PROJECT ATHENA TECHNICAL PLAN

Section E.2.1

Kerberos Authentication and Authorization System

by S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer

Κερβεροσ; also spelled *Cerberus*. *"n. The watch dog of Hades, whose duty it was to guard the entrance—against whom or what does not clearly appear; . . . is known to have had three heads. . ."*

—Ambrose Bierce, *The Enlarged Devil's Dictionary*

This document describes the assumptions, short and long term goals, and system model for a network authentication system, named Kerberos, for the Athena environment. An appendix specifies the detailed design and protocols to support these goals, and a set of UNIX [1] manual pages, not included here, describes an implementation for Berkeley 4.3 UNIX of both user interface commands and also library interfaces for clients and servers. The next section of the technical plan, E.2.2, describes a set of network applications that use Kerberos for authentication.

Definitions

Accounting	Measuring resource usage attributable to a particular client.
Authentication	Verifying the claimed identity of a client or service.
Authorization	Allowing an authenticated client to use a particular service.
Client	A program that makes use of a network service, on behalf of a user.
KDBM	Kerberos Data Base Manager, a system that maintains and provides an interface for update of authoritative Kerberos data consisting of principal identifiers and private keys for both clients and services.
Kerberos	Aside from the 3-headed dog guarding Hades, the name given to the Athena authentication service, the protocol used by that service, and the libraries used to invoke the authentication and authorization services.
KKDS	Kerberos Key Distribution Service, a network service that supplies tickets and temporary session keys; or Kerberos Key Distribution Server, an instance of that service.
Principal	A uniquely named client or server instance that participates in a network communication.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Principal identifier	The name used to uniquely identify each different client and server.
Private key	An encryption key between a principal and the KKDS, distributed outside the system, with a long lifetime;
Seal	To encipher a record containing several fields, in such a way that the fields cannot be individually replaced without either knowledge of the key or leaving evidence of tampering.
Session key	A temporary encryption key used between two principals, with a lifetime limited to the duration of a single communications "session".
Ticket	A record that authenticates a client to a service; it contains the client's identity, a session key, and a timestamp, all of which is sealed by encryption using the service's private key.

The term "principal," being somewhat formal, is replaced with the word "user" in this document wherever the context permits that usage without confusion.

1. Introduction to Kerberos

Purpose of This Plan

Most conventional time-sharing systems require a prospective user to identify him or herself and to authenticate that identity before using its services. In an environment consisting of a network that connects prospective clients with services, a network service has a corresponding need to identify and authenticate its clients. When the client is a user of a time-sharing system, one approach is for the service to trust the authentication that was performed by the time-sharing system. For example, the network applications *lpr* and *rcp* provided with Berkeley 4.3 UNIX trust the user's time-sharing system to reliably authenticate its clients.

In contrast with the time-sharing system, in which a protection wall separates the operating system from its users, a workstation is under the complete control of its user, to the extent that the user can run a private version of the operating system, or even replace the machine itself. As a result, a network service cannot rely on the integrity of the workstation operating system when it (the network service) performs authentication.

This plan extends the conventional notions of authentication, authorization, and accounting to the network environment with untrusted workstations. It establishes a trusted third-party service named Kerberos that can perform authentication to the mutual satisfaction of both clients and services. The authentication approach allows for integration with authorization and accounting facilities. The resulting design is also applicable to a mixed time-sharing/network environment in which a network service is not willing to rely on the authentication performed by the client's time-sharing system.

Goals of Kerberos

Authentication

Authentication is not an end in itself, but rather a tool to support both integrity and authorization. Its basic purpose is to prevent fraudulent connection requests. The goal of Kerberos is to support both one-way and mutual authentication of principals, to the granularity of at least an individual user and specific service instance.

Authorization

Authentication can imply a coarse-grained authorization—for example, some services may allow anyone who can be reliably authenticated by the local Kerberos to use the service. In cases where more selective authorization is needed, the goal of Kerberos is to allow different services to implement different authorization models, and to allow those authorization models to assume that authentication of user identities is reliable.

Accounting

Given an authenticated client, the goal of accounting is to support either quotas charged against the client (to limit consumption), e.g. disk quota, and/or charges based on consumption, e.g. $.01 per page printed. The goal of Kerberos is to permit modular attachment of an integrated, secure, reliable accounting system.

Requirement Examples

Some examples of network services best illustrate the requirements of user authentication and authorization:

- Printing—Only members of a certain group may use a printer that belongs to that group, an expensive and relatively scarce shared resource. On a more public printer, users may be billed for printing, and may have a priori limits on their use.
- Remote File Access—Only designated users may perform operations on a given remote file system or virtual disk. Different users may have different permissions allowed, e.g. only the owner may write, while others may read.
- Remote Login—Only authorized users may *rlogin* to centrally-managed hosts, or to a private workstation.
- Window system—The user of a network-driven display may want to limit the ability of others to create or manipulate windows on that display.
- Mail—Only the addressee should be able to pick up his or her own mail at the Post Office.
- Service Management service—Users may be authorized to create, modify, or destroy records that control various services. For example, system administrators may have unlimited privileges, while the teaching assistant for a subject may only be allowed to authorize use of the libraries belonging to the subject. A user may be able to add or delete his or her own name on a public mailing list, but not to affect any other user's record in that list.

Other Requirements Assumed by the Design

- The authentication requirement is two-way. That is, the service learns with confidence who the client is, and the client, if it wishes, can be certain that the correct service is being used.
- No cleartext passwords should be transmitted over the net;
- No cleartext passwords should be stored on servers;
- At clients, cleartext passwords should be handled for the shortest possible time and then destroyed.

- The design should confine any authentication compromises to the current session or the current user.
- Authentication has a limited lifetime, of the order of a single login session, but may be re-used within that lifetime;
- Network authentication should go on largely unnoticed in normal cases; the traditional model of password-mediated login should be the only point that the user notices that authentication is occurring.
- The design should minimize the effort needed to modify network services that previously used other means of authentication.

Future requirement possibilities

The following are not currently considered essential, but may be re-evaluated as experience increases:

- Forwarding of authentications, so that one service can do part of a job, then invoke another service to complete it, under the credentials of the original client.
- Revocation of authentication or authorization *within* a login session.

2. Assumptions Surrounding Authentication

Assumed Physical and Operational Security Environment

From a security perspective, the environment will include:

- both public and private workstations. Public workstations are in areas with minimal physical security; private workstations are under physical and administrative control of individuals with no responsibility to central network administration.
- a campus network without link encryption, composed of local nets of varying types linked by gateways to a backbone net; the local nets are widely dispersed physically and thus are very vulnerable to security attacks; the backbone and gateways are in locked closets and therefore are moderately secure.
- centrally-operated servers in locked rooms, assumed to operate under moderate physical security with known legitimate software;
- a small number of centrally-operated servers, such as the Kerberos authentication server, that operate under considerable physical security.

Relevant Threats and Risks

The environment is not appropriate for sensitive data or high risk operations, such as bank transactions, classified government data, student grades, controlling dangerous experiments, and such. The risks are primarily uncontrolled use of resources by unauthorized parties, violations of the integrity of either the system's or user's resources, and wholesale violations of privacy such as casual browsing through personal files.

The primary security threats result from the potential of a workstation user to forge the identity of another user in order to gain unauthorized access to data and/or resources. Since a workstation, including its operating system and network interface, is under the complete control of the user, the user can attempt to masquerade as another user or even as another host. In lieu of the authentication provided by a centrally administered time sharing system, an *authentication service* is required to counter such attempts.

Privacy of data being transported across the network is currently a low priority, except where it is necessary to prevent subsequent violations of integrity, e.g. the transmission of passwords. When the cost of providing communications privacy can be significantly reduced, it will attain higher priority.

Traffic analysis and covert channels are not an issue.

Assumptions about Encryption

The private-key Data Encryption Standard (DES), when used in single-encryption mode, is assumed to provide enough security for campus applications that cryptanalysis is not a significant threat.

DES implementations are available in both hardware and software. Because system-integrated hardware implementations are not yet sufficiently low in cost, it is assumed that software implementations will be used for Kerberos except optionally at a small number of sites (Key Distribution Servers) that do a lot of encryption.

DES implementations may not be exported from the U.S. without special license. For this reason, the Kerberos design makes the cryptosystem a modular, replaceable unit. *The initial implementation of Kerberos is based on DES.*

Global Clock Availability

The design of Kerberos assumes that system clocks are loosely synchronized—within a few minutes—on all machines that run Kerberos-authenticated services, and that this *global time* is similarly available to all workstations that use Kerberos. We do not assume that all workstations correctly maintain the time, but in order to request authentication tickets, a workstation is required to maintain its clock within the allowable margin. *Timeservers* provide the official time, and other systems synchronize periodically, for example, at system boot time.

Service Management System

This plan connects with the Athena Service Management System in a several ways. The Athena Service Management System provides authoritative information for Kerberos as well as the related naming system.

3. Naming

This plan assumes a means for numbering network hosts and service ports so that clients may request connection to services, including Kerberos itself. If a naming system for services is also available, it is important that the service names can be congruent with Kerberos principal identifiers (defined in the next paragraph) that are used to authenticate services. In addition, Kerberos clients can make use of such a name service to locate Kerberos service itself. The design of Kerberos is modular; it can operate (somewhat less conveniently) in the absence of name services, and it does not require that the name service itself be secure. *A general network name service, Hesiod, is also an Athena development, described in another Technical Plan section.*

In addition to such network host and service name spaces, Kerberos itself defines a name space of authenticated users and services. For use in authentication the following simple naming model applies.

Unifying Names

There isn't much difference between a client and a service. In fact, a service that wants to use an authorization server must be able to authenticate itself to the authorization server in the same manner a client would authenticate itself to a service. For this reason both client and service names share the same structure so that they can be interchanged as necessary.

