operates on four kinds of data structures: Certificates, Credentials, Tokens, and Certification Authority State. Certificates and Tokens are passed between implementations and thus their exact format must be architecturally specified. This detailed bit-for-bit specification is in section 4. Credentials generally exist only within a single node and their format is therefore not specified here. The contents of all of these data structures is listed below followed by the algorithms for manipulating them.

There are three kinds of services provided by DASS: Certificate Maintenance, Credential Maintenance, and Authentication. The first two kinds exist only in support of the third. Certificate maintenance functions maintain the database of public keys in the naming service. These functions tend to be fairly specialized and may not be supported on all platforms. Before authentication can take place, both authenticating principals must have constructed credentials structures. These are built using the Credential Maintenance calls. The Authentication functions use credential information and certificates, produce and consume authentication tokens and tell the two communicating parties one another's names.

3.1 Certificate Contents

For purposes of this architecture, a certificate is a data structure posted in the naming service which proclaims that knowledge of the private key associated with a stated public key authenticates a named principal. Certificates are "signed" by some authority, are readable by anyone, and can be verified by anyone knowing the public key of the authority.

DASS organizes the CA trust hierarchy around the naming hierarchy. There exists a trusted authority associated with each directory in the naming hierarchy. Generally, each authority creates certificates stating the public keys of each of its children (in the naming hierarchy) and the public key of its parent (in the naming hierarchy). In this way, anyone knowing the public key of any authority can learn the public key of any other by "walking the tree". In order that principals may authenticate even when all of their ancestor directories do not participate in DASS, authorities may also create "cross-certificates" which certify the public key of a named entity which is not a descendent. Rules for finding and following these cross-certificates are described in the Get_Pub_Keys routines. Every principal is expected to know the public key of the CA of the directory in which it is named. This must be securely learned when the principal is initialized and may be maintained in some form of local storage or by having the principal sign a certificate listing the name and public key of its parent and posting that certificate in the naming service.

The syntax and content of DASS certificates are defined in terms of X.509 (Directory - Authentication Framework). While that standard prescribes a single syntax for certificates, DASS considers certificates to be of one of six types:

- Normal Principal certificates are signed by a CA and certify the name and public key of a principal where the name of the CA is a prefix of the name of the principal and is one component shorter.

- Trusted Authority certificates are signed by an ordinary principal and certify the name and public key of the principal's CA (i.e. the CA whose name is a prefix of the principal's name and is one component shorter).

- Child certificates are signed by a CA and certify the name and public key of a CA of a descendent directory (i.e. where the name of the issuing CA is a prefix of the name of the subject CA and is one component shorter).

- Parent certificates are signed by a CA and certify the name and public key of the CA of its parent directory (i.e. whose name is a prefix of the name of the issuer and is one component shorter).

- Cross certificates are signed by a CA and certify the name and public key of a CA of a directory where neither name is a prefix of the other.

- Self certificates are signed by a principal or a CA and the issuer and subject name are the same. They are not used in this version of the architecture but are defined as a convenient data structure in which in which implementations may insecurely pass public keys and they may be used in the future in certain key roll-over procedures.

It is intended that some future version of the architecture relax the restrictions above where prefixes must be one component shorter. Being able to handle certificates where prefixes are two or more components shorter complicates the logic of treewalking somewhat and is not immediately necessary, so such certificates are disallowed for now.

The syntax of certificates is defined in section 4. For purposes of the algorithms which follow, the following is the portion of the content which is used (names in brackets refer to the field names in the ASN.1 encoded structure):

- UID of the issuer (optional)

- Full name of the issuer (the authority or principal signing) [issuer]

- UID of the subject (optional)

- Full name of the subject (the authority or principal whose key is being certified) [subject]

- Public Key of the subject [subjectPublicKey]

- Period of validity (effective date and expiration date) [valid]

- Signature over the entire content of the certificate created using the private key of the issuer.

When parsing a certificate, the reader compares the two name fields to determine what type of certificate it is. For Parent and Trusted Authority certificates, the names are ignored for purposes of all further processing. For Child and Normal Principal certificates, only the suffix by which the child's name is longer than the parent's is used for further processing. The reason for this is so that if a branch of the namespace is renamed, all of the certificates in the moved branch remain valid for purposes of DASS processing. The only purposes of having full names in these certificates are (1) to comply with X.509, (2) for possible interoperability with other architectures using different algorithms, and (3) to allow principals to securely store their own names in trusted authority certificates in the case where they do not have enough local storage to keep it.

3.2 Encrypted Private Key Structure

In order that humans need only remember a password rather than a full set of credentials, and also to make installation of nodes and servers easier, there is a defined format for encrypting RSA secrets under a password and posting in the naming service. This structure need only exist when passwords are used to protect RSA secrets; for servers which keep their secrets in non-volatile memory or users who carry smart cards, they are unnecessary.

This structure includes the RSA private/public key pair encrypted under a DES key. The DES key is computed as a one-way hash of the password. This structure also optionally includes the UID of the principal. It is needed only if a single RSA key is shared by multiple principals (with multiple UIDs).

Since this structure is posted in the name service and may be used by multiple implementations, its format must be architecturally defined. The exact encoding is listed in section 4.

3.3 Authentication Tokens

This section of the document defines the contents of the authentication tokens which are produced and consumed by Create_token and Accept_token. With DASS, the token passed from the client to the server is complex, with a large number of optional parts, while the token passed from server to client (in the case of mutual authentication only) is small and simple.

The authentication token potentially contains a large number of parts, most of which are optional depending on the type of authentication. The following defines the content and purpose of each of the parts, but does not describe the actual encoding (in the belief that such details would be distracting). The encoding is in section 4.

The authentication process begins when the initiator calls Create_token with the name of the target. This routine returns an authentication token, which is sent to the target. The target calls Accept_token passing it the token. Both routines produce a second "mutual authentication token". The target returns this to the initiator to prove that it received the token.

3.3.1 Initial Authentication Token

The components of the initial authentication token are (names in brackets refer to the field names within the ASN.1 encoded structures defined in section 4):

a) Encrypted Shared Key - [authenticatingKey] - This is a Shared (DES) key encrypted under the public key of the target. Also included in the encrypted structure is a validity interval and a recognizable pattern so that the receiver can tell whether the decryption was successful.

b) Login Ticket - [sourcePrincipal.userTicket] - This is a "delegation certificate" signed by a principal's long term private key delegating to a short term public key. Its "active ingredients" are:

   1) UID of delegating principal [subjectUID]

   2) Period of validity [validity]

3) Delegation public key [delegatingPublicKey]

4) Signature by private key of principal
The existence of this signature is testimony that the private key corresponding to the delegation public key speaks for the user during the validity interval.

This data structure is optional and will be missing if the authentication is only on behalf of a Local Username on a node (i.e. proxy) rather than on behalf of a real principal with a real key.

c) Shared Key Ticket - [sourcePrincipal.sharedKeyTicketSignature] - This is a signature of the Encrypted Shared Key by the Delegation Public key in the Login Ticket. The existence of this signature is testimony that the DES key in the encrypted shared key speaks for the user.

This data structure is optional and will be missing if the authentication is only on behalf of a Local Username on a node (i.e. proxy) rather than on behalf of a real principal with a real key. It will also be missing if delegation is taking place.

d) Node Ticket - [sourceNode.nodeTicketSignature] - This is a signature of the Encrypted Shared key and a "Local Username" on the host node by the node’s private key. The existence of this signature is testimony by the node that the DES key in the encrypted shared key speaks for the named account on that node.

e) Delegator - [sourcePrincipal.delegator] - This data structure contains the private login key encrypted under the Shared key. It is optional and is present only if the initiator is delegating to the destination.

f) Authenticator - [authenticatorData] - This data structure contains a timestamp and a message digest of the channel bindings signed by the Shared Key. It is always present.

g) Principal name - [sourcePrincipal.userName] - This is the name of the initiating principal. It is optional and will be missing for strong proxy where bits on the wire are at a premium and where the destination is capable of independently constructing the name.

h) Node name - [sourceNode.nodeName] - This is the name of the initiating node. It is optional and will be missing for strong proxy where bits on the wire are at a premium and the name is present elsewhere in the message being passed.

i) Local Username - [sourceNode.username] - This is the local user name on the initiating node. It is optional and will be missing for strong proxy where bits on the wire are at a premium and where the name is present elsewhere in the message being passed.

3.3.2 Mutual Authentication Token

The authentication buffer sent from the target to the initiator (in the case of mutual authentication) is much simpler. It contains only the timestamp taken from the authenticator encrypted under the Shared Key. It is ASN.1 encoded to allow for future extensions.

3.4 Credentials

DASS organizes its internal state with Credentials structures. There are many kinds of information which can be stored in credentials. Rather than making a different kind of data structure for each kind of data, DASS provides a single credentials structure where most of its fields are optional.

Operating systems must provide some mechanism for having several processes share credentials. An example of a mechanism for doing this would be for credentials to be stored in a file and the name of the file is used as a "handle" by all processes which use those credentials. Some of the calls which follow cause credentials structures to be updated. It is important to the performance of a system that updates to credentials (such as occur during the routines

Verify_Principal_Name and Verify_Node_Name, where the caches are updated) be visible to all processes sharing those credentials.

In many of the calls which follow, the credentials passed may be labeled: claimant credentials, verifier credentials or some such. This indicates whose credentials are being passed rather than a type of credentials. DASS supports only one type of credentials, though the fields present in the credentials of one sort of principal may be quite different from those present in the credentials of another.

An implementation may choose to support multiple kinds of credentials structures each of which will support only a subset of the functions available if it is not implementing the full architecture. This would be the case, for example, if an implementation did not support the case where a server both received requests from other principals and made requests on its own behalf using a single set of credentials.

The following are a list of the fields that may be contained in a credentials structure. They are grouped according to common usage.

3.4.1 Claimant information

This is the information used when the holder of these credentials is requesting something. It includes:

a) Full X.500 name of the principal

b) Public Key of the principal

c) Login Ticket - a login ticket contains:

   1) the UID of the principal

   2) a period of validity (effective date & expiration date)

   3) a delegation public key

   4) a signature of the ticket contents by the principal's long term key

d) Delegation Private Key (corresponding to the public key in c3)

e) Encrypted Shared Key (present only when credentials were created by accept_token; this information is needed to verify a node ticket after credentials are accepted)

3.4.2 Verifier information

This is the information needed by a server to decrypt incoming requests. It is also used by generate_server_ticket to generate a login ticket.

a) RSA private key.

3.4.3 Trusted Authority

This is information used to seed the walk of the CA hierarchy to reliably find the public key(s) associated with a name. Normally, the trusted authority in a set of credentials will be the directory parent of the principal named in Claimant information. In some circumstances, it may instead be the directory parent of the node on which the credentials reside.

a) Full X.500 name of a CA

b) Corresponding RSA Public Key

c) Corresponding UID

3.4.4 Remote node authentication

This information is present only for credentials generated by "Accept_token". It includes information about any remote node which vouched for the request.

a) Full X.500 name of the node

b) Local Username on the node

c) Node ticket.

3.4.5 Local node credentials

This information is added by Combine_credentials, and is used by Create_token to add a node signature to outbound requests.

a) Full X.500 name of the node

b) Local Username on the node

c) RSA private key of the node

3.4.6 Cached outgoing contexts

There may be one (or more) such structures for each server for which this principal has created authentication tokens. These represent a cache: they may be discarded at any time with no effect except on performance. For each association, the following information is kept:

a) Destination RSA Public Key (index)

b) Encrypted Shared key

c) Shared Key Ticket (optional, included if there has been a non-delegating connection)

d) Node Ticket

e) Delegator (optional, included if there has been a delegating connection)

f) Validity interval

g) Shared Key

3.4.7 Cached Incoming Contexts

There may be one such structure for each client from which this server has received an authentication token.[1] These represent a cache: they may be discarded at any time with no effect except on performance. For each association, the following information is kept:

a) Encrypted Shared key (index)

b) Shared Key

[1] An implementation may choose to keep one System-wide Cache (and list of incoming timestamps). While it is unlikely that the same Encrypted Shared Key will result from encryption of Shared keys generated by different clients or for different servers, an implementation must ensure that an entry made for one client/server can not be reused by another client/server. Similarly an implementation may choose to keep separate caches for the Shared Key/Validity Interval/Delegation Public Key, the Nodename/UID/key/username and the Principal name/UID/key.

c) Validity Interval

d) Full X.500 name of Client Principal

e) UID of Client Principal

f) Public Key of Client Principal

g) Name of Client Node

h) UID of Client Node

i) Public Key of Client Node

j) Local Username on Client node

k) Delegation Public key of Client Principal's Login Ticket

The Name, UID and Public key of the Principal are all entered together once the Login Ticket has been verified. Similarly the Node name, Node key and Username are entered together once the Node Ticket has been verified. These pieces of information are only present if they have been verified.

3.4.8 Received Authenticators

A record of all the authenticators received is kept. This is used to detect replayed messages[2]. The entries in this list may be deleted when the timestamp is old enough that they would no longer be accepted. This list is kept separate from the Cached incoming context in order that the information in the cached incoming context can be discarded at any time. An implementation could choose to save these timestamps with the cached incoming context if it ensures that it can never purge entries from the cache before the timestamp has aged sufficiently. This list is accessed based on an extract from the signature from the Authenticator. The extract must be at least 64 bits, to ensure that it is very unlikely that 2 authenticators will be received with matching signatures.

a) Extract from Signature from Authenticator

b) Timestamp

If an implementation runs out of space to store additional authenticators, it may either reject the token which would have overflowed the table or it may temporarily narrow the allowed clock skew to allow it to free some of the space used to hold "old" authenticators. The first strategy will always falsely reject tokens; the second may cause false rejection of tokens if the allowed clock skew gets narrowed beyond the actual clock skew in the network.

3.5 CA State

The CA needs to maintain some internal state in order to generate certificates. This internal state must be protected at all times, and great care must be taken to prevent its being disclosed. A CA may choose to maintain additional state information in order to enhance security. In particular, it is the responsibility of the CA to assure that the same UID is not serially reused by two holders of a single name. In most cases, this can be done by creating the UID at the time the user is registered. To securely permit users to keep their UIDs when transferring from another CA, the CA must keep a record of any UIDs used by previous holders of the name. Since actions of a CA are so security sensitive, the CA should also maintain an audit trail of all certificates signed so that a history can be reconstructed in the event of a compromise. Finally, for the convenience of the CA operator, the CA should record a list of the directories for which it is responsible and their UIDs so that these need not be entered whenever the CA is to be used. The state includes at least the following information:

- Public Key of CA

[2] This list must be common to all targets that could accept the same authenticator (channel bindings will prevent other targets from accepting the same authenticator). This includes different 'servers' sharing the same key.

