# The Evolution of the *Kerberos* Authentication Service

*John T. Kohl*

Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
jtkohl@mit.edu

*ABSTRACT*

The Kerberos Authentication Service, developed at MIT, has been widely adopted by other organizations to eliminate the trusted-host problem in open networks. While a step up from traditional security in networked systems, Kerberos version 4 is not sufficiently flexible for some environments. These inflexibilities and the remedies introduced with the Kerberos version 5 are described.

## Introduction

The Kerberos Authentication Service was originally developed at the Massachusetts Institute of Technology (MIT) for its own use to protect Project Athena's emerging network services. Versions 1 through 3 were internal development versions; protocol version 4 has achieved widespread use. However, it was designed for the envisioned use at MIT, and does not completely "fill the bill" for sites with different models of computer use and administration. Protocol version 5 incorporates new features suggested by experience with version 4 which make it useful in more situations. Version 5 was designed by Clifford Neuman of the University of Washington and the author, based in part upon input from many contributors familiar with version 4.

The first section of this paper briefly discusses the Kerberos model and basic protocol exchanges. Section 2 discusses the shortcomings of version 4. The third section reviews the new features found in version 5. Section 4 discusses the implementation of the new protocol and the compatibility support for converting existing applications from version 4 to version 5. The final section concludes with a status update and considerations of future work.

## Terminology and conventions

A *principal* is the basic entity which participates in network authentication exchanges. A principal usually represents a user or the instantiation of a network service on a particular host. Each principal is uniquely named by its *principal identifier.*

Systems like the Data Encryption Standard (DES) [FIPS46] which use a single key for both encryption and decryption are referred to as *secret-key cryptosystems.* The keys used in such systems are called *secret* keys. Encryption systems like RSA [Riv78] which use different keys for encryption and decryption are referred to as *public-key cryptosystems;* their encryption keys are referred to as *public* or *private* depending on whether the key is widely known or known only to a single

This paper was presented at the Spring 1991 EurOpen Conference, in Tromsø, Norway.

entity.

*Plaintext* refers to an unencrypted message, while *ciphertext* refers to the encrypted form of the message.

In figures, encryption is denoted by showing the plaintext surrounded by curly braces ({})followed by a key (K) whose subscript denotes the principal(s) who possess or have access to that key. Thus, "foo" encrypted under c's key is $\{foo\}K_c$.

## 1. The *Kerberos* Model

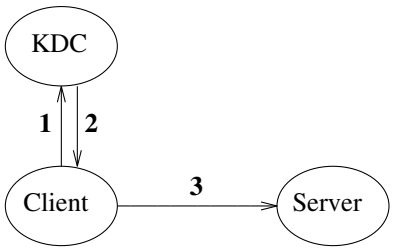
Kerberos was developed to enable network applications to securely identify their peers. To achieve this, the initiating party (the client) conducts a three-party message exchange in order to send the contacted party (the server) an assurance of the client's identity. This assurance takes the form of a *ticket* (shown in figures as $T_{c,s}$) which identifies the client, and an *authenticator* (shown in figures as $A_{c,s}$) which serves to validate the use of that ticket and prevent an intruder from replaying the same ticket to the server in a future session. A ticket is only valid for a given time interval, called its *lifetime.* When the interval ends, the ticket expires; any later authentication exchanges would require a new ticket.

Tickets are issued by a trusted third party *Key Distribution Center* (KDC). As suggested by the Needham and Schroeder protocol [Nee78], the KDC is trusted to hold in confidence secrets known to each client and server on the network (those secrets are established either out-of-band or through an encrypted channel). That trust forms the basis upon which clients and servers can believe the authenticity of the messages they receive.

Each installation establishes its own autonomously administered KDC. Each such installation comprises a *realm.* Most currently-operating sites have chosen realm names that parallel their names under the Internet domain name system (e.g. Project Athena's realm is `ATHENA.MIT.EDU`). Clients in separate realms can authenticate to each other if the administrators of those realms have previously arranged a shared secret.

### 1.1. The initial ticket exchange

Figure 1 shows graphically the messages† exchanged in an application's authentication process. Both Kerberos versions 4 and 5 share the same framework for messages (although the encoding details of the messages differ). A typical application requires a three-message exchange with each server to establish authentication on its first invocation and a single message on subsequent invocations (client caching eliminates the need for the first two messages until the ticket expires).




**1.** Client $\rightarrow$ KDC: c, s
**2.** KDC $\rightarrow$ Client: $\{K_{c,s}\}K_c$ ,$\{T_{c,s}\}K_s$
**3.** Client $\rightarrow$ Server: $\{A_c\}K_{c,s}$ ,$\{T_{c,s}\}K_s$

(In version 4, message 2 is $\{K_{c,s}$ ,$\{T_{c,s}\}K_s\}K_c$)

**Figure 1:** *Getting and using an Initial Ticket*

An application client contacts the KDC to obtain a ticket and associated credentials. The KDC generates a new ticket by selecting a random encryption key $K_{c,s}$, called the *session key,* to include in the ticket, setting the start and expiration times in the ticket as requested, and encrypting the ticket with the server's key $K_s$. It assembles the ticket and session key into the response and encrypts it with the client's secret key $K_c$. The client decrypts the response using its key (which may be algorithmically derived from a password) and caches the ticket and associated session key for future use. It then presents the ticket and a freshly-generated authenticator to an application server formatted as a KRB_AP_REQ (application request) message.

† The figures actually show a simplified version of the messages for clarity. Other message fields are present in the actual messages, but are primarily for "bookkeeping" purposes not relevant to the present discussion.