A *principal identifier* consists of three components:

- a *principal name*
- an *instance name*
- a *realm name*

all three of which are strings of upper and lower case letters and numbers.

Each different client and service has a unique principal name, assigned by negotiation with the manager of Kerberos.

The instance name is a label that permits the possibility that the same client or service may exist in several forms that require distinct authentication; it is useful for both clients and services. In the case of services, an instance may specify the host that provides the service. For example, the *rlogin* service on host *menelaus* is distinct from the *rlogin* service on host *tartaros*. For client principals the instance can be useful when one wishes to have different identifiers for different privileges. For example, *JLSmith* operating as a *class-administrator* may have different privileges from *JLSmith* operating as a normal user. The usual case is that users operate using a name with the null instance.

To allow independently administered sites, such as Athena, the M.I.T. administrative services, and the M.I.T. Laboratory for Computer Science, to inter-operate using Kerberos, a *realm* name is defined to identify each such independent Kerberos site. Thus a *{principal name, instance name }* is qualified by the *realm name* to which it belongs, and is unique only within that realm. Kerberos does not specify any constraints on the form of the realm name; it can be defined to be an ARPA internet domain name which is itself a qualified hierarchical name. That choice makes it possible to use the ARPA internet domain name resolution system to locate the Kerberos authentication service for the realm.

As described below, authentication is accomplished by giving out tickets. Tickets are labeled with the name of the realm for the service for which they are issued. Principal identifiers included in tickets include a non-null realm only if it is different from the realm for which the ticket was issued.

Workstations and service hosts have network names and network addresses, for example those specified by the ARPA internet domain name system.

Each application protocol using the authentication service binds Kerberos {*name*, *instance*} tuples for services to addresses using whatever means its chooses. It may use, for example, the *internet domain name service* or the Hesiod service and cluster location system.

Specifying names

The primary interface where the user will have to be concerned with names is when "logging in" to a workstation. Normally the user would simply enter his or her principal name, which might be the user's last name. Optionally, the user might specify an instance; if not specified, a null instance would be used as default. The realm is normally supplied by the workstation as a default, but the user might override that default, in effect requesting authentication by a different Kerberos server.

The principal name and the instance name are separated by a period ("."). If no "." is included in the name, it is assumed that the instance is null. In order to include a "." as part of the principal name or the instance name, it must be quoted with a backslash.

In order to specify authentication in a realm different from the default for this workstation, a user must specify the realm preceded by an at-sign ("@"). The realm itself may contain periods without the use of a backslash. As an example, consider the user who desires authentication through the LCS.MIT.EDU realm using a system management instance. That user might log in as follows:

```
Kerberos login:  RLSmith.sysadmin@LCS.MIT.EDU
```

Local Names

The namespace used for Kerberos authentication and authorization is independent of any particular host's means of referring to users or services, and any operating system specified conventions. Each host may translate the Kerberos principal identifiers to its own local user names as required. Local translation provides a convenient means of supporting proxies—for example, Kerberos name *{RLSmith,""}* might translate to *guest* on a host where *RLSmith* does not have an account. *Berkeley Unix applications that are modified to use Kerberos authentication generally support only the identity mapping from a Kerberos principal identifier to the same Unix login name.*

4. The Kerberos Authentication Model

In response to the requirements and assumptions sketched above, this section describes the Athena Kerberos model for authentication and authorization, with provision for accounting. This model is based on the Needham and Schroeder key distribution protocols, modified with the addition of timestamps. Their paper (listed in the References section) describes the basic protocol; a tutorial paper by Voydock and Kent provides a broader introduction to the topic and explains the timestamp modifications.

The basic approach for Kerberos authentication is the following: to use a service, a client must supply a *ticket* previously obtained from Kerberos. A ticket for a service is a string of bits with the property that it has been enciphered using the private key for that service. That private key is known only to the service itself and to Kerberos. As a result of that property, the service can be confident that any information found inside the ticket originated from Kerberos. As will be seen, Kerberos will have placed the identity of the client inside the ticket, so the service that receives a ticket has a Kerberos-authenticated opinion of the identity of the client. To help ensure that one user does not steal and reuse another user's tickets, the client accompanies the ticket with an authenticator, explained later. (In addition, tickets expire after a specified lifetime, which is usually on the order of several hours.)

The client obtains a ticket by sending a message to Kerberos naming the principal identifier of the desired service, the principal identifier of the (alleged) client, and mentioning the current time of day. Anyone could send such a message or intercept its response; that response, however, is usable only to the client named in the original request, because Kerberos seals the response by enciphering it in the private key of that client. The response contains three parts: the ticket (which itself is further sealed in the private key of the service), a newly-minted key for use in this client-server session, and a timestamp issued by the Kerberos server.

A legitimate user will be able to unseal this message, obtain the ticket and session key, and verify that the timestamp is current (thereby preventing replays of old responses). No other user, without the named user's private key, can correctly decrypt the reply to produce the sealed tickets and corresponding session key.

Once a client obtains a ticket and sends it to a service, and the service has identified the client, further use of the fact of authentication is specific to the protocol of the service. One application might use the session key (Kerberos seals a copy in the ticket) for secure end-to-end encryption, while at the other extreme, another application might throw everything but the source network address away and assume that all further requests coming on the connection from this particular network address are from the same user.

The *authenticator* mentioned above is a simple mechanism designed to discourage attempts at unauthorized reuse ("replay") of tickets by someone who notices a ticket going by on the network and makes a copy. The authenticator consists of, among other things, the client's principal identifier, network address, and the current time of day all sealed with the key that Kerberos minted for this session. After the service decrypts the ticket, it uses the session key found in that ticket to decrypt the authenticator. If the principal identifier of the authenticator matches the one in the ticket, the network address in the authenticator is the same as the one that sent the packet, and the time in the authenticator is within the last few minutes, the authenticator is probably not a replay, and the service accepts the associated ticket. It is because authenticators expire in a short time that all the clients and servers in a Kerberos realm need to have their clocks loosely synchronized.

If a private key is compromised, another party may successfully pose as the principal until the private key is changed and all tickets previously issued under it expire. If a session key is compromised, another party may successfully pose as the principal until the previously issued tickets expire.

One more mechanism rounds out the complete Kerberos scenario. If a client uses several services, a distinct ticket is needed for each. Not all the services to be used may be known at the beginning of a login session, but that is when the user provides the password used as a private key to decrypt tickets. To avoid storing the private key in the workstation memory for the entire duration of the session, at login time the user obtains a single ticket, useful only for a service provided by Kerberos itself, the ticket-granting service. Whenever the client goes back to Kerberos for an additional, service-specific ticket, the response is actually enciphered in the session key of the ticket-granting service. Thus the private key is needed only for the initial ticket, and the workstation software can immediately destroy its copy of that private key after that single use.

Authentication Scenarios

Here, at the next level of detail, are more complete scenarios of authentication using Kerberos. These scenarios omit several options described in the next section. The reader not interested in security protocols can skip this and the next section without missing anything needed later. The reader interested in full detail will also want to consult the complete protocol specification (in the Appendix to this section), which includes provision for errors, key versions, and protocol versions, and which manipulates timestamps in ways not apparent in this simplified description.

Scenario I. Getting the First Ticket.

1. The user establishes a principal name N_{client} and a private key, K_{client}, through some channel outside the system, for example, by walking up to the system administrator, and presenting his or her identification card. The private key K_{client} becomes the authenticator between the user and the Kerberos Key Distribution Server. The Kerberos Authentication Server stores the user's private key encrypted under its own master key, K_{master}. For the purpose of campus security, a one-way encrypted 8-character secret password serves as the user's private key. (One-way encryption of the original password serves the function of assuring that if the user's Kerberos key is somehow compromised it does not reveal the original password, which the user may also be using on other systems.)

2. The user initiates a workstation session by invoking a ***login*** command, giving as one argument the principal name of the client, N_{client}.

User —> WS N_{client}

The workstation knows the name of its default realm, R. The login command makes a request to the Kerberos Key Distribution Server for realm R, asking for a session key and a ticket for the Kerberos ticket-granting service.

WS —> $KKDS_R$ $\{N_{client}@R, N_{tgs}, T_{current}\}$

where N_{tgs} is the name of the ticket-granting service, and $T_{current}$ is the current date and time.

This request crosses the network in cleartext to the KKDS for realm R.

3. The KKDS looks up N_{client} and N_{tgs}, finding private keys K_{client} and K_{tgs}. It creates a new temporary session key, $K_{temporary_{tgs}}$, for use in this session, and prepares a ticket for the ticket-granting service:

$Ticket_{tgs}$: $\{K_{temporary_{tgs}}, N_{client}, N_{tgs}, T_{current}, WS, Lifetime\}^{K_{tgs}}$

where the notation $\{X\}^{K_y}$ means that message X is enciphered using encryption key K_y. The value WS is the network address of the requesting workstation. The value Lifetime is the ticket lifetime chosen by the KKDS. An explanation of the rules for the ticket lifetime appears in the next section.

4. The KKDS sends a response packet:

$KKDS_R$ —> WS $\{K_{temporary_{tgs}}, N_{tgs}, Lifetime, T_{current}, Ticket_{tgs}\}^{K_{client}}$

Note that authentication has not yet occurred—a sealed response containing a further sealed ticket comes back even if the user has misrepresented his or her identity.

5. At this point, the workstation asks the user for the password.

User —> WS <password>

and the workstation runs the password through the one-way encryption algorithm to produce K_{client}. It immediately destroys its copy of the password.