- Private Key of CA
- Serial number of next certificate to be issued

3.6 Data types used in the routines

There are several abstract data types used as parameters to the routines described in this section. These are listed here

a) Integer

b) Name
Names unless otherwise noted are always X.500 names. While most of the design of DASS is naming service independent, the syntax of certificates and tokens only permits X.500 names to be used. If DASS is to be used in an environment where some other form of name is used, those names must be translated into something syntactically compliant with X.500 using some mechanism which is beyond the scope of this architecture. The only other form of name appearing in this architecture is a "local user name", which corresponds to the simple name of an "account" on a node. As a type, such names appear in parameter lists as "Strings".

c) String
A String is a sequence of printable characters.

d) Absolute Time
A UTC time. The precision of these Times is not stated. A precision of the order of one second in all times is sufficient.

e) Time Interval
A Time interval is composed of 2 times. A Start Time and an End Time, both of which are Absolute Times

f) Timestamp
A Timestamp is a time in POSIX format. I.e. two 32 bit Integers. The first representing seconds, and the second representing nanoseconds.

g) Duration
A Duration is the length of a time interval.

h) Octet String
A sequence of bytes containing binary data

i) Boolean
A value of either True or False

j) UID
A UID is an bit string of 128 bits.

k) OID
An OID is an ISO Object Identifier.

l) Shared key
A Shared key is a DES key, a sequence of 8 bytes

m) CA State
A structure of the form described in §3.5

n) Credentials
A structure of the form described in §3.4

o) Certificate
An ASN.1 encoding of the structure described in §3.1

p) Authentication Token
An ASN.1 encoding of the structure described in §3.3.1

q) Mutual Authentication Token
An ASN.1 encoding of the structure described in §3.3.2

r) Encrypted Credentials
An ASN.1 encoding of the structure described in §3.2

s) Public key
A representation of an RSA Public key, including all the information needed to encode the public key in a certificate.

t) Set of Public key/UID pairs
A set of Public key/UID pairs. This Data type is only used internally in DASS - it does not appear in any interface used to other architectures.

3.7 Error conditions

These routines can return the following error conditions (an implementation may indicate errors with more or less precision):

a) Incomplete chain of trustworthy CAs

b) Target has no keys which can be trusted.

c) Invalid Authentication Token

d) Login Ticket Expired

e) Invalid Password

f) Invalid Credentials

g) Invalid Authenticator

h) Duplicate Authenticator

3.8 Certificate Maintenance Functions

Authentication services depend on a set of data structures maintained in the naming service. There are two kinds of information: Certificates, which associate names and public keys and are signed by off-line Certification Authorities; and Encrypted Credentials, which contain RSA Private Keys and certain context information encrypted under passwords. Encrypted Credentials are only necessary in environments where passwords are used. Credentials may alternatively be stored in some other secure manner (for example on a smart card).

The certificate maintenance services are designed so that the most sensitive - the actual signing of certificates - may be done by an off-line authority. Once signed, certificates must be posted in the naming service to be believed. The precise mechanisms for moving certificates between off-line CAs and the on-line naming service are implementation dependent. For the off-line mechanisms to provide any actual security, the CAs must be told what to sign in some reliable manner. The mechanisms for doing this are implementation dependent. The abstract interface says that the CA is given all of the information that goes into a certificate and it produces the signed certificate.

There are requirements surrounding the auditing of a CA's actions. The details of what actions are audited, where the audit trail is maintained, and what utilities exist to search that audit trail are not specified here. The functions a CA must provide are:

3.8.1 Install CA

```
Install_CA(
            --inputs
                keysize                 Integer,
            --outputs
                CA_state                CA State,
                CA_Public_Key           Public Key)
```

This routine need only generate a public/private key pair of the requested size. Keysize is likely to be in implementation constant rather than a parameter. The value is likely to be either 512 or 640. Key sizes throughout will have to increase over time as factoring technology and CPU speeds improve. Both keys are stored as part of the CA_state; the public key is returned so that other CAs may cross-certify this one. The 'Next Serial number' in the CA state is set to 1.

3.8.2 Create Certificate

```
Create_certificate(
            --inputs
                Renewal                 Boolean,
                Include_UID             Boolean,
                Issuer_name             Name,
                Issuer_UID              UID,
                Effective_date          Absolute Time,
                Expiration_date         Absolute Time,
                Subject_name            Name,
                Subject_UID             UID,
                Subject_public_key      Public Key,
            --updated
                CA_state                CA State,
            --outputs
                Certificate             Certificate)
```

This procedure creates and signs a certificate. Note that the various contents of the certificate must be communicated to the CA in some reliable fashion. The Issuer_name and UID are the name and UID of the directory on whose behalf the certificate is being signed.

This routine formats and signs a certificate with the private key in CA_state. It audits the creation of the certificate and updates the sequence number which is part of CA_state. The Issuer and Subject names are X.500 names. If the CA state includes a history of what UIDs have previously been used by what names, this call will only succeed in the collision case if the Renewal boolean is set true. If the Include_UID boolean is set true, this routine will generate a 1992 format X.509 certificate; otherwise it will generate a 1988 format X.509 certificate.

3.8.3 Create Principal

```
Create_principal(
                --inputs
                     Password                          String,
                     keysize                           Integer,
                     Principal_name                    Name,
                     Principal_UID                     UID,
                     Parent_Public_key                 Public Key,
                     Parent_UID                        UID,
                --outputs
                     Encrypted_Credentials             Encrypted Credentials,
                     Trusted_authority_certificate     Certificate)
```

This procedure creates a new principal by generating a new public/private key pair, encrypting the public and private keys under the password, and signing a trusted authority certificate for the parent CA. In an implementation not using passwords (e.g. smart cards), an alternative mechanism must be used for initially creating principals. If a principal has protected storage for trusted authority information, it is not necessary to create a trusted authority certificate and store it in the naming service. Some procedure analogous to this one must be executed, however, in which the principal learns the public key and UID of its CA and its own name.

This routine creates two output structures with the following steps:

a) Generate a public/private key pair using the indicated keysize. An implementation will likely fix the keysize as an implementation constant, most likely 512 or 640 bits, rather than accepting it as a parameter. Key sizes generally will have to increase over time as factoring technology and CPU speeds improve.

b) Form the encrypted credentials by using the public key, private key, and Principal_UID and encrypting them using a hash of the password as the key.

c) Generate a trusted authority certificate (which is identical in format to a "parent" certificate) getting fields as follows:

   1) Certificate version is X.509 1992.

   2) Issuer name is the Principal name (which is an X.500 name).

   3) Issuer UID is the Principal UID.

   4) Validity is for all time.

   5) Subject name is constructed from the Principal name by removing the last simple name from the hierarchical name.

   6) Subject UID is the CA_UID.

   7) Subject Public Key is the CA_Public_Key

   8) Sequence number is 1.

   9) Sign the certificate with the newly generated private key of the principal.

3.8.4 Change Password

```
Change_password(
        --inputs
            Encrypted_credentials        Encrypted Credentials,
            Old_password                 String,
            New_password                 String,
        --outputs
            Encrypted_credentials        Encrypted Credentials)
```

If credentials are stored encrypted under a password, it is possible to change the password if the old one is known. Note that it is insufficient to just change a user's password if the password has been disclosed. Anyone knowing the old password may have already learned the user's private key. If a password has been disclosed, the secure recovery procedure is to call create_principal again followed by create_certificate to certify the new key.

Using DASS, it may not be appropriate for users to periodically change their passwords as a precaution unless they also change their private keys by the procedure above. The only likely use of the change_password procedure is to handle the case where an administrator has chosen a password for the user in the course of setting up the account and the user wishes to change it to something the user can remember. A future version of the architecture may smooth key roll-over by having the change_password command also generate a new key and sign a "self" certificate in which the old key certifies the new one. As a separate step, a CA which notices a self certificate posted in the naming service could certify the new key instead of the old one when the user's certificate is renewed. While this procedure is not as rapid or as reliable as having the user directly interact with the CA, it offers a reasonable tradeoff between security and convenience when there is no evidence of password compromise.

This routine simply decrypts the encrypted credentials structure supplied using the password supplied. It returns a bad status if the format of the decrypted information is bad (indicating an incorrect password). Otherwise, it creates a new encrypted credentials structure by encrypting the same data with the new password. It would be highly desirable for the user interface to this function to provide the capability to randomly generate passwords and prohibit easily guessed user chosen passwords using length, character set, and dictionary lookup rules, but such capabilities are beyond the scope of this document.

If encrypted credentials are stored in some local secure storage, the above function is all that is necessary (in fact, if the storage is sufficiently secure, no password is needed; credentials could be stored unenciphered). If they are stored in a naming service, this function must be coupled with one which retrieves the old encrypted credentials from the naming service and stores the new. The full protocol is likely to include access control checks that require the principal to acquire credentials and produce tokens. For best security, the encrypted credentials should be accessible only through a login agent. The role of the login agent is to audit and limit the rate of password guessing. If passwords are well chosen, there is no significant threat from password guessing because searching the space is computationally infeasible. In the context of a login agent, change password will be implemented with a specialized protocol requiring knowledge of the password and (for best security) a trusted authority from which the public key of the login agent can be learned. See section 2.3.2 for the plans for the non-X.500 credential storage facility.

3.8.5 Change Name

```
Change_name(
        --inputs
            Claimant_Credentials             Credentials,
            New_name                         Name,
            CA_Public_Key                    Public Key,
            CA_UID                           UID,
        --outputs
            Trusted_Authority_Certificate    Certificate)
```

DASS permits a principal to have many current aliases, but only one current name. A principal can authenticate itself as any of its aliases but verifies the names of others relative to the name by which it knows itself. Aliases can be created simply by using the create_certificate function once for each alias. To change the name of a principal, however, requires that the principal securely learn the public key and UID of its new parent CA. As with create_principal, if a principal has secure private storage for its trusted authority information, it need not create a certificate, but some analogous procedure must be able to install new naming information.

This routine produces a new Trusted Authority Certificate with contents as follows:

a) Issuer name is New_name (an X.500 name)

b) Issuer_UID is Principal UID from Credentials.

c) Validity is for all time.

d) Subject name is constructed from the Issuer name by removing the last simple name from the hierarchical name, and converting to an X.500 name.

e) Subject UID is CA_UID

f) Subject Public Key is CA_Public_Key

g) Sequence number is 1.

h) The certificate is signed with the private key of the principal from the credentials. Note that this call will only succeed if the principal's private key is in the credentials, which will only be true if the credentials were created by calling Create_server_credentials.

3.9 Credential Maintenance Functions

DASS credentials can potentially have information about two principals. This functionality is included to support the case where a user on a node has two identities that might be recognized for purposes of managing access controls. First, there is the user's network identity; second, there is an identity as controlling a particular "account" or "username" on that node. There are two reasons for recognizing this second identity: first, access controls might be specified such that only a user is only permitted access to certain resources when coming through certain trusted nodes (e.g. files that can't be accessed from a terminal at home); and second, before the transition strategy to global identities is complete, as a way to refer to USER@NODE in a way analogous to existing mechanisms but with greater security.

The mapping of global usernames to local user names on a node is outside the scope of DASS. This is done via a "proxy database" or some analogous local mechanism. What DASS provides are mechanisms for adding node oriented credentials into a user's credentials structure, carrying the dual authentication information in authentication tokens, and extracting the information from the credentials structure created by Accept_token.

Some applications of DASS will not make use of the node authentication related extensions. In that case, they will never use the Combine_credentials, Create_credentials, Get_node_info, or Verify_node_name functions.

The "normal" sequence of events surrounding a user logging into a node are as follows:

a) When the user logs in, he types either a local user ID known to the node or a global name (the details of the user interface are implementation specific). Through some sort of local mapping, the node determines both a global name and a local account name. The user also enters a password corresponding to the global name.

b) The node calls network_login specifying the user's global name and the supplied password. The result is credentials which can be used to access network services but which have not yet been verified to be valid.

c) The node calls verify_principal_name using its own credentials to verify the authenticity of the user's credentials (these node credentials must have previously been established by a call to initialize_server during node initialization).

d) If that test succeeds, the node adds its credentials to those of the user by calling combine_credentials.

The set of facilities for manipulating credentials follow:

3.9.1 Network login

```
Network_login(
        --inputs
            Name                    Name,
            password                String,
            keysize                 Integer,
            expiration              Time interval,
            TA_credentials          Credentials,    --optional
        --outputs
            Claimant_credentials    Credentials)
```

This function creates credentials for a principal when the principal "logs into the network".

Name is the X.500 name of the principal.

Password is a secret which authenticates the principal to the network.

Keysize specifies the size of the temporary "login" or "delegation" key. In a real implementation, it is expected to be an implementation constant (most likely 384 or 512 bits).

Expiration sets a lifetime for the credentials created. For a normal login, this is likely to be an implementation constant on the order of 8-72 hours. Some mechanism for overriding it must be provided to make it possible (for example) to submit a background job that might run days or even months after they are submitted.

TA_credentials are used if the encrypted credentials are protected by a login agent. If they are missing, the password will be less well protected from guessing attacks.

This routine does not (as one might expect) securely authenticate the principal to the calling procedure. Since the password is used to obtain the principal's private key, this call will normally fail if the principal supplies an invalid password. A penetrator who has compromised the naming service could plant fake encrypted credentials under any name and impersonate that name as far as this call is concerned. A caller that wishes to authenticate the user in addition to obtaining credentials to be able to act on the user's behalf should call Verify_principal_name (below) with the created credentials and the credentials of the calling process.

This routine constructs a credentials structure from information found in the naming service encrypted using the supplied password.

a) If the encrypted credentials structure is protected with a login agent, retrieve the public key of the login agent:

   1) If TA_credentials are available, use them in a call to Get_Pub_Keys to get the public key of the login agent (whose name is derived from the name of the principal by truncating the last element of the RDN and adding CSS=X509).

   2) If TA_credentials are not available, look up the public key of the login agent in the naming service.

   Login agents limit and audit password guesses, and are important when passwords may not be well chosen (as when users are allowed to choose their own). To fully prevent the password guessing threat, principals may only log onto nodes that already have TA_credentials which can be used to authenticate the login agent. To support nodes which have no credentials of their own and to allow this procedure to support node initialization, it is possible to network login without TA credentials.

A principal who logs into a node that lacks TA credentials is subject to the following subtle security threat: A penetrator who impersonates the naming service could post his own public key and address as those of the login agent. This procedure would then in the process of logging in reveal the the penetrator enough information for the penetrator to mount an unaudited password guessing attack against the principal's credentials.

b) Retrieve the encrypted credentials from the naming service or login agent. In the case of the login agent, the password is one-way hashed to produce proof of knowledge of the password and the hashed value is supplied to the login agent encrypted under its public key as part of the request.

c) Decrypt the encrypted credentials structure using a the supplied password. Verify that the decryption was successful by verifying that the resulting structure can be parsed according the the ASN.1 rules for Encrypted_Credentials and that the two included primes when multiplied together produce the included modulus. If the decryption was unsuccessful then the routine returns the 'Invalid password' error status. The decryption results in both the Private Key and the Public Key.

d) Generate a public/private key pair for the Delegation Key, using the indicated keysize. Key size is likely to be an implementation constant rather than a supplied parameter, with likely values being 384 and 512 bits. Key sizes generally will have to increase over time as factoring technology and CPU speeds improve. Delegation keys can be relatively shorter than long term keys because DASS is designed so that compromise of the delegation key after it has expired does not result in a security compromise. An important advantage of making key size an implementation constant is that nodes can generate key pairs in advance, thus speeding up this procedure. Key generation is the most CPU intensive RSA procedure and could make login annoyingly slow.

e) Construct a Login Ticket by signing with the user's private key a combination of the public key, a validity period constructed from the current time and the expiration passed in the call, and the principal UID found in the encrypted-key structure.

f) Forget the user's private key.

g) Retrieve from the naming service any trusted authority certificates stored with the user's entry. Discard any that are not signed by the user's public key and UID. An implementation in which the login node has credentials of its own may choose its trusted authority information instead of retrieving and verifying trusted authority certificates from the naming service. This will have a subtle effect on the security of the resulting system.

h) Construct a credentials structure from:

   1) Claimant credentials:

      (i) Name of the principal from calling parameter

      (ii) Login Ticket as constructed in (e)

      (iii) Delegation Private key as constructed in (d)

      (iv) Public key from the encrypted credentials structure

   2) No verifier credentials

   3) Trusted Authorities: for the most recently signed trusted authority certificate[3]:

      (i) Name of the CA from the subject field of the certificate

      (ii) Public Key of the CA from the subject public key field

      (iii) UID of the CA from the subject UID field

[3] There is normally only one Trusted Authority Certificate. If there is more than one then an implementation may choose to maintain a list of all the valid keys. They should all refer to the same CA (UID and name).