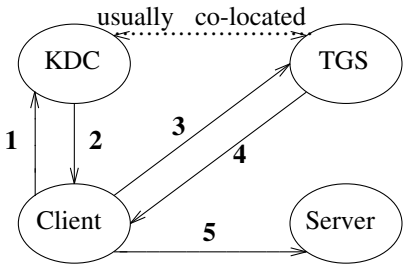
The server can decrypt this ticket using its own secret key (which is kept in secure storage on the server's host) and verify the identity of the client. If the client desires authentication of the server, the server can send a reply to the client using the key $K_{c,s}$ from the ticket, enabling the client to verify the identity of the server (only the proper server could obtain this key, as it is inside the encrypted ticket, and no intruder can gain the server's secret key). More detail on the formats of the messages used in version 4 can be found in [Ste88] and [Mil87]; detail on version 5 formats are in [Koh90].

### 1.2. The additional ticket exchange

In order to reduce the risk of exposure of the client's secret key $K_c$ and make the use of Kerberos more transparent to the user, the above exchange is primarily used to obtain a ticket for a special *ticket-granting server* (TGS). Once this ticket-granting ticket (TGT) is obtained, the client erases the copy of the client's secret key to prevent its disclosure.

The TGS is logically distinct from the KDC which provides the initial ticket service, but runs on the KDC host and has access to the same database of clients and keys used by the KDC. A client presents its TGT (along with other request data) to the TGS as it would present it to any other application server (in a KRB_AP_REQ); the TGS verifies the ticket, authenticator and accompanying request, and replies with a new ticket for the application server. The protected part of the reply is encrypted with the session key from the TGT, so the client need not retain the original secret key $K_c$ to decrypt and use this reply.

The client then uses these new credentials to authenticate itself to the server, and perhaps to verify the identity of the server. Once the authentication is established, the client and server share a common session key $K_{c,s}$, which has never been transmitted over the network without being encrypted. They may use this key to protect or obscure their messages. Kerberos provides message formats which an application may generate as needed with the session key to assure the integrity or both the integrity and privacy of a message.




**1.** Client $\rightarrow$ KDC: c, tgs
**2.** KDC $\rightarrow$ Client: $\{K_{c,tgs}\}K_c$ ,$\{T_{c,tgs}\}K_{tgs}$
**3.** Client $\rightarrow$ TGS: $\{A_c\}K_{c,tgs}$ ,$\{T_{c,tgs}\}K_{tgs}$, s
**4.** TGS $\rightarrow$ Client: $\{K_{c,s}\}K_{c,tgs}$ ,$\{T_{c,s}\}K_s$
**5.** Client $\rightarrow$ Server: $\{A_c\}K_{c,s}$ ,$\{T_{c,s}\}K_s$

(In version 4, message **2** is $\{K_{c,tgs}$ ,$\{T_{c,tgs}\}K_{tgs}\}K_c$, and message **4** is $\{K_{c,s}$ ,$\{T_{c,s}\}K_s\}K_{c,tgs}$)

**Figure 2:** *Getting a service ticket*

## 2. Why change it? Version 4 limitations

Although Kerberos version 4 is in widespread use, it is not sufficiently flexible to meet the needs of some sites. As a result, work on Kerberos version 5 commenced in 1989, fueled by discussions with version 4 users and administrators about their experiences with the protocol and MIT's implementation.

### 2.1. Environmental shortcomings

Since Kerberos version 4 was targeted primarily for the Project Athena environment (described in [Tre88]), it has several features which can be troublesome in other environments:

**Encryption system dependence:** The version 4 protocol uses only the Data Encryption Standard (DES) to encrypt messages. The export of DES from the USA is restricted by the U.S. Government, making truly widespread use of version 4 difficult.

**Internet protocol dependence:** Version 4 requires the use of Internet Protocol (IP) addresses, which makes it unsuitable for some environments.

**Message byte ordering:** Version 4 uses a "receiver makes right" philosophy for encoding multi-byte values in network messages, where the sending

host encodes the value in its own natural byte order and the receiver must convert this byte order to its own native order. While this makes communication between two hosts with the same byte order simple, it does not follow established conventions and will preclude interoperability if some machine with an unusual byte order not understood by the receiver is used.

**Ticket lifetimes:** The valid life of a ticket in version 4 is encoded by a UNIX † timestamp issue date and an 8-bit lifetime quantity in units of five minutes, resulting in a maximum lifetime of 21¼ hours. Some environments require longer lifetimes for proper operation (e.g. a long-running simulation which requires valid Kerberos credentials during its entire execution).

**Authentication forwarding:** Version 4 has no provision for allowing credentials issued to a client on one host to be forwarded to some other host and used by another client. This may be useful if an intermediate service needs to access some resource with the rights of the client (e.g. a print service needs access to the file service to retrieve a client's file for printing), or if a user logs into another host on the network and wishes to pursue activities there with the privileges and authentication she had on the originating host.

**Principal naming:** In version 4, principals are named with three components: name, instance, and realm, each of which may be up to 39 characters long. These sizes are too short for some applications and installation environments. In addition, due to implementation-imposed conventions the normal character set allowed for the name portion excludes the period (.), which is used in account names on some systems. These same conventions dictate that the account name match the name portion of the principal identifier, which is unacceptable in situations where Kerberos is being installed in an existing network with non-unique account names.

**Inter-realm authentication:** Version 4 provides cooperation between authentication realms by allowing each pair of cooperating realms to exchange an encryption key to be used as a secondary key for the ticket-granting service. A client can obtain tickets for services from a foreign realm's KDC by first obtaining a ticket-granting ticket for the foreign realm from its local KDC and then using that TGT to obtain tickets for the foreign application server (see Figure 3). This pair-wise key exchange makes inter-realm ticket requests and verification easy to implement, but requires $O(n^2)$ key exchanges to interconnect $n$ realms (see Figure 4). Even with only a few cooperating realms, the assignment and management of the inter-realm keys is an expansive task.