6. The workstation decrypts the response from $KKDS_R$ using K_{client} and checks its authenticity by comparing $T_{current}$ and N_{tgs} in the response with the corresponding values in the initial request. If the response passes this test, the user knows for certain that the response was prepared by the Kerberos Key Distribution Service, because that is the only other entity in the universe that knows K_{client}. The response is current rather than a replay of a response from yesterday, because it contains $T_{current}$. A fraudulent user finds that the response (including the sealed ticket) is a worthless set of random bits because it is enciphered with the unknown private key of the legitimate user.

The legitimate user stashes away $K_{temporary_{tgs}}$ and $Ticket_{tgs}$ for later use. The workstation destroys its copy of the user's private key K_{client}, because it will not be needed again during this login session.

Scenario II. Using a Kerberos-Mediated Service

To use a service S, the user must have a ticket $Ticket_{service}$ and the corresponding temporary session key for that service, $K_{temporary_{service}}$. Scenario I traced the acquisition of one such ticket. Assume for the moment that the client now has a ticket and temporary session key for service S. (Scenario III, later, demonstrates how the client can get additional tickets without having to again present the user's password.)

1. To use service S, the client first prepares an authenticator.

$Authenticator_{service}$: $\{N_{client}, T_{current}, WS\}^{K_{temporary_{service}}}$

where WS is the workstation's network address, $T_{current}$ is a current timestamp, and $K_{temporary_{service}}$ is the temporary key that came with ticket $Ticket_{service}$.

Now the workstation begins the protocol for the target service S. The protocol has one difference from the corresponding, non-Kerberos protocol for the same service: it is prefaced with the authenticator and the ticket.

WS —> Service $\{Authenticator_{service}, Ticket_{service}\}$

2. When the target service receives this request, it first decrypts the ticket using its private key, $K_{service}$. Since the only two entities in the universe that know $K_{service}$ are the service itself and Kerberos, the service can be confident that if the ticket deciphers properly it must have been originally prepared by Kerberos. The test of whether or not the ticket deciphered properly is whether or not the next step works. A correct ticket decipherment exposes the temporary session key, the client's name, and the timestamp. The temporary session key allows the service to decrypt the authenticator, exposing its data. If the client's name and network address in the ticket and authenticator match, the ticket's timestamp has not expired, the network address in the authenticator matches that in the incoming packet, and the authenticator timestamp is sufficiently recent, then the request is taken as legitimate. The service knows for certain the identity of the requesting client and the service and the client now share a temporary secret key. This authentication remains valid for the lifetime of the client-service connection.

3. Finally, the application protocol begins, typically by transferring an application request from the client to the server, perhaps at the end of the packet that contained the ticket.

If a client has a ticket for some service, that client may reuse the ticket as often as desired, until it expires. Each reuse requires constructing a new authenticator, one that contains a current time stamp.

Scenario III. Getting Additional Tickets

If a client wants to use a service for which a ticket wasn't obtained as part of the initial encounter with Kerberos, the client invokes the Kerberos Ticket-Granting Service. The Kerberos Ticket-Granting Service is simply another protocol for talking to the Kerberos Authentication Service, one that makes use of the ticket-granting ticket passed in the initial encounter, rather than the user's private key, to establish authenticity.

1. The client first prepares an authenticator exactly as before, though with a current timestamp and using the temporary session key that came with the ticket-granting ticket.

$\text{Authenticator}_{tgs}: \quad \{N_{client}, T_{current}, WS\}^{K_{temporary_{tgs}}}$

Now the workstation sends the authenticator, the previously obtained ticket for the ticket-granting service, and the name of the service for which a ticket is wanted to the ticket granting service.

$WS \longrightarrow KTGS_R \quad \{\text{Authenticator}_{tgs}, \text{Ticket}_{tgs}, N_{service}@R\}$

2. The ticket-granting service goes through the same procedure as does any other Kerberos-mediated service, first decrypting the ticket with its private key, and using the temporary session key found inside to decrypt the authenticator. If all the authenticity checks verify correctly, the ticket-granting service knows for certain the identity of the requesting client. In addition, it has recovered the temporary session key which is known only to it and the client; this session key can be used to securely return a ticket to the client. KTGS looks up the service name $N_{service}$ in its database and finds the private key, $K_{service}$, for that service. It now prepares a ticket:

$\text{Ticket}_{service}: \quad \{K_{temporary_{service}}, N_{client}, T_{current}, \text{Lifetime}\}^{K_{service}}$

where $K_{temporary_{service}}$ is a new temporary session key for use between this client and the service; it then sends the response:

$KTGS_R \longrightarrow WS \quad \{K_{temporary_{service}}, N_{service}, T_{current}, \text{Ticket}_{service}\}^{K_{temporary_{tgs}}}$

Note that the form of this response is identical to the form of the original response of the KKDS when it returned the ticket granting ticket.

3. The client, knowing the value of $K_{temporary_{tgs}}$, decrypts the response, verifies its authenticity as before, and stashes away the ticket for the target service.

Scenario III emphasizes that the ticket-granting service is simply another example of a Kerberos-mediated network service. The form of the messages in step one of scenarios II and III is identical, once one realizes that the last field in the second message of scenario III is the application request mentioned in step three of Scenario II.

Some Options

As mentioned, the three scenarios above follow what is expected to be the most common form of use of Kerberos authentication. There are several optional possibilities available for applications that use Kerberos:

- The examples specified no values for the instance name of either the client or the service; those values are optional and default to the null instance.
- An application client may include in the sealed authenticator an application authenticator, such as a checksum of data to be sent. Calculating that checksum is, of course, feasible only if all the data to be transmitted is known at connect time. As an alternative, an application could devise a commit message that appears at the end of the protocol, and that includes a checksum sealed with the session key.
- If the application requires mutual authentication, it sets an option in its service request, and places no application protocol information in the initial packet. The application server responds by adding one to the workstation's request timestamp, encrypting the result using the session key, and sending the encrypted result back to the client. Once the client receives and decrypts this handshake response, it can be certain that the server is authentic, and the application protocol may safely begin.
- The application server may retain state (timestamps) about previous use to aid detecting replay attempts.
- The application may use the application authenticator and the session key to continue a session in which every message is both completely encrypted and authenticated.
- An application may request a ticket with a specified lifetime; if the requested lifetime is less than the default ticket lifetime and less than that specified in the Kerberos database for the service, Kerberos issues a ticket with the shorter lifetime.

Application and User Interface

For the most part, Kerberos is designed to operate under the covers, without separate actions by the user. For network applications that make use of Kerberos authentication there is a library of Kerberos functions that simplify the obtaining of authentication. The primary interface consists of three generic user commands and two generic subroutines that are used by applications.

- User command **kinit**: This command asks the user for a password, obtains a ticket-granting ticket, and destroys the password as soon as it has stored the ticket-granting ticket and associated session key. Note that the function of this command may be combined with the **login** command.
- User command **klist**: Displays the list of tickets obtained so far in this login session.
- User command **kdestroy**: Destroys all tickets. The function of this command may be combined with the **logout** command.
- Subroutine **make_application_request()**: Used by an application to get a copy of, or if necessary obtain, a ticket and session key for a named service, to prepare

an authenticator, and return the result to the application for inclusion in the initial service request.

- Subroutine **read_application_request**: Used by an application server to validate a presented ticket and authenticator. It returns the identity found in the ticket and a judgement about the authenticity of that identity.

Note that the actual names, arguments, and parameters of these generic commands and subroutines are implementation-dependent. The Kerberos library implemented for UNIX, for example, shortens some names, combines kinit and kdestroy with login and logout, contains about a dozen additional supporting subroutines for the convenience of applications that are using optional features, and includes conventions about where to store tickets in the UNIX environment.

Realms

Kerberos provides for partitioning authentication information according to administrative divisions. All users need not be registered with a single organization. In addition, organizations that share authentication need not trust one another. A realm is an authentication domain. It is that part of the namespace of authenticable users and services that relies on a separately administered authentication server (or set of servers sharing the same database) for their authenticity. A service can accept credentials produced by an authentication server only for a realm of which it is a member. Both users and services may belong to multiple realms. Realm names within a network need to be unique. The earlier-mentioned convention of naming realms with ARPA Internet domain names has the side effect of guaranteeing uniqueness.

Realms can be either independent or semi-independent.

Independent Realms

Some users will want to access services from realms with which they aren't registered. Some services will be willing to provide services to users from other realms. These two requirements lead to a mechanism to authenticate users across realms.

This mechanism is provided through the cooperation of the administrators of the two realms involved. The Kerberos for each such realm is a client of the Kerberos in the other, and shares a secret key for a cross-realm ticket-granting service. This mutual client relationship between the Kerberos services allows a client of the Kerberos in one realm to authenticate itself to the Kerberos in the other realm even though no information is shared between the client and the other Kerberos service. Once a client has authenticated itself to the Kerberos in the new realm, that client can request tickets for services issued by that Kerberos.

As an example, consider a user in the LCS realm who wants to access a server in the Athena realm. The user must first authenticate with the LCS Kerberos using the initial authentication protocol. Once this authentication is done, the user can request a ticket for the Athena Kerberos. The user presents this ticket to the Athena Kerberos which accepts the user's identity since the Athena Kerberos is a client of the LCS Kerberos. The user can then request a ticket for an Athena service and the Athena Kerberos will comply. However, the ticket that the Athena Kerberos issues indicates that the user is from the LCS realm. Thus, all the ticket says is that the Athena Kerberos acknowledges that the user has been authenticated by the LCS Kerberos. The client then presents the new ticket to the end service which decides whether or not to accept it, based on its own authorization policy.

Semi-independent Realms

The realm mechanism can also be used to provide authentication services for off-campus independent living groups. The problem is that the ILGs must have a way of authenticating users to local services even when their connection to the campus-based facilities fails. Yet, at the same time, there cannot be a copy of the Kerberos for the Athena realm in the ILG since there would be no guarantee of its security. Instead, each ILG has its own realm.