4) no remote node credentials

5) no local node credentials

6) no cached outgoing associations

7) no cached incoming associations

3.9.2 Create Credentials

```
Create_credentials(
                    --outputs
                       Claimant_credentials          Credentials)
```

This routine creates an "empty" credentials structure. It is needed in the case of a user logging into a node and obtaining node oriented credentials but no global username credentials. Because the "combine_credentials" call wants to modify a set of user credentials rather than create a new set, this call is needed to produce the "shell" for combine_credentials to fill in.

It is unlikely that any real implementation would support this function, but rather would have some functions which combine network_login, create_credentials, and combine_credentials in whatever ways are supported by that node.

3.9.3 Combine Credentials

```
Combine_credentials(
                    --inputs
                       node_credentials              Credentials,
                       localusername                 String,
                    --updated
                       user_credentials              Credentials)
```

This routine is provided by implementations which support the notion of local node credentials. After the node has verified to its own satisfaction that the user_credentials are entitled to access to a particular local account, this call adds node credential information to the user_credential structure. This function may be applied to user_credentials created by network_login, create_credentials, or accept_token.

a) Fill in the local node credentials substructure of user_credentials as follows:

   1) Full name of the node: from Full name of the Principal in node_credentials

   2) Local username on the node: from proxy lookup

   3) RSA private key of the node: from verifier credentials in node_credentials

b) *Optionally,* change the trusted authorities to match the trusted authorities from the node credentials. This is an implementation option, done most likely as a performance optimization. The only case where this option is required is where no trusted authorities existed in the user credentials (because they were created by create_credentials of accept_token). Server credentials should generally keep their own trusted authorities.

It is likely that an implementation will choose not to replicate its node credentials in every credentials structure that it supports, but rather will maintain some sort of pointer to a single copy. This algorithm is stated as it is only for ease of specification.

3.9.4 Initialize_server

```
initialize_server(
                --inputs
                    Name                    Name,
                    password                String,
                    TA_credentials          Credentials,    --optional
                --outputs
                    Server_credentials      Credentials)
```

Somehow a server must get access to its credentials. One way is for the credentials to be stored in the naming service like user credentials encrypted under a service password. The service then needs to gain at startup time access to a service password. This may be easier to manage and is not insecure so long as the service password is well chosen. Alternately, the service needs some mechanism to gain access directly to its credentials. The credentials created by this call are intended to be very long lived. They do not time out, so a node or server might store them in Non-Volatile memory after "initial installation" rather than calling this routine at each "boot". These credentials are shared between all servers which use the same key. This routine works as follows:

a) Retrieve from the naming service or login agent the encrypted credentials structure corresponding to the supplied name. See Network_login for a discussion of the use of TA_credentials and login agents.

b) Decrypt that structure using a one-way hash of the supplied password. Verify that the decryption was successful. Verify that the public key in the structure matches the private key.

c) Retrieve from the naming service any trusted authority certificates stored under the supplied name. Discard any which do not contain the UID from the encrypted credentials structure or are not signed by the key in the encrypted credentials structure.

d) Construct a credentials structure from:

1) Claimant credentials:

(i) Name of the principal from the calling parameter

(ii) UID of the principal from the encrypted-key structure

(iii) No login ticket

(iv) No login secret key

2) Verifier credentials:

(i) Server secret key from the encrypted-key structure

3) Trusted Authorities: from the most recently signed Trusted Authority Certificate:

(i) Name of CA from the Subject Name field

(ii) UID of the CA from the Subject UID field

(iii) Public Key of the CA from the Subject Public Key field

4) no node credentials

5) no cached outgoing associations

6) no cached incoming associations

3.9.5 Generate Server Ticket

```
generate_server_ticket(
                    --inputs
                        expiration                  Time interval,
                    --updated
                        Server_credentials          Credentials)
```

Server credentials created by initialize_server can be used to accept incoming authentication tokens and can act as node_credentials for outgoing authentications, but cannot create user_credentials of their own. If a server initiates connections on its own behalf, it must have a ticket just like any other user might have. That ticket has limited lifetime and the right to act on behalf of the server can be delegated. The server cannot, however, delegate the right to receive connections intended for it. An implementation must come up with a policy for the expiration of server tickets and how long before expiration they are renewed. A likely policy is for this procedure to be implicitly called by Create_token if there is no current ticket present in the credentials. If so, this interface need not be exposed.

This routine is implemented as follows:

a) Generate an RSA public/private key pair.

b) Compute a validity interval from the current time and the expiration supplied.

c) Construct a login ticket from the RSA public key (from a), validity interval (from b), the UID from the credentials, and signed with the server key in the credentials. (Discard previous Login Ticket if there was one).

d) Discard all information in the Cached Outgoing Contexts.

3.9.6 Delete Credentials

```
delete_credentials(
                    --updated
                        credentials                 Credentials)
```

Erases the secrets in the credentials structure and deallocates the storage.

3.10 Authentication Procedures

The guts of the authentication process takes place in the next two calls. When one principal wishes to authenticate to another, it calls Create_token and sends the token which results to the other. The recipient calls Accept_token and creates a new set of credentials. The other calls in this section manipulate the received credentials in order to retrieve its contents and verify the identity of the token creator.

3.10.1 Create Token

```
Create_token(
                    --inputs
                        target_name                 Name,
                        deleg_req_flag              Boolean,
                        mutual_req_flag             Boolean,
                        replay_det_req_flag         Boolean,
                        sequence_req_flag           Boolean,
                        chan_bindings               Octet String,
                        Include_principal_name      Boolean,
                        Include_node_name           Boolean,
                        Include_username            Boolean,
```

```
    --updated
        claimant_credentials            Credentials,
  --outputs
        authentication_token            Authentication token,
        mutual_authentication_token     Mutual Authentication token,
        Shared_key                      Shared Key
        instance_identifier             Timestamp)
```

This routine is used by the initiator of a connection to create an authentication token which will prove its identity. If the claimant credentials includes node/account information, the token will include node authentication.

target_name is the X.500 name of the intended recipient of the token. Only an entity with access to the private key associated with that name will be able to verify the created token and generate the mutual_authentication_token.

deleg_req_flag indicates whether the caller wishes to delegate to the recipient of the token. If it is set, the delegated_credentials returned by Accept_token will be capable of generating tokens on behalf of the caller. Node based authentication information cannot be delegated. The ***mutual_req_flag, replay_det_req_flag***, and ***sequence_req_flag*** are put in the authentication token and passed to the target. This information is included in the token to make it easier to implement the GSSAPI over DASS. DASS itself makes no use of this information.

In most applications, the purpose of a token exchange is to authenticate the principals controlling the two ends of a communication channel. ***chan_bindings*** contains an identifier of the channel which is being authenticated, and thus its format and content should be tied to the underlying communication protocol. DASS only guarantees that the information has been communicated reliably to the named target. If DASS is used with a cryptographically protected channel (such as SP4), this data should contain a one-way hash of the key used to encrypt the channel. If that channel is multiplexed, the data should also include the ID of the subchannel. If the channel is not encrypted, the network must be trusted not to modify data on a connection. The source and target network addresses and a connection ID should be included in the chan_bindings at the source and checked at the target. A token exchange also results in the two ends sharing a key and an instance identifier. If that key and instance identifier are used to cryptographically protect subsequent communications, then ***chan_bindings*** need not have any cryptographic significance but may be used to differentiate multiple entities sharing the public keys of communicating principals. For example, if a service is replicated and all replicas share a public key, ***chan_bindings*** should include something that identifies a single instance of the service (such as current address) so that the token cannot be successfully presented to more than one of the servers.

include_principal_name, include_node_name, and ***include_username*** are flags which determine whether the principal name, node name, and/or username from the credentials structure are to be included in the token. This information is made optional in a token so that applications which communicate this information out of band can produce "compressed" tokens. If this information is included in the token, it will be used to populate the corresponding fields in the credentials structure created by Accept_token.

claimant_credentials are the credentials of the calling procedure. The secrets contained therein are used to sign the token and the trusted authorities are used to securely learn the public key of the target. The cached outgoing contexts portion of the credentials may be updated as a side effect of this call.

The major output of this routine is an ***authentication_token*** which can be passed to the target in order to authenticate the caller.

In addition to returning an authentication token, this routine returns a ***mutual_authentication_token,*** a ***shared_key,*** and an ***instance_identifier***. The mutual authentication token is the same as the one generated by the Accept_token call at the target. If the protocol using DASS wishes mutual authentication, the target should return this token to the source. The source will compare it to the one returned by this routine using Compare_Mutual_Token (below) and know that the token was accepted at its proper destination.

The DES key and instance identifier can be used to encrypt or sign data to be sent to this target. The key and instance will be given to the target by Accept_token, and the key will only be known by the two parties to the authentication.

If a single set of credentials is used to authenticate to the same target more than once, the same DES key is likely to be returned each time. If the parties wish to protect against the possibility of an outside agent mixing and matching messages from one authenticated session with those of another, they should include the instance identifier in the messages. The instance identifier is a timestamp and it is guaranteed that the DES key/instance identifier pair will be unique.

An implementation may wish to "hide" the DES key from calling applications by placing it in system storage and providing calls which encrypt/decrypt/sign/verify using the key.

The primary tasks of this routine are to create its output parameters. As a side effect, it may also update claimant_credentials. It's algorithm is as follows:

a) The login ticket is checked. If it has passed the end of its lifetime an 'Login Ticket Expired' error is returned. If there is a login ticket, but no corresponding private key then an 'Invalid credentials' error is returned (this is the case if the credentials were created by an authentication-without-delegation operation). If there is no login ticket or an expired one and if the long term private key is present in the credentials, an implementation may choose to automatically call create_server_ticket to renew the ticket.

b) Create new timestamp using the current time.[4]

c) The public key and UID of target_name are looked up by calling get_pub_keys, using the target_name and the Trusted Authority section of the claimant_credentials structure. If none is found, an error status is returned. Otherwise, the cached outbound connections portion of credentials are searched (indexed by target Public Key) for a cached Shared key with a validity interval which has not expired. If a suitable one is found skip to step g, else create a cache entry as follows:

d) Destination Public Key is the one found looking up the target. A Shared Key is generated at random. A validity interval is chosen according to node policy but not to exceed the validity interval of the ticket in the credentials (if any).

e) Create the Encrypted Shared Key, using the public key of the Target, and place in the cache.

f) If node authentication credentials are available in the credentials structure, create a "Node Ticket" signature using the node secret and include it in the cache.

g) If delegation is requested and no delegator is present in the cache, create one by encrypting the delegation private key under the Shared key. The delegation private key is represented as an ASN.1 data structure containing only one of the primes (p).

h) If delegation is not requested and no Shared Key Ticket is in the cache, create one by signing the requisite information with the delegation private key.

i) Create the Authenticator. The contents of the Authenticator (including the channel bindings) are encoded into ASN.1, and the signature is computed. The Authenticator is then re-encoded, without including the Channel Bindings but using the same signature.

j) Create output_token as follows:

   1) Encrypted Shared Key from cache

   2) Login Ticket from Claimant Credentials (if present)

   3) Shared Key Ticket from cache (if no delegation and if present)

   4) Node Ticket from cache (if present)

[4] This timestamp must be unique for this Shared Key. The timestamp is a 64 bit POSIX time, with a resolution of 1 nanosecond An implementation must ensure that timestamps cannot be reused.

   5) Delegator from cache (if delegation and if present)

   6) Authenticator

   7) Principal name from credentials (if present and parameter requests this)

   8) Node name from credentials (if present and parameter request this)

   9) Local Username from credentials (if present and parameter requests this)

k) Compute Mutual_authentication_token by encrypting the timestamp from the authenticator using the Shared key.

l) The instance_identifier is the timestamp. This and the Shared key are returned for use by the caller for further encryption operations (if these are supported).

3.10.2 Accept_token

```
Accept_token(
                    --inputs
                        authentication_token            Authentication Token,
                        chan_bindings                   Octet String,
                     --updated
                        verifying_credentials           Credentials,
                    --outputs
                        accepted_credentials            Credentials,
                        deleg_req_flag                  Boolean,
                        mutual_req_flag                 Boolean,
                        replay_det_req_flag             Boolean,
                        sequence_req_flag               Boolean,
                        mutual_authentication_token     Mutual authentication token
                        shared_key                      Shared Key,
                        instance_identifier             Timestamp)
```

This routine is used by the recipient of an authentication token to validate it. ***authentication_token*** is the token as received; ***chan_bindings*** is the identifier of the channel being authenticated. See the description of Create_token for information on the appropriate contents for chan_bindings. DASS does not enforce any particular content, but checks to assure that the same value is supplied to both Create_token and Accept_token.

Verifying_credentials are the credentials of the recipient of the token. They must include the private key of the entity named as the target in Create_token or the call will fail. The cached incoming contexts section of the verifying credentials may be modified as a side effect of this call.

Accepted_credentials will contain additional information about the token creator. If delegation was requested, these credentials can be used to make additional calls to Create_token on the creator's behalf. Whether or not delegation was requested, they can also be used in the calls which follow to gain additional information about the token creator.

The ***deleg_req_flag*** indicates whether the accepted_credentials include delegation which can be used by the recipient to act on behalf of the principal. ***Mutual_req_flag, replay_det_req_flag,*** and ***sequence_req_flag*** are passed through from Create_token in support of the GSSAPI. DASS makes no use of these fields.

The ***mutual_authentication_token*** can be returned to the token creator as proof of receipt. In many protocols, this will be used by a client to authenticate a server. Only the genuine server would be able to compute the mutual_authentication_token from the token.

The ***shared_key*** and ***instance_identifier*** can be used to encrypt or sign data between the two authenticating parties. See Create_token.

This routine verifies the contents of the authentication token in the context of the verifying credentials[5] and returns a information about it. The algorithm updates a cache of information. This cache is not updated if the algorithm exits with an error. The algorithm is as follows:

a) If there is a Login Ticket, but no Shared Key Ticket or Delegator then exit with error 'Invalid Authenticator'. If there is a Shared Key Ticket or Delegator, but no Login Ticket then exit with error 'Invalid Authentication Token'.

   Look up the Encrypted Shared key in the Cached Incoming Contexts of the credentials structure[6]. If it is not found then create a new cache entry as follows:

   1) Encrypted Shared Key, from the Authentication Token.

   2) Shared Key and Validity Interval, by decrypting the Encrypted Shared Key using the server private key in credentials. If the decryption fails then exit with error 'Invalid Authentication Token'.

b) Check that the Validity Interval (in the cache entry) includes the current time; return 'Invalid Authentication Token' if not.

   Check the Timestamp is within max-clock-skew of the current time, return 'invalid Authentication Token' if not.

   Reconstruct the Authenticator including the Channel Bindings passed as a parameter.

   Check that the reconstructed Authenticator is signed by the Shared key. If not then exit with error 'Invalid Authentication Token'.

   Look up the Authenticator Signature in the Received Authenticators. If the same Signature is found in the list then exit with error 'Duplicate Authenticator'. Otherwise add the Signature and timestamp to the list.

   If there is a Login Ticket and the Delegation Public key is in the cache entry, then check that the same key is specified in the Login Ticket, if not then exit with error 'Invalid Authentication Token'. Place the Delegation Public key in the cache if it is not already there.

   If there is a Login Ticket, the Delegation Public key was not previously in the cache entry, and there is a Shared Key Ticket in the Authentication Token, then check that the Shared Key Ticket is signed by the Delegation Public Key in the Login Ticket. If not then exit with error 'Invalid Authentication Token'.