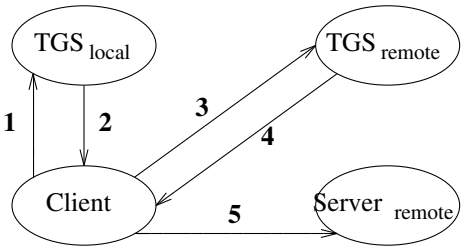
### 2.2. Technical deficiencies

In addition to the environmental problems, there are some technical deficiencies in version 4 and its implementation. [Bel90] provides detailed analyses of some of these problems.

**Double Encryption:** As shown in Figure 1, the ticket issued by the Kerberos server in version 4 is encrypted twice when being transmitted to the client, and only once when sent to the application server. There is no need to encrypt it in the message from the KDC to the client, and doing so can be wasteful of processing time if encryption is computationally intensive (as will be the case for most software-based encryption implementations; see [Mer90] for discussion of fast software-based encryption methods).

**PCBC encryption:** Kerberos version 4 uses a modified mode of DES to encrypt its messages. [FIPS81] describes the normal cipher-block-chaining (CBC) mode of DES.

† UNIX is a Trademark of Bell Laboratories.




**1.** Client $\rightarrow$ TGS $_{local}$ : $\{A_c\}K_{c,tgs}$ ,$\{T_{c,tgs}\}K_{tgs}$ , $tgs_{rem}$
**2.** TGS $_{local}$ $\rightarrow$ Client: $\{K_{c,tgs_{rem}}\}K_{c,tgs}$ ,$\{T_{c,tgs_{rem}}\}K_{tgs_{rem}}$
**3.** Client $\rightarrow$ TGS $_{remote}$ : $\{A_c\}K_{c,tgs_{rem}}$ ,$\{T_{c,tgs_{rem}}\}K_{tgs_{rem}}$, $s_{rem}$
**4.** TGS $_{remote}$ $\rightarrow$ Client: $\{K_{c,s_{rem}}\}K_{c,tgs_{rem}}$ ,$\{T_{c,s_{rem}}\}K_{s_{rem}}$
**5.** Client $\rightarrow$ Server $_{remote}$ : $\{A_c\}K_{c,s_{rem}}$ ,$\{T_{c,s_{rem}}\}K_{s_{rem}}$

(In version 4, message **2** is $\{K_{c,tgs_{rem}}$ ,$\{T_{c,tgs_{rem}}\}K_{tgs_{rem}}\}K_{c,tgs}$, and message **4** is $\{K_{c,s_{rem}}$ ,$\{T_{c,s_{rem}}\}K_{s_{rem}}\}K_{c,tgs_{rem}}$)

**Figure 3:** *Getting a foreign realm service ticket*

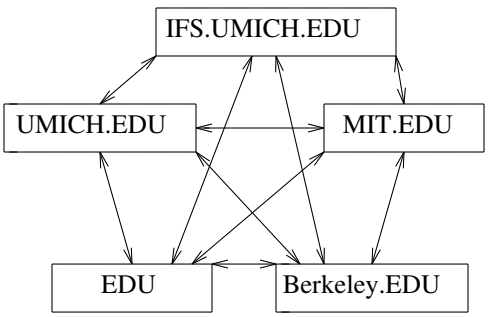



**Figure 4:** *Version 4 realm interconnections*

Kerberos version 4 uses a non-standard modified version called plain- and cipher-block-chaining mode (PCBC ). This mode was an attempt to provide data encryption and integrity protection in one operation. Unfortunately, it is flawed since an intruder can modify a message with a special block-exchange attack and have this modification pass undetected to the recipient [Koh89].

**Authenticators and replay detection:** Kerberos version 4 uses an encrypted timestamp method to verify the freshness of messages and prevent an intruder from staging a successful replay attack. If an authenticator (which contains the timestamp) is out of date or is being replayed, the application server rejects the authentication. However, maintaining a list of unexpired authenticators which have already been presented to a service can be hard to implement properly (and indeed is not implemented in the version 4 implementation MIT distributes).

**Password attacks:** The initial exchange with the Kerberos server encrypts the response with a client's secret key, which in the case of a user is algorithmically derived from a password. An intruder is able to record an exchange of this sort and, without alerting any system administrators, attempt to discover the user's password by decrypting the response with each password guess. Since the response from the Kerberos server includes plaintext she can verify, the intruder can try as many passwords

as she has available, and she will know when she's found the proper password, since the decrypted response will make sense [Lom89].

**Session keys:** Each ticket issued by the KDC contains a key specific to that ticket, called a session key, which may be used by the client and server to protect their communications once authentication has been established. However, since many clients use a ticket multiple times during a user's session, it may be possible for an intruder to replay messages from a previous connection to clients or servers which do not properly protect themselves (again, MIT's version 4 implementation does not properly implement this protection for the KRB_SAFE and KRB_PRIV messages). Additionally, there are situations in which a client wishes to share a session key with several servers. This requires special non-standard application negotiations in version 4.

**Cryptographic checksum:** The cryptographic checksum (sometimes called a message authentication code or hash or digest function) used in version 4 is based on the quadratic algorithm described in [Jue85]. The MIT implementation does not perform this function as described; the suitability of the modified version as a cryptographic checksum function is unknown.

## 3. Remedies and changes introduced with version 5

Version 5 of the protocol has slowly evolved over the past two years based on implementation experience and discussions within the community of Kerberos version 4 users. Its final form is now nearing closure, and a draft description of the protocol is available [Koh90]. It addresses the concerns above and provides additional functionality.