Local services accept authentication by either realm. Most services on campus, however, accept authentication only from the Athena realm. When communication with the campus network is operational, ILG users authenticate themselves to the Athena Kerberos, then use the protocol described above to authenticate themselves to the ILG Kerberos. In this way ILG users have to provide only one password (the one required by the Athena Kerberos) to use both local and campus services. Users on campus who want to use services located at the ILG will also be able to use this mechanism.

If the connection between the ILG and main campus ceases to function, ILG users authenticate themselves directly to the ILG Kerberos and are thus be able to use local services. This local authentication does not allow them to use all the services on campus, but since they are disconnected it doesn't matter.

It is suggested that users choose different keys for the Athena Kerberos and the ILG Kerberos since the ILG Kerberos may be much easier to compromise. We do not plan to enforce such a suggestion, however.

More Complex Realm Relationships

The realm mechanism of Kerberos is not fully developed. In particular, the protocol does not provide the target service with detailed information about the provenance of tickets that have been authenticated in other realms. More work is required on security implications of cross-realm authentication, so that a service examining a ticket can know exactly whom it is trusting for authentication.

5. Management of Kerberos Data

The database underlying Kerberos contains a record for each user identity and for each service (that is, for each principal) known within that Kerberos realm. In order to allow security of the data to be the primary consideration when making operational tradeoffs about management of a Kerberos service, the information that Kerberos stores is the minimum required to accomplish and manage authentication. Thus, although a Kerberos record is a kind of per-user record, it does not contain information such as telephone number and office address, which are not used by Kerberos for authentication. Nevertheless, if there are a large number of users, the Kerberos database can still be quite large and it requires some tools for its management. The data management interface of Kerberos is designed to be used in two ways:

- By a set of manual tools manually from a system manager's workstation. This approach is suitable for management of a Kerberos realm that has a small number of users.
- By an automated Service Management System. This approach is intended for managing a system with thousands of users.

In both cases, the management of the Kerberos service is accomplished remotely via the network, using Kerberos-authenticated secure connections.

The information stored for each principal that Kerberos is prepared to authenticate is the following:

- The principal identifier, including instance identifier.
- The private key (password) for this principal.
- The expiration date for this identity.
- The date that this record was last modified.
- Identity of the principal who last modified this record.
- Maximum lifetime of tickets to be given to this principal.
- Attributes (unused).
- Implementation data, not visible externally:
    - Key version and master key version.
    - Pointer to old values of this record.

One piece of information in each record, the private key, must remain secret. Kerberos reversibly enciphers the private key fields, using a master key for this Key Distribution Service. Encipherment of the private key fields allows a manager to remove copies of the database from the machine and it also allows the Kerberos master to send copies over the network to slave servers without going to extraordinary lengths to protect the privacy of those copies. Kerberos does not store the master key in the database; it manages that one key separately.

Kerberos Database Replication

The Kerberos database for a realm is managed and updated by a single Kerberos Database Management server (the KDBM); authentication requests are handled by one or more Kerberos Key Distribution Servers (KKDS's), each of which contains an identical complete copy of the Kerberos database. Since all KKDS's have identical data any KKDS can handle any authentication request; a client uses a name service to obtain a list of KKDS's, and chooses the one that is nearest in terms of network topology. The separation of responsibility between KDBM and KKDS's does not imply that several distinct host computers are required; in the simplest deployment, one host can run both a KDBM server and a KKDS. The purpose of separation is to simplify update of the database while permitting replicated KKDS's for improved availability and performance. (Since many other network services may depend on it, continuous availability of Key Distribution Service is essential; continuous availability of update service is not nearly so important.)

With respect to the Kerberos database, all operations done by a KKDS are "read-only," so the only coordination among KKDS's and the KDBM is for the KKDS's to receive updates of the information when changes are made at the KDBM. Again for simplicity, the KDBM issues KKDS updates occasionally (e.g., a few times per day) and by copying the entire database. Complete copying eliminates the need for considerably more complex update procedures that would maintain update queues at the KDBM and recovery procedures at the KKDS's. Because updates occur on a batch basis, the KKDS's may have data that is slightly stale; update delay of a few hours is acceptable for this application.

The KDBM copies its database to the KKDS's using a Kerberos-protected protocol. First, using the Kerberos mutual authentication protocol, a secure encryption key is exchanged between the KDBM site and a given KKDS site. The KDBM creates a checkpoint of the data to be transferred, and calculates its (strong) checksum, seeding the checksum with the session key. Then it transfers the actual data using a conventional file transfer protocol. Recall that the data does not include any cleartext passwords or other particularly sensitive information. However, its integrity must be assured. The receiving KKDS temporarily stores all the transferred data, then recalculates the checksum of the received data using the secret session key. It then compares the calculated checksum with the original checksum, which was separately transmitted using the secure Kerberos protocol. If and only if the two checksums match, the newly received data updates the KKDS database.

Updates to the Kerberos Data Base

Updates are done by an update protocol that runs between any authenticated client at a workstation and the KDBM. If the KDBM is not accessible, updates are temporarily not allowed.

There are several routine updates made to the Kerberos database.

1. adding a new user
2. a user changes a password
3. system manager changes a forgotten or compromised password
4. deactivating an old user
5. removing old user identities

In emergencies, a system manager can also tinker directly with raw Kerberos data for repair and other extraordinary maintenance operations. Such tinkering must be done by logging in directly on the host that runs the master Kerberos service.

Adding a New User

Adding a new user to the Kerberos database is accomplished by invoking the add-user message type of the Kerberos protocol, which requires that the user doing the addition be a previously-added user of the system whose identity appears in an add-user access control list maintained by the Kerberos master system.

If an SMS is in use, a different approach is taken that is more suitable for mass production. The intent of this different approach is that a user can choose a principal identifier and register the chosen principal identifier and associated password without actually involving a system manager. Each fall, the SMS is primed with a list of potential new users (obtained from a list of all registered students) including for each user a full name and a student identification number. A prospective user walks up to an Athena workstation, logs in as an unauthenticated user (the user identity "register", with publicly-known password "athena", is used for this purpose) and interacts with a user registration program that obtains from the user his or her full name, student identification number, proposed principal identifier and proposed password. The user registration program first connects to SMS to verify that this user’s full name and student identification number match one in the list of as-yet-unregistered users. If so, it informs SMS of the principal identifier that the user has chosen, and in turn receives an add-user session key from SMS. The user registration program then opens an encrypted connection with the master Kerberos service using the add-user session key. It supplies the user’s chosen principal identifier and password to Kerberos, which checks to see that the principal identifier is not one already on record (rejecting the request if it is) and then records it and the password. If the transaction with Kerberos is successful, the user registration program confirms the success with SMS, which then commits this registration transaction.

This unsupervised registration scenario is a compromise that is only weakly secure, because any one who knows another person's name and student identification number can register as that person. There is some protection against such an attack, however, because when the authentic person with that identity attempts to register, the fraud will be discovered when both SMS and Kerberos reject the second registration attempt. The legitimate user can then appeal to a real system administrator, who can sort things out by forcing into the Kerberos database a new password known only to the legitimate user.

User-Initiated Password Change

The basic scenario for changing a password is that the user does it him or herself by invoking the password-changing program at a workstation. This program demands the old and new passwords, uses the old password to create a completely encrypted session with the master Kerberos server, and sends the new password on the encrypted connection. If the user has reason to believe that the old password is so badly compromised that it is not safe to send the new password this way, the user may appeal to the system manager to install a new password.

System-Manager-Initiated Password Change

Kerberos maintains an access control list, which consists of a list of Kerberos principal identifiers of individuals who are authorized to act as system manager. When a user reports that a password is forgotten or compromised, the system manager opens an encrypted connection from the manager's workstation to the Kerberos master server and runs a password-installation protocol. This protocol requires that the invoker appear in the system manager access control list.

User Deactivation

Kerberos maintains an expiration date and an activation flag for every principal identity that it is prepared to authenticate. Kerberos always rejects attempts to authenticate expired or inactive users, with an appropriate error response. The purpose of deactivation is to provide a simple means of avoiding accidental reuse of principal identifiers, which may continue to appear in access control lists for some time after a user departs from the scene.

There is a secure protocol message type by which the system manager can deactivate or reactivate a principal identifier, or change its expiration date.

Removing Old User Identities

Kerberos maintains a last-modified-date as part of each record of a principal identity. Deactivation updates this date. One use of this date is to allow a system manager to identify old identities that have not been in use for a sufficient period (e.g., one year) that it is safe to remove them. A secure protocol message allows an authorized system manager to remove any specific inactive identity, and to remove all inactive identities that have not been changed since a specified date. This operation is designed under the assumption that it occurs rarely, perhaps two or three times a year, so the only record of identities removed is in the Kerberos log.

Keeping Synchronized with SMS

If a Service Management System is in use, it maintains its own records of registered and prospective users; those records are correlated with the records of Kerberos by principal identifer. Since the principal identifier is the only piece of duplicate information maintained, the only synchronization problem is to insure that every principal identifier that appears in an SMS record also appears in some Kerberos record, and vice-versa. User registration, as described above, is the normal way of creating principal identifiers, and if a user registration operation completes normally, both records will match. Failures, or hand-tinkering, may unsynchronize these two sets of records. No special tools are provided to deal with this problem; the system manager, if trouble is suspected, may extract from Kerberos a list of principal identifers to sort and compare with the corresponding list from SMS.

Database Backup and Reload

The Kerberos database is backed up by running a special backup program on the master Kerberos server, which should be equipped with a private tape drive. The Kerberos master key is not stored on the backup tape. A special reload program is also available, although if the system is completely reset the Kerberos master key must be reinstalled by hand. Reload of slave servers is done by invoking the usual Master-Slave update procedure, which transfers a complete copy of the database.