   If a delegator is present in the message then decrypt the delegator using the Shared key. If the private key does not match the Delegation Public key then exit with error 'Invalid Authentication Token'[7].

   Build the delegation credentials data structure as follows:

   1) Claimant credentials:
      (i) Login Ticket from the Authentication token
      (ii) Delegation Private key from the decrypted delegator if the token is delegating.
      (iii) Encrypted Shared Key from the Authentication token.
   2) There are no verifier credentials.
   3) Trusted authorities are copied from the verifying_credentials passed to this routine.[8]

[5] In particular the Private Key of the server is used. Also the Cached Incoming Contexts and Incoming Timestamp list is used.

[6] This cache entry is used during the execution of this routine. An implementation must ensure that references to the cache entry can not be affected by other users modifying the cache. One way is to use a copy of the cache entry, and update it at exit.

[7] The prime in the delegator is used to find the other prime (from the modulus). The division must not have a remainder. Neither prime may be 1. The two primes are then used to reconstruct any other information needed to perform cryptographic operations.

[8] If an implementation is able to obtain the original Trusted Authorities for the Principal then it may do so instead of using the Servers Trusted Authorities

4) Remote node credentials (Node name, Username, Node Ticket) taken from the Authentication token.
5) There are no local node credentials.
6) There are no cached contexts.

c) The returned boolean values are obtained from the Authenticator.

d) Mutual_authentication_token is computed by encrypting the timestamp from the Authenticator with the Shared key from the cache.

e) Instance_identifier is the timestamp from the Authenticator. This and the Shared key are returned to the caller for further encryption operations (if these are supported).

3.10.3 Compare Mutual Token

```
Compare_mutual_token(
                --inputs
                    Generated_token          Mutual authentication token,
                    Received_token           Mutual authentication token,
                --outputs
                    equality_flag            Boolean)
```

This routine compares two mutual authentication tokens and tells whether they match. In the expected use, the first is the token generated by Create_token at the initiating end and the second is the token generated by Accept_token at the accepting end and returned to the initiating end. This routine can be implemented as a byte by byte comparison of the two parameters.

3.10.4 Get Node Info

```
get_node_info(
                --inputs
                    accepted_credentials     Credentials,
                --outputs
                    nodename                 Name,
                    username                 String)
```

This routine extracts from accepted credentials the name of the node from which the authentication token came and the named account on that node. Because this information is not cryptographically protected within the token, this information can only be regarded as a "hint" by the receiving application. It can, however, be verified using Verify_node_name in a cryptographically secure manner. This information will only be present if these are accepted credentials and it the caller of Create_token set the ***include_node_name*** and/or ***include_username*** flags.

An actual implementation is not likely to have get_node_info and verify_node_name as separate calls. They are specified this way because there are different ways this information might be used. For most applications, the nodename and username will be included in the token, and a single function might extract and verify them (it might in fact be part of accept token). For other applications, the nodename and username will not be in the token but rather will be computed from other information passed during connection initiation so a call would have to take these as inputs. Still other applications such as ACL evaluators that want to support the renaming and aliasing capabilities of DASS would defer verifying node information until they came upon an ACL which allowed access only from a particular node. They would then verify that the name on the ACL was an authenticatable alias for the node which created the token. All of these uses can be defined in terms of calls to get_node_info and verify_node_name.

3.10.5 Get Principal UID

```
get_principal_uid(
                --inputs
                    accepted_credentials        Credentials,
                --outputs
                    uid                         UID)
```

This routine extracts a principal UID from a set of credentials.

As with Get_Node_Info, this interface is not likely to appear in an actual implementation, but rather will be bundled with other routines. It is specified this way because there might be a variety of algorithms by which credentials are evaluated and all of them can be defined in terms of these primitives.

In DASS, it is possible for a principal to have many aliases. This can happen either because the principal was given multiple names to limit the number of CAs that need to be trusted when authenticating to different servers or because the principal's name has changed and the old name remains behind as an alias. Accept_token returns the name by which the principal identified itself when creating its credentials. A service may know the user by some alias. The normal way to handle this is for the service to know the principal's UID (which is constant over name changes) and to compare it with the UID in the token to identify a likely alias situation. It gets the UID from the token using this routine. It then confirms the alias by calling verify_principal_name.

The UID is in a signed portion of accepted credentials, but the signature may not have been verified at the time this call is issued. The information returned by this routine must therefore be regarded as a hint. If a call to Verify_principal_name succeeds, however, then the caller can securely know that the name given to that routine ***and*** the UID returned by this one are the authenticated source of the token.

3.10.6 Get Principal Name

```
get_principal_name(
                --inputs
                    accepted_credentials        Credentials,
                --outputs
                    name                        Name)
```

This routine extracts a principal name from a set of credentials. This name is the name most recently associated with the principal. It may be the name that the principal supplied when the credentials were created (in which case it may not have been verified yet) or it may be a different name that has been verified.

As with Get_Node_Info and Get_Principal_UID, this routine is not likely to appear in an actual implementation, but will be bundled in some fashion with related procedures. The name returned by this procedure is not guaranteed to have been cryptographically verified. Verify_Principal_Name performs that function.

3.10.7 Get Lifetime

```
get_lifetime(
                --inputs
                    Claimant_credentials        Credentials,
                --outputs
                    lifetime                    Duration)
```

This routine computes the life remaining in a set of credentials. Its most common use would be to know to renew credentials before they expire.

Returns the remaining lifetime of the login ticket in the credentials. This can either be the done on the node where the original login took place, or at a server which has been delegated to. It indicates how much longer these credentials can be used for further delegations. This routine will return 0 if the login ticket has passed the end of its life, if there is no login ticket, or if the credentials do not contain the private key certified by the ticket (i.e. where they were created by an authentication-without-delegation operation).

3.10.8 Verify Node Name

```
Verify_node_name(
        --inputs
            nodename                    Name,
            username                    String,
        --updated
            verifying_credentials       Credentials,
            accepted_credentials        Credentials,
        --outputs
            Name matches                Boolean)
```

This routine tests whether the originating node of an authentication token can be authenticated as having the provided name. Like a principal, a node may have multiple aliases. One of them may be returned by Get_node_info, but this call allows a suspected alias to be verified. The verifying credentials supplied with this call must be the same credentials as were used in the Accept_token call. The procedure for completing this request is as follows:

a) If there is no Node Ticket in the claimant credentials then return False.

b) Search the incoming context cache of the verifying credentials for an entry containing the same encrypted shared key as the encrypted shared key subfield of the claimant information of the accepted credentials. In the steps which follow, references to "the cache" refer to this entry. If none is found, initialize such an entry as follows:

   1) Encrypted shared key from the encrypted shared key subfield of the claimant information of the accepted credentials.

   2) The shared key and validity interval are determined by decrypting the encrypted shared key using the RSA private key in the verifier information of the server credentials. If this procedure is called after a call to Accept_token using the same server credentials (as is required for correct use), the shared key and validity interval must correctly decrypt. If called in some other context, the results are undefined. The validity interval is not checked.

   3) Initialize all other entries in the cache to missing.

c) If there is a "local username on client node" in the cache and it does not match the username supplied as a parameter, return False.

d) If there is a "name of client node" in the cache and it matches the nodename supplied as a parameter:

   1) Set the "Full name of the node" subfield of the remote node authentication field of the accepted credentials to be the nodename supplied as a parameter.

   2) Set the "Local Username on the node" subfield of the remote node authentication field of the accepted credentials to be the username supplied as a parameter.

   3) return True.

e) Call the Get_Pub_Keys subroutine with the server_credentials, the nodename supplied as a parameter, and Try_Hard=False.

f) If "Public Key of Client Node" is missing from the cache, check all of the Public keys returned to see if one verifies the node ticket. If one does, set the "Public Key of Client Node" and "UID of Client Node" fields in the cache to be the PK/UID pair that verified the ticket and set the "Local Username on Client node" field to be the username supplied as a parameter..

g) If any of the Public Key/UID pairs match the "Public Key of Client Node" and "UID of Client Node" fields in the cache, then:

   1) Set the "name of client node" in the cache equal to the nodename supplied as a parameter.

   2) Set the "Full name of the node" subfield of the remote node authentication field of the accepted credentials to be the nodename supplied as a parameter.

   3) Set the "Local Username on the node" subfield of the remote node authentication field of the accepted credentials to be the username supplied as a parameter.

   4) Return True.

h) If none of them match, call Get_Pub_Keys again with Try_Hard=True and repeat steps 6 & 7. If Step 7 fails a second time, return False.

3.10.9 Verify Principal Name

```
Verify_principal_name(
                    --inputs
                        principal_name              Name,
                    --updated
                        verifier_credentials        Credentials,
                        claimant_credentials        Credentials,
                    --outputs
                        Name matches                Boolean)
```

This routine tests (in the context of the verifier credentials) whether the claimant credentials are authenticatable as being those of the named principal. This procedure is called with a set of accepted credentials to authenticate their source, or with a set of credentials produced by network_login to authenticate the creator of those credentials. If the claimant credentials were created by Accept_token, then the verifier credentials supplied in this call must be the same as those used in that call. The procedure for completing this request is as follows:

a) If there is no Login Ticket in the claimant credentials, then return False.

b) If the current time is not within the validity interval of the Login Ticket, then return False.

c) If there is an Encrypted Shared Key present in the Claimant information field of the claimant credentials, then find or create a matching cache entry in the Cached Incoming Contexts of the verifier credentials. In the description which follows, references to "the cache" refer to this entry. If the cache entry must be created, its contents is set to be as follows:

   1) Encrypted shared key from the encrypted shared key subfield of the claimant information of the accepted credentials.

   2) The shared key and validity interval are determined by decrypting the encrypted shared key using the RSA private key in the verifier information of the server credentials. If this procedure is called after a call to Accept_token using the same server credentials (as is required for correct use), the shared key and validity interval must correctly decrypt. If called in some other context, the results are undefined. The validity interval is not checked.

   3) Initialize all other entries in the cache to missing.

d) If there is a cache entry and if the "Public Key of Client Principal" field is present and if the "UID of Client Principal" field is present and matches the UID in the Login Ticket, then:

   1) Set the Public Key of the principal field in the Claimant information to be the Public Key of Client Principal.

   2) If the "Full name of the principal" field is missing from the claimant information of the claimant credentials, then set it to the "Name of Client Principal" field from the cache.

e) If there is a cache entry and if the "Name of Client Principal" field is present and if it matches the principal name supplied to this routine and if the UID in the cache matches the UID in the Login Ticket, return True.

f) Call the Get_Pub_Keys subroutine with the name and verifier credentials supplied to this routine and Try_Hard=FALSE. Ignore any keys retrieved where the corresponding UID does not match the UID in the claimant credentials.

g) If the Public Key of the principal is missing from the claimant information of the claimant credentials, then attempt to verify the signature on the login ticket with each public key returned by Get_Pub_Keys. If verification succeeds:

   1) Set the Public Key of the principal in the claimant information of the claimant credentials to be the Public Key that verified the ticket.

   2) If the Full name of the principal in the claimant information of the claimant credentials is missing, set it to the name supplied to this routine.

   3) If there is a cache entry, set the Name of Client Principal to be the name supplied to this routine, the UID of Client Principal to be the UID from the Login Ticket, and the Public Key of Client Principal to be the Public Key that verified the ticket.

   4) Return True.

h) If the Public Key of the principal is present in the claimant information of the claimant credentials, then see if it matches any of the public keys returned by Get_Pub_Keys. If one of them matches:

   1) If the Full name of the principal in the claimant information of the claimant credentials is missing, set it to the name supplied to this routine.

   2) If there is a cache entry, set the Name of Client Principal to be the name supplied to this routine, the UID of Client Principal to be the UID from the Login Ticket, and the Public Key of Client Principal to be the Public Key that verified the ticket.

   3) Return True.

i) If steps 7 & 8 fail, retry the call to Get_Pub_Keys with Try_Hard=TRUE, and retry steps 7 & 8. If they fail again, return false.

3.10.10 Get Pub Keys

```
Get_Pub_Keys(
                --inputs
                    TA_credentials              Credentials
                    Try_Hard                    Boolean,
                    Target Name                 Name,
                --outputs
                    Pub_keys                    Set of Public key/UID pairs
```

This common subroutine is used in the execution of Create_Token, Verify_Principal_Name, and Verify_Node_Name. Given the name of a principal, it retrieves a set of public key/UID pairs which authenticate that principal (normally only one pair). It does this by retrieving from the naming service a series of certificates, verifying the signatures on those certificates, and verifying that the sequence of certificates constitute a valid "treewalk".

The credentials structure passed into this procedure represent a starting point for the treewalk. Included in these credentials will be the public key, UID, and name of an authority that is trusted to authenticate all remote principals (directly or indirectly).

The "Try_Hard" bit is a specification anomaly resulting from the fact that caches maintained by this routine are not transparent to the calling routines. It tells this procedure to bypass caches when doing all name service lookups because the information in caches is believed to be stale. In general, a routine will call Get_Pub_Keys with Try_Hard set false and try to use the keys returned. If use of those keys fails, the calling routine may call this routine again with Try_Hard set true in hopes of getting additional keys. Routinely calling this routine with Try_Hard set true is likely to have adverse performance implications but would not affect the correctness or the security of the operation.

The name supplied is the full X.500 name of the principal for whom public keys are needed as part of some authentication process.

This procedure securely learns the public keys and UIDs of foreign principals by constructing a valid chain of certificates between its trusted TA and the certificate naming the foreign principal. In the simplest case, where the TA has signed a certificate for the foreign principal, the chain consists of a single certificate. Otherwise, the chain must consist of a series of certificates where the first is signed by the TA, the last is a certificate for the foreign principal, and the subject of each principal in the chain is the issuer of the next.

What follows is first a definition of what constitutes a valid chain of certificates followed by a model algorithm which constructs all of (and only) the valid chains which exist between the TA and the target name.

In order to limit the implications of the compromise of a single CA, and also to limit the complexity of the search of the certificate space, there are restrictions on what constitutes a valid chain of certificates from the TA to the Name provided. The only CAs whose compromise should be able to compromise an authentication are those controlling directories that are ancestors of one of the two names and that are not above a common ancestor. Therefore, only certificates signed by those CAs will be considered valid in a certificate chain. Normally, the CA for a directory is expected to certify a public key and UID for the CA of each child directory and one parent directory. A CA may also certify another CA for some remote part of the naming hierarchy, and such certificates are necessary if there are no CAs assigned to directories high in the naming hierarchy.

A certificate chain is considered ***valid*** if it meets the following criteria:

a) It must consist of zero or more ***parent*** certificates, followed by zero or one ***cross*** certificates, followed by zero or more ***child*** certificates.

b) The number of ***parent*** certificates may not exceed the number of levels in the naming hierarchy between the TA name and the name of the least common ancestor in the naming hierarchy between the TA name and the target name.

c) Each ***parent*** certificate must be stored in the naming service under the entry of its issuer.

d) The subject of the ***cross*** certificate (if any) must be an ancestor of the target name but must be a longer name than the least common ancestor of the TA name and the target name.

e) The ***cross*** certificate (if any) must have been stored in the naming service under the entry of its issuer or there must have been an indication in the naming service that certificates signed by this issuer may be stored with their subjects.

f) The issuer of each ***parent*** certificate does not have stored with it in the naming service a ***cross*** certificate with the same issuer whose subject is an ancestor of the target name.

g) Each ***child*** certificate must be stored in the naming service under the entry of its subject.

h) The subject of each ***child*** certificate does not have associated with it in the naming service a ***cross*** certificate with the same subject whose issuer is the same as the issuer of any of the ***parent*** certificates or the ***cross*** certificate of the chain.

i) The subject of each certificate must be the issuer of the certificate that follows in the chain. The equality test can be met by either of two methods:

   1) The public key of the subject in the earlier certificate verifies the signature of the later and the subject UID in the earlier certificate is equal to the issuer UID in the later; or

   2) The public key of the subject in the earlier certificate verifies the signature of the later, the earlier lacks a subject UID and/or the later lacks an issuer UID and the name of the subject in the earlier certificate is equal to the name of the issuer in the later.

j) The Public Key of the TA verifies the signature of the first certificate.

k) The UID of the TA equals the UID of the issuer of the first certificate ***or*** the UID is missing on one or both places and the name of the TA equals the name of the issuer of the first certificate.

l) All of the certificates are valid X.509 encodings and the current time is within all of their validity intervals.

