ULTRIX

digital

Guide to Kerberos

Order Number: AA–PBKVA–TE

ULTRIX

Guide to Kerberos

Order Number: AA-PBKVA-TE
June 1990

Product Version: ULTRIX Version 4.0 or higher

This guide describes Kerberos, its setup, and the network programming connections of the `kerberos` daemon to a Kerberos-authenticated application. Kerberos enhances security by authenticating applications to each other across machine boundaries in a distributed network. ULTRIX Kerberos currently supports the authentication of commonly networked applications, such as `named` and `auditd`.

digital equipment corporation
maynard, massachusetts

Contents

About This Manual

1 The Need for Kerberos Authentication

2 Authentication

3 The Kerberos Daemon and Utilities

4 Setting Up Kerberos

5 Creating an UPGRADE or ENHANCED Distributed Environment

6 Kerberos Programming Interface

Glossary

Examples

Figures

Tables

About This Manual

This guide provides mainly setup information about Kerberos, which enhances the security of a distributed network by authenticating shared applications. It also describes how to increase the security level in stages from BSD to UPGRADE to ENHANCED mode. It also provides some general information about Kerberos and the network programming connections of the `kerberos` daemon to a Kerberos-authenticated application.

Audience

This book is mainly for network administrators who are setting up a network with Kerberos authentication. Other users can read Chapter 1 for basic information about Kerberos.

Organization

This manual consists of six chapters, a glossary, and an index. The six chapters are:

Chapter 1, The Need for Kerberos Authentication

> Discusses how Kerberos enhances security through mutual authentication between client and server in a networked or distributed system service (DSS) environment.

Chapter 2, Authentication

> Discusses the ticket and authenticator in detail and how they implement Kerberos security features.

Chapter 3, The Kerberos Daemon and Utilities

> Discusses Kerberos masters and slaves, database and utilities, and includes an example session using database utility commands.

Chapter 4, Setting Up Kerberos

> Describes how to set up a Kerberos-authenticated `named` daemon on a Kerberos master, slave servers, and clients.

Chapter 5, Creating an UPGRADE or ENHANCED Distributed Environment

> Describes how to increase security level from the default BSD security environment to an ENHANCED security environment.

Chapter 6, Kerberos Programming Interface

> Describes the Kerberos programming interface to a network through both a high-level and low-level client-server example. Twelve interrelated sections of C code are included to facilitate this discussion.

Conventions

The following conventions are used in this guide:

%	The default user prompt is your system name followed by a right angle bracket. In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign is the default superuser prompt.
user input	This bold typeface is used in interactive examples to indicate typed user input.
system output	This typeface is used in interactive examples to indicate system output and also in code examples and other screen displays. In text, this typeface is used to indicate the exact name of a command, option, partition, pathname, directory, or file.
UPPERCASE lowercase	The ULTRIX system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
rlogin	In syntax descriptions and function definitions, this typeface is used to indicate terms that you must type exactly as shown.
macro	In text, bold type is used to introduce new terms.
filename	In examples, syntax descriptions, and function definitions, italics are used to indicate variable values; and in text, to give references to other documents.
[ ]	In syntax descriptions and function definitions, brackets indicate items that are optional.
{ | }	In syntax descriptions and function definitions, braces enclose lists from which one item must be chosen. Vertical bars are used to separate items.
. . .	In syntax descriptions and function definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
cat(1)	Cross-references to the *ULTRIX Reference Pages* include the appropriate section number in parentheses. For example, a reference to cat(1) indicates that you can find the material on the cat command in Section 1 of the reference pages.
RETURN	This symbol is used in examples to indicate that you must press the named key on the keyboard.

CTRL/*x*	This symbol is used in examples to indicate that you must hold down the CTRL key while pressing the key *x* that follows the slash. When you use this key combination, the system sometimes echoes the resulting character, using a circumflex (^) to represent the CTRL key (for example, ^C for CTRL/C). Sometimes the sequence is not echoed.
ESC X	This symbol is used in examples to indicate that you must press the first named key and then press the second named key. In text, this combination is indicated as ESC-X.
`special`	In text, each mention of a specific command, option, partition, pathname, directory, or file is presented in this type.

Related Documentation

For more information on topics related to Kerberos and DSS, see the following documents:

Guide to the BIND/Hesiod Services

Describes how to set up the Berkeley Internet Name Domain (BIND) Service and the Hesiod Name Server supported by BIND.

Security Guide for Users

Describes the security features available in ULTRIX, such as the login procedure, passwords, and sharing and protection of files.

Introduction to Networking and Distributed System Services

Describes how to set up a network in a distributed environment using time services.

Security Guide for Administrators

Describes security policy, implemented by assigning user privileges, performing and reading audits, and initializing and configuring a secure system that includes user authentication and file protection at the system level.

ULTRIX Reference Pages

The reference pages for the commands and files referred to in the manuals.

New and Changed Information

This is a new manual.

The Need for Kerberos Authentication 1

Kerberos is an authentication service that enhances the security of an open network. It is a part of Project Athena, an ongoing research project at the Massachusetts Institute of Technology, funded in part by Digital Equipment Corporation. Project Athena is a software development project for facilitating the communication among file servers and workstations in a distributed network environment. The version of Kerberos supported by ULTRIX Version 4.0 is derived from MIT/Athena's Kerberos Version 4, and will only interoperate with other implementations of that Kerberos version.

1.1 Kerberos and the Security of a Local Area Network

The process of determining the identity of an entity is called **authentication**. Authentication proves that a given entity is genuine; it is required to prevent one entity from masquerading as another entity.

The authentication of an application "X" to an application "Y" that both run on the same machine, "A," is simple. Y need only ask A for the user ID of X. Since Y trusts the integrity of the local machine, if the user ID of X is the user ID that Y expects, then X must be X.

If Y were to use the same method to authenticate X when X runs on a different machine, "B," then Y would be forced to trust machine B to provide a correct answer. The security of this method breaks down as soon as any one machine that Y is willing to trust is subverted by a hostile user. In addition, it breaks as soon as any machines that cannot be trusted by Y are allowed on the physical network to which A and B are connected. Hostile users that have control over these untrusted machines can force them to produce messages that look as though they come from machine B.

Kerberos software enables the authenticaton of an application to another without placing security trust at several points in the network. X trusts Kerberos to give Y only enough information to authenticate itself as Y to X, and Y trusts Kerberos to give X only enough information to authenticate itself as X to Y. Y no longer needs to trust machine B to authenticate X.

The ULTRIX version of Kerberos only provides for the authentication of applications that communicate across a TCP/IP network with the socket interface. Although ULTRIX Kerberos has the ability to authenticate users at log-in time, it is not supported by the base system. Also, inter-realm authentication is not supported by ULTRIX Kerberos. The realm in Kerberos is discussed in detail later, in Chapter 3.

ULTRIX Kerberos was used to authenticate certain networked applications that are critical to network security – for example, `named` and `auditd`. Therefore, each user in a distributed system services (DSS) network must still rely on the `login` program on a given machine.

Kerberos refers to network applications as **principals**. An application that requests a service from an application on another machine is a **client** principal.

The requested application is called a **service** principal.

1.2 Kerberos Security Features and Data Encryption

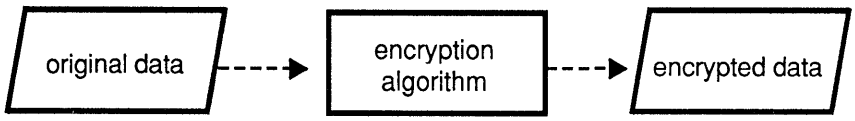
To enhance LAN security, Kerberos provides encryption algorithms and the following features:

- Password security
- Authentication of a Kerberos principal
- Reauthentication of a Kerberos principal
- Protection against the unauthorized interception, modification, and retransmission of data
- Protection against the replay of authentication data

1.2.1 Encryption

Encryption is a way of protecting sensitive data by changing it so that it looks very different from the original data. An encryption algorithm encrypts data, as shown graphically in Figure 1-1.

Figure 1-1: The Process of Data Encryption



ZK-0170U-R

An encryption algorithm prevents a hostile user from reconstructing the unencrypted text from patterns in the encrypted text. For example, after encryption the first eight bytes (64 bits) of the ASCII phrase, ''Now is the time for,'' might look like:

```
3f a4 0e 8a 98 4d 48 15
```

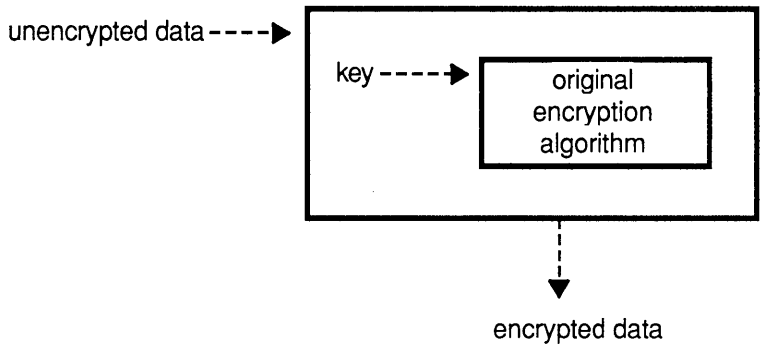
In this example, the internal ASCII representation of each character is expressed by two hexadecimal digits.

The fourth and seventh ASCII characters of the original phrase are both spaces and you might expect that they would be identical after encryption. However, the fourth encrypted character is 8a and the seventh is 48. The value of a given bit in the encrypted text depends on a large subset of bits in the unencrypted text, not on a bit-by-bit encryption mapping. For this reason, it is also possible for the *n*th bit of the unencrypted text to be identical to the *n*th bit of the encrypted text.

A **key** is an input parameter to an encryption algorithm. It modifies the algorithm and produces a new one, making it more difficult to decipher the original data. To reconstruct the original data, a hostile user would need to know both the key and the original algorithm to which it was applied.

The key that is input to the original encryption algorithm and the new encryption algorithm are shown graphically in Figure 1-2.

Figure 1-2: Using a Key to Create a New Encryption Algorithm



Kerberos uses the DES encryption library to encrypt data and passwords. Encryption is expressed by the following notation, which refers to data encrypted with the key of principal X:

```
[data] key(X)
```

Decryption of data is accomplished by reencryption with the same session key. Decryption is expressed by the following notation which refers to the decryption of data that was originally encrypted with the key of principal X:

```
[[data] key(X)] key(X)
```

Kerberos uses encryption to protect sensitive data and to avoid sending unencrypted or **cleartext** passwords over the network. Every Kerberos principal is identified by a unique **encryption key** that is known only by the `kerberos` daemon and the principal. This key identifies a Kerberos principal and is often called the **key of the principal** or the **password of the principal**.

1.2.2 Authentication of a Principal

A **ticket** is a packet of data given by the `kerberos` daemon to enable a Kerberos principal to authenticate itself directly to another Kerberos principal. For example, client application A authenticates itself to application B by sending it a ticket to request a service from B. A ticket contains a **lifespan** that causes it to expire eventually. After the ticket expires, application A cannot authenticate itself to B without obtaining another ticket from the `kerberos` daemon. This protects against the indefinite replay of a ticket stolen by a hostile user. For more detailed information about tickets, refer to Chapter 2 of this guide.

1.2.3 Reauthentication of a Kerberos Principal

After a principal A authenticates itself to principal B by passing a ticket to it, both A and B learn an encryption key called the **session key**, known only to them.

This session key is the key associated with the communication session between the two principals, A and B.

The session key is known only by A and B, so it can be used to reauthenticate principal A to principal B. For example, if A encrypts with the session key a data item of value, *x*, and if principal B decrypts the data item and obtains the same value, *x*, then B knows that A (and not a false application set up by a hostile user) must have sent the data, and A is reauthenticated to B.

1.2.4 Data Integrity

Kerberos generates an **encryption checksum** to detect the modification of data sent over the network. This checksum depends on the value of the unique session key, which is known only by two principals. The checksum would be altered if either a different session key were used, or if the data were changed by a hostile user. Therefore, the checksum enables both the detection of data modified by a hostile user, and the reauthentication of a principal to another principal. Refer to `des_crypt(3krb)` for detailed information about the checksum operation.

1.2.5 Password Security

Before a principal requests a ticket to communicate with *any* principal, it must obtain a **ticket-granting ticket** from Kerberos. The ticket-granting ticket grants permission to obtain other tickets, and it enables a principal to authenticate itself to part of the `kerberos` daemon called the **ticket-granting service**. There is a session key between the ticket-granting service and principal A. Both the session key and the ticket-granting ticket provide authentication between principal A and the ticket-granting service.

Whenever principal A needs a new ticket, it does not have to send its key or password over the network where it might be stolen. It sends it back to the ticket-granting service with a request for a ticket to communicate with another application, B, and includes in this message its ticket-granting ticket and some data encrypted in the session key between the ticket-granting service and A. This prevents sensitive data encrypted with the key of principal A from appearing in the network. It also eliminates the need for principal A to access its password constantly. Both of these features make it difficult for a hostile user to steal the key of principal A.

1.2.6 Protection Against the Replay of Authentication Data

A ticket can be reused for a long time, usually the eight hours of a typical workstation session. A hostile user could steal a ticket and replay it to B and be authenticated as a legitimate principal – for example, as A. However, after the ticket-granting service gives principal A a ticket to authenticate to principal B, principal A must create a nonreusable **authenticator** and send it with the ticket to request a service from principal B.

Since the nonreusable authenticator can only be produced by an application that knows the session key between A and B, and because it must be present to authenticate A, the ticket-authenticator pair is invulnerable to replay by a hostile user. For detailed information about protection against replay through the ticket-authenticator pair, refer to Chapter 2 of this guide.

Kerberos principals depend on network time synchronization for accurate lifespan and timestamp data. Kerberos relies on the `ntpd` daemon that implements Network Time Protocol (NTP) and the `timed` daemon that implements the Berkeley time services. For more information, refer to `ntpd(8)`, `timed(8)`, and to *Introduction to Networking and Distributed System Services.*

Authentication 2

2.1 Authentication Requirements

For an application "X" to authenticate itself to an application "Y," application X uses the `kerberos` daemon's knowledge of two items:

- The secret shared with X: the key of the principal X, which is represented by the notation, `key(X)`
- The secret shared with Y: the key of the principal Y, represented by `key(Y)`. The `kerberos` daemon obtains the keys of X and Y from the Kerberos database. See Chapters 1 and 3 for more information about this database.

  This knowledge is used to produce a new secret: the session key between X and Y, `key(X-Y)`.

By definition, this session key is known only by X and Y. Therefore, to authenticate itself to Y, application X needs only to prove to Y that it knows the session key. Similarly, to prove its identity to X, application Y must prove to X that it knows the session key.

2.2 Producing the Session Key

The first authentication step for X is to request the `kerberos` daemon to produce the session key, `key(X-Y)`, and send it to X in such a way that it cannot be discovered by monitoring the communication between X and the `kerberos` daemon. The `kerberos` daemon must also provide an equally safe way for X to pass the secret to Y. Only if the session key remains a secret can X authenticate itself to Y, and Y to X, by proving that they know the session key.

To accomplish these goals, X first sends a message that includes its ticket-granting ticket, to the `kerberos` daemon through a call to a Kerberos library routine, `krb_mk_req`. Refer to `krb_mk_req(3krb)` for more information about `krb_mk_req`. For more detail on how the ticket-granting ticket (`tgt`) is used, refer to Figure 6-2.

To produce a secret, the `kerberos` daemon simply generates a new key – the session key between X and Y: `key(X-Y)`.

To send this secret safely to X, the `kerberos` daemon includes the session key between X and Y, `key(X-Y)`, within the message it sends to X. To clearly associate the new secret, `key(X-Y)`, with Y, the name of Y is included in the message as well. The entire message is encrypted with the key of X, `key(X)`.

Since only the `kerberos` daemon and X know the key of X, and since DES encryption prevents anyone who does not know the key used to encrypt a block of data from ever discovering the original data, then only X and the `kerberos` daemon will be able to read the message sent to X. (See Chapter 1 for a brief discussion of encryption.)

So far, this message sent to X is represented by the following notation:

```
[ key(X-Y), Y ]key(X)
```

which is the session key between X and Y, encrypted with the key of X.

2.3 The Kerberos Ticket

To provide a secure way of passing the secret to Y through X, the `kerberos` daemon produces a Kerberos **ticket**. A Kerberos ticket is a piece of data which includes the principal name of X, the address of the machine on which X is running, a lifespan value for the ticket, a timestamp, and the key, `key(X-Y)`.

The ticket is encrypted in the key of the principal Y. The ticket to authenticate to Y, `ticket-Y`, is represented as:

```
ticket-Y = [ X, address, key(X-Y), lifespan, timestamp ] key(Y)
```

Since the ticket is encrypted in the key, `key(Y)`, which is known by only the `kerberos` daemon and Y, this ticket cannot be read by any entity other than the `kerberos` daemon and Y. To associate just one Kerberos principal with the key, `key(X-Y)`, the principal name of X is included in the ticket. This association will be important when Y finally decrypts `ticket-Y`.

The address of the machine on which X is running is included in the message to prevent the ticket from being sent to Y from a machine other than that of X. The lifespan and timestamp fields are included to allow the ticket to be used repeatedly over a limited period of time. The `ticket-Y` is included in the message to X and is therefore encrypted with `key(X)`. This encryption step guarantees the confidentiality of `key(X-Y)`. So, the message sent to X by the `kerberos` daemon is represented with the notation:

```
message to X =
[ key(X-Y), Y, [ X, address, key(X-Y), lifespan, timestamp ]key(Y) ] key(X)
```

At this point, X must receive and read this message from the `kerberos` daemon. X interprets the message with the help of a Kerberos library routine, `krb_mk_req`. This routine understands the format of responses, has access to the secret key of X, `key(X)`, and can use the DES algorithm for decryption. The library call decrypts the response. The decrypted message from the `kerberos` daemon is:

```
key(X-Y), Y, [ X, address, key(X-Y), lifespan, timestamp]key(Y)
```

Since the message decrypted correctly, the sender of the message must know the key, `key(X)`. But, only the `kerberos` daemon and X know `key(X)`, so the sender of the message must be the `kerberos` daemon. Because Kerberos is trusted to give X correct information, `key(X-Y)` must be the session key to use with Y. So far, X has learned the secret, and the confidentiality of the secret, `key(X-Y)`, has been successfully protected.

2.4 The Authenticator

At this point, X must pass the secret that it wants to share with Y, `key(X-Y)`, to Y and then demonstrate to Y that it knows this secret. The first step is for X to send the ticket, `ticket-Y` to Y. Next, to prove that it knows the secret, `key(x-y)`, X produces a piece of data called the authenticator. The **authenticator** includes the principal name of X and a timestamp. It is encrypted with the session key between X

and Y, `key(X-Y)`. So, the authenticator built by X, `auth-X`, is represented by the expanded notation:

```
auth-X = [ X, timestamp ] key(X-Y)
```

The authenticator serves as proof of the knowledge of `key(X-Y)`, since, if the authenticator is decrypted with `key(X-Y)` and the decrypted authenticator contains both a valid name, *X* and logical time information (*timestamp*), then it must have been encrypted with `key(X-Y)`. Otherwise, the decrypted message would not have been a well-formatted authenticator. So, if the authenticator is understandable, the sender must have known `key(X-Y)`.

The authenticator also prevents the replay of tickets. That is, a hostile user might steal a ticket and attempt to replay or use it before it expires. But, the lifespan of the authenticator is short enough, at five minutes, to make the authenticator nonreusable. Since the authenticator is associated with only one address, *address*, to use it, a hostile user would have only five minutes to read the ticket-authenticator pair and create a false name and address for his workstation. So, since an authenticator is essentially nonreusable, and since the ticket to use a principal is useless without it, the authenticator prevents the replay of tickets. Both the authenticator and ticket to authenticate Y, `ticket-Y`, are sent through the Kerberos libraries to Y. This message is represented by the notation:

```
message to Y = [ auth-X, ticket-Y ]
```

which expands to:

```
[ [X, timestamp]key(X-Y),
[X, address, key(X-Y), lifespan, timestamp]key(Y) ]
```

The confidentiality of the secret is protected with this message as well, because `key(X-Y)` is encrypted with `key(Y)`.

2.5 Authentication of Kerberos Principals with the Ticket and Authenticator

Next, Y uses `auth-X` and `ticket-Y` to authenticate the identity of the sender of the message it received. To do so, it must first decrypt `ticket-Y`, by calling a Kerberos library routine that interprets authentication messages – `krb_rd_req`– and supply it with the key of Y and the message sent by X to Y. Refer to `krb_mk_req(3krb)` for more information about `krb_rd_req`.

The decryption of `ticket-Y` yields:

```
X, address, key(X-Y), lifespan, timestamp
```

After the Kerberos library routine decrypts the ticket, it examines the fields in the ticket. If the address in the ticket, *address*, does not match the address of the sender of the message, then the message was replayed by a hostile user and the authentication fails. If the ticket is no longer valid according to *lifespan* and *timestamp*, then the authentication attempt fails. Otherwise, since the fields in the ticket are valid, the ticket must have decrypted correctly. This implies several things. Since Y knows that only it and the `kerberos` daemon know `key(Y)`, then the ticket must have been produced by the `kerberos` daemon and, since the name of the principal, *X*, associated with `key(X-Y)` is included in the ticket, `key(X-Y)` must be the correct session key between X and Y.

Next, the Kerberos library routine decrypts `auth-X` with the key, `key(X-Y)`, that it read from the ticket.