6. Authorization Model

The Kerberos authentication model provides only a certification of the identity of a requesting client; by itself it provides no information as to whether or not that client is actually authorized to use the service. There are three forms in which authorization could be integrated with the Kerberos authentication model:

- The Kerberos database could also contain authorization information for each service, and issue service tickets only to authorized users of each service.
- A separate authorization service could maintain authorization information by keeping access lists for each service and allowing the client to obtain sealed certification of list membership. The client would present that certification, rather than a Kerberos ticket, to the ultimate service.
- Each service could maintain its own authorization information, with the optional help of a service that stores shared public lists and provides certification of public list membership.

The first of these alternatives places the large, dynamically updated authorization database in the midst of the small, slowly changing, high-security encryption key database. Operational parameters such as primary and secondary memory size, degree of replication, nature of backup, and physical security must be chosen as a compromise between the requirements of the two services. It also locks in one particular authorization model for all applications.

The second alternative separates the authorization database from the authentication database, thereby improving separation of administration and making the authentication service simpler and smaller, which should make it more reliable and easier to secure. But this alternative leads to an extraordinarily complex (and therefore potentially fragile) collection of interacting protocols among the client and the authentication, authorization, and target services. It also creates a rendezvous problem, in that the client must know which membership certification to request from the authorization server.

The Kerberos authorization model is based on the principle that each service knows best who its users should be and what form of authorization is appropriate, so it adopts the third of these alternatives. This choice has several advantages:

- Many services will have short, private lists of authorized users. For example, the display server on a private workstation may have as its list of authorized users only one entry—the current user of the workstation—and that user's identity is already known by the workstation. (In addition, the identity of the user allowed to use the display on a public workstation changes as often as someone logs in.) By far the simplest way to manage that information is to place it in the server. Completely private services (e.g., a dating service exported from a private workstation) thus require no central registration, yet can take advantage of Kerberos-quality authentication and implement access control.

- Services that maintain their own lists (e.g., the display server) or that do not require an access control list (e.g., a public library) do not depend on availability of and network continuity to an authorization service.

- Rendezvous is limited to getting the client together with the service; the client does not need to figure out what kind of authorization to request for this particular service.

- No one authorization model applies to all services; by making authorization the responsibility of the server, the designer of the service has the option of using a standard library authorization model, or creating a different model that is better adapted to the particular service it is offering.

- Since the amount of information storage required for authorization information is proportional to the number of services offered, storing and managing the authorization information at the service scales up well. This scaling advantage is of particular interest when one realizes that every workstation exports at least its display service, and may export others. It is also administratively preferable to have each service provide its own authorization list storage, rather than burdening a public storehouse with this responsibility.

- Administrative authority to set and change the authorization information for a service tends to be automatically delegated to the appropriate entity—the administration of the service itself.

There is one significant disadvantage to requiring the service to do its own authorization: Services that cannot depend on other network services (for example, because they are single-threaded and should not block waiting for a network reply) cannot make use of shared public access control lists.

Authorization Mechanics

A standard authorization model based on access control lists is provided, and an authorization library package is available for incorporation into any service that finds the standard model useful. Under this standard model, the service takes the (known, authenticated) identity of the client and inquires whether or not that client is a member of a named list. The access list library package maintains any number of named lists in the local storage of the server. A list may contain three kinds of names:

1. Kerberos-authenticable principal identifiers,
2. names of other local lists, and
3. names of shared, public access control lists.

The access list library undertakes a search of the named list, local sublists stored at the service host, and shared, public lists. If the client's identity is found in this search, the operation is authorized.

Rather than associating operation-specific permissions with access list entries, the service maintains distinct, named access lists for each different kind of operation.

The lists are maintained as simple ASCII text string files in a special access list directory that is protected from modification except by administrators of the target service. Their format allows, in simple cases, maintenance by use of standard text editors, or in more complex cases, automatic maintenance by the Athena Service Management System.

The Public List Server

A public list server provides Kerberos-quality certifications that principal identifier A is (or is not) in list B. The ability to use remote servers for such a certification allows the possibility of shared, centrally managed lists. The ability to use local lists allows the possibility of lists whose contents are unknown to any central authority. The architecture allows that these two possibilities can be mixed and matched in any way desired by the implementer or manager of the host that offers the service. (The detailed design of a public list service has not yet been undertaken. Issues such as what action to take in the face of a cycle in a list, and management of very large lists, have not yet been addressed.)

Authentication/Authorization Scenario with Name Service

A complete scenario for integrating name service, Kerberos, and authorization is as follows (there are a lot of services flying around in this discussion—the one the client really wants to invoke is called the "desired service"):

1. Assume for starters that each client (and service) knows the internet address of a name service and the name of Kerberos. As part of its initialization, the client invokes the name service to determine the internet address of Kerberos. It also performs an initial transaction with Kerberos to obtain a ticket-granting ticket. Each service that cares about authorization has done the same thing as part of its initialization.

2. The person exporting the desired service has previously registered the name of that service with the name service. If this step hasn't happened, it doesn't prevent use of the desired service, but it does mean that the client has to invoke it by discovering and using a host name and port number, rather than by name.

3. The user learns the name of a desired service. Learning may happen one of any number of ways. Here are a few examples:

- A prospective user reads the name on a bulletin board.
- The user copies a program from a public place; the program has the name buried in it.
- The name is embedded in a system-provided library program.
- The name is embedded in a class-provided library program.
- The user learns about the service name from a system staff member.

4. The client invokes the Kerberos ticket-granting service, requesting a ticket for the desired service name. If Kerberos has never heard of the desired service, that doesn't cause the scenario to abort; it may simply be that the desired service doesn't require authentication.

5. The client invokes the name service to learn the host name and port of the desired service. The client can cache this information at its own risk, to allow future invocations of the desired service without using the name service again. The name service provides a time-to-live value for the information that gives the client a hint about how long it is safe to cache it.

6. The client invokes the name service again, to transform the host name of the desired service into an internet address.

7. The client invokes the desired service, presenting its Kerberos ticket (if by now it has one) certifying the client's identity.

8. The desired service decides whether or not it wishes to deal with this client. To decide, it may invoke the access list library, giving the name of the client and the name of an access control list. The access list library performs a recursive descent through that list and any lists, local or remote, named in that list, trying to verify list membership of the client.

Because the desired service is depending on the authenticity of the certifications of the list membership service, each connection with a remote list membership service must be initiated via Kerberos and the responses from the service need to be integrity-assured. Integrity assurance is provided by having the remote list membership service return a copy of the original request, with a yes or no bit added, enciphered in the session key that the invoker obtained at initial connection with the list membership service.

Acknowledgments

Many people have provided ideas, or have been involved with the implementation of this design. In addition to the authors of this document, they include: John Ostlund, Mark Colan, Bob Baldwin, Dan Geer, Stan Zanarotti, Bill Sommerfeld, John Kohl, Jim Aspnes, Chris Reed, and Brian Murphy. The name "Kerberos" was suggested by Bill Bryant.

7. Appendix I—Design Specifications

This section contains detailed design specifications for the current implementation of Kerberos. It is of interest primarily to implementers.

7.1. Design

7.1.1. Conventions

The following conventions apply:

- encryption or decryption implies DES private key in a modified[2] cipher-block-chaining mode
- "{data}K_x" means that "data" is encrypted using "x"s DES key;
- all data to be encrypted is padded with trailing 0 bytes to an integral multiple of 8 bytes;
- all references to *session key* imply a distinct random session key valid only for that particular session;
- bit 0 refers to the least significant bit;
- all field sizes are expressed in numbers of 8-bit bytes, unless otherwise stated, and whether or not the value is signed (s), unsigned (u) or only printable ASCII, null terminated (a);
- strings are sequences of printable ASCII bytes, null terminated;
- all messages are self-framing, that is, do not depend on packet boundaries to determine their extent;
- where not otherwise stated, *name* implies the local *realm*; similarly, a null *realm* implies the local one;
- principal, indicated in the protocols by either subscript *p* or *principal*, refers to the subject requesting authentication and/or authorization, i.e either a user's or service's *{name, instance}* pair.
- service, indicated in the protocols by either a subscript *s* or *service* refers to the end service, object, or other user for which authentication/authorization was requested. This is most often a service's *{name, instance}* pair, but could also be any user's to allow secure key distribution between two users.

[2]Modified to provide forward error propagation of a single bit error in the ciphertext thru to the end of the resulting cleartext. Refer to Voydock and Kent [17].