If a chain is ***valid***, the name which it authenticates can be constructed as follows:

a) If the chain contains a ***cross*** certificate, the name authenticated can be constructed by taking the subject name from the cross certificate and appending to it a relative name for each child certificate which follows. The relative name is the extension by which the subject name in the child certificate extends the issuer name.

b) If the chain does not contain a ***cross*** certificate, the name authenticated can be constructed by taking the TA name, truncating from it the last ***n*** name components where ***n*** is the number of ***parent*** certificates in the chain, and appending to the result a relative name for each child certificate. The relative name is the extension by which the subject name in the child certificate extends the issuer name.

In the common case, the authenticated name will be the subject name in the last certificate. The authenticated name is constructed by the rules above to deal with namespace reorganization. If a branch of the namespace is renamed (due to, for example, a corporate acquisition or reorganization), only the certificates around the break point need to be regenerated. Certificates below the break will continue to contain the old names (until renewed), but the algorithms above assure the principals in that branch will be able to authenticate as their new names. Further, if the certificates at the branch point are maintained for both the old and new names for an interim period, principals in the moved branch will be able to authenticate as either their old or new names for that interim period without having duplicate certificates.

A final complication that the algorithm must deal with is the location of ***cross*** certificates. If a key is compromised or for some other reason it is important to revoke a certificate ahead of its expiration, it is removed from the naming service. This algorithm will only use certificates that it has recently retrieved from the naming service, so revocation is as effective as the mechanisms that prevent impersonation of the naming service. There are plans to eventually use DASS mechanisms to secure access to the naming service; until they are in place, name service impersonation is a theoretical threat to the security of revocation. Opinions differ as to whether it is a practical threat. ***Child*** certificates are always stored with the subject and will not be found unless stored in the name server of the subject. ***Parent*** certificates are always stored with the issuer and will not be found unless stored in the name server of the issuer. For best security, ***cross*** certificates should be stored with the issuer because the name server for the subject may not be adequately trustworthy to perform revocation. There are performance and availability penalties, however, in doing so. The architecture and the algorithm therefore support storing ***cross*** certificates with either the issuer or the subject. There must be some sort of flag in the name service associated with the issuer saying whether ***cross*** certificates from that issuer are permitted to be stored in the subject's name service entry, and if that flag is set such certificates will be found and used.

In order to make revocation effective, DASS must assure that naming service caches do not become arbitrarily stale (the allowed age of a cache entry is included in the sum of times with together make up the revocation time). If DASS uses a naming service such as DNS that does not time out cache entries, it must bypass cache on all calls and (to achieve reasonable performance) maintain its own naming service cache. It may be advantageous to maintain a cache in any case so the that the fact that the certificates have been verified can be cached as well as the fact that they are current.

3.10.10.1 Basic Algorithm

For ease of exposition, this first description will ignore the operation of any caches. Permissible modifications to take advantage of caches and enhance performance will be covered in the next section. This path will be followed if the Try_Hard bit is set True on the call.

Rather than trying construct all possible chains between the TA and the name to be authenticated (in the event of multiple certificates per principal, there could be exponentially many valid chains), this algorithm computes a set of PK/UID/Name triples that are valid for each principal on the path between the TA and the name to be authenticated. By doing so, it minimizes the processing of redundant information.

a) Determining path and initialization

   Several state variables are manipulated during the tree walk. These are called:

   1) Current-directory-name
      This is the name indicating the current place in the tree walk. Initially, this is the name of the TA.

   2) Least-Common-Ancestor-Name
      This is the portion of the names which is common to both the CA and the Target. This is computed at initialization and does not change during the treewalk.

   3) Trusted-Key-Set
      For each name which is an ancestor of either the TA or the Target but not of the Least-Common-Ancestor, a list of PK/UID/Name triples. This is initialized to a single triple from the TA information in the supplied credentials.

   4) Search-when-descending
      This is a list of PK/UID/Name triples of issuers that will be trusted when descending the tree. This set is initially empty.

   5) Saved-RDNs
      This is a sequence of Relative Distinguished Names (RDNs) stripped off the right of the target name to form Least-common-ancestor-name. This "stack" is initially empty and is populated during Step 3.

b) Ascending the "TA side" of the tree

   While Current-directory-name is not identical to Common-point-Name the algorithm moves up the tree. At each step it does the following operations.

   1) Find all cross certificates stored in the naming service under Current-directory-name in which the subject is an ancestor of the principal to be authenticated or an indication that cross certificates from this issuer are stored in the subject entry. If there is an indication that such certificates are stored in the subject entry, copy

all triples in Trusted-Key-Set for Current-directory-name into the "Search-when-descending" list. If any such certificates are found, filter them to include only those which meet the following criteria:

(i) For some triple in the Trusted-Key-Set corresponding to the Current-directory-name, the public key in the triple verifies the signature on the certificate ***and either*** the UID in the triple matches the issuer UID in the certificate ***or*** the UID in the triple and/or the certificate is missing and the name in the triple matches the issuer name in the certificate.

(ii) No certificates were found signed by this issuer in which the subject name is longer than the subject name in this certificate (i.e. if there are cross certificates to two different ancestors, accept only those which lead to the closest ancestor).

(iii) The current time is within the validity interval of the certificate.

2) If any cross certificates were found (whether or not they were all eliminated as part of the filtering process), set Current-directory-name to the longest name that was found in any certificate and construct a set of PK/UID/Name triples for that name from the certificates which pass the filter and place them in the Trusted Key Set associated with their subject. Exit the ascending tree loop at this point and proceed directly to step 3. Note that this means that if there are cross certificates to an ancestor of the target but they are all rejected (for example if they have expired), the treewalk will ***not*** construct a chain through the least common ancestor and will ultimately fail unless a crosslink from a lower ancestor is found stored with its subject. This is a security feature.

3) If no cross certificates are found, find all the parent directory certificates for the directory whose name is in the Current-directory-name. Filter these to find only those which meet the following criteria:

(i) The current time is within the validity interval.

(ii) For some triple corresponding to the Current-directory-name, the public key in the triple verifies the signature on the certificate ***and either*** the UID in the triple matches the issuer UID in the certificate ***or*** the UID in the triple and/or the certificate is missing and the name in the triple matches the issuer name in the certificate.

4) Construct PK/UID/Name triples from the remaining certificates for the directory whose name is constructed by stripping the rightmost simple name from the Current-directory-name and place them in the Trusted-Key-Set.

5) Strip the rightmost simple name of the Current-directory-name.

6) Repeat from step (a) (testing to see if current-directory-name is the same as Common-point-Name).

c) Searching the "target side" of the tree for a crosslink:

1) Initialization: set Current-directory-name to the name supplied as input to this procedure.

2) Retrieve from the naming service all cross certificates associated with Current-directory-name. Filter to only those that meet the following criteria:

(i) The current time is within their validity interval.

(ii) The subject name is equal to Current-directory-name.

(iii) For some PK/UID/Name triple in the "Search-when-descending" list compiled while ascending the tree, the Public Key verifies the signature on the certificate and ***either*** the UID matches the issuer UID in the certificate ***or*** a UID is missing from the triple and/or the certificate and the Name in the triple matches the issuer name in the certificate.

(iv) There are no certificates found meeting criteria (ii) and (iii) matching a PK/UID/Name triple in the Search-when-descending list whose subject is a directory lower in the naming hierarchy.

3) If any qualifying certificates are found, construct PK/UID/Name triples for each of them; these should ***replace*** rather than supplement any triples already in the Trusted-key-set for that directory.

4) If after steps (b) and (c), there are no PK/UID/Name triples corresponding to Current-directory-name in Trusted-Key-Set, shorten Current-directory-name by one RDN (pushing it onto the Saved-RDNs stack) and repeat this process until Current-directory-name is equal to Least-common-ancestor-name ***or*** there is at least one triple in Trusted-key-set corresponding to Current-directory-name..

d) Descending the tree

While the list Saved-RDNs is not Empty the algorithm moves down the tree. At each step it does the following operations.

1) Remove the first RDN from Saved-RDNs and append it to the Current-directory-name.

2) Find all the child directory certificates for the directory whose name is in the current-directory-name.

3) Filter these certificates to find only those which meet the following criteria:

(i) The current time is within the validity interval.

(ii) For some PK/UID/Name triple in the Current-key-set for the parent directory, the Public Key verifies the signature on the certificate ***and either*** the UID matches the issuer UID of the certificate ***or*** the UID is missing from the triple and/or the certificate and the Name in the triple matches the issuer name in the certificate.

(iii) The issuer name in the certificate is a prefix of the subject name and the difference between the two names is the final RDN of Current-directory-name.

4) Take the key, UID, and name from each remaining certificate and form a new triple corresponding to Current-directory-name in Trusted-Key-Set. If this set is empty then the algorithm exits with the 'Incomplete-chain-of-trustworthy-CAs' error condition.

5) repeat from step (a), appending a new simple name to Current-directory-name.

e) Find public keys:

If there are no triples in the Trusted-Key-Set for the named principal, then the algorithm exits with the 'Target-has-no-keys-which-can-be-trusted' error condition. Otherwise, the Public Key and UID are extracted from each pair, duplicates are eliminated, and this set is returned as the Pub_keys.

3.10.10.2 Allowed Variations - Caching

Some use of caches can be implemented without affecting the semantics of the Get_Pub_Keys routine. For example, a crypto-cache could remember the public key that verified a signature in the past and could avoid the verification operation if the same key was used to verify the same data structure again. In some cases, however, it is impossible (or at least inconvenient) for a cache implementation to be completely transparent.

In particular, for good performance it is important that certificates not be re-retrieved from the naming service on every authentication. This must be balanced against the need to have changes to the contents of the naming service be reflected in DASS calls on a timely basis. There are two cases of interest: changes which cause an authentication which previously would have succeeded to fail and changes which cause an authentication which previously would have failed to succeed. These two cases are subject to different time constraints.

In general, changes that cause authentication to succeed must be reflected quite quickly - on the order of minutes. If a user attempts an operation, it fails, the user tracks down a system manager and causes the appropriate updates to take place, and the user retries the operation, it is unacceptable for the operation to continue to fail for an extended period because of stale caches.

Changes that cause authentication to fail must be reflected reliably within a bounded period of time for security reasons. If a user leaves the company, it must be possible to revoke his ability to authenticate within a relatively short period of time - say hours.

These constraints mean that a naming service cache which contains arbitrarily old information is unacceptable. To meet the second constraint, naming service cache entries must be timed out within a reasonable period of time unless in implementation verifies that the certificate is still present (a crypto-cache which lasted longer would be legal; rather than deleting a name service cache entry, in implementation might instead verify that the entry was still present in the naming service. This would avoid repeating the cryptographic "verify").

In order to assure that information cached for even a few hours not deny authentication for that extended period, it must be possible to bypass caches when the result would otherwise be a failure. Since the performance of authentication failures is not a serious concern, it is acceptable to expect that before an operation fails a retry will be made to the naming service to see if there are any new relevant certificates (or in certain obscure conditions, to see if any relevant certificates have been deleted).

If on a call to Get_Pub_Keys, the Try_Hard bit is True, then this procedure must return results based on the contents of the naming service no more than five minutes previous (this would normally be accomplished by ignoring name service caches and making all operations directly to the naming service). If the Try_Hard bit is False, this procedure may return results based on the contents of the naming service any time in the previous few hours, in the sense that it may ignore any certificate added in the previous few hours and may use any certificate deleted in the previous few hours. Procedures which call this routine with Try_Hard set to false must be prepared to call it again with Try_Hard True if their operation fails possibly from this result.

The exact timer values for "five minutes" and "a few hours" are expected to be implementation constants.

In the envisioned implementation, the entire "ascending treewalk" is retrieved, verified, and its digested contents cached when a principal first establishes credentials. A mechanism should be provided to refresh this information periodically for principals whose sessions might be long lived, but it would probably be acceptable in the unlikely event of a user's ancestor's keys changing to require that the user log out and log back in. This is consistent with the observed behavior of existing security mechanisms.

The descending treewalk, on the other hand, is expected to be maintained as a more conventional cache, where entries are kept in a fixed amount of memory with a "least recently used" replacement policy and a watchdog timer that assures that stale information is not kept indefinitely. A call to Get_Pub_Keys with Try_Hard set false would first check that cache for relevant certificates and only if none were found there would it go out to the naming service. If there were newer certificates in the naming service, they might not be found and an authentication might therefore fail.

When Try_Hard is false, an implementation may assume that certificates not in the cache do not exist so long as that assumption does not cause an authentication to falsely succeed. In that case, it may only make that assumption if the certificates have been verified to not exist within the revocation time (a few hours).

3.11 DASSlessness Determination Functions

In order to provide better interoperability with alternative authentication mechanisms and to provide backward compatibility with older (insecure) authentication mechanisms, it is sometimes important to be able to determine in a secure way what the appropriate authentication mechanism is for a particular named principal. For some applications, this will be done by a local mechanism, where either the person creating access control information must know and specify the mechanism for each principal or a system administrator on the node must maintain a database mapping names to mechanisms. Three applications come to mind where scaleability makes such mechanisms implausible:

a) To transparently secure proxy-based applications (like rlogin) in an environment where some hosts have been upgraded to support DASS and some have not, a node must be willing to accept connections authenticated only by their network addresses but only if they can be assured that such nodes do not have DASS installed. Access to a resource becomes secure without administrative action when all nodes authorized to access it have been upgraded.

   In this scenario, the server node must be able to determine whether the client node is DASSless in a secure fashion.

b) Similarly, in a mixed environment where some servers are running DASS and some are not, it may be desirable for clients to authenticate servers if they can but it would be unacceptable for a client to stop being able to access a DASSless server once DASS is installed on the client. In such a situation where server authentication is desirable but not essential, the client would like to determine in a secure fashion whether the server can accept DASS authentication.

c) In a DASS/Kerberos interoperability scenario, a server may decide that Kerberos authentication is "good enough" for principals that do not have DASS credentials without introducing trust in on-line authorities when DASS credentials are available. In parallel with case 1, we want it to be true that when the last principal with authority to access an object is upgraded to DASS, we automatically cease to trust PasswdEtc servers without administrative action on the part of the object owner. For this purpose, the authenticator must learn in a secure fashion that the principal is incapable of DASS authentication.

Reliably determining DASSlessness is optional for implementations of DASS and for applications. No other capabilities of DASS rely on this one.

The interface to the DASSlessness inquiry function is specified as a call independent of all others. This capability must be exposed to the calling application so that a server that receives a request and no token can ask whether the named principal should be believed without a token. It might improve performance and usability if in real interfaces DASSlessness were returned in addition to a bad status on the function that creates a token if the token is targeted toward a server incapable or processing it. An application could then decide whether to make the request without a token (and give up server authentication) or to abort the request.

3.11.1 Query DASSlessness

```
Query_DASSlessness(
                    --inputs
                        verifying_credentials          Credentials,
                        principal_name                 Name,
                    --outputs
                        alternate_authentication       Set of OIDs)
```

This function uses the verifying credentials to search for an alternative authentication mechanism certificate for the named principal or for any CA on the path between the verifying credentials and the named principal. Such a certificate is identical to an DASS X.509 certificate except that it lists a different algorithm identifier for the public key of the subject than that expected by DASS.