### 3.1. Changes between versions 4 and 5

**Use of encryption**

To modularise the system and ease export-regulation considerations for version 5, the use of encryption is separated into distinct software modules which can be replaced or removed by the programmer as needed. When encryption is used in a protocol message, the ciphertext is tagged with a type identifier so that the recipient can identify the appropriate decryption algorithm necessary to interpret the message.

Each encryption algorithm is responsible for providing sufficient integrity protection for the plaintext so that the receiver can verify that the ciphertext was not altered in transit. If the algorithm does not have such properties, it can be augmented by including a checksum in the plaintext before encryption. By doing this, we can discard the flawed PCBC DES mode, and use the standard CBC mode with an embedded checksum over the plaintext.

Encryption keys are tagged with a type and length when they appear in messages. Since it is conceivable to use the same key type in multiple encryption systems (e.g. different variations on DES encryption), the key type may not map one-to-one to the encryption type.

**Network addresses**

When network addresses appear in protocol messages, they are similarly tagged with a type and length field so the recipient can interpret them properly. If a host supports multiple network protocols or has multiple addresses of a single type, all types and all addresses can be provided in a ticket.

**Message encoding**

Network messages in version 5 are described using the Abstract Syntax Notation One (ASN.1) syntax [ISO8824] and encoded according to the basic encoding rules [ISO8825]. This avoids the problem of independently specifying the encoding for multi-byte quantities as was done in version 4. It makes the protocol description look quite different from version 4, but it is primarily the presentation of the message fields that changes; the essence of the Kerberos version 4 protocol remains.

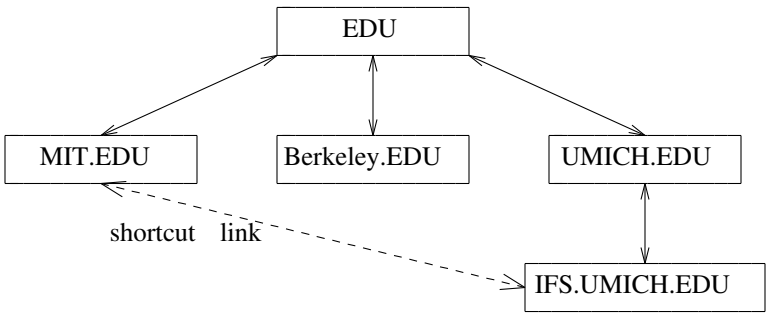



**Figure 5:** *A version 5 hierarchy of realms*

**Ticket changes**

The Kerberos version 5 ticket has an expanded format to accommodate the required changes from the version 4 ticket. It is split into two parts, one encrypted and the other in plaintext. The server's name in the ticket is in plaintext, since it need not be encrypted to provide secure authentication. The server's name is retained since it may be needed to select a key with which to decrypt the ticket if a server has multiple identities (such as an inter-realm TGS). Everything else remains encrypted. The ticket lifetime is encoded as a starting time and an expiration time (rather than a specific lifetime field), affording nearly limitless ticket lifetimes. The new ticket also contains a new flags field and other new fields used to enable the new features described below (such as authentication forwarding).

**Naming principals**

Principal identifiers are multi-component names in Kerberos version 5. The identifier is encoded in two parts, the realm and the remainder of the name. The realm is separate to facilitate easy implementation of realm-traversal routines and realm-sensitive access checks. The remainder of the name is a sequence of however many components are needed to name the principal. The realm and each component of the remainder are each encoded as an ASN.1 `GeneralString`, so there are few practical restrictions on the characters available for principal names.

**Inter-realm support**

In version 5, different Kerberos realms cooperate by establishing a hierarchy of realms (based on the name of the realm; see Figure 5). Any realm can interoperate with any other realm in the hierarchy as long as they can interoperate with the realms between them in the hierarchy. Each realm exchanges a different inter-realm key with its parent node and each child node, and uses that key and a common encryption system to obtain tickets for each successive realm along the path. This arrangement reduces the number of key exchanges to $O(\log(n))$.

When an application needs to contact a server in a foreign realm, it "walks" up and down the tree toward the destination realm, contacting each realm's KDC in turn, asking for a ticket-granting ticket to the foreign realm. In most cases, the KDC will issue a ticket for the next node in the proper direction on the tree. If a realm has established a "shortcut" spanning link with some realm further in the path, it issues a ticket-granting ticket for that realm instead. This way every realm can interoperate, and heavily-traveled paths can be optimised with a direct link.

When a ticket for the end service is finally issued, it will contain an enumeration of all the realms consulted in the process of requesting the ticket. An application server which applies strict authorization rules is permitted to reject authentication which passes through certain untrusted realms.

### 3.2. New protocol features in version 5

#### Ticketing options

In addition to the ticket changes discussed above, there are a set of timestamps and flags which allow more flexible use of tickets than was available in version 4.

Each ticket issued by the KDC is flagged as having been issued based on an initial ticket exchange or an additional ticket exchange. Some application servers (such as password changing programs) may require that a client present a ticket obtained by direct use of the client's secret key $K_c$, so that intruders who might try to abuse that service cannot simply steal cached credentials from an unattended user session.

Tickets may be issued as renewable tickets with two expiration times, one for a time in the near future, and one for a farther point. The ticket expires like normal at the earlier time, but if it is presented to the KDC in a renewal request before this earlier expiration time, a replacement ticket is returned which is valid for an additional period of time. The KDC will not renew a ticket beyond the second expiration indicated in the ticket. This mechanism has the advantage that although the credentials can be used for long periods of time, the KDC may refuse to renew tickets which are reported as stolen and thereby thwart their continued use.