Common fields used in messages

field	**size**	**u,s,a**	**description**
version	1	u	Protocol version number;
auth_msg_type	1	u	Protocol message type and byte order; = m_type << 1 + byte_order ;
m_type	7bits	u	Protocol message type;
byte_order	1bit	u	Byte order of sender;
name	>=0	a	Athena principal name (user or service);
instance	>=0	a	Athena principal instance (user or service);
realm	>=0	a	Authentication realm name;
group	>=0	a	Athena group name;
time_sec	4	u	UTC timestamp, sec since 0000 GMT 1/1/70; may also have direction encoded in msbit;
time_5ms	1	u	rest of UTC timestamp, 5ms units;
lifetime	1	u	valid ticket lifetime, 5 minute units;
key	8	u	64 bit encryption key;
kvno	1	u	key version number;
n	1	u	count of service entries;
address	4		Internet host address, IP format and order;
length	1	u	length of a field, 0 - 255, bytes;
length_2	2	u	length of a field, 0 - 65535, bytes;
length_4	4	u	length of a field, 0 - 4,294,967,295, bytes;
exp_date	4	u	UTC expiration date, sec since 0000 GMT 1/1/1970;
direction	1bit	u	within an association, zero if sending {addr, port } < receiving {addr,port}, else one; multiplex into msb of time_sec;
app_data	n		application specific data, arbitrary length;
checksum_4	4	u	4 byte checksum;
checksum_16	16	u	16 byte checksum;
flags	1	u	bit-flags within ticket, set by Kerberos;
err_code	4	s	Kerberos error code;
err_text	>=0	a	description of Kerberos error;

Network Representations

byte ordering — The least significant bit of *auth_msg_type* will encode the byte ordering for the transmitting host. LSB_FIRST, one, implies least signigicant byte in lowest address, e.g. VAX and IBM PC's. MSB_FIRST, zero, implies most significant byte in lowest address, e.g. Sun 68000 and IBM RT's. The transmitter of a message always transmits in natural host order, and marks its byte ordering in *auth_msg_type*. The receiver, if necessary, converts fields to its own byte ordering.

alignment — to avoid possible incompatibilities between compiler alignment rules, all protocol messages must be defined without use of structures. All protocol messages have no holes for alignment. Each field begins on the next byte boundary.

Protocol Message pattern

{ version, auth_msg_type, $name_p$, $instance_p$, $realm_p$, time_sec, cleartext, ciphertext }
where unneeded parts are omitted.

The protocol message specifications should be read in increasing byte order within the message as you read from left to right, with no holes.

7.1.2. KKDS

7.1.2.1. Protocol. All the Kerberos protocols described are layered on a UDP datagram between the client and the KKDS. The client interface may retransmit a request up to <AUTH_RETRY_MAX> times if a response is not received within time interval <AUTH_RETRY_WAIT>. All protocol messages between a client and the KKDS must be idempotent. To minimize retransmissions, all requests should generate a response, either an *auth_reply* or an *err_reply*, even if the response only implies failure.

auth_request = { version, auth_msg_type, $name_p$, $instance_p$, $realm_p$, $time\_sec_{ws}$, $lifetime_s$, $name_s$, $instance_s$ }
where
auth_msg_type = <AUTH_MSG_KDC_REQUEST>
The service requested is local to the realm managed by the Kerberos receiving the request.

auth_reply = { version, auth_msg_type, $name_p$, $instance_p$, $realm_p$, $time\_sec_{ws}$, exp_date_p, $kvno_p$, length_2, $\{cipher\}K_{p_{kvno}}$ }
where
auth_msg_type = <AUTH_MSG_KDC_REPLY>
length_2 = length of cipher; zero if *{name $_p$, instance $_p$}* is unknown;
cipher =
$\{K_{session}$, $name_s$, $instance_s$, $realm_s$, $lifetime_s$, $kvno_s$, $\{ticket_s\}K_{s_{kvno}}$,

time_sec $_{kkds}$ }
where
ticket = { flags, name $_p$, instance $_p$, realm $_p$, address $_p$, $K_{session}$, lifetime, time_sec $_{kkds}$, name $_s$, instance $_s$ }
note:
the *lifetime* returned is the minimum of the principal's, server's, and the lifetime requested.

err_reply = { version, auth_msg_type, name $_p$, instance $_p$, realm $_p$, time_sec $_{ws}$, err_code, err_text }
where
auth_msg_type = <AUTH_MSG_ERR_REPLY> ,
err_code = Kerberos error code, defined in *prot.h* ,
err_text = text string describing error.

7.1.2.2. Protocol Vulnerability.

- replay -- The timestamp serves to prevent replay attempts by limiting the lifetime of the key. If the server retains *all* the still valid timestamps for previous associations for the user, all replay attempts can be prevented. The latter requires stable store across process and machine crashes.
- modification -- The *timestamps* and *name* can serve as effective integrity checks to detect modification to the packet. If the ciphertext was changed or forged, with extremely high probability the *timestamp* would no longer be valid, and the *name* in the ticket and in the *authenticator* would not match.

7.1.2.3. Administrative Protocol. A set of protocols is required for interaction between administrators, users, and the Kerberos Database Manager, for example to create new principals and to change keys. These protocols are not yet specified.

7.1.2.4. Authentication Database. Each Kerberos *realm* maintains an independent set of databases. The following are represented:

- Private keys of clients and services; estimate 10,000 users + <= 15000 services x 1 record; tag each key with an index number noting which KKDS master key was used to store it.[3]
  record = {name, instance, kvno, $\{key_{kvno}\}K_{KKDS_{KKDS\text{-}kvno}}$, KKDS-kvno, exp_date, max_life, last_modified_by_name, last_modified_by_instance, last_modified_date}
- Audit trail -- A management audit trail of selected database operations, not yet specified, will be maintained[4].
- Statistics - To be specified.

---

[3]In case the KKDS master key needs to be changed, this allows a more orderly transition to a new master key.

[4]Probably as a side effect of journaling the database.

7.1.2.5. Database management. Kerberos is built on a database management layer with a very simple set of lookup operations that can be implemented using any available database system. The initial implementation of that layer uses Ingres as the supporting database system; a second implementation uses the UNIX dbm package. Slave servers use the second implementation. The master server can use either implementation; the advantage of the Ingres implementation is that administration of a large number of users (e.g., producing a list of all users whose accounts will expire in the next six months) can be done with more potent tools.

7.1.2.6. User interface. An implementation of a user interface to obtain, list, and destroy Kerberos tickets for Berkeley 4.3 UNIX is described in a set of UNIX *man* pages named *kerberos(1), kinit(1), klist(1)*, and *kdestroy(1)*. A command to change a user's Kerberos password is described in *kpasswd(1)*, and the Kerberos database administrator's program, used for registering new Kerberos principals and setting or changing passwords, is explained in the *kadmin(8)* manual page.

7.1.3. Application Authentication Protocols

7.1.3.1. Request Interface. The changes involved in using a service should be as transparent as possible. When a user uses *lpr, lpr* should automatically include the authenticator in its request without the user having to do anything extra. In the event that the ticket for a service has not been obtained, or has expired, the service should obtain a ticket on the user's behalf using the ticket granting ticket obtained when the user *logged in*.

7.1.3.2. Client Request. The following KKDS block normally would be transmitted from the client to the server before any user data as the first packet sent, though this need not be first. It serves to identify the requestor, present his or her ticket, and authenticate the request. By appropriately decrypting and checking the integrity, the service may proceed to offer or deny the requested service.

appl_request { version, auth_msg_type , $kvno_s$, $realm_s$, length_2, $\{ticket\}K_{s_{kvno}}$, $\{authenticator\}K_{session}$ }
where
auth_msg_type = <AUTH_MSG_APPL_REQUEST>
i.e. one-way authentication, or
<AUTH_MSG_APPL_REQUEST_MUTUAL>
i.e. mutual (two-way) authentication request
length_2 = length of ticket, then length of authenticator
ticket = { flags, $name_p$, $instance_p$, $realm_p$, $address_p$, $K_{session}$, $lifetime_s$, $time\_sec_{kkds}$, $name_s$, $instance_s$ }
authenticator = {$name_p$, $instance_p$, $realm_p$, checksum_4, $time\_5ms_{ws\_now}$, $time\_sec_{ws\_now}$ }
checksum_4 = optional data checksum to be used by service, checksum algorithm selected by service.

7.1.3.3. Server Verification and Response. The server decrypts request, checking *name*, *instance*, *realm*, *address*, and *time_sec*, and optionally checks for a *recent playback attempt*. If the authentication is invalid, the client's request is denied, and an *appl_err* message is returned. Otherwise, the service may then request the client's authorizations from the authorization service, if need be. It then performs the requested operation within the bounds of the authorizations granted.

If *auth_msg_type* requests mutual authentication (two-way), the server replies with the message noted below. If the client is satisfied with the server's response, it then begins the normal operation.

appl_reply { version, auth_msg_type, {svc_authent}$K_{session}$ }
where
auth_msg_type = <AUTH_MSG_APPL_REPLY_MUTUAL>
svc_authent = { $time\_sec_{ws\_now}$ +1 }

appl_err = { version, auth_msg_type, err_code, err_text }
where
auth_msg_type = <AUTH_MSG_APPL_ERR> ,
err_code = Kerberos error code, defined in *prot.h* ,
err_text = text string describing error.

7.1.3.4. Secure Conversations. The authentication protocols described previously create a secure session key exchange and authenticate the principals. This is sufficient for many purposes, but other services, such as the *authorization service* and the *KDBM service* require protection for every message exchanged, not just for initial "connections". Such protection may take two alternate forms:

- message authentication -- guarantee that a given message has not been modified, forged, replayed, or made out of sequence; the message is still readable on the network;
- message secrecy -- in addition to offering message authentication, providing message secrecy by encrypting the contents of the message.

Two additional protocol message envelopes are provided for these purposes; *safe_msg* provides message authentication, and *private_msg* provides both message authentication and privacy. The *app_data* field is application specific data. Each application determines the pattern of message types needed -- *private_msg*, *safe_msg*, *appl_err*, and application specific messages.

A *safe_msg* provides strong means to detect any modification attempts, forgery, or replays, but does not provide privacy.

safe_msg = { version, auth_msg_type, safe_data, checksum_16($K_{session}$, safe_data) }
where
auth_msg_type = <AUTH_MSG_SAFE>
length_4 = length of safe_data,
safe_data = { $length\_4_{safe\_data}$, app_data, $time\_5ms_{ws\_now}$, $address_{source}$, direction, $time\_sec_{ws\_now}$ }
checksum_16 is a function of both $K_{session}$ and *safe_data*, using the quad_cksum() algorithm.