This function is implemented identically to Get_Pub_Keys *except*:

a) If in any set of certificates found, no valid DASS certificate is found and one or more certificates are found that would otherwise be valid except for an invalid subject public key OID, the OID from that certificate or certificates is returned and the algorithm terminates.

b) On initial execution, Try_Hard=False. If the first execution fails to retrieve any valid PK/UID pairs but also fails to find any invalid OID certificates, repeat the execution with Try_Hard=True.

c) If the either execution finds PK/UID pairs or if neither finds and invalid OID certificates, fail by returning a null set.

4 Certificate and message formats

4.1 ASN.1 encoding

Some definitions are taken from X.501 and X.509.

```
Dass DEFINITIONS ::=

BEGIN

--CCITT Definitions:
joint-iso-ccitt           OBJECT IDENTIFIER ::= {2}
ds                        OBJECT IDENTIFIER ::= {joint-iso-ccitt 5}
algorithm                 OBJECT IDENTIFIER ::= {ds 8}
encryptionAlgorithm       OBJECT IDENTIFIER ::= {algorithm 1}
hashAlgorithm             OBJECT IDENTIFIER ::= {algorithm 2}
signatureAlgorithm        OBJECT IDENTIFIER ::= {algorithm 3}
rsa                       OBJECT IDENTIFIER ::= {encryptionAlgorithm 1}

iso                       OBJECT IDENTIFIER ::= {1}
identified-organization   OBJECT IDENTIFIER ::= {iso 3}
ecma                      OBJECT IDENTIFIER ::= {identified-organization 12}
member-company            OBJECT IDENTIFIER ::= {ecma 2}
digital                   OBJECT IDENTIFIER ::= {member-company 1011}

--1989 OSI Implementors Workshop "Stable" Agreements
oiw                       OBJECT IDENTIFIER ::= {identified-organization 14}
dssig                     OBJECT IDENTIFIER ::= {oiw 7}
oiwAlgorithm              OBJECT IDENTIFIER ::= {dssig 2}
oiwEncryptionAlgorithm    OBJECT IDENTIFIER ::= {oiwAlgorithm 1}
oiwHashAlgorithm          OBJECT IDENTIFIER ::= {oiwAlgorithm 2}
oiwSignatureAlgorithm     OBJECT IDENTIFIER ::= {oiwAlgorithm 3}
oiwMD2                    OBJECT IDENTIFIER ::= {oiwHashAlgorithm 1} --null parameter
oiwMD2withRSA             OBJECT IDENTIFIER ::= {oiwSignatureAlgorithm 1} --null parameter

--X.501 definitions
AttributeType ::= OBJECT IDENTIFIER
AttributeValue ::= ANY
AttributeValueAssertion ::= SEQUENCE {AttributeType,AttributeValue}

Name ::= CHOICE {         --only one for now
                          RDNSequence
               }
```

```
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

DistinguishedName ::= RDNSequence

RelativeDistinguishedName ::= SET OF AttributeValueAssertion

--X.509 definitions (with proposed 1992 extensions presumed)

ENCRYPTED MACRO        ::=
BEGIN
TYPE NOTATION          ::= type(ToBeEnciphered)
VALUE NOTATION         ::= value(VALUE BIT STRING)
END          -- of ENCRYPTED

SIGNED MACRO           ::=
BEGIN
TYPE NOTATION          ::= type (ToBeSigned)
VALUE NOTATION         ::= value (VALUE
SEQUENCE{
             ToBeSigned,
             AlgorithmIdentifier,        --of the algorithm used to generate the signature
             ENCRYPTED OCTET STRING              --where the octet string is the result
                                                 --of the hashing of the value of
                                                 --"ToBeSigned"
             }
                                         )
END          -- of SIGNED

SIGNATURE MACRO        ::=
BEGIN
TYPE NOTATION          ::= type (OfSignature)
VALUE NOTATION         ::= value (VALUE
SEQUENCE {
             AlgorithmIdentifier,        --of the algorithm used to compute the signature
             ENCRYPTED OCTET STRING              -- where the octet string is a function
                                                 -- (e.g. a compressed or hashed version)
                                                 -- of the value 'OfSignature', which may
                                                 -- include the identifier of the algorithm
                                                 -- used to compute the signature
             }
                                         )
END          -- of SIGNATURE

Certificate ::= SIGNED SEQUENCE {
             version [0]                 Version DEFAULT v1988,
             serialNumber                CertificateSerialNumber,
             signature                   AlgorithmIdentifier,
             issuer                      Name,
```

```
        valid                   Validity,
        subject                 Name,
        subjectPublicKey        SubjectPublicKeyInfo,
        issuerUID [1]           IMPLICIT UID OPTIONAL,          -- v1992
        subjectUID [2]          IMPLICIT UID OPTIONAL           -- v1992
        }

        --The Algorithm Identifier for both the signature field and in the signature itself is:
        --          oiwMD2withRSA (1.3.14.7.2.3.1)

Version ::= INTEGER {v1988(0), v1992(1)}

CertificateSerialNumber ::= INTEGER

Validity ::= SEQUENCE {
        NotBefore     UTCTime,
        NotAfter      UTCTime
        }


AlgorithmIdentifier ::= SEQUENCE {
        algorithm     OBJECT IDENTIFIER,
        parameter     ANY DEFINED BY algorithm OPTIONAL
        }

--The algorithms we support in one context or another are:
        --oiwMD2withRSA (1.3.14.7.2.3.1) with parameter NULL
        --rsa (2.5.8.1.1) with parameter keysize INTEGER which is the keysize in bits
        --decDEA (1.3.12.1001.7.1.2) with optional parameter missing
        --decDEAMAC (1.3.12.2.1011.7.3.3) with optional parameter missing

SubjectPublicKeyInfo  ::=  SEQUENCE {
        algorithm     AlgorithmIdentifier,
                      -- rsa (2.5.8.1.1)
        subjectPublicKey        BIT STRING
                                -- the "bits" further decode into a DASS public key
        }

UID ::= BIT STRING

-- the following definitions are for Digital specified Algorithms

cryptoAlgorithm OBJECT IDENTIFIER ::= {digital 7}

decEncryptionAlgorithm    OBJECT IDENTIFIER ::= {cryptoAlgorithm 1}
decHashAlgorithm          OBJECT IDENTIFIER ::= {cryptoAlgorithm 2}
decSignatureAlgorithm     OBJECT IDENTIFIER ::= {cryptoAlgorithm 3}
decDASSLessness           OBJECT IDENTIFIER ::= {cryptoAlgorithm 6}
```

```
decMD2withRSA         OBJECT IDENTIFIER ::= {decSignatureAlgorithm 1}
decMD4withRSA         OBJECT IDENTIFIER ::= {decSignatureAlgorithm 2}
decDEAMAC             OBJECT IDENTIFIER ::= {decSignatureAlgorithm 3}

decDEA                OBJECT IDENTIFIER ::= {decEncryptionAlgorithm 2}

decMD2                OBJECT IDENTIFIER ::= {decHashAlgorithm 1}
decMD4                OBJECT IDENTIFIER ::= {decHashAlgorithm 2}


ShortPosixTime ::= INTEGER          -- number of seconds since base time

LongPosixTime ::= SEQUENCE {
            INTEGER,                -- number of seconds since base time
            INTEGER                 -- number of nanoseconds since second
            }


ShortPosixValidity ::=  SEQUENCE {
            notBefore   ShortPosixTime,
            notAfter    ShortPosixTime }

--Note: Annex C of X.509 prescribes the following format for the representation of a public key, --but
does not give the structure a name.

DASSPublicKey ::=  SEQUENCE {
            modulus     INTEGER,
            exponent    INTEGER
            }

DASSPrivateKey ::= SEQUENCE {
            p           INTEGER ,                          -- prime p
            q [0]       IMPLICIT INTEGER OPTIONAL ,        -- prime q
            mod[1]      IMPLICIT INTEGER OPTIONAL,         -- modulus
            exp [2]     IMPLICIT INTEGER OPTIONAL,         -- public exponent
            dp [3]      IMPLICIT INTEGER OPTIONAL ,        -- exponent mod p
            dq [4]      IMPLICIT INTEGER OPTIONAL ,        -- exponent mod q
            cr [5]      IMPLICIT INTEGER OPTIONAL ,        -- Chinese remainder coefficient
            uid[6]      IMPLICIT UID OPTIONAL,
            more[7]     IMPLICIT BIT STRING OPTIONAL       --Reserved for future use
            }


LocalUserName         ::= OCTET STRING
ChannelId             ::= OCTET STRING
VersionNumber         ::= OCTET STRING (SIZE(3))
                                    -- first octet is major version
```

```
                                        -- second octet is minor version
                                        -- third octet is ECO rev.
versionZero  VersionNumber ::= '000000'H

Authenticator ::= SIGNED SEQUENCE {
          type                          BIT STRING,
                                          -- first bit 'delegation required'
                                          -- second bit 'Mutual Authentication Requested'
          whenSigned                    LongPosixTime ,
          channelId  [3]                IMPLICIT ChannelId OPTIONAL
                    -- channel bindings are included when doing the
                    -- signature, but excluded when transmitting the
                    -- Authenticator
          }

                    -- uses decDEAMAC (1.3.12.2.1011.7.3.3)

EncryptedKey ::= SEQUENCE {
          algorithm             AlgorithmIdentifier,
                                        -- uses rsa (2.5.8.1.1)
          encryptedAuthKey              BIT STRING
                                        -- as defined in section 4.4.5
          }

SignatureOnEncryptedKey ::=  SIGNATURE EncryptedKey
                    -- uses oiwMD2withRSA (1.3.14.7.2.3.1)
                    -- Signature bits computed over EncryptedKey structure


LoginTicket ::= SIGNED SEQUENCE {
          version [0]                   IMPLICIT VersionNumber DEFAULT versionZero,
          validity                      ShortPosixValidity ,
          subjectUID                    UID ,
          delegatingPublicKey           SubjectPublicKeyInfo
          }
          -- uses oiwMD2withRSA (1.3.14.7.2.3.1)

Delegator ::= SEQUENCE {
          algorithm                     AlgorithmIdentifier
                                        -- decDEA encryption (1.3.12.1001.7.1.2)
          encryptedPrivKey              ENCRYPTED  DASSPrivateKey,
                                        -- (only p is included)
          }

UserClaimant ::=  SEQUENCE {
          userTicket [0]                IMPLICIT LoginTicket,
          evidence  CHOICE {
                    delegator [1]                 IMPLICIT Delegator ,
```

```
                                        -- encrypted delegation private key
                                        -- under DES authenticating key
                                        -- present if delegating
                        sharedKeyTicketSignature [2]
                                    IMPLICIT SignatureOnEncryptedKey
                                        -- present if not delegating
                        } ,
            userName [3]            IMPLICIT Name OPTIONAL
                                        -- name of user principal
            }

EncryptedKeyandUserName ::= SEQUENCE {
            encryptedKey            EncryptedKey ,
            username                LocalUserName
            }

SignatureOnEncryptedKeyandUserName ::=
            SIGNATURE EncryptedKeyandUserName
                         -- uses oiwMD2withRSA (1.3.14.7.2.3.1)
                        -- Signature bits computed over
                        -- EncryptedKeyandUserName structure
                        -- using node private key
            }

NodeClaimant ::= SEQUENCE {
            nodeTicket Signature[0]     IMPLICIT
                        SignatureOnEncryptedKeyandUserName,
            nodeName [1]            IMPLICIT Name OPTIONAL,
            username  [2]           IMPLICIT LocalUserName OPTIONAL
            }


AuthenticationToken ::= SEQUENCE {
            version [0]             IMPLICIT VersionNumber DEFAULT versionZero,
            authenticator [1]       IMPLICIT Authenticator ,
            encryptedKey [2]        IMPLICIT EncryptedKey OPTIONAL ,
                                        -- required if initiating token
            userclaimant [3]        IMPLICIT UserClaimant OPTIONAL ,
                                        -- missing if only doing node authentication
                                        -- required if not doing node authentication
            nodeclaimant [4]        IMPLICIT NodeClaimant OPTIONAL
                                       -- missing if only doing principal authentication
                                       -- required if not doing principal authentication
            }

MutualAuthenticationToken ::= CHOICE {
            v1Response [0] IMPLICIT  OCTET STRING (SIZE(6))
                                    -- Constructed as follows:  A single DES block
```

```
                                -- of eight octets is constructed from the two
                                -- integers in the timestamp.  First four bytes
                                -- are the high order integer encoded MSB
                                -- first; Last four bytes are the low order
                                -- integer encoded MSB first.  The block is
                                -- encrypted using the shared DES key, and
                                -- the first six bytes are the OCTET STRING.
                                -- With the [0] type and 6-byte length, the
                                -- MutualAuthenticationToken has a fixed
                                -- length of eight bytes.
        }

END
```

4.2 Encoding Rules

Whenever a structure is to be signed it must always be constructed the same way. This is particularly important where a signed structure has to be reconstructed by the recipient before the signature is verified. The rules listed below are taken from X.509.

- the definite form of length encoding shall be used, encoded in the minimum number of octets;
- for string types, the constructed form of encoding shall not be used;
- if the value of a type is its default value, it shall be absent;
- the components of a Set type shall be encoded in ascending order of their tag value;
- the components of a Set-of type shall be encoded in ascending order of their octet value;
- if the value of a Boolean type is true, the encoding shall have its contents octet set to 'FF'$_{16}$;
- each unused bits in the final octet of the encoding of a BitString value, if there are any, shall be set to zero;
- the encoding of a Real type shall be such that bases 8, 10 and 16 shall not be used, and the binary scaling factor shall be zero.

4.3 Version numbers and forward compatibility

The LoginTicket and AuthenticationToken structures contain a three octet version identifier which is intended to ease transition to future revisions of this architecture. The default value, and the value which should always be supplied by implementations of this version of the architecture is 0.0.0 (three zero octets). The first octet is the major version. An implementation of this version of the architecture should refuse to process data structures where it is other than zero, because changing it indicates that the interpretation of some subsidiary data structure has changed. The second octet is the minor version. An implementation of this version of the architecture should ignore the value of this octet. Some future version of the architecture may set a value other than zero and may specify some different processing of the remainder of the structure based on that different value. Such a change would be backward compatible and interoperable. The third octet is the ECO revision. No implementation should make any processing decisions based on the value of that octet. It may be logged, however, to help in debugging interoperability problems.

In the CDC protocol, there is also a three octet version numbering scheme, where versions 1.0.0 and 2.0.0 have been defined. Implementations should follow the same rules above and reject major version numbers greater than 2.

ASN.1 is inherently extensible because it allows new fields to be added "onto the end" of existing data structures in an unambiguous way. Implementations of DASS are encouraged to ignore any such additional fields in order to enhance backwards compatibility with future versions of the architecture. Unfortunately, commonly available ASN.1 compilers lack this capability, so this behavior cannot reasonably be required and may limit options for future extensions.

4.4 Cryptographic Encoding

Some of the substructures listed in the previous sections are specified as ENCRYPTED OCTET STRINGs containing encrypted information. DASS uses the DES, RSA, and MD2 cryptosystems Each of those cryptosystems specifies a function from octet string into another in the presence of a key (except MD2, which is keyless). This section describes how to form the octet strings on which the DES and RSA operations are performed.