A similar mechanism is available to assist in authentication during batch processing. A ticket issued as postdated and invalid will not be valid until its post-dated starting time passes and it is replaced with a validated ticket. The client validates the ticket by presenting it to the KDC as described above for renewable tickets.

Authentication forwarding can be implemented by contacting the KDC with the additional ticket exchange and requesting a ticket valid for a different set of addresses than the TGT used in the request. The KDC will not issue such tickets unless the presented TGT has a flag set indicating that this is a permissible use of the ticket. When the entity on the remote host is granted only limited rights to use the authentication, the forwarded credentials are referred to as a *proxy* (after the proxy used in legal and financial affairs). Proxies are handled similarly to forwarded tickets, except that new proxy tickets will not be issued for a ticket-granting service; they will only be issued for application server tickets.

In certain situations, an application server (such as an X Window System server) will not have reliable, protected access to an encryption key necessary for normal participation as a server in the authentication exchanges. In such cases, if the server has access to a user's ticket-granting ticket and associated session key (which in the case of single-user workstations may well be the case), it can send the server's ticket-granting ticket to the client, who then presents it and the user's own ticket-granting ticket to the KDC. The KDC then issues a ticket encrypted in the session key from the server's ticket-granting ticket; the application server has the proper key to decrypt and process this ticket. [Dav90] provides details on the fine points of this exchange.

#### Authorization data

Some network operating system applications need to provide tamper-proof arbitrary data to an application server (for example, such information might include group membership information). It is convenient to collect or generate such information at a KDC and insert it into a ticket as *authorization data,* where it is encrypted and protected from any client or intruder tampering. In the protocol's most general form, a client may request that the KDC include or add to such data in a new ticket. The KDC does not remove any authorization data from a ticket; the TGS always copies it from the TGT into the new ticket, and then adds any requested additional authorization data. Upon decryption of a ticket, the authorization data is available to the application server. While Kerberos makes no interpretation of these data, the application server is expected to use the authorization data to appropriately restrict the client's access to its resources.

This field can be used in a proxy ticket to create a capability. The client requesting the proxy ticket from the KDC specifies any

authorization restrictions in the authorization data, then securely transmits the proxy ticket and session key to another entity, which can then use the ticket to obtain limited service from an application server. [Neu91] discusses in more detail some uses of this field.

**Pre-authentication data**

In an effort to help alleviate the ever-present problem of stolen passwords, the Kerberos version 5 protocol has fields available in the initial- and additional-ticket exchange messages to enable alternative identification methods, such as hand-held authenticators (devices which have internal circuitry to help a user identify herself to the system). In the initial ticket exchange, these fields might be used to alter the client's key $K_c$ in which the reply is encrypted; they may also be used to implement a challenge/response protocol which must be completed before the issuance of a ticket. Both alternatives can help alleviate the password attack problems discussed above, if they make the derivation of the key from a typed password hard or impossible to compute without the additional information utilised in the exchange, or if they eliminate the use of passwords to derive or protect encryption keys.

This pre-authentication data field is used by the client in the additional ticket exchange to pass the ticket-granting ticket to the KDC; since it is a variable-length array, other values may also be sent in the additional-ticket exchange.

**Subsession key negotiation**

Tickets may be cached by clients for later use before their expiration dates. In order to avoid problems caused by re-using a ticket's session key (which is held for the duration of the user's login session) for several application sessions, a server and client can cooperate to choose a new *subsession key* to protect just that application session. This subsession key is discarded after the application session concludes.

A clever use of this negotiation allows an application to use a broadcast medium while protecting its messages to several recipients. The application can negotiate individually with each recipient to use the same subsession key before beginning its broadcasts.

**Sequence numbers**

Kerberos provides two messages for applications to protect their communications. The KRB_SAFE message uses a cryptographic checksum to insure data integrity. The KRB_PRIV message uses encryption to insure integrity and privacy. In version 4 these messages included as control information a timestamp and the sender's network address. With version 5, an application may elect to use a timestamp (as before) or a sequence number. If the timestamp is used, the receiver must record the known timestamps to avoid replay attacks; if a sequence number is used the receiver must verify that the messages arrive in the proper order without gaps. There are situations where one choice makes applications simpler (or even possible) to implement; see the discussions in [Koh90].

## 4. Implementation features

### 4.1. The base implementation

The MIT implementation of the version 5 protocols is composed of several run-time libraries with which a program may link. The core library functions will probably be used by all applications; other libraries or subsystems may be replaced or omitted as needed by an application programmer. All code is currently written in "C."

**The base functions:** The core Kerberos library contains the routines which assemble, disassemble and interpret the network messages. This includes ASN.1 encoding and decoding functions which convert from a machine's native format to the network encoding (currently based on the ISODE library, but used in a way to allow easy replacement of the ASN.1 routines), routines which verify that requests are answered as expected, and routines to determine which messages are necessary. This core set of routines calls out to the remaining portions of the library as required. A

programmer may replace those portions at certain specified interfaces.

**Encryption routines:** Since multiple encryption types may be in use simultaneously, the core functions call encryption routines through a function table which has entries provided by each encryption system implementation. Table 1 shows the fields in a cryptosystem table entry. The core library provides a default cryptosystem table, initialised to list the known encryption types. A programmer may load his own cryptosystem table to replace the default table and avoid linking with the default encryption libraries.

In an attempt to alleviate some possible export restrictions, MIT's implementation distributes its encryption systems separately from the remainder of the system. Only DES is currently available from MIT.