The decryption of auth-X yields:

```
X, timestamp
```

After the Kerberos library routine decrypts the authenticator, it examines the fields within it. If the authenticator is still no longer valid according to the timestamp, then the ticket and authenticator may be a replay to Y by a hostile user. If this is so, the authentication attempt is rejected. If the principal name, *X*, within auth-X, does not match the principal name in ticket-Y, then the ticket and the authenticator do not match and the authentication request is rejected. This could indicate that a hostile user has sent a ticket from one authentication request with the authenticator of another. Otherwise, since the fields in the authenticator are valid and match the fields of the ticket, the authenticator must be the one originally sent with the ticket, ticket-Y, and the authenticator must have decrypted correctly.

Since the authenticator was correctly decrypted with key(X-Y), the sender of the message must have known key(X-Y). But, Y knows that key(X-Y) can only be known to a Kerberos principal if that principal can read a message encrypted in the key of principal X. This is the message sent by the kerberos daemon to X (in Section 2.3). So, that principal must have known key(X). But, the only principal that knows key(X) is X itself, so the sender of the message must be X.

2.6 Mutual Authentication

Mutual authentication is accomplished by Y when it sends some data back to X encrypted with the session key between X and Y, key(X-Y). This data proves to X that the message sender knows key(X-Y). Since the only way the sender can know key(X-Y) is to read it from ticket-Y, the sender must be able to read ticket-Y. But, ticket-Y is encrypted in the key, key(Y), so the sender of the message must know key(Y). Therefore, the sender of the message must be Y, because only Y knows key(Y).

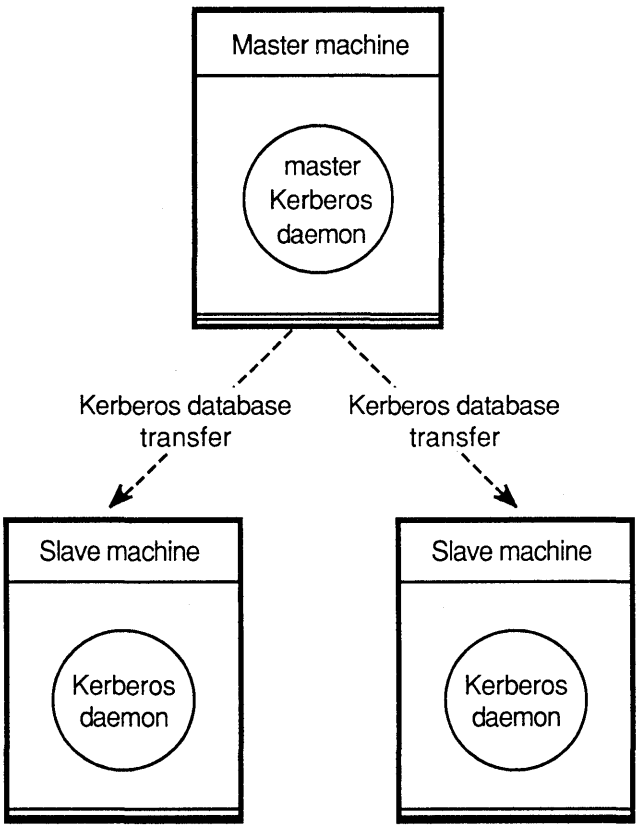
The Kerberos Daemon and Utilities 3

3.1 The Kerberos Daemon in a Network

The `kerberos` daemon resides on one **master** and on possibly several **slave** servers in a local area network (LAN). There can be only one `kerberos` daemon per machine and only one master per LAN.

The master `kerberos` daemon is associated with changes to the Kerberos database. This database is updated on the Kerberos master machine and then transferred to the Kerberos slave machines, as shown in Figure 3-3. The only difference between a Kerberos master and a Kerberos slave is that the database on the master can be modified; on a slave machine, the database is read-only.

Figure 3-1: Kerberos Database Transfer from Master to Slave



ZK-0163U-R

Kerberos slaves provide a backup function in the event of a system failure on the master machine and increase the availability of Kerberos throughout the network for better response time.

Network connections between the `kerberos` daemon and a Kerberos-authenticated application are socket-based and implemented through the TCP/IP standard Internet protocol suite. For more information about the Kerberos programming interface, refer to Chapter 6 in this guide.

3.2 The Database of Kerberos Principals

The Kerberos database maps Kerberos principals to their corresponding keys. This enables Kerberos to look up the key of a principal from its name.

The syntax for a Kerberos principal name is:

```
primaryname.instance@realm
```

Kerberos identifies a principal by primary name, instance, and a realm. The *primary name* is the name of the application. For example, the ULTRIX version of BIND is Kerberos-authenticated. The daemon that implements the BIND protocol is called `named`. The primary Kerberos name for this `named` daemon is "named."

The *instance* names the machine where the application resides. For example, the `named` that runs on machine `JUPITER.dec.com` has an instance name of `jupiter`. Note that the instance name is in lowercase and the domain name has been stripped.

Kerberos regards identical applications on different machines as different principals.

The *realm* is associated with all the principals in a single Kerberos database. It is the name of a group of machines, such as those on a LAN. For example, the `named` running on `jupiter` could be a member of the `dec.com` realm at Digital Equipment Corporation. The full Kerberos principal name for this network entity would be: `named.jupiter@dec.com`.

The Kerberos database, in `ndbm` format, resides in the directory `/var/dss/kerberos/dbase`. It consists of three files, `principal.pag`, `principal.dir`, and `principal.ok`. For more information about an `ndbm`-formatted database, see `ndbm(3)`.

Each Kerberos client has an entry in the Kerberos database and one client can authenticate itself to another.

3.2.1 Kerberos Database Utilities

There are four Kerberos database utilities for producing the intial database, editing existing principals, changing passwords, producing a readable ASCII format from the `ndbm`-formatted database (and vice versa), and for destroying the Kerberos database. These four database utilities, each of which has a reference page, are:

- kdb_init
- kdb_edit
- kdb_util
- kdb_destroy

3.2.1.1 The kdb_init Utility – The `kdb_init` Kerberos database utility creates and initializes the Kerberos database on the master. It creates the following database files:

- `/var/dss/kerberos/dbase/principal.dir`
- `/var/dss/kerberos/dbase/principal.pag`
- `/var/dss/kerberos/dbase/principal.ok` (a semaphore file)

This utility also initializes the database by adding three database entries: the master database principal, a Kerberos default principal, and the initial ticket-granting service principal, `krbtkt`. For more information about the ticket-granting ticket, refer to `kdb_init(8krb)`.

The master database principal is the entry to the database itself. You cannot use or modify the database without the master database password. The `krbtkt` ticket-granting service provides tickets that enable an application "A" to authenticate itself to "B."

To use `kdb_init`, you must supply the realm name of the Kerberos database, or you will be prompted for it. You will also be prompted for the master database key, which is necessary for any database modifications.

For more information about this utility, refer to `kdb_init(8krb)`.

3.2.1.2 The kdb_edit Utility – The `kdb_edit` Kerberos database utility is used for editing the Kerberos database. Use the `kdb_edit` command to create or change principals in the Kerberos database. When you invoke `kdb_edit`, the command prompts you for the Kerberos master key and verifies that the key is the same as the one in the database.

After the master key is verified, `kdb_edit` prompts you for the principal and instance name that you want to modify. If `kdb_edit` does not find an entry, you can create one. After `kdb_edit` finds or creates an entry, you can set the password, expiration date, and maximum ticket lifetime.

The `kdb_edit` command displays the default values for the expiration dates, maximum ticket lifetimes, and attributes in brackets. You can select any default by pressing the RETURN key. By displaying a message, the `kdb_edit` command indicates whether you have successfully created or changed an entry.

There is no default password. However, if you enter RANDOM as the password for a principal, `kdb_edit` selects a random key for the principal from the Data Encryption Standard (DES) library. For information about DES keys, refer to `des_crypt(3krb)` and to Chapter 6 of this guide.

For more information about this utility, refer to `kdb_edit(8krb)`.

3.2.1.3 The kdb_util Utility – Unlike `kdb_edit`, the `kdb_util` Kerberos database utility can change the characteristics of the entire Kerberos database in a single operation.

When used with its `dump` option, `kdb_util` converts the Kerberos database from `ndbm` format to ASCII text format. Refer to `krb_dbase(5krb)` for a description of the ASCII format of this database.

When used with its `load` option, `kdb_util` replaces the `ndbm`-formatted Kerberos database with the data supplied by a `krb_dbase` file. These two operations can be

used together to eliminate entries in the Kerberos database or to change the characteristics of several Kerberos principals at once. Although both the dump and load operations can be performed on slave and master Kerberos servers, the database should only be altered on the master server.
Use kprop and kpropd to propagate changes in the master database to the slaves.

The kdb_util utility can be used with kprop to transfer a Kerberos database from a master Kerberos server to a slave Kerberos server. If the slave_dump option of kdb_util is used, the ndbm-formatted Kerberos database on the master Kerberos server is converted to a krb_dbase-formatted file, which can be read by kprop and transferred over the network to a slave server.

When used with the new_master_key option, kdb_util changes the master key of the Kerberos database on a master or slave Kerberos server. The master key of the Kerberos database encrypts portions of the database. If the key has been discovered by a hostile user, or if enough time has passed to suspect that it might have been, then the key of the master database must be changed. Otherwise, the Kerberos authentication system will be compromised.

When used with the new_master_key option, kdb_util dumps the database into a krb_dbase-formatted file with a new master key supplied by the user. The krb_dbase file can then be used with the load option of kdb_util to load the new database into the ndbm-formatted Kerberos database. The kerberos daemon must be restarted before it can use the new database. The master keys of the Kerberos slaves should always be the same as the master key of the master kerberos daemon. See Section 4.6 in this guide for detailed instructions about using kdb_util to change the master key of the Kerberos database. Refer also to kdb_util(8krb) for more information about the master key.

3.2.1.4 The kprop, kpropd, and krb_push Utilities – The kprop and kpropd Kerberos database utilities and krb_push are used to transfer a Kerberos database from a Kerberos master server to a Kerberos slave server. A shell script, krb_push, must be created to run on the Kerberos master server. The text for the krb_push shell script is shown in Section 4.4, step 5, as part of the procedure for setting up Kerberos slave servers.

The krb_push script determines whether the database has changed since the last time it was sent over the network. If it has, then krb_push uses kdb_util with the slave_dump option, together with kprop to transfer the database over the network. The kprop command runs on the Kerberos master server and transfers the master database over the network. It takes as input a file in krb_dbase format and a list of slave machines (krb.slaves) and transmits the file to the kpropd daemons that run on the Kerberos slave machines.

The kpropd daemon waits on a well-known socket for the database transfer from kprop, places the new database in a krb_dbase-formatted file and then calls krb_util with the load option to write over the ndbm-formatted Kerberos slave database. For more information about transferring Kerberos databases, refer to kprop(8krb), kpropd(8krb), and krb_slaves(5krb).

3.2.1.5 The kdb_destroy Utility – The kdb_destroy Kerberos database utility destroys the Kerberos master database. Used only on the master database host, it removes the Kerberos master database by unlinking the
/var/dss/kerberos/dbase/principal.dir,
/var/dss/kerberos/dbase/principal.pag, and
/var/dss/kerberos/dbase/principal.ok files.

For more information about these files, refer to `kdb_destroy(8krb)`.

3.2.2 Other Kerberos Utilities

The Kerberos utilities in the following sections do not modify the Kerberos master database. The `kstash` utility hides the master database password, and the `kdestroy` utility destroys unwanted Kerberos tickets, usually at the end of a work session.

3.2.2.1 The kstash Utility – After using `kdb_init` to set up the master database, you may want to use the `kstash` utility to hide the master database password on the database host machine. This enables Kerberos administration programs to access and manipulate the master database without needing the password to be entered manually.

For more information about the `kstash` utility, refer to `kstash(8krb)`.

3.2.2.2 The kdestroy Utility – The `kdestroy` utility destroys Kerberos tickets by writing zeros to the file that contains them. If the ticket file does not exist, `kdestroy` displays an appropriate message.

After overwriting the file, `kdestroy` removes the file from the system. The utility displays a message indicating the success or failure of the operation. If `kdestroy` is unable to destroy the ticket file, the utility issues a warning by making the terminal beep.

Only the tickets in the current user ticket file are destroyed. There are separate ticket files for holding root instance and password changing tickets. If all tickets are kept in a single ticket file, they will not have to be destroyed separately. Although user-level authentication is not supported, `kdestroy` is useful for testing the requirements of setting up local user authentication. You can place the `kdestroy` command in your `.logout` file, so that your tickets are destroyed automatically at logout time.

For more information about destroying tickets, refer to `kdestroy(8krb)`.

Note

Before destroying Kerberos tickets, you may want to look at certain data about the Kerberos ticket file. The `klist` command enables you to print the name of the ticket file, the identity of the principal requesting the tickets (as listed in the ticket file), and the principal names of all the Kerberos tickets, including issue date and expiration time for each authenticator. For more information about the `klist` utility, refer to `klist(8krb)`.

3.3 Session Using Kerberos Database Utilities

Example 3-1 is an example session using the `kdb_init`, `kstash`, `kdb_util`, `kdb_edit`, `kdestroy`, and `kdb_destroy` utilities.

Example 3-1: Using Kerberos Database Utilities in a Session

The first two commands show that the Kerberos database utilities reside in /var/dss/kerberos/bin:

```
# pwd
/var/dss/kerberos/bin

# ls -gal
total 778
drwxr-xr-x  2 root     system        512 Oct 28 18:46 .
drwxr-xr-x  6 root     system        512 Oct 28 18:47 ..
-rwxr-xr-x  1 root     system     102400 Sep 25 11:14 ext_srvtab
-rwxr-xr-x  1 root     system      24576 Sep 25 11:14 kdb_destroy
-rwxr-xr-x  1 root     system     110592 Sep 25 11:14 kdb_edit
-rwxr-xr-x  1 root     system      94208 Sep 25 11:14 kdb_init
-rwxr-xr-x  1 root     system     102400 Sep 25 11:14 kdb_util
-rwxr-xr-x  1 root     system     233472 Sep 25 11:15 kprop
-rwxr-xr-x  1 root     system      77824 Sep 25 11:14 kstash
```

The kdb_init command prompts you for the database master password and creates the database files /var/dss/kerberos/dbase/principal.dir, /var/dss/kerberos/dbase/principal.pag, and /var/dss/kerberos/dbase/principal.ok as shown:

```
# kdb_init
Realm name: zk3.dec.com
You will be prompted for the database Master Password.
It is important that you NOT FORGET this password.

Enter Kerberos master key:
Verifying, please re-enter
Enter Kerberos master key:
```

Next, we must the change directory location to see if the three files have been created:

```
# cd ../dbase

# ls -gal
total 8
drwxr-xr-x  2 root     system        512 Jan 24 10:46 .
drwxr-xr-x  6 root     system        512 Oct 28 18:47 ..
-rw-------  1 root     system       4096 Jan 24 10:46 principal.dir
-rw-------  1 root     system          0 Jan 24 10:46 principal.ok
-rw-------  1 root     system       2048 Jan 24 10:46 principal.pag
```

As shown above, the Kerberos master database is stored in the three principal files. This is ndbm database format. The database is stored in the .dir and .pag files. The file, principal.dir is a directory containing a bit map, and the principal.pag file contains all the data. The principal.ok file is a semaphore file used for timestamp information.

If we want to look at the database, which is in ndbm format, we can use the dump option of the kdb_util command and look at the result with the cat command, as shown:

```
# ../bin/kdb_util dump ./dump1

# cat dump1
changepw kerberos 255 1 1 0 11a3b47 3c1ae9af 200459 191546 db_creation *
K M 255 1 1 0 6caaec01 49b87372 200001010459 199001241546 db_creation *
```

```
default * 255 1 1 0 0 0 200001010459 199001241546 db_creation *
krbtgt zk3.dec.com 255 1 1 0 47bfb34a 8cbf1455 200459 199046 db_creation *
```

As you can see, the database consists of 12 fields; an asterisk (*) denotes a blank entry for that field. The first field of the third record, `krbtgt`, is the ticket-granting ticket. For a detailed explanation of each of these fields, refer to `krb_dbase(5krb)`.

Next, `kdb_edit` is used to add a Kerberos principal, `named`, to the database:

```
# ../bin/kdb_edit
Opening database...

Enter Kerberos master key:
Verifying, please re-enter
Enter Kerberos master key:

Current Kerberos master key version is 1.

Master key entered.  BEWARE!
Previous or default values are in [brackets] ,
enter return to leave the same, or new value.

Principal name: named
Instance: verhogen

<Not found>, Create [y] ? y

Principal: named, Instance: verhogen, kdc_key_ver: 1
New Password:
Verifying, please re-enter
New Password:
Mismatch - try again

New Password:
Verifying, please re-enter
New Password:

Random password [y] ? y

Principal's new key version = 1
Expiration date (enter yyyy-mm-dd) [ 1999-12-31 ] ?
Max ticket lifetime (*5 minutes) [ 255 ] ?
Attributes [ 0 ] ?
Edit O.K.
Principal name:
```

The following `kdb_util` command, with the `dump` option, is used to create the `dump2` file to see (with the `cat` command) whether the new principal has been added to the database:

```
# ../bin/kdb_util dump ./dump2

# cat dump2
changepw kerberos 255 1 1 0 11a3b47 3c1ae9af 200459 199546 db_creation *
K M 255 1 1 0 6caaec01 49b87372 200001010459 199001241546 db_creation *
default * 255 1 1 0 0 0 200001010459 199001241546 db_creation *
krbtgt zk3.dec.com 255 1 1 0 47bfb34a 8cbf1455 200459 191546 db_creation *
named verhogen 255 1 1 0 d18378f8 3545f62e 200001010459 199001241553 * *
```

Next, the `kdestroy` and `kdb_destroy` commands are used to destroy any tickets and the Kerberos database:

```
# kdestroy
No tickets to destroy.

# ../bin/kdb_destroy
You are about to destroy the Kerberos database on this machine.
Are you sure you want to do this (y/n)? y
Database deleted at /var/dss/kerberos/dbase/principal
```

Use the `ls` command to see if the database (consisting of the `principal.dir` and `principal.pag` files) has been deleted:

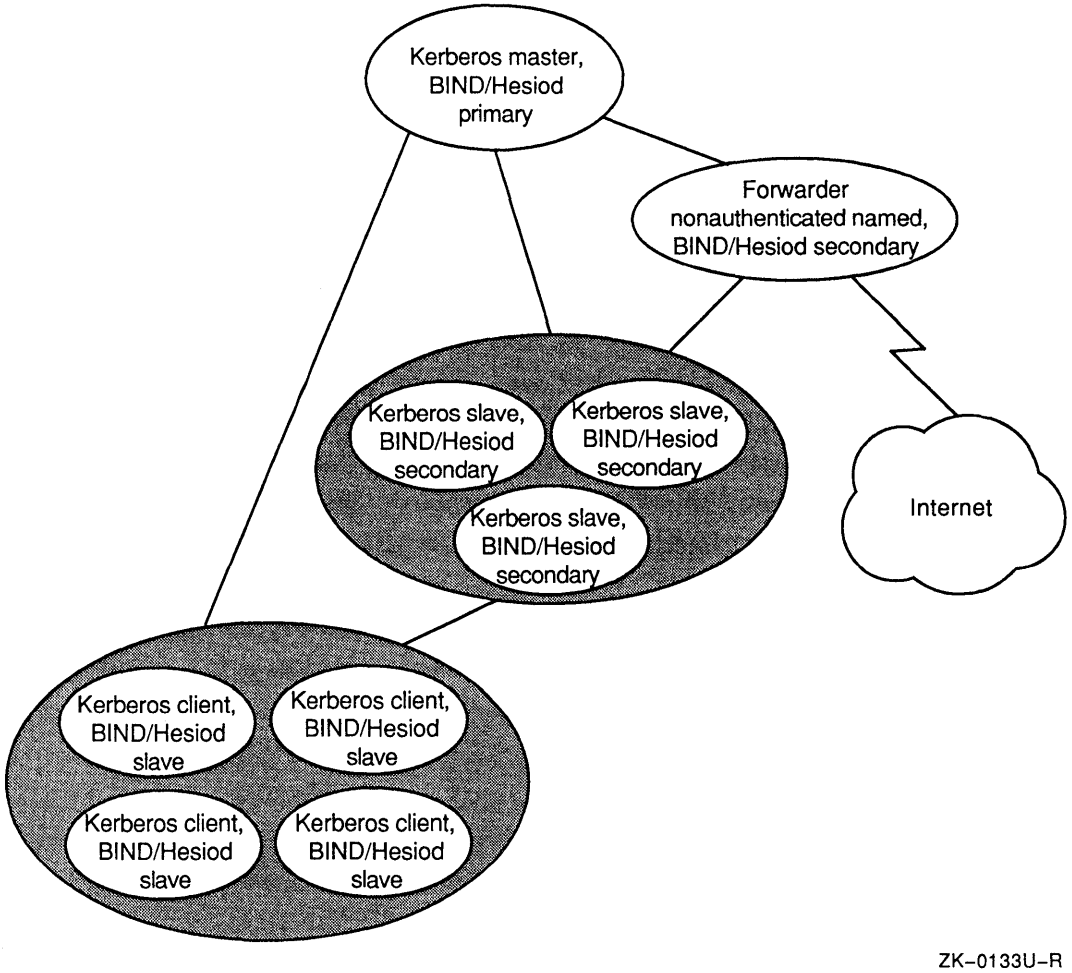
```
# ls -gal
total 4
drwxr-xr-x  2 root     system         512 Jan 24 10:57 .
drwxr-xr-x  6 root     system         512 Oct 28 18:47 ..
-rw-r--r--  1 root     system         308 Jan 24 10:51 dump1
-rw-r--r--  1 root     system         381 Jan 24 10:53 dump2
-rw-------  1 root     system           0 Jan 24 10:46 principal.ok
```

Setting Up Kerberos 4

4.1 Planning a Kerberos Authenticated Distributed Environment

In the environment described here, the BIND/Hesiod primary and the Kerberos master are the same. Figure 4-1 illustrates a network setup for distributed Kerberos authentication:

Figure 4-1: Network of Distributed Kerberos Authentication



In Figure 4-1, the **Kerberos master** is the system on which the master Kerberos database resides, and it can also run the Kerberos-authenticated `named` daemon. The Kerberos master also refers to the **BIND/Hesiod primary server** that loads the BIND/Hesiod database from a file on disk. It is highly recommended that the

Kerberos master reside on the best administered and controlled machine that is available in the LAN.