4.4.1 Algorithm Independence vs. Key Parity

All of the defined encodings for DASS for secret key encryption are based on DES. It is intended, however, that other cryptosystems could be substituted without any other changes for formats or algorithms. The required "form factor" for such a cryptosystem is that it have a 64 bit key and operate on 64 bit blocks (this appears to be a common form factor for a cryptosystem). For this reason, DES keys are in all places treated as though they were 64 bits long rather than 56. Only in the operation of the algorithm itself are eight bits of the key dropped and key parity bits substituted. Choosing a key always involves picking a 64 bit random number.

4.4.2 Password Hashing

Encrypted credentials are encrypted using DES as described in the next section. The key for that encryption is derived from the user's password and name by the following algorithm:

a) Put the rightmost RDN of the user's name in canonical form according to BER and the X.509 encoding rules. For any string types that are case insensitive, map to upper case, and where matching is independent of number of spaces collapse all multiple spaces to a single space and delete leading and trailing spaces.

   Note: the RDN is used to add "salt" to the hash calculation so that someone can't precompute the hash of all the words in a dictionary and then apply them against all names. Deriving the salt from the last RDN of the name is a compromise. If it were derived from the whole name, all encrypted keys would be obsoleted when a branch of the namespace was renamed. If it were independent of name, interaction with a login agent would take two extra messages to retrieve the salt. With this scheme, encrypted keys are obsoleted by a change in the last RDN and if a final RDN is common to a large number of users, dictionary attacks against them are easier; but the common case works as desired.

b) Compute TEMP as the MD2 message digest of the concatenation of the password and the RDN computed above.

c) Repeat the following 40 times: Use the first 64 bits of TEMP as a DES key to encrypt the second 64 bits; XOR the result with the first 64 bits of TEMP; and compute a new TEMP as MD2 of the 128 bit result.

d) Use the final 64 bits of the result (called hash1) as the key to decrypt the encrypted credentials. Use the first 64 bits (called hash2) as the proof of knowledge of the password for presentation to a login agent (if any).

4.4.3 Digital DEA encryption

DES encryption is used in the following places:

- In the encryption of the encrypted credentials structure

- To encrypt the delegator in authentication tokens
- To encrypt the time in the mutual authenticator

In the first two cases, a varying length block of information coded in ASN.1 is encrypted. This is done by dividing the block of information into 8 octet blocks, padding the last block with zero bytes if necessary, and encrypting the result using the CBC mode of DES. A zero IV is used.

In the third case, a fixed length (8 byte) quantity (a timestamp) is encrypted. The timestamp is mapped to a byte string using "big endian" order and the block is encrypted using the ECB mode of DES.

4.4.4 Digital MAC Signing

DES signing is used in the Authenticator. Here, the signature is computed over an ASN.1 structure. The signature is the CBC residue of the structure padded to a multiple of eight bytes with zeros. The CBC is computed with an IV of zero.

4.4.5 RSA Encryption

RSA encryption is used in the Encrypted Shared Key. RSA encryption is best thought of as operating on blocks which are integers rather than octet strings and the results are also integers. Because an RSA encryption permutes the integers between zero and (modulus-1), it is generally thought of as acting on a block of size (keysizeinbits-1) and producing a block of size (keysizeinbits) where keysizeinbits is the smallest number of bits in which the modulus can be represented.

DASS only supports key sizes which are a multiple of eight bits.[9]

The encrypted shared key structure is laid out as follows:

- The DES key to be shared is placed in the last eight bytes
- The POSIX format creation time encoded in four bytes using big endian byte order is placed in the next four (from the end) bytes
- The POSIX format expiration time encoded in four bytes using big endian byte order is placed in the next four (from the end) bytes
- Four zero bytes are placed in the next four (from the end) bytes
- The first byte contains the constant '64' (decimal)
- All remaining bytes are filled with random bytes (the security of the system does not depend on the cryptographic randomness of these bytes, but they should not be a frequently repeating or predictable value. Repeating the DES key from the last bytes would be good).

The RSA algorithm is applied to the integer formed by treating the bytes above as an integer in big endian order and the resulting integer is converted to a BIT STRING by laying out the integer in 'big endian' order.

On decryption, the process is reversed; the decryptor should verify the four explicitly zero bytes but should not verify the contents of the high order byte or the random bytes.

[9] This restriction is only required to support interoperation with certain existing implementations. If the key size is not a multiple of eight bits, the high order byte may not be able to hold values as large as the mandated '64'. This is not a problem so long as the two high order bytes together are non-zero, but certain early implementations check for the value '64' and will not interoperate with implementations that use some other value.

4.4.6 oiwMD2withRSA Signatures

RSA-MD2 signatures are used on certificates, login tickets, shared key tickets, and node tickets. In all cases, a signature is computed on an ASN.1 encoded string using an RSA private key. This is done as follows:

- The MD2 algorithm is applied to the ASN.1 encoded string to produce a 128 bit message digest
- The message digest is placed in the low order bytes of the RSA block (big endian)
- The next two lowest order bytes are the ASN.1 'T' and 'L' for an OCTET STRING.
- The remainder of the RSA block is filled with zeros
- The RSA operation is performed, and the resulting integer is converted to an octet string by laying out the bytes in big endian order.

On verification, a value like the above ***or*** one where the message digest is present but the 'T' and 'L' are missing (zero) should be accepted for backwards compatibility with an earlier definition of this crypto algorithm.

4.4.7 decMD2withRSA Signatures

This algorithm is the same as the oiwMD2withRSA algorithm as defined above. We allocated an algorithm object identifier from the Digital space in case the definition of that OID should change. It will not be used unless the meaning of oiwMD2withRSA becomes unstable.

Annex A
Typical Usage

This annex describes one way a system could use DASS services (as described in section 3) to provide security services. While this example provided motivation for some of the properties of DASS, it is not intended to represent the only way that DASS may be used. This goes through the steps that would be needed to install DASS "from scratch".

A.1 Creating a CA

A CA is created by initializing its state. Each CA can sign certificates that will be placed in some directory in the name service. Before these certificates will be believed in a wider context than the sub-tree of the name space which is headed by that directory, the CA must be certified by a CA for the parent directory. The procedure below accomplishes this. For most secure operation, the CA should run on an off-line system and communicate with the rest of the network by interchanging files using a simple specialized mechanism such as an RS232 line or a floppy disk. It is assumed that access to the CA is controlled and that the CA will accept instructions from an operator.

- Call Install_CA to create the CA State.
  This state is saved within the CA system and is never disclosed.

- If this is the first CA in the namespace and the CA is intended to certify only members of a single directory, we are done. Otherwise, the new CA must be linked into the CA hierarchy by cross-certifying the parent and children of this CA. There is no requirement that CA hierarchies be created from the root down, but to simplify exposition, only this case will be described. The newly created CA must learn its name, its UID, the UID of its parent directory, and the public key of the parent directory CA by some out of band reliable means. Most likely, this would be done by looking up the information in the naming service and asking the CA operator to verify it. The CA then forms this information into a *parent* certificate and signs it using the Create_certificate function. It communicates the certificate to the network and posts it in the naming service.

- This name, UID, and public key of the new CA are taken to the CA of the parent directory, which verifies it (again by some unspecified out-of-band mechanism) and calls Create_Certificate to create a *child* certificate using its own Name and UID in the issuer fields. This certificate is also placed in the naming service.

A CA can sign certificates for more than one directory. In this case it is possible that a single CA will take the role of both CAs in the example above. The above procedure can be simplified in this case, as no interchange of information is required.

A.2 Creating a User Principal

A system manager may create a new user principal by invoking the Create_principal function supplying the principal's name, UID, and the public key/UID of the parent CA. The public key and UID must be obtained in a reliable out of band manner. This is probably by having knowledge of that information "wired into" the utility which creates new principals. At account creation time, the system manager must supply what will become the user's password. This might be done by having the user present and directly enter a password or by having the password selected by some random generator.

The trusted authority certificate and corresponding user public key generated by the Create_principal function are sent to the CA which verifies its contents (again by an out-of-band mechanism) and signs a corresponding certificate. The encrypted credentials, CA signed certificate, and trusted authority certificates are all placed in the naming service.

The process by which the password is made known to the user must be protected by some out-of-band mechanism.

In some cases the principal may wish to generate its own key, and not use the Encrypted_Credentials. (E.g. if the Principal is represented by a Smart Card). This may be done using a procedure similar to the one for creating a new CA.

A.3 Creating a Server Principal

A server also has a public/private key pair. Conceptually, the same procedure used to create a user principal can be used to create a server. In practice, the most important difference is likely to be how the password is protected when installing it on a server compared to giving it to a user.

A server may wish to retrieve (and store) its Encrypted Credentials directly and never have them placed in the naming service. In this case some other mechanism can be used (e.g. passing the floppy disk containing the encrypted credentials to the server). This would require a variant of the Initialize_Server routine which does not fetch the Encrypted Credentials from the naming service.

A.4 Booting a Server Principal

When the server first boots it needs its name (unreliably) and password (reliably). It then calls Initialize_Server to obtain its credentials and trusted authority certificates (which it will later need in order to authenticate users). These credentials never time out, and are expected to be saved for a long time. In particular the associated Incoming Timestamp List must be preserved while there are any timestamps on it. It is desirable to preserve the Cached Incoming Contexts as long as there are any contexts likely to be reused.

If a server wants to initiate associations on its own behalf then it must call Generate_Server_Ticket. It must repeat this at intervals if the expiration period expires.

A node that wishes to do node authentication (or which acts as a server under its own name) must be created as a server.

A.5 A user logs on to the network

The system that the user logs onto finds the user's name and password. It then calls Network_Login to obtain credentials for the user. These credentials are saved until the user wants to make a network connection. The credentials have a time limit, so the user will have to obtain new credentials in order to make connections after the time limit. The credentials are then checked by calling Verify_Principal_Name, in order to check that the key specified in the encrypted credentials has been certified by the CA.

If the system does source node authentication it will call Combine_credentials, once the local username has been found. (This can either be found by looking the principal's global name up in a file, or the user can be asked to give the local name directly. Alternatively the user can be asked to give his local username, which the system looks up to find the global name).

A.6 An Rlogin (TCP/IP) connection is made

When the user calls a modified version of the rlogin utility, it calls Create_token in order to create the Initial Authentication Token, which is passed to the other system as part of the rlogin protocol. The rlogind utility at the destination node calls Accept_token to verify it. It then looks up in a local rhosts-like database to determine whether this global user is allowed access to the requested destination account. It calls Verify_principal_name and/or Verify_node_name to confirm the identity of the requester. If access is allowed, the connection is accepted and the Mutual Authentication Token is returned in the response message.

The source receives the returned Mutual Authentication Token and uses it to confirm it communicating with the correct destination node.

Rlogind then calls Combine_credentials to combine its node/account information with the global user identification in the received credentials in case the user accesses any network resources from the destination system.

A.7 A Transport-Independent Connection

As an alternative to the description in A.6, an application wishing to be portable between different underlying transports may call create_token to create an authentication token which it then sends to its peer. The peer can then call accept_token and verify_principal_name and learn the identity of the requester.

Annex B
Support of the GSSAPI

In order to support applications which need to be portable across a variety of underlying security mechanisms, a "Generic Security Service API" (or GSSAPI) was designed which gives access to a common core of security services expected to be provided by several mechanisms. The GSSAPI was designed with DASS, Kerberos V4, and Kerberos V5 in mind, and could be written as a front end to any or all of those systems. It is hoped that it could serve as an interface to other security systems as well.

Application portability requires that the security services supported be comparable. Applications using the GSSAPI will not be able to access all of the features of the underlying security mechanisms. For example, the GSSAPI does not allow access to the "node authentication" features of DASS. To the extent the underlying security mechanisms do not support all the features of GSSAPI, applications using those features will not be portable to those security mechanisms. For example, Kerberos V4 does not support delegation, so applications using that feature of the GSSAPI will not be portable to Kerberos V4.

This annex explains how the GSSAPI can be implemented using the primitive services provided by DASS.

B.1 Summary of GSSAPI

The latest draft of the GSSAPI specification is available as an internet draft. The following is a brief summary of that evolving document and should not be taken as definitive. Included here are only those aspects of GSSAPI whose implementation would be DASS specific.

The GSSAPI provides four classes of functions: Credential Management, Context-Level Calls, Per-message calls, and Support Calls; two types of objects: Credentials and Contexts; and two kinds of data structures to be transmitted as opaque byte strings: Tokens and Messages. Credentials hold keys and support information used in creating tokens. Contexts hold keys and support information used in signing and encrypting messages.

The Credential Management functions of GSSAPI are "incomplete" in the sense that one could not build a useful security implementation using only GSSAPI. Functions which create credentials based on passwords or smart cards are needed but not provided by GSSAPI. It is envisioned that such functions would be invoked by security mechanism specific functions at user login or via some separate utility rather than from within applications intended to be portable. The Credential Management functions available to portable applications are:

- GSS_Acquire_cred: get a handle to an existing credential structure based on a name or process default.

- GSS_Release_cred: release credentials after use.

The Context-Level Calls use credentials to establish contexts. Contexts are like connections: they are created in pairs and are generally used at the two ends of a connection to process messages associated with that connection. The Context-Level Calls of interest are:

- GSS_Init_sec_context: given credentials and the name of a destination, create a new context and a token which will permit the destination to create a corresponding context.

- GSS_Accept_sec_context: given credentials and an incoming token, create a context corresponding to the one at the initiating end and provide information identifying the initiator.

- GSS_Delete_sec_context: delete a context after use.

The Per-Message Calls use contexts to sign, verify, encrypt, and decrypt messages between the holders of matching contexts. The Per-Message Calls are:

- GSS_Sign: Given a context and a message, produces a string of bytes which constitute a signature on a provided message.

- GSS_Verify: Given a context, a message, and the bytes returned by GSS_Sign, verifies the message to be authentic (unaltered since it was signed by the corresponding context).

- GSS_Seal: Given a context and a message, produces a string of bytes which include the message and a signature; the message may optionally be encrypted.

- GSS_Unseal: Given a context and the string of bytes from GSS_Seal, returns the original message and a status indicating its authenticity.

The Support Calls provide utilities like translating names and status codes into printable strings.

B.2 Implementation of GSSAPI over DASS

B.2.1 Data Structures

The objects and data structures of the GSSAPI do not map neatly into the objects and data structures of the DASS architecture. This section describes how those data structures can be implemented using the DASS data structures and primitives

Credential handles correspond to the credentials structures in DASS, where the portable API assumes that the credential structures themselves are kept from applications and handles are passed to and from the various subroutines.

Context initialization tokens correspond to the tokens of DASS. The GSSAPI prescribes a particular ASN.1 encoded form for tokens which includes a mechanism specific bit string within it. An implementation of GSSAPI should enclose the DASS token within the GSSAPI "wrapper".

Context handles have no corresponding structure in DASS. The Create_token and Accept_token calls of DASS return a shared key and instance identifier. An implementation of the GSSAPI must take those values along with some other status information and package it as a "context" opaque structure. These data structures must be allocated and freed with the appropriate calls.

Per-message tokens and sealed messages have no corresponding data structure within DASS. To fully support the GSSAPI functionality, DASS must be extended to include this functionality. These data structures are created by cryptographic routines given the keys and status information in context structures and the messages passed to them. While not properly part of the DASS architecture, the formats of these data structures are included in section C.3.

B.2.2 Procedures

This section explains how the functions of the GSSAPI can be provided in terms of the Services Provided by DASS. Not all of the DASS features are accessible through the GSSAPI.

B.2.2.1 GSS_Acquire_cred

The GSSAPI does not provide a mechanism for logging in users or establishing server credentials. It assumes that some system specific mechanism created those credentials and that applications need some mechanism for getting at them. A model implementation might save all credentials in a node-global pool indexed by some sort of credential name. The credentials in the pool would be access controlled by some local policy which is not concern of portable applications. Those applications would simply call GSS_Acquire_cred and if they passed the access control check, they would get a handle to the credentials which could be used in subsequent calls.