**Checksum routines:** In a similar fashion to the encryption routines, the core routines call any needed checksum functions through a function table, and compute any necessary sizes based on the information in the table. Certain applications of checksum technology require that the checksum have certain properties. The table entry indicates whether the checksum is keyed (its algorithm is perturbed by an encryption key which cannot be discovered with knowledge only of the algorithm and the checksummed text) and whether the checksum is collision proof (it is computationally infeasible to discover a different checksum text which has the same checksum). Table 2 shows the fields in a checksum table entry. The core library provides a replaceable default checksum table.

Three checksums are currently available from MIT: the CRC-32, which is neither keyed nor collision proof (but it is useful for integrity checks within encryption systems); the DES message authentication code (MAC), which is both keyed and collision proof, and MD4 [Riv90], which is collision proof but not keyed.

**Credentials cache and key table routines:** When clients store tickets and credentials in a cache, the core routines call out through a credentials cache table entry to a separate library module which implements the storing and searching routines for credentials caches (see Table 3). An environment variable can be used to specify the default type and location of a credentials cache, so a user can switch between different types and locations of caches as needed (perhaps if she is working in two roles and wants to keep the credentials for each role separate). MIT's implementation provides two credentials cache implementations, one built on C "standard I/O" routines and the other built on UNIX file-descriptor semantics. Other implementations could provide shared-memory or kernel-resident caches.

Servers likewise store their secret keys $K_s$ in key tables accessed by the core routines through a key table function table entry (see Table 4). MIT's implementation provides a key table library built on C "standard I/O" routines.

**KDC database support:** All accesses to the KDC's principal database by the KDC and administrative programs are mediated by a database library which can be replaced if needed. MIT's implementation uses the UNIX *dbm* database system. Since *dbm* does not do any record or database locking, it is augmented with separate locking code to mediate between writers and readers. Administrative requests (e.g. adding entries, changing keys or passwords) can be handled on-line.

**Operating system support:** Although it is targeted for UNIX systems, the MIT implementation is careful to access operating system features only in a few well-contained modules. An operating system support library performs all the accesses required by the rest of the code, such as transmitting

| *Field name* | *Field use* |
|---|---|
| `encrypt_func()` | Entry point to encrypt an input. |
| `decrypt_func()` | Entry point to decrypt an input. |
| `process_key()` | Entry point to perform any necessary key processing (must be called before either `encrypt_func` or `decrypt_func`). |
| `finish_key()` | Entry point to clean up from any key processing (called after `encrypt_func` or `decrypt_func`). |
| `string_to_key()` | Entry point to convert a string to a key. |
| `init_random_key()` | Entry point to initialise state for generating random keys. |
| `finish_random_key()` | Entry point to clean up state from random key generation. |
| `random_key()` | Entry point to generate a random key. |
| `block_length` | The minimum size of input and output for this encryption system. |
| `pad_minimum` | The minimum padding space required of any input (used to insert integrity checks). |
| `keysize` | The length (in octets) of keys used by this system. |
| `proto_enctype` | The encryption type value used in the protocol. |
| `proto_keytype` | The key type value used in the protocol. |

**Table 1:** *A cryptosystem table entry*

| *Field name* | *Field use* |
|---|---|
| `sum_func()` | Entry point for the checksum function. |
| `checksum_length` | The length (in octets) of the checksum produced by the `sum_func`. |
| `is_collision_proof` | Binary value indicating whether this checksum is collision proof. |
| `uses_key` | Binary value indicating whether this checksum is keyed. |

**Table 2:** *A checksum table entry*

| *Field name* | *Field use* |
|---|---|
| `prefix` | The string prefix used to name this variety of credentials cache. |
| `get_name()` | Entry point to return the name of a key table. |
| `resolve()` | Entry point to reserve a cache name, prepare to access it, and return an access handle. |
| `gen_new()` | Entry point to generate a unique credentials cache, prepare to access it, and return an access handle. |
| `init()` | Entry point to create or erase a cache. |
| `destroy()` | Entry point to destroy a cache and invalidate the access handle. |
| `close()` | Entry point to close a cache and invalidate the access handle. |
| `store()` | Entry point to store an entry in the cache. |
| `retrieve()` | Entry point to retrieve an entry from the cache. |
| `get_princ()` | Entry point to retrieve the primary principal named in the cache. |
| `get_first()` | Entry point to prepare to sequentially read all entries in the cache. |
| `get_next()` | Entry point to read the next entry in the cache. |
| `end_get()` | Entry point to stop reading every entry in the cache. |
| `remove_cred()` | Entry point to remove an entry or entries from the cache. |
| `set_flags()` | Entry point to set various flags for the cache routines. |

**Table 3:** *A credentials cache table entry*

| *Field name* | *Field use* |
|---|---|
| `resolve()` | Entry point to resolve a key table name, prepare to access it, and return an access handle. |
| `get_name()` | Entry point to return the name of a key table. |
| `close()` | Entry point to close a key table and invalidate its handle. |
| `get()` | Entry point to search the key table and return a requested entry. |
| `start_seq_get()` | Entry point to prepare to read every key in the table. |
| `get_next()` | Entry point to read the next key in the table. |
| `end_get()` | Entry point to stop reading every key in the table. |
| `add()` | Entry point to add an entry to the table. |
| `remove()` | Entry point to delete an entry from the table. |

**Table 4:** *A key table function table entry*

and receiving network messages, examining configuration files, checking the system's time-of-day, translating from account names to Kerberos names (and *vice versa* ), and performing rudimentary account access checks.

### 4.2. User interaction

If all parts of Kerberos are working properly, a user will normally not be aware that Kerberos authentication is in use by her applications. The normal login process obtains and caches an initial ticket-granting ticket, and applications automatically obtain and cache service tickets as required. Only when authentication fails will the user become aware of the underlying use of Kerberos.