The BIND/Hesiod primary server distributes the master BIND/Hesiod database to BIND/Hesiod secondaries and also answers queries. The system that receives the propagated Kerberos databases from the Kerberos master is the **Kerberos slave server**. It can run the Kerberos-authenticated `named` daemon, and act as a backup if the Kerberos master cannot be accessed. It is also a BIND/Hesiod secondary server. The **BIND/Hesiod secondary server** receives the BIND/Hesiod database from the BIND/Hesiod primary server and answers queries. It is also a Kerberos slave server here.

In an authenticated system, the **Kerberos client** runs an application that uses Kerberos – for example, a system running the Kerberos-authenticated `named` daemon. A Kerberos client can also be a **BIND/Hesiod slave**, which answers queries and runs the `named` daemon. A system that uses the BIND/Hesiod service to resolve host names and addresses is a **BIND/Hesiod client**. At the UPGRADE or ENHANCED security levels, all BIND/Hesiod clients must convert to BIND/Hesiod slaves and run a Kerberos-authenticated `named` daemon.

The time master and time client synchronize time among network entities. The **time master** is the system that runs the Network Time Protocol (NTP) daemon `ntpd`, to synchronize time over a wide area network. The time master also runs the Berkeley `timed` as master, to distribute time to all network workstations. The **time client** is the system that runs `timed` as a client. The master can change the time of the client.

Table 4-1 summarizes the processes running in an authenticated environment.

Table 4-1: Processes in an Authenticated Environment

Kerberos	BIND/Hesiod	Processes and Files

Kerberos master	BIND/Hesiod primary	Authenticated `named` (optional)
		Kerberos master database
		BIND/Hesiod master database
		`kerberos`
		`krb.conf`
		`/etc/srvtab`
		Master `timed`
		`ntpd`

Kerberos slave server	BIND/Hesiod secondary	Propagated Kerberos database
		BIND/Hesiod secondary database
		Authenticated `named`
		`kerberos`
		`krb.conf`
		`/etc/srvtab`
		Master `timed`
		`ntpd`

Table 4-1: (continued)

Kerberos	BIND/Hesiod	Processes and Files

Kerberos client	BIND/Hesiod slave	Authenticated `named`
		`krb.conf`
		`/etc/srvtab`
		`timed`

Kerberos commands are described fully in corresponding section `8krb` reference pages. The Kerberos-authenticated `named` daemon, as well as the `kerberos` and `kprop` commands, depend on time synchronization among network entities. You must set up the network time services before attempting to configure either the UPGRADE or ENHANCED security levels. Refer to *Introduction to Networking and Distributed System Services* for more information about planning time services and configurable security levels that depend on Kerberos authentication.

4.2 Preparing to Set Up a Kerberos Authenticated named Daemon

The rest of this chapter describes how to set up a Kerberos-authenticated `named` daemon at the BSD security level. The `named` daemon guarantees that all Hesiod information distributed by the Kerberos-authenticated `named` daemon comes from an authenticated source. Chapter 5 describes the procedure for making the transition to an UPGRADE or ENHANCED security level. It is highly recommended that an experienced administrator perform all Kerberos setup procedures.

Setting up the Kerberos-authenticated `named` daemon performs these major functions:

1. Starting Kerberos on the Kerberos master
2. Starting Kerberos on the Kerberos slave servers, and propagating the Kerberos databases to them
3. Starting the `named` daemon

If you are already running `bind`, you must rerun `bindsetup` during the Kerberos setup procedure. You must be logged on as superuser to run the commands and edit the files in these procedures.

Note

Kerberos-authenticated `named`, `kerberos`, and `kprop` depend upon time synchronization among the systems on which they run. If time differs by more than five minutes between two systems running the Kerberos-authenticated `named` daemon, then the authenticated `named` may fail. To synchronize the time, start the `ntpd` daemon and the `timed` services as described in the *Guide to System and Network Setup.*

Before starting these procedures, the network must be installed. Refer to *Introduction to Networking and Distributed System Services*) for information about network installation. You should also be familiar with *Guide to the BIND/Hesiod Service* and the reference pages: `kerberos(8krb)`, `ntpd(8)`, and `timed(8)`.

4.3 Setting Up the Kerberos Master Server

Log in as superuser and ensure that the `ntpd` and `timed` daemons are running before you start to set up Kerberos on the Kerberos master. Proceed as follows (the system, `CACTUS.dec.com`, is the Kerberos master):

1. If necessary, install the Kerberos software subset, ULTKERB400:

   ```
   # setld -l ULTKERB400
   ```

   If you add this subset, you do not need to rebuild the kernel.

2. Change to the `/var/dss/kerberos/bin` directory, created during the ULTRIX operating system installation. The Kerberos utilities are located here:

   ```
   # cd /var/dss/kerberos/bin
   ```

3. Use the `kdb_init` command to create a Kerberos database. This command prompts you for a realm and a master key (password). This example shows the creation of a Kerberos database on realm ZONE:

   ```
   CACTUS.dec.com # kdb_init
   Realm name: ZONE
   You will be prompted for the database Master Password.
   It is important that you NOT FORGET this password.

   Enter Kerberos master key:
   Verifying, please re-enter
   Enter Kerberos master key:
   ```

   The `kdb_init` command creates the files: `/var/dss/kerberos/dbase/principal.dir`, `/var/dss/kerberos/dbase/principal.pag`, and `/var/dss/kerberos/dbase/principal.ok`. These files must be saved and protected if the machine is reinstalled, and backups should be made also. Choose a master key according to the recommendations of *Security Guide for Users and Programmers.*

4. Use the `kstash` command to store the master key. The `kstash` command stores the master key in a hidden master key file. If you store the master key in this file, programs that usually prompt for the master key can find the key in this file. You do not have to enter a password from the keyboard:

   ```
   # kstash
   ```

5. Create a `/etc/krb.conf` configuration file to describe the Kerberos realm and the Kerberos key distribution center (KDC) for each realm. A line in a configuration file has two parts, the realm and the hostname of the Kerberos master.

   For more information about the configuration file, refer to `krb.conf(5krb)`. The `krb.conf` file must be saved and protected if the machine is reinstalled. This example shows the lines in a `krb.conf` file for `CACTUS.dec.com`:

   ```
   CACTUS.dec.com # cat /etc/krb.conf
   ZONE
   ZONE        CACTUS.dec.com
   ```

6. Start `kerberos`. This example shows the command and the output for a network with ZONE as the realm:

   ```
   CACTUS.dec.com # kerberos &
   Kerberos server starting
   ```

```
    Sleep forever on error
    Log file is /var/dss/kerberos/log/kerberos.log
Current Kerberos master key version is 1.

Master key entered.  BEWARE!

Current Kerberos master key version is 1.
Local realm:  ZONE
CACTUS.dec.com #
```

The command has no options or arguments.

7. Place an entry in `/etc/rc.local` to automatically start `kerberos`. Place the entry after the `syslog` entry, as shown in this example:

```
echo -n 'local daemons:'  > /dev/console
[ -f /etc/syslog ] && {
    /etc/syslog & echo -n ' syslog'   > /dev/console
}
[ -f /usr/etc/kerberos ] && {
        /usr/etc/kerberos & echo ' kerberos'   > /dev/console
}
```

4.4 Setting Up Kerberos Slave Servers

This section describes the procedure for running `kerberos` on systems other than the Kerberos master. The only difference between the Kerberos slave servers and a Kerberos master is that the master periodically updates the database on each slave server.

Note

It is very highly recommended that Kerberos slave servers are run in the environment. If every machine that is a Kerberos master or slave is inoperative, then the utilities that use Kerberos (for example, `named`), will not work. Running slave servers makes such an occurrence much more unlikely. In addition, slave servers increase the number of queries that the Kerberos system can answer. The number of slave servers required is proportional to the number of Kerberos principals in the network.

This procedure uses the following script, utility, daemon, and log file:

- `krb_push` – a shell script that runs periodically on the Kerberos master. This script determines whether the database has changed since the last time the master database was distributed to the Kerberos slave servers. If it has, the script runs `kprop` to propagate the new databases to the Kerberos slave servers.
- `kprop` – a process running on the Kerberos master; it propagates the databases to the Kerberos slave servers.
- `kpropd` – the Kerberos propagation daemon. This daemon runs on the Kerberos slave servers, receives a new copy of the master database from `kprop`, and updates the database on the Kerberos slave servers.
- `kpropd.log` – a file that records the propagation of database files by the `kprop` daemon to the Kerberos slave.

The procedure includes the steps for setting up the Kerberos master to propagate the master database and the steps for setting up the slave servers for Kerberos and for receiving the database.

1. *On the Kerberos master*, from the `/var/dss/kerberos/bin` directory, use `kdb_edit` to create a principal entry for `kprop` for each system (instance) that is to run `kerberos`: the Kerberos master and any Kerberos slave servers. In this example, `barrel` and `desert`, the Kerberos slaves, and `CACTUS.dec.com`, the Kerberos master, must have `kprop` principal entries. The slave servers can also run the Kerberos-authenticated `named` daemon, but they are not required to. In the following sample session, `kdb_edit` creates principals in the Kerberos database. You need to enter the password of the service for each entry. If you type RANDOM as the password for a principal, `kdb_edit` selects a random Data Encryption Standard (DES) key for the principal.

   Note

   The instance name for `CACTUS.dec.com` is `cactus`. Refer to Section 3.2 for a discussion of Kerberos principal names.

   The `kdb_edit` command prompts you for the principal name and password:

```
CACTUS.dec.com # cd /var/dss/kerberos/bin
CACTUS.dec.com # kdb_edit
Opening database...

Enter Kerberos master key:
Verifying, please re-enter
Enter Kerberos master key:

Current Kerberos master key version is 1.

Master key entered.  BEWARE!
Previous or default values are in [brackets] ,
enter return to leave the same, or new value.
Principal name: kprop
Instance: cactus

<Not found>, Create [y] ? y

Principal: kprop, Instance: cactus, kdc_key_ver: 1
New Password:
Verifying, please re-enter
New Password:

Random password [y] ? y

Principal's new key version = 1
Expiration date (enter yyyy-mm-dd) [ 1999-12-31 ] ?
Max ticket lifetime (*5 minutes) [ 255 ] ?
Attributes [ 0 ] ?
Edit O.K

Principal name: kprop
Instance: desert
.
.
.
```

   Repeat for each Kerberos slave server.

2. *On the Kerberos master*, use the `ext_srvtab` command to create a `new-srvtab` file for the Kerberos master and each Kerberos slave. This step adds `kprop` to the `srvtab` file for `CACTUS.dec.com`. Each time you add a principal, you must create a new `srvtab`. If you run `ext_srvtab` with the `-n` option, `ext_srvtab` fetches the master key from the master key file. This example shows how `ext_srvtab` creates a `cactus-new-srvtab` file for `CACTUS.dec.com`:

```
CACTUS.dec.com # ext_srvtab cactus

Enter Kerberos master key:
Verifying, please re-enter
Enter Kerberos master key:

Current Kerberos master key version is n.

Master key entered.  BEWARE!
Generating 'cactus-new-srvtab'...
CACTUS.dec.com # ext_srvtab barrel
.
.
.
```

Repeat for each Kerberos slave.

3. *On the Kerberos master*, copy each `new-srvtab` file to `/etc/srvtab` on the host for which you created the file. You must ensure that the slave host names are in the `.rhosts` file on the master before you can use `rcp` to copy the files. The `srvtab` file should be owned by root and set readable and writable only by root. It must be saved and protected if the machine is reinstalled. Use `rcp` as follows:

```
CACTUS.dec.com # cp cactus-new-srvtab /etc/srvtab
CACTUS.dec.com # rcp barrel-new-srvtab barrel:/etc/srvtab
CACTUS.dec.com # rcp desert-new-srvtab desert:/etc/srvtab
```

4. *On the Kerberos master*, create the `/etc/krb.slaves` file, which lists the systems to which `kprop` distributes the Kerberos master database.

   On a network where `CACTUS.dec.com` is the Kerberos master and `barrel` and `desert` are Kerberos slave servers, the `/etc/krb.slaves` file should look like this:

```
barrel
desert
```

5. *On the Kerberos master*, create the `krb_push` script (with permission level set to 755) to check for changes to the database and to run `kprop`, which propagates changes to the Kerberos slave servers. The `krb_push` script is shown by using the `cat` command in the appropriate directory:

```
CACTUS.dec.com # cd /var/dss/kerberos/dbase
CACTUS.dec.com # cat krb_push
KRB_DBASE=/var/dss/kerberos/dbase
KRB_LOG=/var/dss/kerberos/log
KRB_BIN=/var/dss/kerberos/bin

if test -f $KRB_DBASE/principal.dir
then
   if test -f $KRB_DBASE/dbase
   then
       find $KRB_DBASE/principal.dir -newer $KRB_DBASE/dbase -exec \
            $KRB_BIN/kdb_util slave_dump $KRB_DBASE/dbase \; -exec \
```

```
            $KRB_BIN/kprop $KRB_DBASE/dbase /etc/krb.slaves \
                2>&1 >> $KRB_LOG/kprop.log \;
    else
            $KRB_BIN/kdb_util slave_dump $KRB_DBASE/dbase
            $KRB_BIN/kprop $KRB_DBASE/dbase /etc/krb.slaves \
                2>&1 >> $KRB_LOG/kprop.log
    fi
fi

CACTUS.dec.com #
```

6. *On each Kerberos slave server*, create a `krb.conf` configuration file that lists the Kerberos master. A line in a configuration file has two parts: the name of the realm, and the name of the Kerberos master. The following example shows the `krb.conf` file for a network in which `CACTUS.dec.com` is the master Kerberos server:

```
ZONE
ZONE        CACTUS.dec.com
```

7. *On each Kerberos slave server*, create the Kerberos database files. If the files do not exist when you start the `kpropd` daemon, the process fails. You can create the files with the `touch` command as shown:

```
barrel # cd /var/dss/kerberos/dbase
barrel # touch principal.pag
barrel # touch principal.dir
barrel # touch principal.ok

desert # cd /var/dss/kerberos/dbase
desert # touch principal.dir
desert # touch principal.pag
desert # touch principal.ok
```

8. *On each Kerberos slave server*, start the `kpropd` daemon. Before starting the daemon, you must set the `PATH` environment variable to include `/var/dss/kerberos/bin`. The following commands add the `PATH` variable and run `kpropd`:

```
barrel # set path=($path /var/dss/kerberos/bin)
barrel # /usr/etc/kpropd /var/dss/kerberos/dbase/dbase &

desert # set path=($path /var/dss/kerberos/bin)
desert # /usr/etc/kpropd /var/dss/kerberos/dbase/dbase &
```

9. *On each Kerberos slave server*, add entries to `/etc/rc.local` for `kerberos` and `kpropd`. Place the entries after the `syslog` entry. The following shows the lines to add to `/etc/rc.local` to start `kerberos`:

```
echo -n 'local daemons:' /dev/console
[ -f /etc/syslog ] && {
    /etc/syslog & echo -n ' syslog' >/dev/console
}
[ -f /usr/etc/kerberos ] && {
        /usr/etc/kerberos & echo' kerberos' >/dev/console
}
```

The following shows the lines to add to `/etc/rc.local` to start `kprop`:

```
# %KPROPDSTART%

PATH=/bin:/usr/ucb:/usr/bin:/var/dss/kerberos/bin
```

```
export PATH

echo -n 'kprop daemon:' >/dev/console

[ -f /usr/etc/kpropd ] && {

    /usr/etc/kpropd /var/dss/kerberos/dbase/dbase & echo ' kpropd' \
       >/dev/console

}

echo '.'

# %KPROPDEND%
```

Note

The `kpropd` command line above is broken with a backslash (\) because of the short line length in this manual. In `/etc/rc.local`, this command is on a single line.

10. *On the Kerberos master*, update the `/var/dss/kerberos/dbase/principal.dir` file by using the `touch` command. This makes `principal.dir` newer than `dbase`, so that `kprop` propagates the database files to the Kerberos slave. For example:

```
CACTUS.dec.com # touch /var/dss/kerberos/dbase/principal.dir
CACTUS.dec.com #
```

11. *On the Kerberos master*, run `krb_push` to start `kprop`. The following example shows the command in a network in which `CACTUS.dec.com` is the Kerberos master:

```
CACTUS.dec.com # krb_push
```

12. *On each Kerberos slave server*, verify that the `kpropd` daemon successfully received the database by looking at the `/var/dss/kerberos/log/kpropd.log` file. The following shows an entry into the log file for a successful transfer:

```
barrel # more /var/dss/kerberos/log/kpropd.log
***** kpropd started *****
28-Sep-89 11:13:38 Established socket
28-Sep-89 11:16:03 Connection from CACTUS.dec.com, 130.180.4.13
28-Sep-89 11:16:04 kpropd: Connection from kprop.cactus@ZONE
28-Sep-89 11:16:04 File received.
28-Sep-89 11:16:04 Temp file renamed to /var/dss/kerberos/dbase/dbase
28-Sep-89 11:16:05
.
.
.
desert # more /var/dss/kerberos/log/kpropd.log
.
.
.
```

13. *On each Kerberos slave server*, run `kstash` to store the master Kerberos key:

```
# kstash
```

14. *On each Kerberos slave server*, start `kerberos`. The following shows the commands for a network in which `barrel` and `desert` are Kerberos slave servers:

```
barrel # /usr/etc/kerberos &

desert # /usr/etc/kerberos &
```

15. *On the Kerberos master*, place an entry in `/etc/crontab` to run the `krb_push` shell script every five minutes to determine whether the database has changed. If it has, the script runs `kprop` to propagate the database to the Kerberos slave servers.

    The following shows such an entry:

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * * /var/dss/kerberos/dbase/krb_push
```

16. After you verify that the Kerberos database is propagating successfully to the Kerberos slave servers, add each Kerberos slave server to the `krb.conf` file on any system running the Kerberos-authenticated `named` daemon on the Kerberos master, and on Kerberos slaves. If the first system does not respond, Kerberos tries the other servers listed. The entry for a network in which `CACTUS.dec.com` is the master server, and `barrel` and `desert` are Kerberos slave servers is as follows:

```
CACTUS.dec.com # cat /etc/krb.conf
ZONE
ZONE CACTUS.dec.com
ZONE barrel
ZONE desert
```

4.5 Starting the Kerberos-Authenticated named Daemon

This section describes how to set up and start a Kerberos-authenticated `named` daemon.

The examples for each step show a network consisting of `CACTUS.dec.com`, as Kerberos master, and `desert` and `barrel` as two other systems that will be running the Kerberos-authenticated `named` daemon. Although these show the Kerberos master running the `named` daemon, it is not a requirement.

1. *On the Kerberos master*, change to the `/var/dss/kerberos/bin` directory:

```
# cd /var/dss/kerberos/bin
```

2. *On the Kerberos master*, use `kdb_edit` to create principal entries in the Kerberos database for `named` and `hesiod` for each system that will run the Kerberos-authenticated `named` daemon (including the Kerberos master if it is going to run `named`)

   When you invoke `kdb_edit`, the command prompts you for the master key (password). If you invoke `kdb_edit` with the `-n` option, `kdb_edit` fetches the key from the master key file. (You must have first created the master key file with `kstash`.

   The `kdb_edit` command then prompts you for the principal name. If `kdb_edit` does not find an entry in the Kerberos database file, it prompts you

to create one. Create a principle entry for hesiod and named. For the instance, use the name of a host on which the Kerberos-authenticated named daemon will run. If you are going to run the named daemon on more than one system, then, for each system, you must use the procedure shown in the example below.

You need to enter the password of the service for each entry. If you type RANDOM as the password for a principal, kdb_edit selects a random DES key for the principal. You also need to enter values for expiration dates, maximum ticket lifetimes, and attributes. The kdb_edit command displays default values in brackets.

This is an example of a kdb_edit session that adds the hesiod and named principals for the machine, CACTUS.dec.com. In the example, the default settings are selected for expiration dates, maximum ticket lifetimes, and attributes. Repeat the procedure for each instance (system) on which you are going to run the named daemon. Press the RETURN key to return to the prompt after entering the last principal and instance:

Note

The instance name of CACTUS.dec.com is cactus. See Section 3.2 for a discussion of Kerberos principal names.

```
CACTUS.dec.com # kdb_edit
Opening database...

Enter Kerberos master key:
Verifying, please re-enter
Enter Kerberos master key:

Current Kerberos master key version is 1.

Master key entered.  BEWARE!
Previous or default values are in [brackets] ,
enter return to leave the same, or new value.