B.2.2.2 GSS_Release_cred

This call corresponds to the "delete_credentials" call of DASS.

B.2.2.3 GSS_Init_sec_context

In the course of a normal mutual authentication, this routine will be called twice. The procedure can determine whether this is the first or second call by seeing whether the "input_context_handle" is zero (it will be on the first call). On the first call, it will use the DASS Create_token service to create a token and it will also allocate and populate a "context" structure. That structure will hold the key, instance identifier, and mutual authentication token returned by Create_token and will in addition hold the flags which were passed into the Init_sec_context call. The token returned by Init_sec_context will be the DASS token included in the GSSAPI token "wrapper". The DASS token will include the optional principal name.

If mutual authentication is not requested in the GSSAPI call, the mutual authentication token returned by DASS will be ignored and the initial call will return a COMPLETE status. If mutual authentication is requested, the mutual authentication token will be stored in the context information and a CONTINUE_NEEDED status returned.

On the second call to GSS_Init_sec_context (with input_context_handle non-zero), the returned token will be compared to the one in the context information using the Compare_mutual_token procedure and a COMPLETE status will be returned if they match.

B.2.2.4 GSS_Accept_sec_context

This routine in GSSAPI accepts an incoming token and creates a context. It combines the effects of a series of DASS functions. It could be implemented as follows:

- Remove the GSSAPI "wrapper" from the incoming token and pass the rest and the credentials to "Accept_token". Accept_token produces a mutual authentication token and a new credentials structure. If delegation was requested, the new credentials structure will be an output of GSS_Accept_sec_context. In any case, it will be used in the subsequent steps of this procedure.

- Use the DASS Get_principal_name function to extract the principal name from the credentials produced by Accept_token. This name is one of the outputs of "GSS_Accept_sec_context.

- Apply the DASS Verify_principal_name to the new credentials and the retrieved name to authenticate the token as having come from the named principal.

- Create and populate a context structure with the key and timestamp returned by Accept_token and a status of COMPLETE. Return a handle to that context.

- If delegation was requested, return the new credentials from GSS_Accept_sec_context. Otherwise, call Delete_credentials.

- If mutual authentication was requested, wrap the mutual authentication token from Accept_token in a GSSAPI "wrapper" and return it. Otherwise return a null string.

B.2.2.5 GSS_Delete_sec_context

This routine simply deletes the context state. No calls to DASS are required.

B.2.2.6 GSS_Sign

This routine takes as input a context handle and a message. It creates a per_msg_token by computing a digital signature on the message using the key and timestamp in the context block. No DASS services are required. If additional cryptographic services were requested (replay detection or sequencing), a timestamp or sequence number must be prepended to the message and sent with the signature. The syntax for this message is listed in section C.3.

B.2.2.7 GSS_Verify

This routine repeats the calculation of the sign routine and verifies the signature provided. If replay detection or sequencing services are provided, the context must maintain as part of its state information containing the sequence numbers or timestamps of messages already received and this one must be checked for acceptability.

B.2.2.8 GSS_Seal

This routine performs the same functions as Sign but also optionally encrypts the message for privacy using the shared key and encapsulates the whole thing in a GSSAPI specified ASN.1 wrapper.

B.2.2.9 GSS_Unseal

This routine performs the same functions as GSS_Verify but also parses the data structure including the signature and message and decrypts the message if necessary.

B.3 Syntax

The GSSAPI specification recommends the following ASN.1 encoding for the tokens and messages generated through the GSSAPI:

```
--optional top-level token definitions to frame
-- different mechanisms

GSSAPI DEFINITIONS ::=

BEGIN

MechType ::= OBJECT IDENTIFIER
-- data structure definitions

ContextToken ::=
-- option indication (delegation, etc.) indicated
-- within mechanism-specific token
[APPLICATION 0] IMPLICIT SEQUENCE {
      thisMech MechType,
      responseExpected BOOLEAN,
      innerContextToken ANY DEFINED BY MechType
         -- contents mechanism-specific
      }
```

```
PerMsgToken ::=
-- as emitted by GSS_Sign and processed by
-- GSS_Verify
[APPLICATION 1] IMPLICIT SEQUENCE {
      thisMech MechType,
      innerMsgToken ANY DEFINED BY MechType
        -- contents mechanism-specific
      }

SealedMessage ::=
-- as emitted by GSS_Seal and processed by
-- GSS_Unseal
[APPLICATION 2] IMPLICIT SEQUENCE {
      sealingToken PERMSGTOKEN,
      confFlag BOOLEAN,
      userData OCTET STRING
        -- encrypted if confFlag TRUE
      }
```

The object identifier for the DASS MechType is 1.3.12.2.1011.7.5.

The innerContextToken of a token is a DASS token or mutual authentication token.

The innerMsgToken is a null string in the case where the message is encrypted and the token is included as part of a SealedMessage. Otherwise, it is an eight octet sequence computed as the CBC residue computed using a key and string of bytes defined as follows:

- Pad the message provided by the application with 1-8 bytes of pad to produce a string whose length is a multiple of 8 octets. Each pad byte has a value equal to the number of pad bytes.

- Compute the key by taking the timestamp of the association (two four byte integers laid out in big endian order with the most significant integer first), complementing the high order bit (to avoid aliasing with mutual authenticators), and encrypting the block in ECB mode with the shared key of the association.

The userData field of a SealedMessage is exactly the application provided byte string if confFlag=FALSE. Otherwise, it is the application supplied message encrypted as follows:

- Pad the message provided by the application with 1-8 bytes of pad to produce a string whose length = 4 (mod 8). Each pad byte has a value equal to the number of pad bytes.

- Append a four byte CRC32 computed over the message + pad.

- Compute a key by taking the timestamp of the association (two four byte integers laid out in big endian order with the most significant integer first), complementing the high order bit (to avoid aliasing with mutual authenticators), and encrypting the block in ECB mode with the shared key of the association.

- Encrypt the message + pad + CRC32 using CBC and the key computed in the previous step.

A note of the logic behind the above:

- Because the shared key of an association may be reused by many associations between the same pair of principals, it is necessary to bind the association timestamp into the messages somehow to prevent messages from a previous association being replayed into a new sequence. The technique above of generating an association key accomplishes this and has a side benefit. An implementation may with to keep the long term keys out of the hands of applications for purposes of confinement but may wish to put the encryption associated with an association in process context for reasons of performance. Defining an association key makes that possible.

- The reason that the association specific key is not specified as the output of Create_token and Accept_token is that the DCE RPC security implementation requires that a series of associations between two principals always have the same key and we did not want to have to support a different interface in that application.
- The CRC32 after pad constitutes a cheap integrity check when data is encrypted.
- The fact that padding is done differently for encrypted and signed messages means that there are no threats related to sending the same message encrypted and unencrypted and using the last block of the encrypted message as a signature on the unencrypted one.

Annex C
Imported ASN.1 definitions

This annex contains extracts from the ASN.1 description of X.509 and X.500 definitions referenced by the DASS ASN.1 definitions.

```
CCITT DEFINITIONS ::=

BEGIN
joint-iso-ccitt  OBJECT IDENTIFIER ::= {2}
ds OBJECT IDENTIFIER ::= {joint-iso-ccitt 5}
algorithm OBJECT IDENTIFIER ::= {ds 8}

iso OBJECT IDENTIFIER ::= {1}
identified-organization OBJECT IDENTIFIER ::= {iso 3}
ecma OBJECT IDENTIFIER ::= {identified-organization 12}
digital OBJECT IDENTIFIER ::= { ecma 1011 }

-- X.501 definitions

AttributeType ::= OBJECT IDENTIFIER
AttributeValue ::= ANY
            -- useful ones are
                    --          OCTET STRING ,
                    --          PrintableString ,
                    --          NumericString ,
                    --          T61String ,
                    --          VisibleString

AttributeValueAssertion ::= SEQUENCE {AttributeType, AttributeValue}

Name ::= CHOICE {-- only one possibility for now --
                    RDNSequence}

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

DistinguishedName ::= RDNSequence
```

```
RelativeDistinguishedName ::= SET OF AttributeValueAssertion

-- X.509 definitions

Certificate ::= SIGNED SEQUENCE {
                    version         [0]         Version DEFAULT 1988 ,
                    serialNumber                SerialNumber ,
                    signature                   AlgorithmIdentifier ,
                    issuer                      Name,
                    valid                       Validity,
                    subject                     Name,
                    subjectPublicKey        SubjectPublicKeyInfo }

Version ::= INTEGER { 1988(0)}
SerialNumber INTEGER
Validity ::= SEQUENCE{
          notBefore             UTCTime,
          notAfter              UTCTime}

SubjectPublicKeyInfo  ::=  SEQUENCE {
          algorithm             AlgorithmIdentifier ,
          subjectPublicKey      BIT STRING
          }

AlgorithmIdentifier ::= SEQUENCE {
          algorithm     OBJECT IDENTIFIER ,
               parameters          ANY DEFINED BY algorithm OPTIONAL}

ALGORITHM MACRO
BEGIN
TYPE NOTATION           ::= "PARAMETER" type
VALUE NOTATION          ::= value (VALUE OBJECT IDENTIFIER)
END -- of ALGORITHM

ENCRYPTED MACRO
BEGIN
TYPE NOTATION           ::=type(ToBeEnciphered)
VALUE NOTATION          ::= value(VALUE BIT STRING)
          -- the value of the bit string is generated by
          -- taking the octets which form the complete
          encoding (using the ASN.1 Basic Encoding Rules)
          -- of the value of the ToBeEnciphered type and
          -- applying an encipherment procedure to those octets--
END
```

```
SIGNED MACRO            ::=
BEGIN
TYPE NOTATION           ::= type (ToBeSigned)
VALUE NOTATION          ::= value(VALUE
SEQUENCE{
            ToBeSigned,
            AlgorithIdentifier, -- of the algorithm used to generate the signature
            ENCRYPTED OCTET STRING
            -- where the octet string is the result
            -- of the hashing of the value of
            "ToBeSigned"
END -- of SIGNED

SIGNATURE MACRO         ::=
BEGIN
TYPE NOTATION           ::= type(OfSignature)
VALUE NOTATION          ::= value(VALUE
            SEQUENCE{
                        AlgorithmIdentifier,
                        -- of the algorithm used to compute the signature
                        ENCRYPTED OCTET STRING
                        -- where the octet string is a function (e.g. a compressed or
                        -- hashed version) of the value "OfSignature", which may
                        -- include the identifier of  the algorithm used to compute
                        -- the signature--}
                                    )
END -- of SIGNATURE

-- X.509 Annex H (not part of the standard)

encryptionAlgorithm OBJECT IDENTIFIER ::= {algorithm 1}

rsa ALGORITHM
            PARAMETER KeySize
            ::= {encryptionAlgorithm 1}

KeySize ::= INTEGER

END
```

Glossary

authentication The process of determining the identity (usually the name) of the other party in some communication exchange.

authentication context
Cached information used during a particular instance of authentication and including a shared symmetric (DES) key as well as components of the authentication token conveyed during establishment of this context.

authentication token
Information conveyed during a strong authentication exchange that can be used to authenticate its sender. An authentication token can, but is not necessarily limited to, include the claimant identity and ticket, as well as signed and encrypted secret key exchange messages conveying a secret key to be used in future cryptographic operations. An authentication token names a particular protocol data structure component.

authorization The process of determining the rights associated with a particular principal.

certificate The public key of a particular principal, together with some other information relating to the names of the principal and the certifying authority, rendered unforgeable by encipherment with the private key of the certification authority that issued it.

certification authority
An authority trusted by one or more principals to create and assign certificates.

claimant The party that initiates the authentication process. In the DASS architecture, claimants possess credentials which include their identity, authenticating private key and a ticket certifying their authenticating public key.

credentials Information "state" required by principals in order to for them to authenticate. Credentials may contain information used to initiate the authentication process (claimant information), information used to respond to an authentication request (verifier information), and cached information useful in improving performance.

cryptographic checksum
Information which is derived by performing a cryptographic transformation on the data unit. This information can be used by the receiver to verify the authenticity of data passed in cleartext

decipher To reverse the effects of encipherment and render a message comprehensible by use of a cryptographic key.

delegation The granting of temporary credentials that allow a process to act on behalf of a principal.

delegation key A short term public/private key pair used by a claimant to act on behalf of a principal for a bounded period. The delegation public key appears in the ticket, whereas the delegation private key is used to sign secret key exchange messages.

DES Data Encryption Standard: a symmetric (secret key) encryption algorithm used by DASS. An alternate encryption algorithm could be substituted with little or no disruption to the architecture.

DES key A 56-bit secret quantity used as a parameter to the DES encryption algorithm.

digital signature A value computed from a block of data and a key which could only be computed by someone knowing the key. A digital signature computed with a secret key can only be verified by someone knowing that secret key. A digital signature computed with a private key can be verified by anyone knowing the corresponding public key.

encipher To render incomprehensible except to the holder of a particular key. If you encipher with a secret key, only the holder of the same secret can decipher the message. If you encipher with a public key, only the holder of the corresponding private key can decipher it.

initial trust certificate
A certificate signed by a principal for its own use which states the name and public key of a trusted authority.

global user name A hierarchical name for a user which is unique within the entire domain of discussion (typically the network).

local user name A simple (non-hierarchical) name by which a user is known within a limited context such as on a single computer.

principal Abstract entity which can be authenticated by name. In DASS there are user principals and server principals.

private key Cryptographic key used in asymmetric (public key) cryptography to decrypt and/or sign messages. In asymmetric cryptography, knowing the encryption key is independent of knowing the decryption key. The decryption (or signing) private key cannot be derived from the encrypting (or verifying) public key.

proxy A mapping from an external name to a local account name for purposes of establishing a set of local access rights. Note that this differs from the definition in ECMA TR/46.

public key Cryptographic key used in asymmetric cryptography to encrypt messages and/or verify signatures.

RSA The Rivest-Shamir-Adelman public key cryptosystem based on modular exponentiation where the modulus is the product of two large primes. When the term RSA key is used, it should be clear from context whether the public key, the private key, or the public/private pair is intended.

secret key Cryptographic key used in symmetric cryptography to encrypt, sign, decrypt and verify messages. In symmetric cryptography, knowledge of the decryption key implies knowledge of the encryption key, and vice-versa.

sign A process which takes a piece of data and a key and produces a digital signature which can only be calculated by someone with the key. The holder of a corresponding key can verify the signature.

source The initiator of an authentication exchange.

strong authentication
Authentication by means of cryptographically derived authentication tokens and credentials. The actual working definition is closer to that of "zero knowledge" proof: authentication so as to not reveal any information usable by either the verifier, or by an eavesdropping third party, to further their potential ability to impersonate the claimant.

target The intended second party (other than the source) to an authentication exchange.

ticket A data structure certifying an authenticating (public) key by virtue of being signed by a user principal using their (long term) private key. The ticket also includes the UID of the principal.

trusted authority The public key, name and UID of a certification authority trusted in some context to certify the public keys of other principals.

UID A 128 bit unique identifier produced according to OSF standard specifications.

user key A "long term" RSA key whose private portion authenticates its holder as having the access rights of a particular person.

verify To cryptographically process a piece of data and a digital signature to determine that the holder of a particular key signed the data.

verifier The party who will perform the operations necessary to verify the claimed identity of a claimant.

Author's Address

Charles Kaufman
Digital Equipment Corporation
LKG 1-2/A19
550 King Street
Littleton, MA 01460

Phone: (508) 486-7329

Email: kaufman@dsmail.enet.dec.com

General comments on this document should be sent to cat-ietf@mit.edu. Minor corrections should be sent to the author.