If the user needs to refresh tickets (say, if they expire), then she can use the *kinit* program, which will get a new ticket-granting ticket after reading her password from the keyboard. She may examine the cached tickets with *klist* and destroy the cache with *kdestroy*.

When principal names need to be displayed to human users, by convention† they are represented as the sequence of name components separated by slashes (*/*), followed by an at-sign (@), and the realm name. Thus, a principal with two name components `jtkohl` and `role2` in the realm `ATHENA.MIT.EDU` would be represented as `jtkohl/role2@ATHENA.MIT.EDU`.

† Please note that this is only a *convention,* and other implementations may display the principal names differently.

**Password to key conversion**

Since users are not good at remembering binary encryption keys, we provide routines to convert passwords into keys. The algorithm used to convert a password into an encryption key performs a non-invertible transformation, so that an attacker cannot discover a user's password if he knows the $K_c$. The conversion can be seeded with an additional string which perturbs the output key, so that a user who is registered in multiple realms and uses the same password in two of those realms will have a different $K_c$ in each realm. Without this perturbation, someone discovering the user's key in one realm could impersonate that user in the other realm (without knowing her password!). When no additional perturbation string is supplied, the resulting key is the same as the key produced by the version 4 algorithm.

### 4.3. Compatibility support for version 4

There is a small but growing base of Kerberos version 4 applications, and a number of sites running a Kerberos version 4 authentication server. Several features of MIT's implementation of version 5 can help sites and programmers convert to using the newer protocol.

**Interface compatibility:** MIT's implementation of version 5 includes a "glue library" which may be used to convert applications which are coded to use the version 4 application programming interface (API) to use version 5 protocol messages and routines. This

library converts data structures as much as possible between the differing version 4 and version 5 data structures. In many cases (especially those that use only a common subset of the version 4 library functions), an application need only be re-linked with this library and the remainder of the version 5 code to use version 5 protocols. However, such applications will no longer be compatible with older peer processes, which would still expect the version 4 messages, and continued maintenance may be made more difficult.

**A generic authentication interface:** An authentication-system independent programming interface [Lin90] has been discussed by representatives from several computer system manufacturers. MIT provides a binding of this interface to the Kerberos version 5 implementation. For new applications which desire the most flexibility to have different authentication systems (even ones not yet invented) supplied by the system, this offers an attractive abstraction boundary. If an application needs more detailed access to a particular authentication system, it would probably do better to code to that system's native interface.

**Protocol compatibility:** For those sites which wish to convert the Kerberos server to provide the features of version 5, a compatibility mode may be enabled on the server to access the version 5 style authentication database but provide version 4 format tickets and messages. This allows an administrator to convert a version 4 installation to version 5 slowly, by supporting the old users with the compatibility code. After some grace period, the version 4 compatibility would be turned off. If a user wishes to use both version 4 and version 5 programs simultaneously, the compatibility code can utilise the pre-authentication data in the ticket responses to indicate which algorithm should be used to convert her password to an encryption key.

**Interface coexistence:** The MIT version 5 libraries were purposely designed to allow an application to simultaneously support both versions 4 and 5, and this is the suggested compatibility mode. The telnet [Pos83] program distributed with the MIT code can automatically choose an authentication system to use when it connects to a remote system, based on what credentials the user holds and what versions of authentication the remote telnet server will accept. It implements the current draft specifications of the authentication [Bor90a] and encryption [Bor90b] options.

**Program compatibility:** Another possible compatibility mode can be fabricated by maintaining separate copies of network applications which use version 4 and version 5 protocol messages. The user would use a generic name for the application, and the application would try each authentication system in turn, by executing a separate copy of the program for each system (see Figure 6). When authentication is successfully completed, the application would proceed as normal. On both the client and server sides of the application, this approach requires two copies of the same program, each linked with a different authentication system. The different versions of the server would each accept requests at different network ports, and the different clients would only send a request to the server which supports its authentication type.

This approach could be mixed with the glue library and/or single-server approaches, by creating the separate clients using the glue library and/or using a single server program which understands both protocols.

## 5. Status and future work

Kerberos version 5 is a large step forward toward generalising Kerberos to make it globally useful. We believe its framework will be flexible enough to accommodate future requirements. Some items we

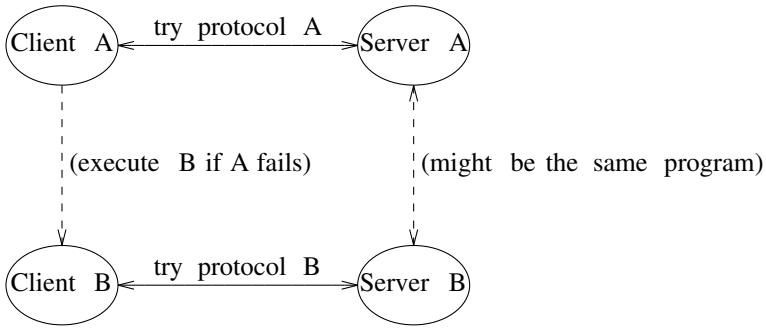



**Figure 6:** *Implementing protocol compatibility by executing separate programs*

expect to be incorporated into Kerberos in the near future include:

**Public-key cryptosystems:** The encryption specifications in Kerberos version 5 are designed primarily for secret-key cryptosystems, but there is some ongoing work into the integration of public-key cryptosystems into Kerberos, and we hope to be able to better support them in future code releases. However, public-key cryptosystems have different characteristics than secret-key systems, and their use in Kerberos may not take advantage of those characteristics.