Principal name: named
Instance: cactus

<Not found>, Create [y] ? y

Principal: named, Instance: cactus, kdc_key_ver: 1
New Password:
Verifying, please re-enter
New Password:

Random password [y] ? y

Principal's new key version = 1
Expiration date (enter yyyy-mm-dd) [ 1999-12-31 ] ?
Max ticket lifetime (*5 minutes) [ 255 ] ?
Attributes [ 0 ] ?
Edit O.K.
Principal name: hesiod
Instance: cactus

<Not found>, Create [y] ? y

Principal: hesiod, Instance: cactus, kdc_key_ver: 1
New Password:
```

```
Verifying, please re-enter
New Password:

Random password [y] ? y

Principal's new key version = 1
Expiration date (enter yyyy-mm-dd) [ 1999-12-31 ] ?
Max ticket lifetime (*5 minutes) [ 255 ] ?
Attributes [ 0 ] ?
Edit O.K.
Principal name: named
Instance: desert
.
.
.
$
```

Repeat for each system that will run the Kerberos-authenticated named daemon.

3. *On the Kerberos master*, use the ext_srvtab command to create a new-srvtab file for each system on which you are going to run the Kerberos-authenticated named daemon. If you are going to run named on the Kerberos master, you must create a new-srvtab file for the Kerberos master. If you run ext_srvtab with the -n option, ext_srvtab fetches the master key from the master key file. The following shows the use of ext_srvtab to create a barrel-new-srvtab file for barrel:

```
CACTUS.dec.com # ext_srvtab barrel

Enter Kerberos master key:
Verifying, please re-enter
Enter Kerberos master key:

Current Kerberos master key version is n.

Master key entered.  BEWARE!
Generating 'barrel-new-srvtab'...
CACTUS.dec.com # ext_srvtab desert
.
.
.
```

4. *On the Kerberos master*, copy each new-srvtab file to /etc/srvtab on the host for which you created the file. For example:

```
CACTUS.dec.com # cp cactus-new-srvtab /etc/srvtab
CACTUS.dec.com # rcp barrel-new-srvtab barrel:/etc/srvtab
CACTUS.dec.com # rcp desert-new-srvtab desert:/etc/srvtab
```

5. *On the Kerberos master*, copy the krb.conf file to /etc on each host that is going to run the Kerberos-authenticated named daemon. These are the systems for which you created new-srvtab files. For example:

```
CACTUS.dec.com # rcp /etc/krb.conf barrel:/etc
CACTUS.dec.com # rcp /etc/krb.conf desert:/etc
```

6. *On each system that is going to run Kerberos-authenticated* named, (the systems for which you created new-srvtab files) run bindsetup from the /var/dss/namedb directory. In the example network used in these

procedures, you must run `bindsetup` on `CACTUS.dec.com`, `barrel`, and `desert`.

Respond to the prompts as you did when you set up BIND/Hesiod, until prompted for running a Kerberos-authenticated `named` daemon. Respond to the prompt with **yes**, as shown:

```
Do you want to run Kerberos Authenticated named (y/n) [n] ? y
```

The `bindsetup` script terminates the unauthenticated `named` daemon and starts a Kerberos-authenticated daemon.

Note

Kerberos-authenticated `named` and `named` cannot both serve Hesiod information in the same network. They can, however, exchange Internet host information.

The script replaces the line in `/etc/rc.local` that starts the `named` daemon with the line shown here:

```
# %BINDSTART% - BIND daemon
[ -f /usr/etc/named ] && {
     /usr/etc/named -s -a kerberos one -b /var/dss/namedb/named.boot;\
        echo -n ' named ' > /dev/console
}
```

Note

The example breaks the `named` command line with a backslash (\) because of the short line length in this manual. In `/etc/rc.local`, the command appears on a single line.

4.6 Changing the Master Key of the Kerberos Database

The Kerberos database on both slave and master servers contains fields that are encrypted with the master key of the database. This key, like any other DES key, is vulnerable over time to attacks by a hostile user. If the master key is not changed occasionally, and a user discovers the value of the master key, the entire Kerberos service may be compromised. The following procedure describes how to change the master key of the Kerberos database in an environment in which `barrel` and `desert` are Kerberos slave servers and `CACTUS.dec.com` is the Kerberos master server:

1. *On the Kerberos master*, disable the ability of `krb_push` to transfer the Kerberos master database over the network. To do this, (the `ed` editor is used here) eliminate the `krb_push` entry from `/etc/crontab` (discussed in Section 4.4 step 15):

```
CACTUS.dec.com # cd /etc
CACTUS.dec.com # cp crontab crontab.orig
CACTUS.dec.com # ed crontab
1278
/krb_push
0,5,10,15,20,25,30,35,40,45,50,55 * * * * /var/dss/kerberos/dbase/krb_push
d
w
1203
q
CACTUS.dec.com #
```

2. *On the Kerberos master*, use `kdb_util` to produce a file in `kdb_dbase` format that uses the new master key of the Kerberos database to encrypt portions of the database.

```
CACTUS.dec.com # cd /var/dss/kerberos/dbase
CACTUS.dec.com # ../bin/kdb_util new_master_key ./new_key_dbase


Enter the CURRENT master key.
Enter Kerberos master key:< old kerberos key entered here >
Verifying, please re-enter
Enter Kerberos master key:< old kerberos key entered here >

Current Kerberos master key version is 2.

Master key entered.  BEWARE!


Now enter the NEW master key.  Do not forget it!!
Enter Kerberos master key:< new Kerberos key entered here >
Verifying, please re-enter
Enter Kerberos master key:< new Kerberos key entered here >

Don't forget to do a 'kdb_util load ./new_key_dbase' to reload
the database!
CACTUS.dec.com #
```

3. *On the Kerberos master*, use `kstash` to replace the old master key of the Kerberos database with the new key:

```
CACTUS.dec.com # cd /var/dss/kerberos/bin
CACTUS.dec.com # ./kstash

Enter Kerberos master key:< new Kerberos key entered here >
Verifying, please re-enter
Enter Kerberos master key:< new Kerberos key entered here >

Current Kerberos master key version is 2.

Master key entered.  BEWARE!
CACTUS.dec.com #
```

4. *On the Kerberos master*, stop the execution of the `kerberos` daemon. If any machine in the network depends completely on the master `kerberos` daemon for Kerberos services, this step could be risky. To avoid this risk, make sure at least one Kerberos server is running in the network and that every machine in the network that uses Kerberos has the name of at least one functioning Kerberos server listed in its `krb.conf` file:

```
CACTUS.dec.com # ps -gax | grep kerberos
22853 ?  I     0:29 /usr/etc/kerberos
9798 p5 S     0:00 grep kerberos
CACTUS.dec.com # kill -9 22853
```

5. *On the Kerberos master*, use `kdb_util` to transfer the database that uses the new master key from the `new_key_dbase` file, to the ndbm-formatted Kerberos database:

```
CACTUS.dec.com # cd /var/dss/kerberos/dbase
CACTUS.dec.com # ../bin/kdb_util load ./new_key_dbase
```

6. *On the Kerberos master*, restart the `kerberos` master daemon:

```
CACTUS.dec.com # /usr/etc/kerberos
CACTUS.dec.com #
```

7. *On the Kerberos master*, edit the `/etc/krb.slaves` file so that only the first Kerberos slave appears in the file:

```
CACTUS.dec.com # cd /etc
CACTUS.dec.com # cp /etc/krb.slaves /etc/krb.slaves.orig
CACTUS.dec.com # cat > /etc/krb.slaves
barrel
^D
CACTUS.dec.com #
```

8. *On the first Kerberos slave*, use `kstash` to replace the old master key of the Kerberos database with the new key:

```
barrel # cd /var/dss/kerberos/bin
barrel # ./kstash

Enter Kerberos master key:< new kerberos key entered here >
Verifying, please re-enter
Enter Kerberos master key:< new kerberos key entered here >

Current Kerberos master key version is 2.

Master key entered.  BEWARE!
barrel #
```

9. *On the Kerberos slave*, stop the execution of the `kerberos` daemon. If any machine in the network depends completely on this slave `kerberos` daemon for Kerberos services, then this step could be risky. To avoid this risk, make sure at least one Kerberos server is running in the network and that every machine in the network that uses Kerberos has the name of at least one functioning Kerberos server listed in its `krb.conf` file:

```
barrel # ps -gax | grep kerberos
22853 ?  I     0:29 /usr/etc/kerberos
9798 p5 S     0:00 grep kerberos
barrel # kill -9 22853
```

10. *On the Kerberos master*, use `krb_push` to transfer the database encrypted with the new master key to the slave server listed in `/etc/krb.slaves`:

```
CACTUS.dec.com # cd /var/dss/kerberos/dbase
CACTUS.dec.com # touch principal.dir
CACTUS.dec.com # ./krb_push
CACTUS.dec.com #
```

11. *On the Kerberos slave*, verify that the transfer of the database was successful by examining the `/var/dss/kerberos/log/kpropd.log` file. A successful transfer looks like the following:

```
15-Feb-90 14:04:54 Connection from CACTUS.dec.com, 4.5.6.7
15-Feb-90 14:04:54 kpropd: Connection from kprop.cactus@ZONE
15-Feb-90 14:04:56 File received.
15-Feb-90 14:04:56 Temp file renamed to /var/dss/kerberos/dbase/dbase
```

12. *On the Kerberos slave*, restart the `kerberos` master daemon:

```
barrel # /usr/etc/kerberos
barrel #
```

13. *For every Kerberos slave*, repeat steps 7 through 12.
14. *On the Kerberos master*, replace the original versions of the `krb.slaves` file and the `/etc/crontab` file:

```
CACTUS.dec.com # cd /etc
CACTUS.dec.com # cp crontab.orig crontab
CACTUS.dec.com # cp krb.slaves.orig krb.slaves
```

Creating an UPGRADE or ENHANCED Distributed Environment 5

Before setting up an UPGRADE or ENHANCED distributed environment, you must first set up a distributed BSD BIND/Hesiod environment as described in *Guide to the BIND/Hesiod Services.* The BSD distributed environment verifies that the BIND/Hesiod primary and secondary servers are accessible and working correctly.

At BSD level, the `/etc/passwd` file distributes passwords in a network. In ULTRIX, the `passwd`, `login`, `su`, `dxsession`, and `named` programs are modified to use Kerberos for accessing the distributed `auth` data required by the UPGRADE or ENHANCED security levels.

Workstations and other less-used machines in the distributed (UPGRADE or ENHANCED) environment must run as BIND/Hesiod slaves. Every system in the distributed (UPGRADE or ENHANCED) environment must be running the Kerberos-authenticated `named` daemon, as described in Chapter 4. For more information, refer to *Security Guide for Administrators.*

After setting up a distributed BSD environment, you can set up an UPGRADE environment.

Note

If you skip the UPGRADE level, no user can log in. The UPGRADE level converts the current BSD style passwords into the ENHANCED `auth`-style passwords.

Users who do not upgrade their BSD passwords when the environment is at the UPGRADE level cannot log in at the ENHANCED level, unless the superuser runs `passwd` for them.

5.1 Transition from BSD to UPGRADE Security Level

Before converting from BSD to UPGRADE security level, you must perform these procedures:

1. Set up `ntpd` or `timed`, or both, on all systems in the distributed environment, as described in the *Guide to System and Network Setup.*
2. Set up and run a distributed BSD BIND/Hesiod environment, as described in *Guide to the BIND/Hesiod Services.*
3. Start the Kerberos-authenticated `named` daemon, as described in Chapter 4.

The procedures in Section 5.2 create the `auth` database and restarts a Kerberos-authenticated `named` daemon that uses the database.

The procedures in Section 5.3 perform the transition to UPGRADE level.

5.2 Preparing for the Transition to UPGRADE Level

When you have BIND/Hesiod and Kerberos-authenticated named running, you can start the transition to UPGRADE security level. The steps in the procedure assume that the same host (in this case CACTUS.dec.com) is the BIND/Hesiod primary and the Kerberos master.

The first part of the procedure creates a distributed auth file and copies it to the BIND/Hesiod master server.

1. For maximum security, perform this step while in single user mode (/, /usr, and /var must be mounted).

   On any system in the network (preferably a workstation that will not disrupt the current environment) copy /etc/passwd to a backup file. Create a distributed auth database for named by appending the /etc/passwd file that you want to distribute to the local /etc/passwd. The following commands show a distributed auth file being created on workstation mesa from a password file selected for distribution (passwd.dist).

   ```
   mesa # cp /etc/passwd passwd.bak
   mesa # cat /etc/passwd.dist >> /etc/passwd
   ```

   Delete any duplicate entries in the new /etc/passwd.

2. Run /usr/etc/sec/secsetup as shown:

   ```
   mesa # /usr/etc/sec/secsetup
   ```

   Do not select any other security options. Select UPGRADE level at the prompt. The secsetup script also creates a local auth database. For a detailed description of secsetup, refer to the *Security Guide for Administrators*.

3. Run getauth to create a distributed auth file:

   ```
   mesa # /usr/etc/sec/getauth > auth
   ```

   Remove those entries in auth for user IDs that should not be distributed. For example, remove entries for user IDs -2, 0 – 10, 25, and any others that are specific to your local environment. The first field in the auth entry is the uid field. The passwd database must have a user entry for each uid in the auth database.

4. Copy the auth database to the BIND/Hesiod primary server – in this example, CACTUS.dec.com:

   ```
   mesa # rcp auth CACTUS.dec.com:/var/dss/namedb/src/auth
   ```

   You should now have a passwd and auth database file in the named source directory on the BIND/Hesiod primary, /var/dss/namedb/src.

5. Restore the original /etc/passwd file to the workstation, as shown:

   ```
   mesa # cp passwd.bak /etc/passwd
   ```

6. When you ran secsetup to create the auth file, the workstation made the transition to UPGRADE level. You must return the workstation to BSD level before completing the transition by the entire network. To return the workstation to BSD level, edit /etc/svc.conf on the workstation and set the security level to BSD (SECLEVEL=BSD) .

7. Run bindsetup on the BIND/Hesiod primary server to set up a Kerberos-authenticated named daemon.

8. Run `bindsetup` on the BIND/Hesiod secondary servers to set up a Kerberos-authenticated `named` daemon.
9. Run `bindsetup` on the BIND/Hesiod slave servers to set up a Kerberos-authenticated `named` daemon.
10. Allow approximately four minutes for the BIND/Hesiod secondary servers to obtain the databases from the BIND/Hesiod primary. Then, run `nslookup` to determine if the databases have been distributed. *Guide to BIND/Hesiod Services* describes the `nslookup` command.

At this point, your environment is still running at BSD level, and both Kerberos and the Kerberos-authenticated `named` daemon are running. The next section describes the procedure for making BIND/Hesiod use Kerberos, which brings the environment to UPGRADE level.

5.3 Making the Transition to UPGRADE Level

Make the transition from BSD to UPGRADE level first on the BIND/Hesiod primary, then on the BIND/Hesiod secondaries and slaves. The `/usr/etc/sec/secsetup` command changes the security level.

1. On the Kerberos master BIND/Hesiod primary server, run `/usr/etc/sec/secsetup` to change the security level from BSD to UPGRADE.
2. Change the root password on the Kerberos master BIND/Hesiod primary with `passwd`.
3. Run `svcsetup` or edit the `/etc/svc.conf` file to set the `auth` and `passwd` lookup switches to `bind` as shown:

```
auth=local,bind
passwd=local,bind
```

4. Allow approximately two minutes for the distribution delay and then try to log into the server system as a normal user. (In some cases, the delay may be as long as five minutes.) Next, change your password with `passwd`.
5. Use `nslookup` to examine the new `auth` password and old password entry. If the entries are asterisks, then the Kerberos BIND/Hesiod primary server is running at UPGRADE level. To check the password entry for user `lizard`, use the commands that select Hesiod class data:

```
cactus # nslookup
> set type=txt
> set cl=hs
> lizard.passwd
lizard:*:.....
```

6. Repeat steps 1 to 5 on each BIND/Hesiod secondary Kerberos slave server in sequence.
7. Repeat steps 1 to 5 on each BIND/Hesiod slave Kerberos client.

Your environment is now running at UPGRADE level. Users who do not change their passwords at this time cannot log in when the environment is set to ENHANCED. As root on the BIND/Hesiod primary, you can use `passwd` to reset the password of users who cannot log in.

You can check the progress of the upgrade by examining the `/var/dss/namedb/src/passwd` file, which has an asterisk (*) in place of an encryption string for all upgraded passwords.

After running in UPGRADE level long enough to collect and change all user passwords, you can make the transition to ENHANCED level.

5.4 Making the Transition to ENHANCED Level

If all user passwords have been collected and changed, you can make the transition to ENHANCED level. This section describes the steps in the transition.

1. Starting with the primary server, run `/usr/etc/sec/secsetup` to change the security level from UPGRADE to ENHANCED.

   Note

   For maximum security, the Kerberos master and BIND/Hesiod primary server should limit login access to only those people in the local `passwd` and `auth` files. To prevent distributed lookups on the master, in `/etc/svc.conf` on the Kerberos master BIND/Hesiod primary, set the `auth` and `passwd` variables to local as shown:

   ```
   auth=local
   passwd=local
   ```

2. Run `/usr/etc/sec/secsetup` on each Kerberos slave, BIND/Hesiod secondary server.
3. Run `/usr/etc/sec/secsetup` on each BIND/Hesiod slave and on all other systems running the Kerberos-authenticated `named` daemon.

Kerberos Programming Interface 6

This chapter describes the network programming connections of the `kerberos` daemon to a Kerberos-authenticated application. The Kerberos libraries provide an authentication interface for applications that are Kerberos principals. They enable the application to contact the `kerberos` daemon for formatting and sending authentication information to other Kerberos principals.

6.1 Kerberos Libraries

The Kerberos software components are:

- Kerberos applications library (`libkrb`)
- Encryption library (`libdes`)
- Access control list library (`libacl`)
- Kerberos communications library (`libknet`)
- Kerberos database library (`libkdb`)

The Kerberos programming interface consists chiefly of four of these libraries: `libkrb`, `libdes`, `libknet`, and `libkdb`. These libraries are used by all Kerberos-authenticated utilities. They make up the authentication and communication interface between clients and the `kerberos` daemon, and between two principals.

The applications library, `libkrb`, provides the programming interface for authentication. This library contains routines for creating or reading authentication requests, and for creating messages that cannot be tampered with by a hostile user. This library can also send messages across TCP/IP connections to the `kerberos` daemon as well as to another Kerberos principal. The applications library contains references to routines in `libdes`, `libknet`, and `libkdb`, so these libraries must also be linked with any program if the `libkrb` routines are used. Refer to the following reference pages for detailed information:

- `kerberos(3krb)`
- `krb_get_lrealm(3krb)`
- `krb_sendauth(3krb)`
- `krb_sendmutual(3krb)`
- `krb_set_tkt_string(3krb)`
- `krb_svc_init(3krb)`

The encryption library (`libdes`) provides routines that support the Data Encryption Standard. The DES library routines do not actually perform encryption, but are support routines for it. Refer to `des_crypt(3krb)` for detailed information.

The communications library (`libknet`) provides, for `libkrb`, support routines for communicating with the `kerberos` daemon. There are no routines in `libknet` available to be used by an applications programmer.

The Kerberos database library (`libkdb`) provides support routines, for `libkrb`, that access the Kerberos database. Only the routines in `libkrb` use the `libkdb` routines.

The access control list (ACL) library, `libacl`, has routines that can be used by an application to perform authorization. **Authorization** is the process of deciding whether a principal is allowed to perform a particular action. Often, authorization checks are performed immediately after a principal is authenticated, to distinguish between the privileges accorded to two different principals. The `libacl` library uses access control lists to store lists of authorized principals. Refer to `acl_check(3krb)` for more information about access control lists.

6.2 Kerberos Programming Examples

The server is designed to store, modify, and access an access control list while the client is designed to ask questions about the list. The access control list used by the server is created and serviced by the routines of the `libacl` library. Refer to `acl_check(3krb)` for more information about `libacl`. All data passed between the client and server is authenticated by Kerberos.

The client uses a TCP socket to communicate with the server, but the code that the client and server use for communication does not need to be described here. Only those sections of the code that describe how the client and server use the Kerberos library calls are discussed.

The examples consist of two client-server pairs. In both, the client and server programs do the same task. However, the low-level pair use the Kerberos routines that create packets of authentication information that can be placed inside the ''on-the-wire'' protocol of the application.

The high-level pair use the Kerberos routines for applications whose ''on-the-wire'' protocol cannot be altered. The Kerberos routines in the high-level example encapsulate the ''on-the-wire'' protocol of the application inside Kerberos-formatted packets.

The high-level client differs from the low-level client in that the low-level client runs as a daemon and is a Kerberos principal. The high-level client is not a Kerberos principal; it takes the identity of the user, principal, that runs the utility as its own and authenticates itself as the user.

6.2.1 Organization of Example Files

The next two sections of this chapter explain the low-level and high-level examples by referring to 12 files, which are grouped together after the explanations. This organization is necessary to facilitate the explanations of high-level and low-level client-server pair examples, which share much of the same programming code.

These shared code files are organized as follows:

- Common header file – `all.h`
- Code common to the high-level and low-level server and client – `comm.c`
- Code specific to the high-level and low-level client-server pairs:

  - `low_level.c`
  - `high_level.c`

- High-level and low-level server code:
  - `server.h`
  - `l_server.c`
  - `h_server.c`
  - `server.c`

- High-level and low-level client code:
  - `client.h`
  - `l_client.c`
  - `h_client.c`
  - `client.c`

You will need to refer to this C code and its embedded comments while reading the next two sections, "Low-Level Example Explanation" and "High-Level Example Explanation." Each explanation often points to a specific section of C code by using a numbered box such as point 1, whose counterpart 1 is in Example 6-3.

It is necessary that the high-level and low-level examples share code extensively, to more easily explain *both* of the "on-the-wire" protocols:

- The placing of packets of authentication information inside the "on-the-wire" protocol, explained in Section 6.2.2 (low-level example)
- Applications whose "on-the-wire" protocol cannot be altered, explained in Section 6.2.3 (high-level example)

This makes it necessary to explain different sections of code *nonsequentially* among the 12 example files (including the code not explicitly labeled as a common header file). To make this task easy, all numbered boxes in the *entire* block of files continue as a sequence. For example, the last numbered box in Example 6-3 is point 14. Therefore, the first numbered box in Example 6-4 is point 15.

Whenever the explanation shifts to another file of code, the new file is denoted by name and example number. However, it is always easier to simply refer to a numbered point in the block of files. The entire block of files starts on an odd-numbered page (at Section 6.2.4), and you may find it easier to refer to them by removing them from the binder and placing them nearby.

These examples were compiled on a "one-pass-through" C compiler, whose efficiency is maximized by placing `main` at the *end* of a section of code. That is why the explanation below starts at point 63, which is `main` in the file, `server.c`.

6.2.2 Low-Level Example Explanation

In `server.c` (Example 6-8), the low-level server begins execution at point 63.

At point 64, the server sets up the default names of the log file, `log_file`, the access control list (ACL) file, `acl_file`, the primary name of the server, `p_state.primary_name`, the ticket file, `p_state.tkt_file`, and the service table (srvtab) file, `p_state.srvtab_file`. The log file is used to print error

messages, the ACL file stores the access control list that the server will manipulate, the ticket file stores any tickets that the server receives from Kerberos, and the srvtab file stores the primary key of the server.

At point 65, the user of the server can change any of the default settings.

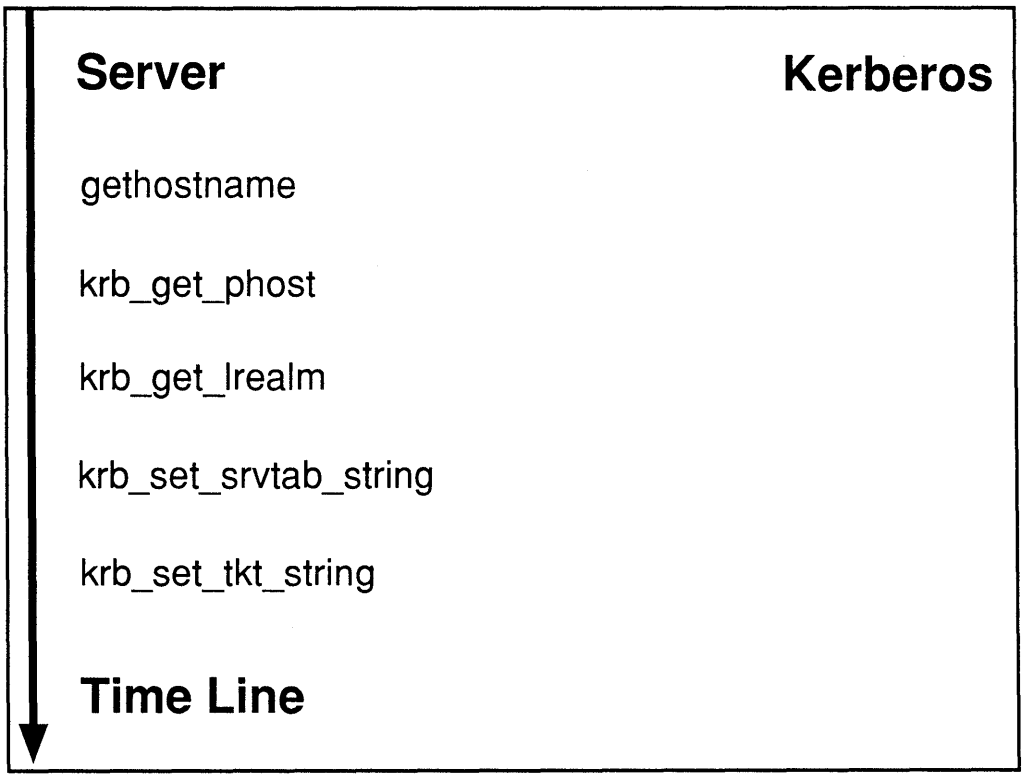
At point 66, the server calls `init_files`, which is located at point 62, to open the log file and make sure that the ACL file exists.

At point 67, the server calls `init_comm`, which is located at point 61, to initialize the TCP socket it uses to communicate with the client. Next, `init_comm` stores the socket attributes in the `prin_state` structure, `p_state`.

In `server.h` (Example 6-5), the `prin_state` structure is defined at point 20. A variable of `main`, `p_state` is a `prin_state` structure used throughout the client and server for storing some of the attributes associated with the server. The `p_state` variable stores the server primary name, `primary_name`, instance name, `instance`, and realm name, `realm`. It stores the name of the `srvtab_file` and the `tkt_file` that the server uses, and in `stream_sock`, and `stream_addr`, it stores information about the socket that the server uses to communicate with the client.

In `server.c` (Example 6-8), the low-level server continues executing by calling `init_krb`, at point 68. The `init_krb` call initializes the Kerberos libraries for the server. See Figure 6-1, which illustrates the server Kerberos initialization by showing each subroutine call on a time line, and how that call interacts with Kerberos (for the server initialization, there is no interaction).

Figure 6-1: Server Initialization



In `init_krb` (point 35), the server calls `gethostname` (point 36), and `krb_get_phost` (point 37), to produce the instance name of the server from the hostname of the local host. Refer to `krb_get_lrealm(3krb)` for more information about `krb_get_phost`, and to `gethostname(2)` for more information about `gethostname`.

The instance name will be a lowercase version of the hostname with the BIND domain name stripped. The call to `krb_get_lrealm` (point 38), reads the name of the realm of which the local host is a member, and stores it in the `p_state` structure. Refer to `krb_get_lrealm(3krb)` for more information about the realm name.

The calls to `krb_set_srvtab_string` (point 39) and to `krb_set_tkt_string` (point 40) initialize memory internal to the Kerberos libraries, with the names of the srvtab file and the ticket file. Refer to `krb_set_tkt_string(3krb)` for more information about the ticket file. The server does not need to contact the `kerberos` daemon (Figure 6-1), to initialize itself. The clients will be required to contact Kerberos.

After the Kerberos libraries are initialized, the server calls `begin_assoc` (at point 69) to begin an association with a client.

In the routine, `begin_assoc` (at point 52), the server determines if there is a client ready to communicate (see the first half of a `while` loop at point 53 .)

Before the client contacts the server, the client begins executing the file `client.c` (Example 6-12) in `main`, which is located at point 133 . The client reads in the arguments sent to the client daemon at point 134 . The `read_args` routine for the high-level client is different from that for the low-level client, so the `read_args` routine is in file `l_client.c` (Example 6-10) at point 91 .

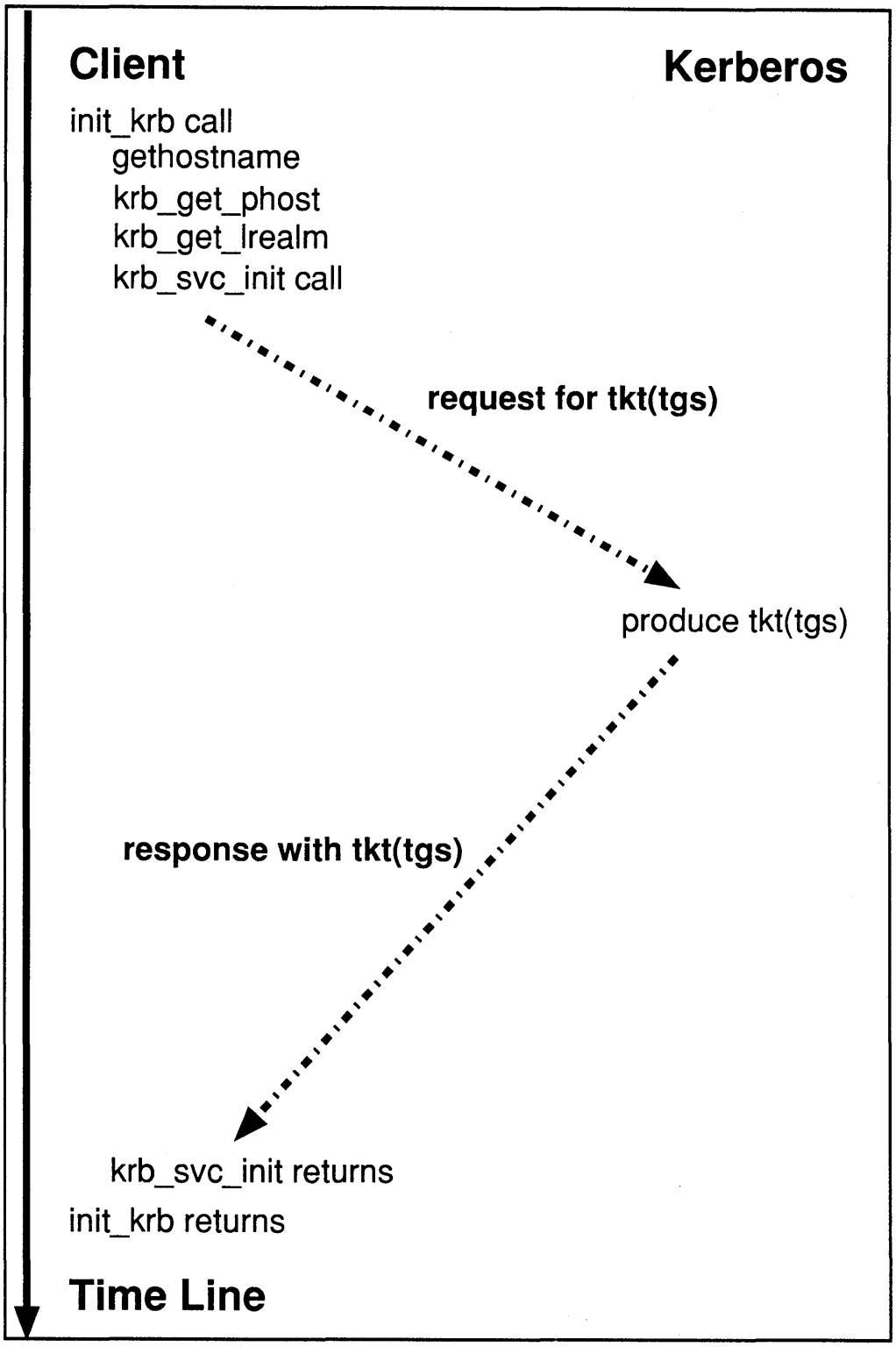
At point 92 , the low-level client sets default values for the log file, `log_file`, the ticket file, `p_state->tkt_file`, the srvtab file, `p_state->srvtab_file`, and the primary name of the service in the same way in which the server sets defaults. However, the low-level client uses two extra files: the output file, `out_file`, and the command file, `comm_file`. The output file is the file into which results from the low-level client operations are placed. The command file is the file from which commands sent by the low-level client to the server are read. The low-level client does not use an ACL file. The low-level client allows the user to change the default values for the files it uses at point 93 .

In `client.c` (Example 6-12), the client calls the routine `init_files` at point 135 . With the `init_files` procedure in `l_client.c` (Example 6-10) at point 90 , the low-level client opens the output file, command file, and log file.

With the `init_comm` procedure call in `client.c` (Example 6-12) at point 136 , the client initializes a socket that it uses to communicate with the server. The `init_comm` procedure itself is located in `client.c` at point 132 .

In the `init_krb` call at point 137 in `client.c` (Example 6-12), the low-level client initializes the Kerberos libraries so that the connection between the client and server can be authenticated. See Figure 6-2, which illustrates the low-level client initialization through `init_krb`, showing each client side subroutine call on a time line, with the interaction with Kerberos.

Figure 6-2: Client Low-Level Initialization



In the routine `init_krb` in file `l_client.c` (Example 6-10), at point 85, the low-level client calls the `gethostname`, (point 86), the `krb_get_phost` (point 87), and the `krb_get_lrealm` (point 88) routines for exactly the same reasons as the server. Refer to `krb_get_lrealm(3krb)` and `gethostname(2)` for more information about these program calls.

Next, the low-level client calls `krb_svc_init` (point 89), which contacts the `kerberos` daemon (as shown in Figure 6-2) so that the client can obtain a ticket-granting ticket, **tgt**. The ticket-granting ticket is used by the client to authenticate itself to the ticket-granting service, which is a part of the kerberos daemon, without having continuous access to the client password. The ticket-granting service gives, to the client, tickets with which it can authenticate itself to other services.

The `krb_svc_init` routine stores the ticket-granting ticket in the ticket file input to `krb_svc_init`, and reads the client password from the srvtab file input to `krb_svc_init`. For details of the operation of `krb_svc_init`, refer to `krb_svc_init(3krb)`.

The `init_krb` routine uses `p_state`, a `prin_state` structure. In `client.h` (Example 6-9) the `prin_state` structure is defined at point 73. The `p_state` structure of the client is the same as the `p_state` structure for the server, except that the client does not need a `stream_sock` variable to wait for connections.

After the Kerberos libraries are initialized, the client calls `begin_assoc` at point 138 in `client.c` (Example 6-12) to begin an association with a server.

In the `begin_assoc` routine (point 119) at the `while` loop (point 120) the client first calls `read_comm_file` at point 121 to fill in the variable `comm` (a command structure) with a command from the command file.

In `client.h` (Example 6-9), the definition of a command structure is at point 72. The `command` struct lists the action the client should take as well as the principal name of the client to which the action should be applied. Actions are usually ACL commands such as `check`, `exact_match`, and `add`. (See the reference page for `acl_check.3krb` for more information.) There are two other commands, `begin` and `end`. The `begin` command tells the client to begin an association with a server, and `end` tells the client to end the session.

In the `read_comm_file` procedure of `client.c` (Example 6-12) at point 107, the action type and the principal name is read from the command file. At point 108, `kname_parse` splits the principal name into its constituent primary, instance, and realm names. Refer to `acl_check(3krb)` for more information on the format of a principal name and the usage of `kname_parse`.

At point 122, (after the call to `read_comm_file`) the client determines whether the command is a `begin` command. A `begin` command includes the principal name of the server that the low-level client should contact. The rest of the `while` loop through point 125, contacts the appropriate server and fills in sections of the association state structure, `a_state`.

In `client.h` (Example 6-9), at point 74, the association state structure `assoc_state`, is defined. It describes the association between the client and the server, and includes the local address of the client, `l_addr`, the foreign address of the server's socket, `f_addr`, the file descriptor that the client uses to communicate with the server, `comm_sock`, and the principal name of the server: `f_primary_name`, `f_instance`, and `f_realm`.

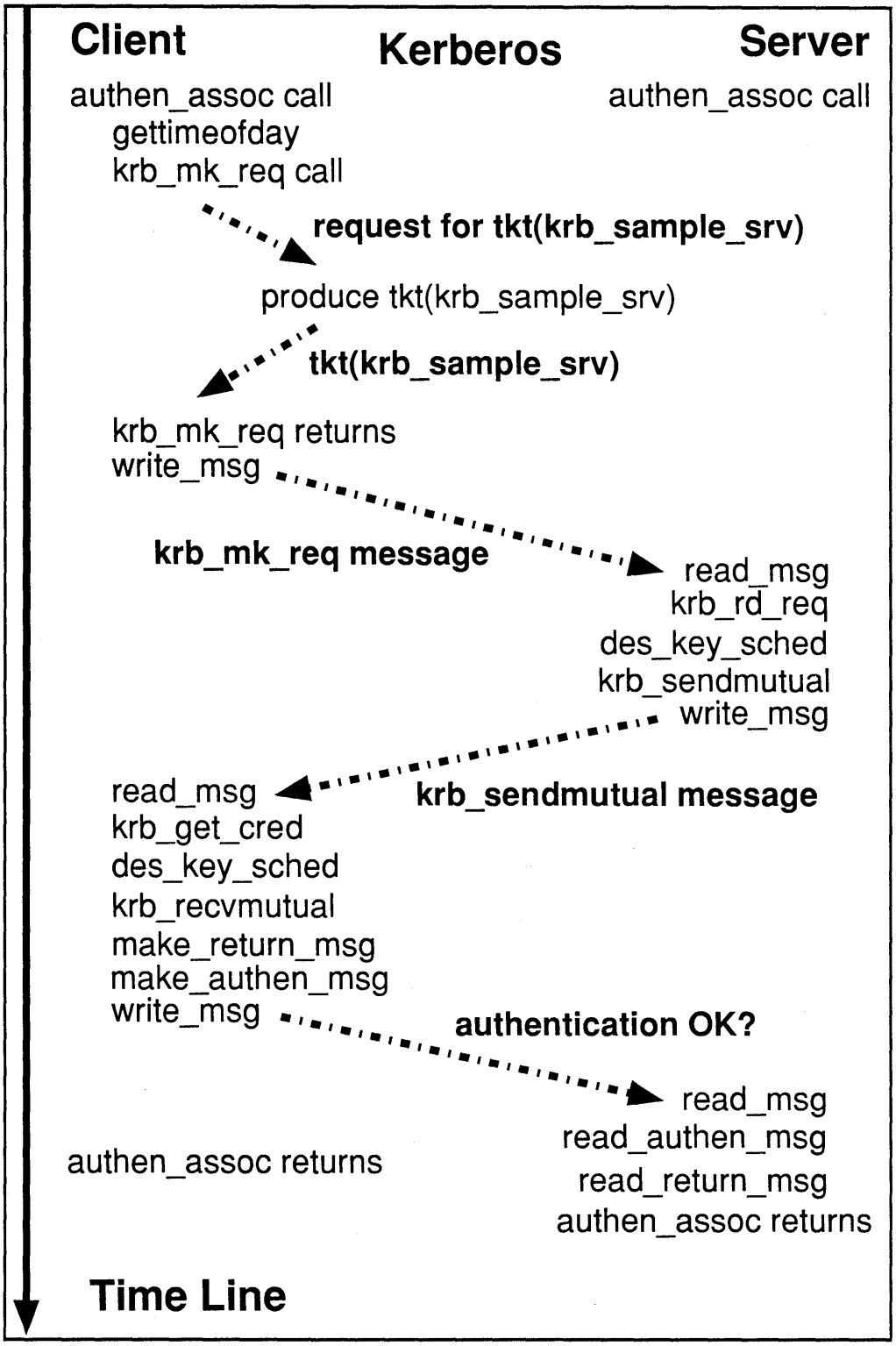
In `client.c` (Example 6-12), at point 123, the `gethostbyname` routine uses the instance name of the server as the name of the machine on which the server runs

in order to find the address of the server's machine. At point 124, the client determines the port that the server is using to communicate with the client. Both of these values are stored in `a_state->f_addr`. At point 126, `a_state->l_addr` is filled with the local address of the client. At point 127, `a_state->f_primary_name`, `a_state->f_instance`, and `a_state->f_realm` are filled with the name of the foreign server. All of these values will be used to authenticate the identity of the server.