A *private_msg* provides strong means to detect any modification attempts, forgery, or replays, and in addition provides privacy. However, to provide the privacy, it incurs significant additional run-time overhead for encryption. Since the lifetime of a session key may be greater than that of a process, timestamps are used instead of sequence numbers.

private_msg = { version, auth_msg_type, $length\_4_{cipher}$, cipher }
where
auth_msg_type = <AUTH_MSG_PRIVATE>
$length\_4_{cipher}$ = length of the encrypted portion of the message,
cipher ={ private_data } $K_{session}$
private_data = { $length\_4_{app}$, app_data, $time\_5ms_{ws\_now}$, $address_{source}$, direction, $time\_sec_{ws\_now}$ }
$length\_4_{app}$ = length of *app_data*,
app_data = application specific data,

Rules for *safe_msg* **and** *private_msg***:**

- Both sides discard messages with duplicate timestamps and messages with the wrong direction (replay attempts);
- Both sides retain state of both the transmitted and the received timestamps; messages with out of order timestamps are discarded (limited pipelining is possible if one were ambitious);
- Messages with invalid checksums are discarded;
- (Discarded messages cause a security log entry to be made either locally or sent to a security audit trail log process ???).

7.1.4. Library Routines

Kerberos uses two major libraries. The first is a general purpose DES encryption library, and the second is a Kerberos-specific library to help interface to the Kerberos protocols.

7.1.4.1. DES Encryption Library. The DES encryption library created for Kerberos is a software only implementation of the DES algorithm, certain modes of operation, and related utilities. It may be used independently of Kerberos, or may be replaced (for example, for export) by any other 64-bit block cipher algorithms which maintain a compatible interface.

The routines supported include *ecb mode, cbc mode,* and *pcbc mode* [5] encryption and decryption, a *cbc checksum mode*, a *quadratic checksum mode*, (not DES), a DES *random key* generator, a routine to *prompt* and read a password without echoing, a routine to one-way-encrypt an arbitrary string into a DES key, and a routine to create a DES key schedule from a DES key.

The implementation for Berkeley 4.3 UNIX is described in a UNIX man page labelled *des_crypt(3).*

[5]pcbc is a modified cbc mode to provide indefinite error propagation on decryption.

7.1.4.2. Kerberos Protocol Library. A Kerberos Protocol Library provides a callable interface to the protocol described earlier.

The implementation for Berkeley 4.3 UNIX is described in a UNIX man page labelled *kerberos(3)*.

7.2. Issues

Master key management for the servers is a yet unresolved operational problem. To maintain security during maintenance operation it is preferable not to store the master key on disk on the server, yet it is an operational headache to manually enter the master key at each server every time it is restarted. One possible solution is to build a simple hardware box that supplies the master key from a set of thumbwheels, over a serial port. This box could remain plugged in to the KKDS in case a power loss causes it to reboot, yet it could be unplugged (or the thumbwheels set to zero) when it is necessary to turn the machine over to a field service engineer for maintenance. A related requirement is to completely clear all copies of the master key, including any that may be in virtual memory swap areas on the disk, when sanitizing the KKDS for service.

Key management for user keys also presents some problems. In order to make this authentication mechanism as familiar and transparent to the user as possible, keys are based on a password of the user's choice. Because of this, Kerberos suffers from some of the same problems as passwords. In particular, users may choose keys which are easy to guess, or they may record them where others can find them.

Servers may require stable storage for the recently used *authenticators*, in order to eliminate replay attempts that cross system boot or process restart boundaries. Whether this is needed depends on the difference between the expected maximum downtime for the service and the size of the service's timestamp window.[6]

The KKDS workload needs to be estimated and measured, since it (they) can easily become a bottleneck. We will then need to determine how to tune the KKDS's, and how many are needed where.

The server's private key is needed to decrypt the ticket for every application request. This subjects it to potential exposure much more than is desirable for a private key. In the future, a means to automatically change the server's private key on a daily basis, using a higher level key, is desirable. Also, a hardware implementation of DES supporting write-only master keys is highly desirable for the Kerberos servers.

Another problem that is not easily dealt with at the moment is authenticating the workstation to the user. How does a user know that an adversary hasn't modified the software on the machine he or she is using so that it will store the secret key? One approach to this problem is to have the user carry around a *boot disk*. The user would then boot the machine off that disk, and upon logging in, the authentication would be taken care of by software on that disk. The problem with this approach though, is that it requires the user to carry something extra around.

Another approach, although not practical at the moment, is the use of *smart cards* that

---

[6]The service's timestamp window is the valid range for $time\_sec_{ws\_now}$ for which the service will honor a request.

would do the encryption for initial authentication internally. With this approach, the key never leaves the card, thus, there's nothing for a spoofer to store except the session key (which has a limited lifetime).

The representation of names as entered by the user is somewhat awkward.

The timestamp granularity for requests -- 5 ms. -- is more than sufficient for software encryption, 4.3BSD, and current processors, but may be too large for systems 5 years from now. (The granularity will have to be reduced and the fields extended, and the systems will have to provide higher resolution timestamps than the 10ms currently provided by 4.3BSD UNIX.)

The timestamp base used in the protocols is based on the Berkeley UNIX clock standard rather than the ARPA internet clock standard used elsewhere in TCP/IP protocol family; the IP standard should be used instead.

7.3. Well Known Services

All Kerberos installations should adhere to the following conventions:

- The following literals are reserved Kerberos principal names: *{K,M}, krbtgt, changepw, default*.
- The Kerberos service is accessible at a well known UDP port, 750. The Kerberos administration protocol is carried on via UDP port 751. [7] In UNIX implementations, these ports are named *kerberos* and *kerberos_master*, respectively.

7.4. Revision History

7.4.1. Revision 7 --> Release v1.1

Revision 7 represents the definitive specification for the August 1986 Athena staff release of Kerberos.

- Protocol changed to only allow one ticket request up front. This was done to decrease the complexity of the protocol, and to allow implementations that are forced to limit the number of tickets returned to interact with others. This change was made after reliability problems resulted from the complextity of the old protocol, and network limitations. For a while, both the old protocol (V3) and the new protocol (V4) will be supported.

7.4.2. Revision 6 --> Release v1.0

Revision 6 represents the definitive specification for the May 1986 Athena staff release of Kerberos.

Major changes:

- Moved the design proposals for authorization into a new document, entitled "Project Athena Technical Plan -- Authorization Proposals".

[7] These two port assignments are not official ones. An official assignment is needed.

- Added Kerberos *err_reply* message type and an *appl_err*, the latter message for use with *safe_msg* and *private_msg*.
- Disallowed wildcard lookups for ticket requests (either via an authentication request or ticket-granting-ticket request); removed the cleartext service *{name,instance}* and *lifetime* from the corresponding reply messages.
- Added a *flags* field to the beginning of the ticket, to include the byte order of the system granting the ticket.
- Changed the name of the *des_set_key* routine to *key_sched*.
- Modified the *safe_msg* and *private_msg* protocols to streamline them, removed the *app_code*, and replaced the *sequence* number with timestamps.
- Added the cleartext *exp_date* of the requesting principal to the *auth_reply* message.

7.4.3. Rev 5

Major changes:

- Split authentication and authorization into two independent services; removed authorization information from the authentication protocols. Redefined the term *KDC/AS* to be the Key Distribution Center/Authentication Server.
- Changed the naming of users and services to a single, unified name model of *{name, instance}*, with an optional *realm* specified. Modified protocols to reflect the new naming model.
- Added a discussion of replication for the authentication database.
- Added more discussion of realms.
- Added protocols for secure conversations.
- Deleted most references to the existing *athena_reg* Athena Unix login database.

7.4.4. Rev 4

Major changes:

- Added an authentication realm *realm* to qualify all uses of the authentication name *{name,instance}*. This allowed future enhancements to support authentication across administratively independent Kerberos services, for example between Athena's Kerberos and one at LCS. (This is similar to the Internet domains, but not necessarily equivalent.)
- Added the cleartext service *{name,instance}* and *lifetime* to the authentication reply message, *auth_reply*. This supported the use of wildcard requests by returning to the requestor a readable version of the specific servers and instances selected.
- Specified byte ordering in the least significant bit of the *auth_msg_type*. The transmitter of each message sends in its natural byte order, while the receiver converts the byte order as needed.

8. Appendix II—The Kerberos Encryption Library

The Kerberos encryption library supports various encryption related operations. Its contents differ from the *crypt, setkey,* and *encrypt* library routines. In this description, eight bit bytes are assumed; bit numbers start with the least significant bit. Array and bit indices start with 0. Operation of the library is described below.

For each key that may be simultaneously active, create a *Key_schedule* structure, defined in "krb.h" as a structure of 64 bit-fields:

```
        typedef bit_64  Key_schedule[16];
```

Next, create key schedules (from the 8-byte keys) as needed, using *krb_key_sched,* prior to using the encryption or checksum routines. Then set up the input and output areas. Make sure to note the restrictions on lengths being multiples of eight bytes. Finally, invoke an encryption/decryption routine such as *pcbc_encrypt,* or, to generate a cryptographic checksum, use a routine such as *quad_cksum.*

A *C_Block* structure is an 8 byte block used as the fundamental unit for data and keys, defined as:

```
        typedef unsigned char C_Block[8];
```

The individual library functions *krb_read_password*, *krb_string_to_key*, *krb_random_key*, *krb_key_sched*, *pcbc_encrypt*, and *quad_cksum* will now be described.

```
int krb_read_password(key, prompt, verify)
        C_Block *key;
        char    *prompt;
        int     verify;
```

krb_read_password writes the string specified by *prompt* to the standard output, turns off echo (if possible) and reads an input string from standard input until terminated with a newline. If *verify* is non-zero, it prompts and reads input again, for use in applications such as changing a password; both versions are compared, and the input is requested repeatedly until they match. Then *krb_read_password* converts the input string into a valid key, internally using the *krb_string_to_key* routine. The newly created key is copied to the area pointed to by the *key* argument. *krb_read_password* returns a zero if no errors occurred, or -1 indicating that an error occurred trying to manipulate the terminal echo.

```
int krb_string_to_key(s, k)
        char    *s;
        C_Block *k;
```

krb_string_to_key converts a null-terminated string of arbitrary length (e.g., a user's password) into an 8 byte key, with odd byte parity, per the FIPS Data Encryption Standard (DES) specification. A one-way function is used to convert the string to a key, making it very difficult to reconstruct the string, given the key. The *s* argument is a pointer to the string, and *k* should point to a *C_Block* supplied by the caller to receive the generated key. No meaningful value is returned. Void is not used for compatibility with other compilers. The algorithm for the conversion is described below.