**"Smartcards":** Several companies manufacture hand-held devices which can be used to augment normal password security methods, and there is strong interest within the industry to integrate one or more of these systems with Kerberos.

**Remote administration:** The current protocol specifications do not specify any administrative interface to the KDC database. MIT's implementation provides a sample remote administration program which allows administrators to add and modify entries and users to change their keys. We would eventually like to standardise such a protocol. Some features we would like to add include remote extraction of server key tables, password "quality checks," and a provision for servers to change their secret keys automatically every so often.

**Directional inter-realm keys:** The protocols will support the use of a different inter-realm key for each direction of an inter-realm link, but our implementation only allows for the same key to be used for both directions. We would like to allow separate keys in our implementation, to reduce the exposure from a disclosed key.

**Database propagations:** The current implementation provides reliable KDC service by a periodic bulk-copy of the KDC database to slave KDC machines. It might be more convenient and/or efficient to build the KDC on a distributed database technology. However, the technology must provide a secure, private transmission of the database elements to each server, to insure that an attacker cannot illegitimately obtain any database entry.

**Validation suites:** The current implementation does not include a complete validation suite to verify that the protocol is properly implemented. Such a suite could prevent future security problems in the case of a faulty implementation, and would help facilitate interoperation of diverse implementations.

**Applications:** There are many more network applications that would benefit from the addition of authentication which we have not had time or resources to convert. Among the highly visible examples are electronic mail and popular bulletin-board systems (such as Usenet).

**Acknowledgements**

The work described here has been the result of many MIT Project Athena and MIT Network Services staff members' visions, ideas, and hard work.

The author would especially like to thank Steve Bellovin, Clifford Neuman, Jennifer Steiner, and Ralph Swick for their comments on early drafts of this paper.

**REFERENCES**

Bel90. S. M. Bellovin and M. Merritt, ''Limitations of the Kerberos Authentication System,'' *Computer Communications Review* **20**(5), pp. 119-132 (October 1990).

Bor90a. D. Borman, Editor, ''Telnet Authentication Option,'' Internet-Draft, Internet Engineering Task Force, Telnet Working Group (July 1990).

Bor90b. D. Borman, Editor, ''Telnet Encryption Option,'' Internet-Draft, Internet Engineering Task Force, Telnet Working Group (April 1990).

Dav90. Don Davis and Ralph Swick, ''Workstation Services and Kerberos Authentication at Project Athena,'' Technical Memorandum TM-424, MIT Laboratory for Computer Science (February 1990).

Jue85. R. R. Jueneman, S. M. Matyas, and C. H. Meyer, ''Message Authentication,'' *IEEE Communications* **23**(9), pp. 29-40 (September 1985).

Koh89. John T. Kohl, ''The Use of Encryption in Kerberos for Network Authentication,'' in *Crypto '89 Conference Proceedings* , International Association for Cryptologic Research, Santa Barbara, CA (August 1989).

Koh90. John T. Kohl and B. Clifford Neuman, ''The Kerberos Network Authentication Service,'' RFC DRAFT 4, Project Athena, Massachusetts Institute of Technology (December 1990).

Lin90. John Linn, *Generic Security Service Application Program Interface,* Digital Equipment Corporation (September 1990). Version C.3.

Lom89. T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham, ''Reducing Risks from Poorly Chosen Keys,'' *Operating Systems Review* **23**(5), pp. 14-18 (December 1989).

Mer90. Ralph C. Merkle, ''Fast Software Encryption Functions,'' in *Crypto '90 Conference Proceedings* , International Association for Cryptologic Research, Santa Barbara, CA (August 1990).

Mil87. S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, *Section E.2.1: Kerberos Authentication and Authorization System,* M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).

FIPS46. National Bureau of Standards, U.S. Department of Commerce, ''Data Encryption Standard,'' Federal Information Processing Standards Publication 46, Washington, DC (1977).

FIPS81. National Bureau of Standards, U.S. Department of Commerce, ''DES Modes of Operation,'' Federal Information Processing Standards Publication 81, Springfield, VA (December 1980).

Nee78. Roger M. Needham and Michael D. Schroeder, ''Using Encryption for Authentication in Large Networks of Computers,'' *Communications of the ACM* **21**(12), pp. 993-999 (December, 1978).

Neu91. B. Clifford Neuman, ''Proxy-Based Authorization and Accounting for Distributed Systems,'' Technical Report 91-02-01, Department of Computer Science and Engineering, University of Washington (February 1991).

Pos83. J. Postel and J. Reynolds, ''TELNET Protocol Specification,'' RFC 854, University of Southern

California, Information Sciences Institute (May 1983).

Riv90. R. Rivest, ''The MD4 Message Digest Algorithm,'' RFC 1186, MIT Laboratory for Computer Science (October 1990).

Riv78. R. L. Rivest, A. Shamir, and L. Adleman, ''A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,'' *Communications of the ACM* **21**(2), pp. 120-126 (February 1978). See also U.S. Patent 4,405,829.

ISO8824. International Organization for Standardization, ''Information Processing Systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1),'' IS 8824 (December 1987). First Edition.

ISO8825. International Organization for Standardization, ''Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1),'' IS 8825 (November 1987). First Edition.

Ste88. J. G. Steiner, B. C. Neuman, and J. I. Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' pp. 191-202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).

Tre88. G. W. Treese, ''Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD,'' pp. 175-182 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).

Project Athena, Athena, Athena MUSE, Discuss, Hesiod, Kerberos, Moira, and Zephyr are trademarks of the Massachusetts Institute of Technology (MIT). No commercial use of these trademarks may be made without prior written permission of MIT.

X Window System is a trademark of MIT.

UNIX is a registered trademark of Unix Systems Laboratories, Inc. in the USA and other countries.