Likewise, in `server.c` (Example 6-8) at point 54, the server fills in the `a_state->f_addr` values, and at point 55, the `a_state->l_addr` values.

In `server.h` (Example 6-5) at point 21, the association state structure (`assoc_state`) is the same as the one described in `client.h` (Example 6-9), at point 74, except that it does not include the principal name of the foreign host. This value will be stored in the `authen_data` structure.

Figure 6-3: Low-Level Authentication



At point 56 in the server, `server.c` (Example 6-8), and at point 128 in the client, `client.c` (Example 6-12), the client is connected to the server and is ready to send authentication information to the server. Both the client and the server authenticate the association with `authen_assoc`. The way in which messages are passed between client and server is pictured in Figure 6-3, which illustrates low-level authentication by presenting client and server code (and the interaction with Kerberos) along a time line.

In `l_client.c` (Example 6-10), the low-level client uses `krb_mk_req` at point 76, to format an authentication packet to send to the low-level server. The `krb_mk_req` routine contacts the `kerberos` daemon to obtain a ticket to communicate with the server, as illustrated in Figure 6-3. In addition, `krb_mk_req` creates an authenticator, packages it with the ticket, and returns the result in variable `krb_txt`. The `gettimeofday` call, at point 75, produces a random number, `checksum`, that is input to `krb_mk_req`. The random value is used to provide mutual authentication. Refer to `kerberos(3krb)` for more details about `krb_mk_req`. The low-level client sends the authentication information to the server at point 77.

In `l_server.c` (Example 6-6), at point 22, the low-level server receives the authentication information and uses `krb_rd_req` at point 23 to interpret the authentication information. If the `krb_rd_req` call succeeds, then the low-level client really is Kerberos-authenticated to the low-level server. Refer to `kerberos(3krb)` for more information about the `krb_rd_req` call.

At point 24, the low-level server uses the `des_key_sched` procedure to convert the session key between the low-level server and the low-level client (which was returned by `krb_rd_req` in `a_state->authen_data`) into a key schedule. Refer to `des_crypt(3krb)` for more details about `des_key_sched`. It then formats a message with `krb_sendmutual`, at point 25, that will authenticate the low-level server to the client. The message is placed by `krb_sendmutual` into `krb_txt`. Refer to `krb_sendmutual(3krb)` for more information about `krb_sendmutual`. At point 26, the low-level server sends the message formatted by `krb_sendmutual` to the low-level client.

In `l_client.c` (Example 6-10), at point 78, the low-level client reads the message sent by the low-level server, uses `krb_get_cred` at point 79 to get access to the session key between the low-level client and the low-level server (`a_state->cred`), converts the session key into a key schedule at point 80 with `des_key_sched` and attempts to authenticate the low-level server with `krb_recvmutual` at point 81.

The `krb_recvmutual` routine is given the message formatted by `krb_sendmutual` in `krb_txt` as input. If `krb_recvmutual` succeeds, the low-level server is authenticated to the low-level client. Refer to `kerberos(3krb)` for information about `krb_get_cred` and to `krb_sendmutual(3krb)` for information about `krb_recvmutual`.

At point 82, the low-level client uses `make_return_msg` to format a message to the server that indicates the status of the low-level client's authentication of the low-level server. It then authenticates the message at point 83 with `make_authen_msg`, and sends the message to the low-level server, at point 84. The routine, `make_authen_msg`, will be described in detail later.

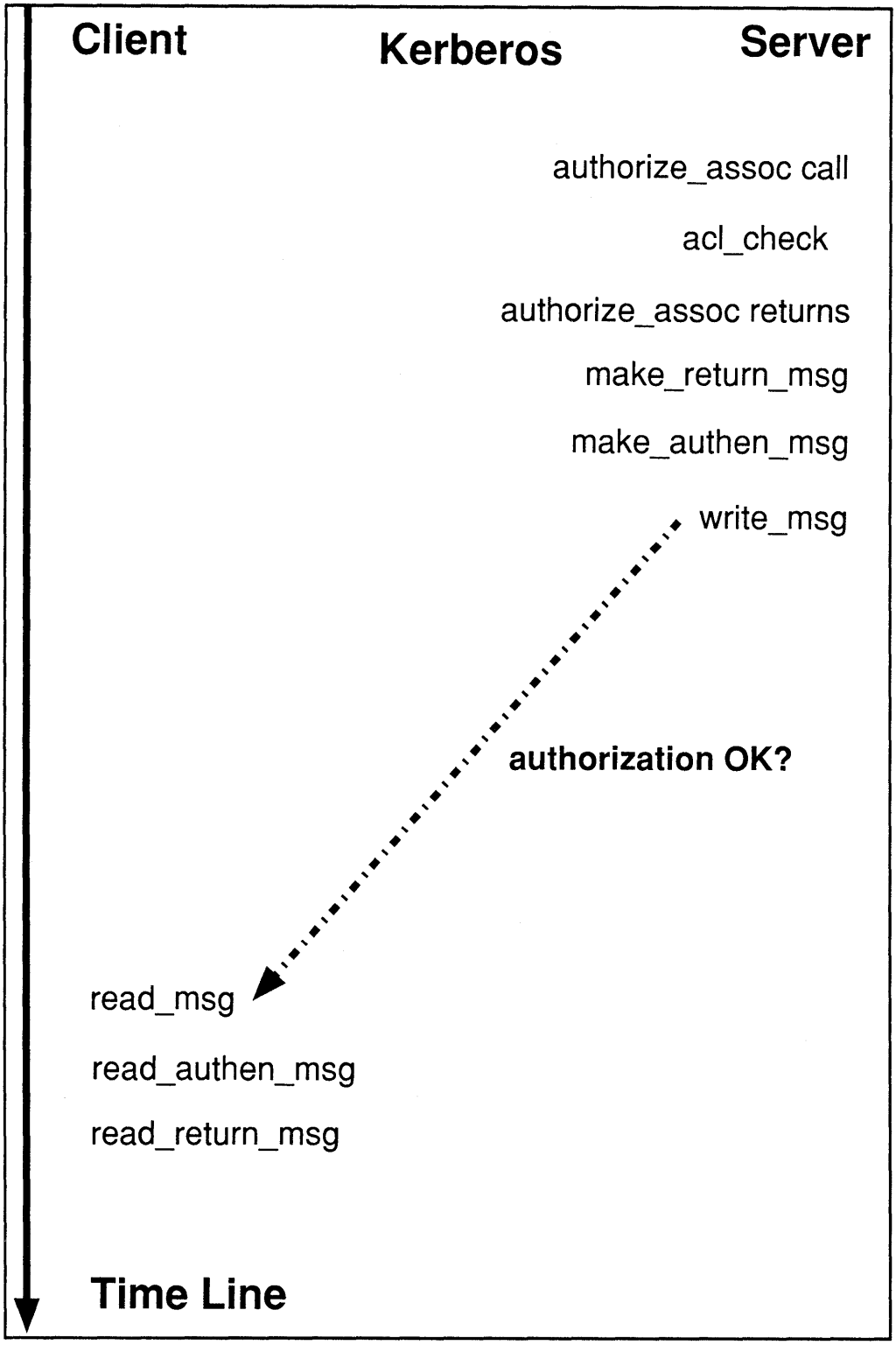
In `l_server.c` (Example 6-6), at point 27, the low-level server determines the status of its authentication to the low-level client in three steps:

1. by reading the message sent by the low-level client with `read_msg`,
2. by determining its authenticity with `read_authen_msg` at point 28 , and
3. by determining the status of the authentication with `read_return_msg` at point 29 .

The `read_authen_msg` routine will be discussed in detail later.

At point 57 , in the server, `server.c` (Example 6-8) and at point 129 in the client, `client.c` (Example 6-12), the client is authenticated to the server and the server is authenticated to the client. Next, the server must determine if the client is authorized to send commands to the server. The messages passed between the client and the server to achieve authorization are illustrated in Figure 6-4.

Figure 6-4: Client Authorization



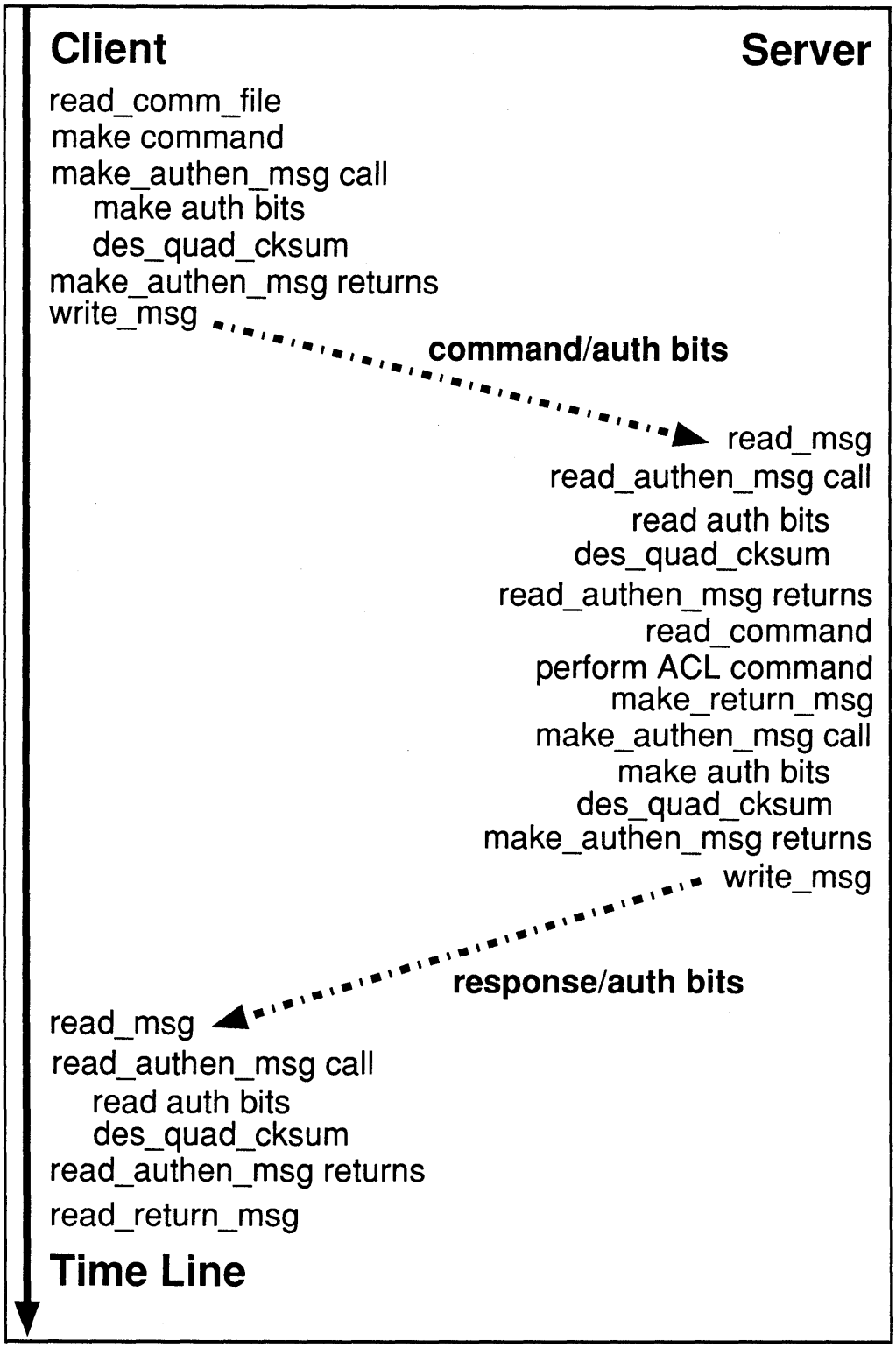
In `server.c` (Example 6-8) at point 57, the server calls the `authorize_assoc` procedure at point 41 to make sure that the client is allowed to communicate with the server. The decision is made in the `authorize_assoc` routine through a call to `acl_check`, at point 42. The `acl_check` routine checks to make sure that the client principal is a member of the ACL file, `acl_file`. If the client is a member, then the client should be able to ask the server questions about the ACL file. The principal name of the client that the server is communicating with, is stored in the `authen_data` structure returned by `krb_rd_req`. Refer to `acl_check(3krb)` for more information about `acl_check`.

At point 58, a message is formatted telling the client whether it has access to the server. At point 59, the message is authenticated, and at point 60, the message is sent to the client.

In `client.c` (Example 6-12), at point 129, the client determines the status of the authorization through the use of `read_msg`, `read_authen_msg`, at point 130, and `read_return_msg` at point 131.

At point 139 here in the client and at point 70 in the server, `server.c` (Example 6-8), the client is authenticated to the server and the server to the client. The server is ready to service the commands of the client and the client is ready to send commands. The `service_assoc` routine is used by the client to format and send commands to the server, and `service_assoc` is used by the server to receive, process, and answer these commands. The way in which data is passed between the client and server in `service_assoc` is illustrated in Figure 6-5, which shows the low-level service of commands by presenting the client and server command interaction along a time line.

Figure 6-5: Low-Level Access Control List (ACL) File Service



In `client.c` (Example 6-12), at point 110, the client reads a command from a file into the `comm_to_do` structure with `read_comm_file` at point 107. It converts the command into an "on-the-wire" format with `make_command` at point 111. The possible commands are `check`, `exact_match`, `add`, `delete`, and `end`. Every command except `end`, includes the name of the principal associated with the command. Refer to `acl_check(3krb)` for more information about these commands.

At point 112, the command is authenticated by `make_authen_msg`, and sent to the server with `write_msg`, at point 113.

In `server.c` (Example 6-8), at point 44, the server reads the command with the `read_msg` procedure and authenticates it with `read_authen_msg` at point 45.

In `low_level.c` (Example 6-3), the `make_authen_msg` routine at point 10 and the `read_authen_msg` at point 1 are designed to write and read the authentication information placed on every packet sent between the low-level client and the low-level server. The authentication information guarantees that every message sent between the low-level server and low-level client comes from the right application and has not been altered during its journey. The low-level `make_authen_msg` and `read_authen_msg` routines do not encapsulate the "on-the-wire" protocol of the low-level client and low-level server. They alter the "on-the-wire" protocol by adding authentication bits to a command or a response.

The `make_authen_msg` routine adds the time at which the packet was formatted, `time` (at point 11), the address of the machine that is formatting the message, `l_addr` (at point 12), and a direction value, `direction`, (at point 13), that indicates whether the low-level client or the low-level server formatted the message input to `make_authen_msg`.

At point 14, the `make_authen_msg` call made by the low-level client uses `des_quad_cksum` to form a checksum of the message and add it as a part of the authentication bits. The checksum is formed in such a way as to identify the producer of the message (authentication), and prevent anyone except the low-level client and the low-level server from producing the checksum (modification protection). Refer to `des_crypt(3krb)` for more information about `des_quad_cksum`.

The `read_authen_msg` procedure, at point 1, called by the low-level client, reads the information written by the `make_authen_msg` call (point 10), at points 2, 3, and 4.

At point 5, `des_quad_cksum` is used to recreate the checksum that should have been created by the low-level client for this message.

At point 6, if the checksum in the message is different from the checksum produced by `des_quad_cksum`, then the message is not from the low-level client, or it was modified by an intruder; so, the message is rejected.

At point 7, if the address for which the packet was formatted is not the address from which the packet was sent, then the packet may have been stolen from the network and sent by an intruder; so, the message is rejected.

At point 8, if the message is marked as if it were sent by a server, an intruder must have sent back to the server a message sent by the server; so, the message is rejected.

At point 9, if the message is too old, it is rejected.

In `server.c` (Example 6-8) at point 46, after the server knows the message is authentic, the command is read from the message with `read_command`.

At point 47, the command is converted into an ACL command and performed. The ACL commands access the ACL file, `acl_file`. For more information about the ACL commands, refer to `acl_check(3krb)`.

At point 48, the result of the command is formatted by `make_return_msg`, authenticated by `make_authen_msg`, at point 49, and written to the client with `write_msg`, at point 50.

In `client.c` (Example 6-12), at point 114, the result is read by the client, authenticated by `read_authen_msg` at point 115, and interpreted by `read_return_msg` at point 116.

At point 117, the result of the command is printed by the client.

In `server.c` (Example 6-8), at point 51, if the command sent to the server by the client is an `end` command, then the server returns to `main` from `service_assoc`. The client does the same in `client.c` (Example 6-12), at point 118.

In `server.c` (Example 6-8), at point 71, the server closes its communication channel to the client with `end_assoc`, and waits for another client with `begin_assoc` at point 69.

In `client.c` (Example 6-12), the client closes its communication channel to the server with `end_assoc`, at point 140. It attempts to contact another server with `begin_assoc`, at point 138.

6.2.3 High-Level Example Explanation

The high-level versions of the client and server perform the same service in almost the same way as the low-level client and server, so they are able to share a large amount of code. The explanation of the way in which the high-level server-client pair work focuses on the differences between the high-level and low-level server-client pairs.

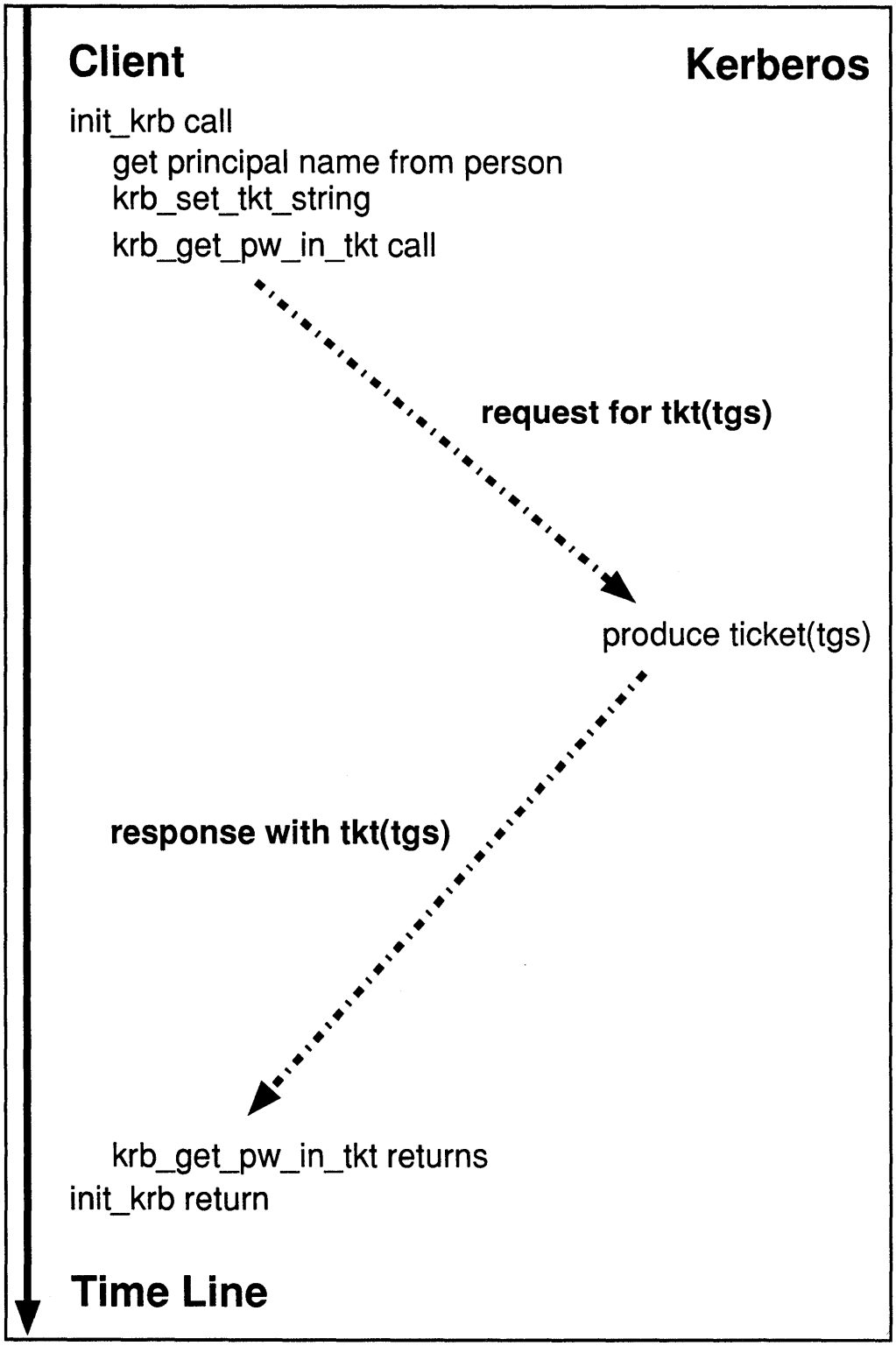
In `server.c` (Example 6-8) at point 63, the high-level server begins like the low-level server. In fact, the execution of the high-level server is equivalent to the low-level server until the high-level server executes the the `authen_assoc` call (see point 56) inside of `begin_assoc` at point 52.

In `client.c` (Example 6-12) at point 133, the high-level client also begins, like the low-level client. But, the high-level client begins to differ from the low-level in the calls to `read_args` at point 134 and to `init_files` at point 135.

The high-level versions of `init_files` and `read_args` are in `h_client.c` at points 103 and 104, respectively. The high-level client is not itself a principal, but assumes the identity of the user that starts the high-level client. In addition, the high-level client gets its commands from the user, and sends the user back the results of its messages to the high-level server.

So, to enable the client to communicate with the user, the command file, `fio_comm`, is set equal to `stdin`, the output file, `fio_comm`, is set equal to `stdout`, and the log file, `fio_log` is set equal to `stderr`, in `init_files` at point 103. As a result, in the `read_args` routine at points 105 and 106, the user is not allowed to change the values of the log, output, or command files. The service table (srvtab) file is not used by the high-level client because the user supplies a password.

Figure 6-6: Client High-Level Initialization



The next portion of code in which the low-level client differs from the high-level is the `init_krb` call in `client.c` (Example 6-12), at point 137. See Figure 6-6.

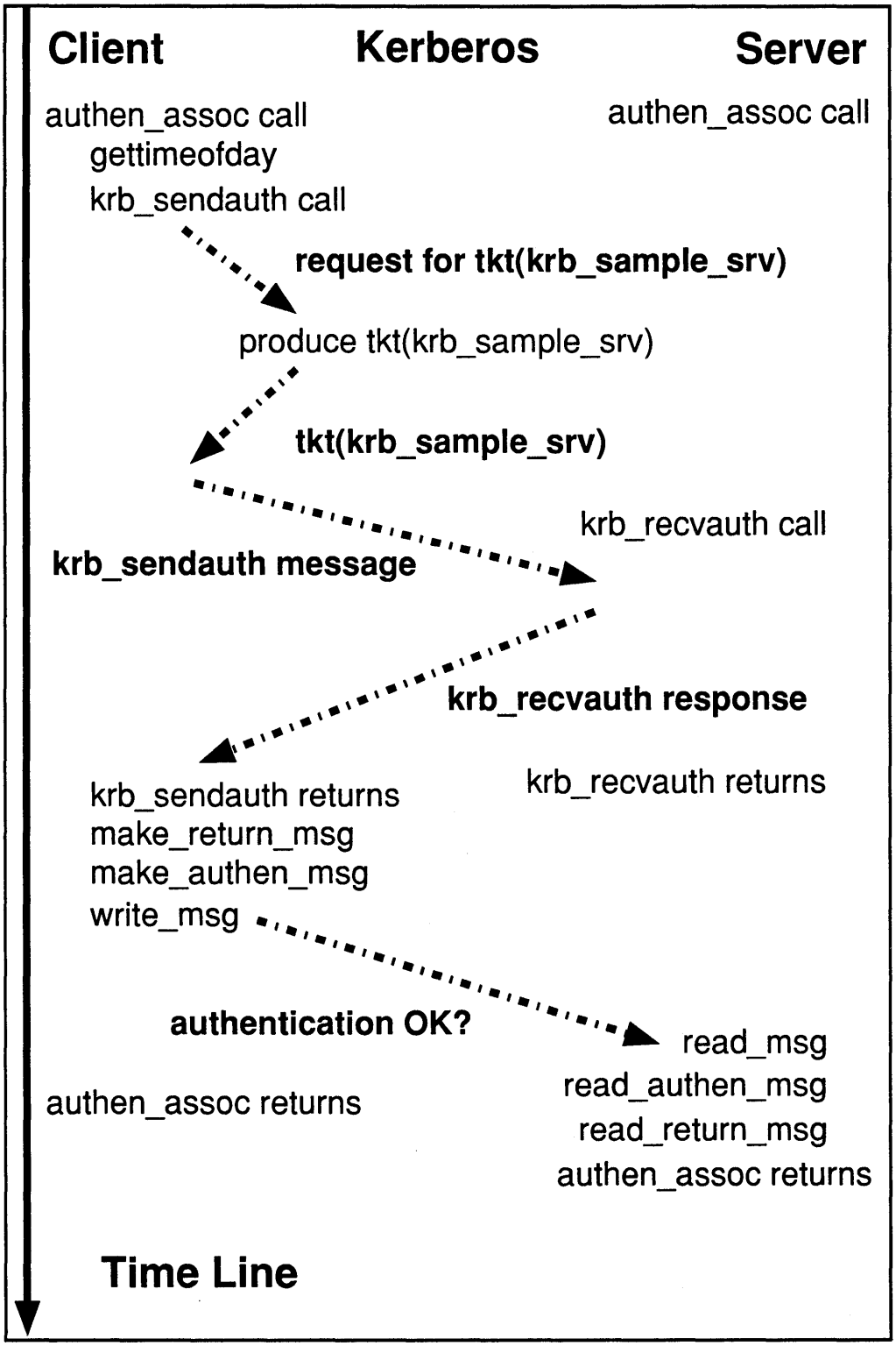
In `h_client.c` (Example 6-11) at point 99, there is the `init_krb` code for the high-level client. The `init_krb` call of the high-level client performs the same function as the `init_krb` call of the low-level client, except that it initializes the Kerberos libraries for a user instead of for the client daemon.

The high-level client prompts the user for the user principal, instance, and realm name. Like the low-level server, these three values are stored in the `p_state` structure at point 100.

In the low-level client file, `l_client.c` (Example 6-10), the `krb_svc_init` call at point 89 sets the default ticket file, but the `krb_get_pw_in_tkt` call in `h_client.c` (Example 6-11) at point 102 does not set the default ticket file value. Therefore, the high-level client calls `krb_set_tkt_string` at point 101. For more details about `krb_set_tkt_string`, refer to `krb_set_tkt_string(3krb)`.

At point 102, the high-level client calls `krb_get_pw_in_tkt` to obtain a ticket-granting ticket from the `kerberos` daemon for the user. The `krb_get_pw_in_tkt` string call prompts the user for a password, while the `krb_svc_init` call in `l_client.c` (Example 6-10) at point 89 of the low-level client, obtained the password of the low-level client from the service table (srvtab) file. For more details about `krb_get_pw_in_tkt`, refer to `krb_svc_init(3krb)`.

Figure 6-7: High-Level Authentication

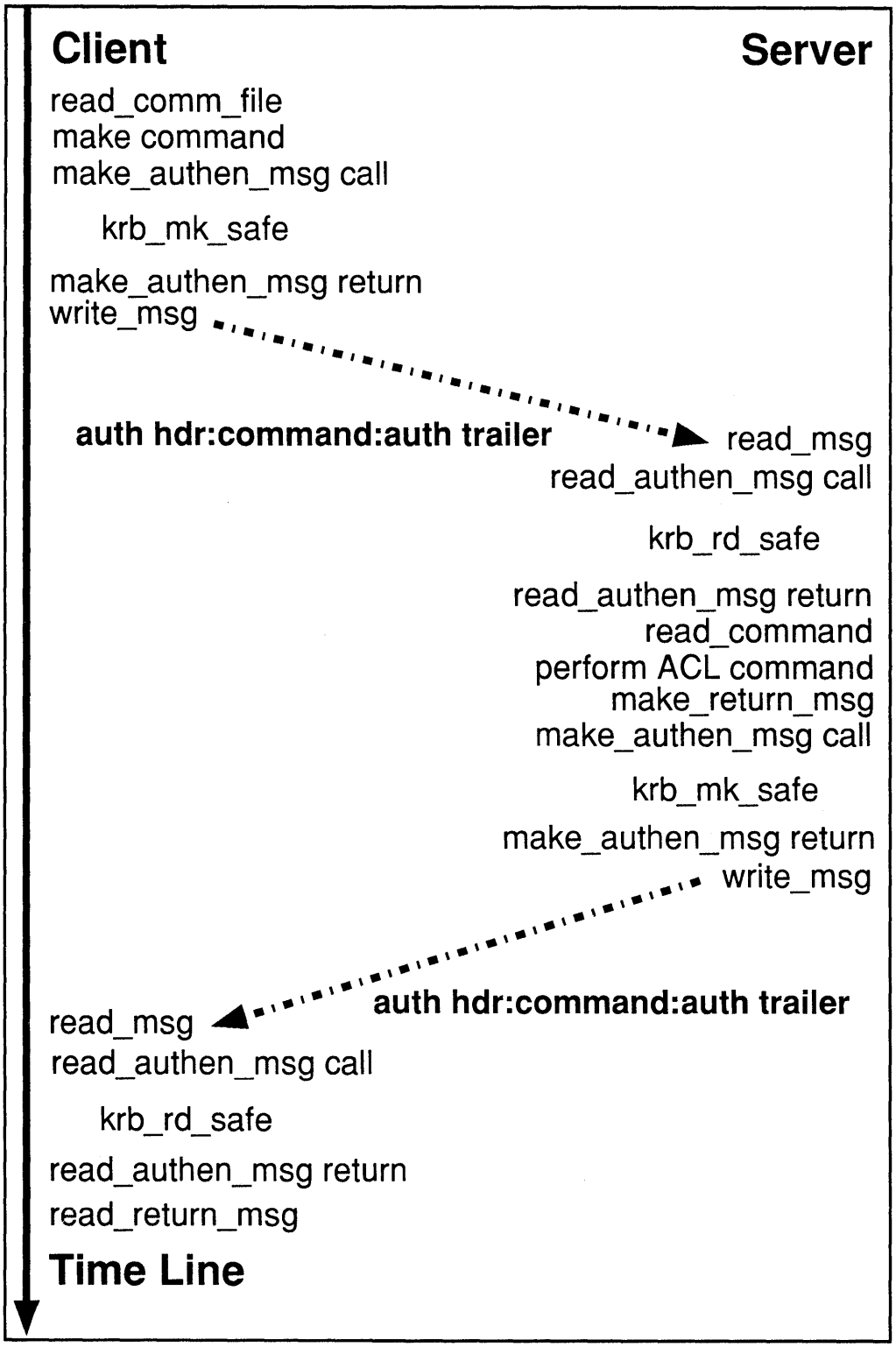


Recall that the high-level server first differs from the low-level server in the `authen_assoc` call in `server.c` (Example 6-8) at point 56, within the `begin_assoc` procedure at point 52. The client's `authen_assoc` procedure, called in `client.c` (Example 6-12) at point 128 within the `begin_assoc` procedure at point 119, also differs from the client's low-level version of `authen_assoc`. See Figure 6-7.

The high-level server's version of `authen_assoc` is in file `h_server.c` (Example 6-7), at point 30, while the high-level client's version is in `h_client.c` (Example 6-11) at point 94. The high-level client calls `gettimeofday` to produce a checksum just like the low-level client. However, it then calls `krb_sendauth`, at point 95. Next, `krb_sendauth` requests a ticket to communicate with the server from the `kerberos` daemon, receives the ticket, and produces a ticket-authenticator pair just like `krb_mk_req`, in `l_client.c` (Example 6-10) at point 76. Then, however, unlike `krb_mk_req`, the `krb_sendauth` routine sends the ticket-authenticator pair to the high-level server on the file descriptor provided, `a_state->comm_sock`.

In `h_server.c` (Example 6-7), the high-level server calls `krb_recvauth` at point 31, to read the message sent by `krb_sendauth` with the file descriptor provided, `a_state->comm_sock`, and, like `krb_rd_req`, at point 23 in `l_server.c` (Example 6-6), it authenticates the high-level client. Because the KOPT_DO_MUTUAL option is set, `krb_recvauth` formats a message that will authenticate the high-level server to the high-level client, like `krb_sendmutual` does at point 25, and returns the message to the high-level client. Next, `krb_sendauth` receives the message from the high-level server, and authenticates the high-level server in the same way as `krb_recvmutual`, in `l_client.c` (Example 6-10) at point 81. Both procedures then return. Refer to `krb_sendauth(3krb)` for more information about `krb_sendauth` and `krb_recvauth`.

Figure 6-8: High-Level Access Control List (ACL) File Service



In the service_assoc routine in client.c (Example 6-12), located at point 109, the client calls make_authen_msg to authenticate a command sent to the server. The server, in its version of service_assoc, at point 43 in server.c (Example 6-8), calls the read_authen_msg routine to authenticate the client that sent the message. The high-level versions of the make_authen_msg call and the read_authen_call in file high_level.c (Example 6-4), differ greatly from the low-level versions, in file low_level.c (Example 6-3). See Figure 6-8.

The high-level version of make_authen_msg, at point 17 calls the krb_mk_safe routine, at point 18 to authenticate the sender of the message and to prevent the message from being tampered with during transit. Unlike the low-level version of make_authen_msg, in file low_level.c (Example 6-3) the krb_mk_safe routine does not add authentication information on the end of the client's command, but encapsulates the client's message in a message formatted by krb_mk_safe. The high-level version of read_authen_msg in file high_level.c (Example 6-4) calls krb_rd_safe at point 16 to unpackage the message formatted by krb_mk_safe. For more details about krb_mk_safe and krb_rd_safe, refer to kerberos(3krb).

6.2.4 The all.h File

The `all.h` file is a common header file, shown in Example 6-1:

Example 6-1: The all.h Header File