The first step is to flatten the input string into a stream of 7*length(s) bits *b* as follows:

```
        b[0] = bit 0 of s[0]
        b[1] = bit 1 of s[0]
        ...
        b[6] = bit 6 of s[0]
        b[7] = bit 0 of s[1]
```

```
        b[8] = bit 1 of s[1]
        ...
        b[7n + m] (0<=m<=6) = bit m of s[n]
```

In other words, the eighth (most significant) bit of each byte of *s* is dropped, and the remaining bits are shifted over to fill in the gaps.

The second step is to "fan-fold" and XOR *b* into a string *b'* exactly 56 bits long. For example, if *b* is 63 bits long:

```
        b'[55] = b[55] XOR b[56],
        b'[54] = b[54] XOR b[57],
        ...
        b'[49] = b[49] XOR b[62]
```

(The two steps described above can easily be combined.)

A key is 8 bytes long, but with odd parity in each byte; the least significant bit of the byte is the parity bit. The key is formed from *b'* above in two steps. The first step is to form the key with zero parity as follows:

```
        bit 1 of k[0] = b'[0]
        bit 2 of k[0] = b'[1]
        bit 1 of k[1] = b'[7]
        ...
        bit m of k[n] = b'[7n+m-1]  (1<=m<=7) and
        bit 0 of k[n] = 0
```

In other words, a zero parity bit is inserted into the stream *b'* every seven bits, resulting in the array *k* of eight 8-bit bytes. The second step is to set or clear the parity bit in each byte of *k* as appropriate.

Next, the DES key schedule of *k* is computed using *krb_key_sched.* Then the 64 bit DES cipher-block-chaining (CBC) checksum of the original string is computed, and finally, the CBC checksum is forced to odd parity. The generated checksum is the resulting key.

[CBC checksumming produces an 8 byte cryptographic checksum by cipher-block-chain encrypting the cleartext data. All of the ciphertext output is discarded, except the last 8-byte ciphertext block. If the cleartext length is not an integral multiple of eight bytes, the last cleartext block is zero filled (highest addresses). The output is always eight bytes.]

```
int krb_random_key(key)
        C_Block         *key;
```

krb_random_key generates a random encryption key (eight bytes), set to odd parity per FIPS specifications. The routine may use any algorithm it wishes to generate a key at random. The caller must supply space for the output key, pointed to by the argument *key,* then after calling *krb_random_key* should call the *krb_key_sched* routine when needed. No meaningful value is returned. Void is not used for compatibility with other compilers.

```
int krb_key_sched(k, schedule)
        C_Block         *k;
        Key_schedule    schedule;
```

krb_key_sched calculates a DES key schedule from all eight bytes of the input key, pointed to by the *k* argument, and outputs the schedule into the *Key_schedule* indicated by the *schedule* argument. Make sure to pass a valid eight byte key; no padding is done. The key schedule may then be used in subsequent encryption/decryption/checksum operations.

Many key schedules may be cached by the user for later use. The user is responsible for clearing keys and schedules as soon as they are no longer needed, to prevent their disclosure. The routine also checks the key parity, and returns 0 if the key is good, -1 indicating a key parity error, or -2 indicating use of an illegal weak key. If an error is returned, the key schedule was not created.

```
int pcbc_encrypt(input, output, length, schedule, ivec, encrypt)
        C_Block         *input;
        C_Block         *output;
        long            length;
        Key_schedule    schedule;
        C_Block         *ivec;
        int             encrypt;
```

pcbc_encrypt encrypts/decrypts using a modified block chaining mode. It differs in its error propagation characteristics from the DES cipher-block-chaining (CBC) mode, in that modification of a single bit of the ciphertext will affect ALL the subsequent (decrypted) cleartext; whereas with CBC, modifying a single bit of the ciphertext, then decrypting, only affects the resulting cleartext from the modified block and the succeeding block. PCBC mode, on encryption, "xors" both the cleartext of block N and the ciphertext resulting from block N with the cleartext for block N+1 prior to encrypting block N+1. By "ciphertext", we mean ciphertext generated using the DES Electronic Code Book (ECB) encryption mode.

If the *encrypt* argument is non-zero, the routine encrypts the cleartext data pointed to by the *input* argument into the ciphertext pointed to by the *output* argument, using the key schedule provided by the *schedule* argument, and initialization vector provided by the *ivec* argument. If the *length* argument is not an integral multiple of eight bytes, the last block is copied zero filled (highest addresses). The output is always an integral multiple of eight bytes.

If *encrypt* is zero, the routine decrypts the (now) ciphertext data pointed to by the *input* argument into (now) cleartext pointed to by the *output* argument using the key schedule provided by the *schedule* argument, and initialization vector provided by the *ivec* argument. Decryption ALWAYS operates on integral multiples of 8 bytes, so it will round the *length* provided up to the appropriate multiple. Consequently, it will always produce the rounded-up number of bytes of output cleartext. The application must determine if the output cleartext was zero-padded due to original cleartext lengths that were not integral multiples of 8.

No errors or meaningful values are returned. Void is not used for compatibility with other compilers.

```
unsigned long quad_cksum(input, output, length, out_count, seed)
        C_Block         *input;
        C_Block         *output;
        long            length;
        int             out_count;
        C_BLOCK         *seed;
```

The *quad_cksum* routine is based on the Quadratic Congruential Manipulation Detection Code described by Jueneman et al. *quad_cksum* produces a checksum by chaining quadratic operations on the cleartext data pointed to by the *input* argument. The *length* argument specifies the length of the input -- only exactly that many bytes are included for the checksum, without any padding.

The algorithm may be iterated over the same input data, if the *out_count* argument is 2, 3 or 4, and the optional *output* argument is a non-null pointer . The default is one iteration, and it will not run more than 4 times. Multiple iterations run slower, but provide a longer checksum if desired. The *seed* argument provides an 8-byte seed for the first iteration. If multiple iterations are requested, the results of one iteration are automatically used as the seed for the next iteration.

It returns both an unsigned long checksum value, and if the *output* argument is not a null pointer, up to 16 bytes of the computed checksum are written into the output.

Modifications to the algorithm described by Jueneman et al. are as follows. The accumulator (referred to as Z in the paper) is 64 bits, as is its initial value (referred to as C); and the modulus N is 2**63 - 1 rather than the suggested 2**31-1. The optional secret seed S is not implemented.

References

1. Bauer, R.K., Berson, A., and Feiertag, R.J. "A Key Distribution Protocol Using Event Markers". *ACM Transactions on Computer Systems 1*, 3 (August 1983), 249-255.

2. Birrell, Andrew D. et. al. "Grapevine: An Exercise in Distributed Computing". *CACM 25*, 4 (April 1982), 260-274.

3. Birrell, A.D. "Secure Communication Using Remote Procedure Calls". *ACM Transactions on Computer Systems 3*, 1 (February 1985), 1-14.

4. Denning, Dorothy E. and Sacco, Giovanni Maria. "Timestamps in Key Distribution Protocols". *CACM 24*, 8 (August 1981), 533-536.

5. National Bureau of Standards. "DES Modes of Operation". *Federal Information Processing Standards Publication 81* (1980).

6. National Bureau of Standards. "Data Encryption Standard". *Federal Information Processing Standards Publication 46* (1977).

7. Gifford, D.K. "Cryptographic Sealing for Information Secrecy and Authentication". *CACM 25*, 4 (April 1982), 274-286.

8. Girling, C. G. *Representation and Authentication on Computer Networks*. Ph.D. Th., University of Cambridge, April 1983. Technical report 37.

9. Jaeger, Eric. Protocol for Trusted Third Party Access Control. Bachelor Thesis, Massachusetts Institute of Technology, February 1985.

10. Jueneman, R.R. et. al. "Message Authentication". *IEEE Communications 23*, 9 (September 1985), 29-40.

11. Kent, Steven T. Encryption-Based Protection Protocols for Interactive User-Computer Communications. Master Th., Massachusetts Institute of Technology,May 1976. MIT-LCS Tech Report TR-162.

12. Miller, Steven P. Security for Local Area Networks. Tech. Rept. TR-227, Digital Equipment Corporation, August, 1983.

13. Needham, R.M. and Herbert, A.J.. *The Cambridge Distributed Computing System.* Addison-Wesley, London, 1982.

14. Needham, R. M. and Schroeder M. D. "Using Encryption for Authentication in Large Networks of Computers". *CACM 21*, 12 (Dec 78), 993-999.

15. Neuman, Barry Clifford. Sentry, A Discretionary Access Control Server. Bachelor Thesis, Massachusetts Institute of Technology, May 1985.

16. Popek, Gerald J. and Kline, Charles S. "Encryption and Secure Computer Networks". *Computing Surveys 11*, 4 (December 1979), 331-356.

17. Voydock, Victor L., and Kent, Stephen T. "Security Mechanisms in High-Level Network Protocols". *Computing Surveys 15*, 2 (June 1983), 135-171.