```
/* all.h includes macro definitions for all the low- and
   high-level server and client C files.
*/

#include <stdio.h>
#include <krb.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/uio.h>
#include <sys/stat.h>

/* procedure return values */
#define OK       0
#define FAILURE  -1
#define STOP     -2

/* size constraints */
#define FILE_NAME_SZ      80
#define HOST_NAME_SZ      80
#define MAX_COMM_LINE_SZ  80
#define MAX_MSG_SZ      1024

#define ALL_FDS  32
#define TKT_LIFETIME  255

extern  char   *krb_get_phost();
extern  int    errno;
```

6.2.5 The comm.c File

The `comm.c` file consists of code that is common to the high- and low-level server and client. It is shown in Example 6-2:

Example 6-2: The comm.c Routine

```
/* comm.c contains code which is used by the high and
   low-level servers as well as the high- and low-level
   clients.
*/

#include "all.h"

extern FILE   *fio_log; /* file descriptor of log file */

/* read_bytes attempts to read size bytes from comm_sock.
   It returns FAILURE if an error occurs or OK if the bytes
   were correctly read.
*/
read_bytes(comm_sock, buf, size)
   int  comm_sock; /* file descriptor used to read (input) */
   char *buf;      /* buff in which to place bytes read (input) */
```

Example 6-2: (continued)

```
   int  size;      /* the number of bytes to read (input) */
{
   int  len;   /* bytes currently read */

   while(size > 0 && (len = read(comm_sock, buf, size)) > 0)
   {
      buf += len;
      size -= len;
   }

      if(len <= 0)
      {
         fprintf(fio_log,"Bad read_bytes call.\n");
         return(FAILURE);
      }
      else
      {
         return(OK);
      }

}

/* read message attempts to read the next message sent on
   socket comm_sock.  A message always begins with the
   number of bytes in the message.  FAILURE is returned if
   an error occurs, OK otherwise.
*/
int read_msg(comm_sock, msg, msg_len)
   int  comm_sock;  /* file descriptor used to read (input)*/
   char *msg;  /* buff in which to place the message (input)*/
   int  msg_len;  /* the size of the message (output) */
{
   int read_mask;   /* the read mask for select */
   int result_mask; /* the mask of fd ready */
   int comm_not_ready = 1; /* 0 if bytes are ready to read */
   int size = sizeof(u_long);     /* size of bytes to read */
   int sizeofmsg;    /* size of the message in network form */
   int num_ready;    /* number of file descriptors ready */
   int result;

   while(comm_not_ready)
   {
      read_mask = 1 << comm_sock;

      if((num_ready = select(ALL_FDS, &read_mask,
         (int *)NULL, (int *)NULL,
         (struct timeval *)NULL)) < 0)
      {
         if(errno == EINTR)
         {
            fprintf(fio_log, "interrupted sys call");
            continue;
         }
         else
         {
            fprintf(fio_log, "select error");
            return(FAILURE);
         }
      }
      else
         comm_not_ready = 0;
   }
```

Example 6-2: (continued)

```
   if((result = read_bytes(comm_sock,
                           (char *)&sizeofmsg, size)) != OK)
   {
      fprintf(fio_log, "read error\n");
      return(FAILURE);
   }

   sizeofmsg = ntohl(sizeofmsg);

   if(sizeofmsg > msg_len)
   {
      fprintf(fio_log, "msg too large\n");
      return(FAILURE);
   }

   if((result = read_bytes(comm_sock, msg, sizeofmsg)) != OK)
   {
      fprintf(fio_log, "read error\n");
      return(FAILURE);
   }

   return(sizeofmsg);
}

/* write_msg writes the length of a message followed by the
   message to a file descriptor.  It returns FAILURE if an
   error occurred, OK otherwise.
*/
write_msg(comm_sock, msg, size)
   int         comm_sock; /* the file descriptor written to (input)*/
   char        *msg;      /* buff that holds the message (input)*/
   int         size;      /* the size of the message (input) */
{
   struct iovec iov[2];/* vector of output */
   int result;
   u_long len = htonl((u_long)size);  /* length of output */

   iov[0].iov_base = (caddr_t)&len;
   iov[0].iov_len = sizeof(u_long);
   iov[1].iov_base = (caddr_t)msg;
   iov[1].iov_len = size;

   if((result = writev(comm_sock, iov, (int)2))
                        != sizeof(u_long) + size)
   {
      fprintf(fio_log,"Bad writemsg call.\n");
      return (FAILURE);
   }
   return (OK);
}

/* read_return_msg interprets a message returned to the
   receiver in response to an earlier message.  read_return
   message returns FAILURE if an error occurred or OK,
   otherwise.
*/
read_return_msg(ret_msg, len, status)
   char  *ret_msg;  /* the response msg to a msg (input) */
   int   len;       /* length of the above (input) */
   char  *status;   /* status string in the msg (output) */
{
   char        *cp;
   u_long   length;
```

Example 6-2: (continued)

```
    u_long   long_val;

    cp = ret_msg;
    bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
    length = ntohl(long_val);

    cp += sizeof(u_long);

    strcpy(status, cp);
    if(strlen(status) + 1 != (int)length)
    {
        fprintf(fio_log, "Bad length in return msg\n");
        return(FAILURE);
    }

    return(OK);
}

/* make_return_msg formats a response message to a message
    sent earlier to the sender.  It places a status string
    inside the message.  read_return message returns FAILURE
    if an error occurred or OK, otherwise.
*/
make_return_msg(ret_msg, len, status)
    char  *ret_msg;   /* the response msg to a msg (output)*/
    int   *len;       /* length of the above (ouput) */
    char  status;     /* status code for the msg (input) */
{
    u_char   *length;  /* pointer to the length of the msg */
    u_long   long_val;
    u_char   *cp;

    cp = (u_char *)ret_msg;
    length = cp;
    cp += sizeof(u_long);

    if(status == OK)
        strcpy(cp, "OK");
    else
        strcpy(cp, "FAILURE");

    cp += strlen(cp) + 1;

    *len = cp - (u_char *)ret_msg;
    long_val = htonl(*len - sizeof(u_long));
    bcopy((char *)&long_val, (char *)length, sizeof(u_long));

    return(OK);
}
```

6.2.6 The low_level.c File

The `low_level.c` file (like the `high_level.c` file) contains code that is specific to the high- and low-level client-server pairs. The `low_level.c` file is Example 6-3:

Example 6-3: The low_level.c Routine

```
/* low_level.c contains code common to both the low level
   client and the low level server.
*/

#ifdef SERVER

#include "server.h"
int   direction = 0;    /* direction is 0 if the message is
                           generated or read by these routines
                           by the client, 1 otherwise. */

#else /* SERVER */

#include "client.h"
int   direction = 1;    /* direction is 0 if the message is
                           generated or read by these routines
                           by the client, 1 otherwise. */

#endif /* SERVER */

extern FILE   *fio_log;  /* file descriptor of the log file */

/* read_authen_msg takes a message which has authentication
   bits formatted by make_authen_msg at the end of the message
   and guarantees, that the message was formatted by the sender
   associated with the key, session_key, and that the message
   was sent by the principal that formatted the message.  In
   addition, it guarantees that the message was not altered in
   transit, that it is probably not old enough to be a replayed
   message, and that it is not a replay of a message sent earlier
   by the receiver of the message.  It returns FAILURE if any of
   the guarantees are broken, otherwise it returns OK.
*/

1
read_authen_msg(session_key, f_addr, l_addr, safe_msg,
                length, msg, msg_len)

   des_cblock   session_key;  /* the session key between
                                 client and server (input) */
   struct sockaddr_in  *f_addr;/* addr of sender (input)*/
   struct sockaddr_in  *l_addr;/* addr of receiver (input)*/
   u_char   *safe_msg;        /* msg received (input)  */
   int   length; /* length of the above(input)*/
   char  *msg;  /* message without authentication info(output)*/
   int   *msg_len;   /* length of above(output)*/
{
   u_long   length;   /* length of the msg without
                         authentication bytes */
   u_long   length_auth;     /* length of authentication
                                section of message */
   u_long   my_cksum[8];     /* checksum of the message */
   struct   timeval  time;   /* time now */
   struct   timeval  time_packet;   /* time packet was sent */
   struct   timezone zone;     /* time zone */
```

Example 6-3: (continued)

```
long     time_diff;       /* time msg spent in transit */
u_long   direction_packet;  /* direction bit in packet */
u_long   f_addr_packet;     /* receiver address in packet*/
u_char   *cp;
u_long   long_val;


if(gettimeofday(&time, &zone))
{
   fprintf(fio_log, "bad gettimeofday call.\n");
   return(FAILURE);
}

cp = safe_msg;
bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
length = ntohl(long_val);
cp += sizeof(u_long);
cp += length;

bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
length_auth = ntohl(long_val);
cp += sizeof(u_long);
```

2

```
bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
time_packet.tv_usec = ntohl(long_val);
cp += sizeof(u_long);

bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
time_packet.tv_sec = ntohl(long_val);
cp += sizeof(u_long);
```

3

```
bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
f_addr_packet = ntohl(long_val);
cp += sizeof(u_long);
```

4

```
bcopy((char *)cp, (char *)&long_val, sizeof(u_long));
direction_packet = ntohl(long_val);
cp += sizeof(u_long);
```

5

```
long_val = des_quad_cksum(safe_msg, my_cksum,
           (long)(cp - safe_msg), (int)4, session_key);
```

6

```
if(bcmp((char *)my_cksum, cp, 32))
{
   fprintf(fio_log, "bad checksum in read_authen_msg.\n");
   return(FAILURE);
}
```

7

```
if(f_addr_packet != (u_long)f_addr->sin_addr.s_addr)
{
   fprintf(fio_log, "bad source address.\n");
   return(FAILURE);
}
```

8

```
if(direction_packet == direction)
{
```

Example 6-3: (continued)

```
      fprintf(fio_log, "bad direction in read_authen_msg.\n");
      return(FAILURE);
   }

   time_diff = time_packet.tv_sec - time.tv_sec;

9
   if(time_diff > 60 || time_diff < -60)
   {
      fprintf(fio_log, "bad time in read_authen_msg.\n");
      return(FAILURE);
   }

   bcopy(safe_msg, msg, length + sizeof(u_long));

   *msg_len = length + sizeof(u_long);

   return(OK);
}
/* make_authen_msg adds authentication bits onto the end of
   a message to guarantees, that the sender of the message can
   be determined.  In addition, it adds bits to guarantee that
   the message cannot be altered in transit, and it helps
   prevent replays by placing the time that the message was
   sent and the direction in which it was sent in the
   authentication bits.  It returns FAILURE if any error
   occurs, otherwise it returns OK.
*/
10
make_authen_msg(session_key, f_addr, l_addr, msg, len)
   des_cblock   session_key;  /* the session key between
                                 client and server (input) */
   struct sockaddr_in  *f_addr;/* addr of receiver (input)*/
   struct sockaddr_in  *l_addr;/* addr of sender (input)*/
   char  *msg;    /* message to which authentication info
                     is added (input) */
   int  *len;     /* length of above(output)*/
{
   struct timeval   time;    /* the current time */
   struct timezone  zone;    /* time zone */
   u_char *length;          /* pointer to the length section
                               of the authentication bits */
   u_long  chksum[8];/* checksum to place in message */
   u_char  *cp;
   u_long  long_val;

   if(gettimeofday(&time, &zone))
   {
      fprintf(fio_log, "bad gettimeofday call.\n");
      return(FAILURE);
   }

   length = (u_char *)msg + *len;
   cp = length + sizeof(u_long);
11
   long_val = htonl((u_long)time.tv_usec);
   bcopy((char *)&long_val, (char *)cp, sizeof(u_long));
   cp += sizeof(u_long);

   long_val = htonl((u_long)time.tv_sec);
   bcopy((char *)&long_val, (char *)cp, sizeof(u_long));
   cp += sizeof(u_long);
```

Example 6-3: (continued)

```
12
   long_val = htonl((u_long)l_addr->sin_addr.s_addr);
   bcopy((char *)&long_val, (char *)cp, sizeof(u_long));
   cp += sizeof(u_long);

13
   long_val = htonl((u_long)direction);
   bcopy((char *)&long_val, (char *)cp, sizeof(u_long));
   cp += sizeof(u_long);

   *len = (cp - (u_char *)msg) + 32;
   long_val = htonl((u_long)(cp  + 32 -
                  ((u_char *)length + sizeof(u_long))) );

   bcopy((char *)&long_val, (char *)length, sizeof(u_long));

14
   long_val = des_quad_cksum((u_char *)msg, chksum,
              (long)(cp - (u_char *)msg), (int)4, session_key);

   bcopy((char *)chksum, (char *)cp, 8 * sizeof(u_long));
   return(OK);
}
```

6.2.7 The high_level.c File

The `high_level.c` file (like the `low_level.c` file) contains code that is specific to the high-level and low-level client-server pairs. Example 6-4 consists of the `high_level.c` file:

Example 6-4: The high_level.c Routine

```
/* high_level.c contains code common to both the high-level
   client and the high-level server.
*/

#ifdef SERVER
#include "server.h"

#else /* SERVER */
#include "client.h"

#endif /* SERVER */

extern FILE   *fio_log;   /* file descriptor of the log file */

/* read_authen_msg takes a message which is encapsulated by
   krb_mk_safe and uncovers it with krb_rd_req.  With krb_rd_rq
   it guarantees that the message was formatted by the sender
   associated with the key, session_key, and that the message
   was sent by the principal that formatted the message.  In
   addition, it guarantees that the message was not altered
   in transit, that it is probably not old enough to be a
   replayed message, and that it is not a replay of a message
   sent earlier by the receiver of the message.  It returns
   FAILURE if any of the guarantees are broken, otherwise it
   returns OK.                                                */
15
read_authen_msg(session_key, f_addr, l_addr, safe_msg,
                length, msg, msg_len)
```

Example 6-4: (continued)

```
   des_cblock  session_key;      /* the session key between
                                    client and server (input) */
   struct  sockaddr_in  *f_addr; /* addr of sender (input)    */
   struct  sockaddr_in  *l_addr; /* addr of receiver (input)  */
   u_char  *safe_msg;            /* msg received (input)      */
   int  length;                  /* length of the above(input)*/
   char  *msg;  /* message without authentication info(output)*/
   int  *msg_len;   /* length of above(output)*/
{
   MSG_DAT  message_data;   /* information about message sent */
   long        result;

16
   if((result = krb_rd_safe(safe_msg, (u_long)length, session_key,
               f_addr, l_addr, &message_data)) != RD_AP_OK)
   {
      fprintf(fio_log, "krb_rd_safe kerberos error %ld\n", result);
      return(FAILURE);
   }
   else
   {
      bcopy(message_data.app_data, msg, message_data.app_length);
      *msg_len = message_data.app_length;
      return(OK);
   }
}

/* make_authen_msg encapsulates with krb_mk_safe a message.
   With krb_mk_safe, it guarantees, that the sender of the
   message can be determined.  In addition, it guarantees
   that the message cannot be altered in transit, and that
   replay of the message is hindered.  It returns FAILURE
   if any error occurs, otherwise it returns OK.
*/
17
make_authen_msg(session_key, f_addr, l_addr, msg, len)
   des_cblock           session_key; /*session key between
                                       client and server (input)*/
   struct sockaddr_in   *f_addr;  /*addr of the receiver (input)*/
   struct sockaddr_in   *l_addr;  /*addr of sender (input)*/
   char                 *msg;  /*message without authentication
                                 info (input), and the message
                                 with authentication info (output)*/
   int                  *len;  /*length of above (input) and (output)*/
{
   u_char      temp_msg[MAX_MSG_SZ];

   bcopy(msg, (char *)temp_msg, *len);


18
   if((*len = (int)krb_mk_safe(temp_msg, (u_char *)msg,
              (u_long)*len, session_key, l_addr, f_addr)) < 0)
   {
      fprintf(fio_log, "krb_mk_safe call kerberos
                        error %d\n", *len);
      return(FAILURE);
   }

   return(OK);
}
```

6.2.8 The server.h File

The `server.h` file is one of four files that contain code for both the high-level and low-level servers. Example 6-5 consists of `server.h`:

Example 6-5: The server.h Routine

```
/* server.h */

#include "all.h"

#define SERVER_NAME"krb_sample_srv"

/* The command structure stores the action performed by
   the command as well as the principal name the action
   should be performed upon */
19
struct command {
   char        action[80];/* command action */
   char        principal[80];/* principal on which action is done */
};
typedef struct command command;

/* The prin_state structure stores the server state associated
   only with the running server.
20
*/struct prin_state {
   char        primary_name[SNAME_SZ]; /* primary name of server */
   char        instance[INST_SZ];      /* instance name of server */
   char        realm[REALM_SZ];        /* realm name of server */
   char        version[KRB_SENDAUTH_VLEN];  /* version of server */

   char        srvtab_file[FILE_NAME_SZ]; /* srvtab file of server */
   char        tkt_file[FILE_NAME_SZ];    /* ticket file of server */

   int  stream_sock;               /* socket clients connect to */
   struct sockaddr_in  stream_addr;/* address of the above */
};
typedef struct prin_state prin_state;

/* The assoc_state structure stores the server state related
   to an association with a client.
*/
21
struct assoc_state {
   int         comm_sock;   /* socket used to talk with client. */
   char        version[KRB_SENDAUTH_VLEN];  /*version of client */

   struct sockaddr_in  l_addr;   /* address of the server */
   struct sockaddr_in  f_addr;   /* address of the client */

   AUTH_DAT  authen_data;   /* authentication data for client */
};
typedef struct assoc_state assoc_state;
```

6.2.9 The l_server.c File

The `l_server.c` file is one of four files that contain code for both the high-level and low-level servers. Example 6-6 consists of `l_server.c`:

Example 6-6: The l_server.c Routine

```
/* l_server.c contains server code that is only used by
   the low level server.
*/

#include "server.h"

extern FILE  *fio_log;   /* file descriptor of the log file */

/* authen_assoc attempts to authenticate the association
   between the low-level client and the low-level server
   for the server.  It returns FAILURE if the authentication
   fails and OK, otherwise.
*/
authen_assoc(p_state, a_state)
   assoc_state *a_state;  /* association descriptor (input)*/
   prin_state  *p_state;  /* state of the principal(input) */
{
   char  auth_init_msg[MAX_MSG_SZ];    /* krb_rd_req message */
   int   auth_init_length;      /* length of the above */
   KTEXT_ST  krb_txt;/* input to krb_rd_req,krb_sendmutual*/
   Key_scheduleschedule;    /* key schedule for session key */

   char  return_buf[MAX_MSG_SZ]; /*status buffer from client */
   int   return_buf_len;         /* length of the above */
   char  return_msg[MAX_MSG_SZ]; /* status message from client*/
   char  return_msg_len;         /* length of the above */
   char  status[MAX_MSG_SZ];     /* status sent by client */
   int   result;

22
   if((auth_init_length = read_msg(a_state->comm_sock,
         auth_init_msg, MAX_MSG_SZ)) == FAILURE)
   {
      fprintf(fio_log, "Bad read_msg call.\n");
      return(FAILURE);
   }

   krb_txt.length = auth_init_length;

   bcopy(auth_init_msg, krb_txt.dat, krb_txt.length);

23
   if((result = krb_rd_req(&krb_txt, p_state->primary_name,
     p_state->instance,
     (u_long)a_state->f_addr.sin_addr.s_addr,
     &a_state->authen_data, p_state->srvtab_file)) != RD_AP_OK)
   {
      fprintf(fio_log, "Bad krb_rd_req call.\n");

      if(des_key_sched(a_state->authen_data.session, schedule))
      {
         fprintf(fio_log, "Bad des_key_schedule call.\n");
         return(FAILURE);
      }

      if((result = krb_sendmutual(KOPT_NORDWR, &krb_txt,
```

Example 6-6: (continued)

```
            KFAILURE, (int)NULL, &a_state->f_addr,
            &a_state->l_addr, &a_state->authen_data,
            schedule)) != KSUCCESS)
      {
         fprintf(fio_log, "Bad krb_sendmutual call.\n");
         return(FAILURE);
      }
   }
   else
   {
24
      if(des_key_sched(a_state->authen_data.session, schedule))
      {
         fprintf(fio_log, "Bad des_key_schedule call.\n");
         return(FAILURE);
      }

25
      if((result = krb_sendmutual(KOPT_NORDWR, &krb_txt,
            KSUCCESS, (int)NULL, &a_state->f_addr,
            &a_state->l_addr, &a_state->authen_data,
            schedule)) != KSUCCESS)
      {
         fprintf(fio_log, "Bad krb_sendmutual call.\n");
         return(FAILURE);
      }
   }

26
   if(write_msg(a_state->comm_sock, krb_txt.dat,
        krb_txt.length) != OK)
   {
      fprintf(fio_log, "Bad write_msg call.\n");
      return(FAILURE);
   }


27
   if((return_buf_len = read_msg(a_state->comm_sock, return_buf,
         sizeof(return_buf))) == FAILURE)
   {
      fprintf(fio_log, "Bad read_msg call.\n");
      return(FAILURE);
   }

28
   if((result = read_authen_msg(a_state->authen_data.session,
       &a_state->f_addr, &a_state->l_addr, return_buf,
       return_buf_len, return_msg, &return_msg_len)) == FAILURE)
   {
      fprintf(fio_log, "Intruder alert, bad safe msg.\n");
      return(FAILURE);
   }


29
   if((result = read_return_msg(return_msg, return_msg_len,
         status)) != OK)
   {
      fprintf(fio_log, "Bad return message\n");
      return(FAILURE);
   }
```

Example 6-6: (continued)

```
   if(!strcmp("FAILURE", status))
   {
      return(FAILURE);
   }

   return(OK);
}
```

6.2.10 The h_server.c File

The `h_server.c` file is one of four files that contain code for both the high-level and low-level servers. Example 6-7 consists of `h_server.c`:

Example 6-7: The h_server.c Routine

```
/* h_server.c contains server code that is only used by the
   high-level server.
*/

#include "server.h"

extern FILE  *fio_log;   /* file descriptor of the log file */

/* authen_assoc attempts to authenticate the association
   between the high-level client and the high-level server
   for the server.  It returns FAILURE if the authentication
   fails and OK, otherwise.
*/
30
authen_assoc(p_state, a_state)
   assoc_state *a_state;  /* association descriptor (input) */
   prin_state  *p_state;  /* state of the principal(input) */
{
   KTEXT_ST  ticket;  /* ticket/authenticator pair from client*/
   Key_scheduleschedule;  /* key schedule for session key */

   char  return_buf[MAX_MSG_SZ]; /*status buffer from client */
   int   return_buf_len;         /* length of the above */
   char  return_msg[MAX_MSG_SZ]; /* status message from client*/
   char  return_msg_len;         /* length of the above */
   char  status[MAX_MSG_SZ];     /* status sent by client */
   int   result;

31
   if((result = krb_recvauth((long)KOPT_DO_MUTUAL,
               a_state->comm_sock, &ticket,
               p_state->primary_name, p_state->instance,
               &a_state->f_addr, &a_state->l_addr,
               &a_state->authen_data, p_state->srvtab_file,
               schedule, a_state->version)) != KSUCCESS)
   {
      fprintf(fio_log, "Bad krb_recvauth call.\n");
      return(result);
   }

32
   if((return_buf_len = read_msg(a_state->comm_sock,
         return_buf, sizeof(return_buf))) == FAILURE)
   {
      fprintf(fio_log, "Bad read_msg call.\n");
      return(FAILURE);
```

Example 6-7: (continued)

```
    }

33
    if((result = read_authen_msg(a_state->authen_data.session,
        &a_state->f_addr, &a_state->l_addr, return_buf,
        return_buf_len, return_msg, &return_msg_len)) == FAILURE)
    {
       fprintf(fio_log, "Intruder alert, bad safe msg.\n");
       return(FAILURE);
    }

34
    if((result = read_return_msg(return_msg, return_msg_len,
          status)) != OK)
    {
       fprintf(fio_log, "Bad return message\n");
       return(FAILURE);
    }

    if(!strcmp("FAILURE", status))
    {
       return(FAILURE);
    }

    return(OK);
}
```

6.2.11 The server.c File

The `server.c` file is one of four files that contain code for both the high-level and low-level servers. Example 6-8 consists of `server.c`:

Example 6-8: The server.c Routine

```
/* server.c contains the code that forms the basis of the low-
   and high-level servers.
*/

#include "server.h"

char  log_file[FILE_NAME_SZ];        /* the name of the log file */
FILE  *fio_log;              /* file descriptor for the log file */
char  acl_file[FILE_NAME_SZ];  /* filename of access control list*/
FILE  *fio_acl;           /* the file descriptor of the ACL file */


/* init_krb initializes the kerberos libraries for the server.
   If the server is successful, init_krb returns OK. Otherwise,
   the server exits.
 */
35
init_krb(p_state)
   prin_state  *p_state;   /* state of the principal(input) */
{
   char        hostname[HOST_NAME_SZ];  /* name of the local host */
   char        *char_ptr;
   int         result;

36
   if((result = gethostname(hostname, sizeof(hostname))) < 0)
   {
```

Example 6-8: (continued)

```
      fprintf(fio_log, "gethostname failure\n");
      exit(FAILURE);
   }

37
   char_ptr = krb_get_phost(hostname);
   strcpy(p_state->instance, char_ptr);

38
   krb_get_lrealm(p_state->realm, 0);
   if(p_state->srvtab_file[0] != '\0')
      krb_set_srvtab_string(p_state->srvtab_file);
39

   if(p_state->tkt_file[0] != '\0')
      krb_set_tkt_string(p_state->tkt_file);

40
   return(OK);
}

/* read_command reads the command "on-the-wire" protocol.
   If an error occurs, read_command returns FAILURE.
   Otherwise, OK is returned.
 */
read_command(msg, msg_length, command_todo)
   char        *msg;           /* the "on-the-wire" message (input) */
   int         msg_length;     /* length of the message (input) */
   command  *command_todo;   /* command_todo stores the
                                command (output) */
{
   char        *action;    /* pointer to action section of the comm */
   char        *principal; /* principal name in the command */
   u_long  command_length;  /* length of the command */

   bcopy(msg, (char *)&command_length, sizeof(u_long));
   command_length = ntohl(command_length);

   action = msg + sizeof(u_long);

   if(action + strlen(action) > msg + msg_length - 1)
      return(FAILURE);

   strcpy(command_todo->action, action);

   if(!strcmp(action, "end"))
   {
      if(action + strlen(action) != msg + msg_length - 1)
         return(FAILURE);
      else
         return(OK);
   }

   principal =  action + strlen(action) + 1;
   if(principal + strlen(principal) != msg + msg_length - 1)
      return(FAILURE);

   strcpy(command_todo->principal, principal);

   return(OK);
}

/* authorize_assoc determines if the client is authorized to
```

Example 6-8: (continued)

```
   talk to the server.  If the client is authorized
   authorize_assoc returns OK, otherwise, FAILURE.
*/
41
authorize_assoc(acl_file, authen_data)
   char  *acl_file;     /* An access control list (input) */
   AUTH_DAT  *authen_data;   /* name of principal (input) */
{
   char principal[80];  /* principal name in acl format */

   sprintf(principal, "%s.%s@%s", authen_data->pname,
            authen_data->pinst, authen_data->prealm);
42
   if(acl_check(acl_file, principal))
   {
      return(OK);
   }
   else
   {
      return(FAILURE);
   }
}

/* end_assoc reinitializes an a_state structure after an
   association ends.  It returns OK if successful, and
   FAILURE otherwise.
 */
end_assoc(a_state)
   assoc_state  *a_state;   /* association descriptor (input) */
{
   close(a_state->comm_sock);
   a_state->version[0] = '\0';
}

/* service_assoc receives and processes commands from the
   client described by association, a_state.  service_assoc
   returns OK if the association has ended, or FAILURE if
   there was an error.
*/
43
service_assoc(p_state, a_state)
   assoc_state  *a_state; /* association descriptor (input)*/
   prin_state   *p_state; /* state of the principal(input) */
{
   char  safe_msg[MAX_MSG_SZ];   /* the message received */
   int   safe_msg_len;           /* length of the above */
   char  msg[MAX_MSG_SZ];        /* the command sent */
   int   msg_len;                /* length of the above */
   struct command  command_todo; /* the command structure */
   char  return_buf[MAX_MSG_SZ]; /* the return message */
   int  return_len;              /* length of the above */
   int  result;

   while(1)
   {
44
      if((safe_msg_len = read_msg(a_state->comm_sock, safe_msg,
         sizeof(safe_msg))) == FAILURE)
      {
         fprintf(fio_log, "Bad read_msg call.\n");
         return(FAILURE);
      }
```

Example 6-8: (continued)

45

```
if((result = read_authen_msg(
      a_state->authen_data.session,
      &a_state->f_addr, &a_state->l_addr, safe_msg,
      safe_msg_len, msg, &msg_len)) == FAILURE)
{
   fprintf(fio_log, "Intruder alert, bad safe msg.\n");
   continue;
}
```

46

```
if(read_command(msg, msg_len, &command_todo) == FAILURE)
{
   fprintf(fio_log, "Bad command format\n");
   continue;
}
```

47

```
if(!strcmp(command_todo.action, "add"))
{
   if((result = acl_add(acl_file,
      command_todo.principal)) == 0)
   {
```

48

```
      make_return_msg(return_buf, &return_len, OK);
   }
   else
   {
      make_return_msg(return_buf, &return_len, FAILURE);
   }
}
else if(!strcmp(command_todo.action, "delete"))
{
   if((result = acl_delete(acl_file,
      command_todo.principal)) == 0)
   {
      make_return_msg(return_buf, &return_len, OK);
   }
   else
   {
      make_return_msg(return_buf, &return_len, FAILURE);
   }
}
else if(!strcmp(command_todo.action, "check"))
{
   if((result = acl_check(acl_file,
      command_todo.principal)) > 0)
   {
      make_return_msg(return_buf, &return_len, OK);
   }
   else
   {
      make_return_msg(return_buf, &return_len, FAILURE);
   }
}
else if(!strcmp(command_todo.action, "exact_match"))
{
   if((result = acl_exact_match(acl_file,
      command_todo.principal)) > 0)
   {
      make_return_msg(return_buf, &return_len, OK);
   }
```

Example 6-8: (continued)

```
            else
            {
               make_return_msg(return_buf, &return_len, FAILURE);
            }
         }
         else if(!strcmp(command_todo.action, "end"))
            make_return_msg(return_buf, &return_len, OK);
         else
            make_return_msg(return_buf, &return_len, FAILURE);

49
         if((result = make_authen_msg(a_state->authen_data.session,
            &a_state->f_addr, &a_state->l_addr,
            return_buf, &return_len)) != OK)
            continue;

50
         if(write_msg(a_state->comm_sock, return_buf,
               return_len) != OK)
         {
            fprintf(fio_log, "Bad write_msg call.\n");
            return(FAILURE);
         }

51
         if(!strcmp(command_todo.action, "end"))
            break;
      }

      return(OK);
}


/* begin_assoc attempt to begin communicating with a client
   as well as authenticate the client and make sure the client
   is authorized to talk to the server.  begin_assoc returns
   FAILURE if an error occurs or OK if an association has been
   established.
*/
52
int begin_assoc(p_state, a_state)
   assoc_state  *a_state;  /* association descriptor (input) */
   prin_state   *p_state;  /* state of the principal(input) */
{
   int read_mask;    /* the read mask for select */
   int result_mask; /* the mask of fd ready */
   int comm_not_ready = 1; /* 0 if client wants to communicate */
   int continue_begin = 0; /* 1 if an error has occured */
   int num_ready;    /* the number of file descriptors ready */
   char return_buf[MAX_MSG_SZ];    /* the message returned */
   int return_len;  /* the length of the above */
   int len_addr;
   int result;


53
   while(comm_not_ready)
   {
      read_mask = 1 << p_state->stream_sock;

      if((num_ready = select(ALL_FDS, &read_mask,
         (int *)NULL, (int *)NULL,
         (struct timeval *)NULL)) < 0)
```

Example 6-8: (continued)

```
{
   if(errno == EINTR)
   {
      fprintf(fio_log, "interrupted sys call\n");
      continue;
   }
   else
   {
      fprintf(fio_log, "select error\n");
      return(FAILURE);
   }
}

len_addr = sizeof(struct sockaddr);

54

if((a_state->comm_sock = accept(p_state->stream_sock,
   (struct sockaddr *)&a_state->f_addr,
   &len_addr)) < 0)
{
   fprintf(fio_log, "accept failed\n");
   continue;
}

len_addr = sizeof(struct sockaddr);

55

if((result = getsockname(a_state->comm_sock,
  (struct sockaddr *)&a_state->l_addr, &len_addr)) < 0 ||
  len_addr != sizeof(struct sockaddr_in))
{
   fprintf(fio_log, "getsockname failure\n");
   continue;
}

56

if((result = authen_assoc(p_state, a_state)) != OK)
{
   fprintf(fio_log, "authen_assoc error\n");
   continue;
}

57

if(authorize_assoc(acl_file,
                   &a_state->authen_data) == FAILURE)
{
 fprintf(fio_log,
   "principal: %s instance: %s realm:%s not authorized\n",
      a_state->authen_data.pname,
      a_state->authen_data.pinst,
      a_state->authen_data.prealm);
 continue_begin = 1;
 make_return_msg(return_buf, &return_len, FAILURE);
}
else
{
58

   make_return_msg(return_buf, &return_len, OK);
}

59

if((result = make_authen_msg(a_state->authen_data.session,
   &a_state->f_addr, &a_state->l_addr,
```

Example 6-8: (continued)

```
            return_buf, &return_len)) != OK)
        {
            continue;
        }

60
        if(write_msg(a_state->comm_sock, return_buf,
                    return_len) != OK)
        {
            fprintf(fio_log, "Bad write_msg call.\n");
            continue;
        }

        if(continue_begin)
        {
            continue_begin = 0;
            continue;
        }

        comm_not_ready = 0;
    }

    return(OK);
}

/* init_comm initializes the socket that the server will use
   to accept connections from clients.  The server exits if
   an error occurs and returns OK otherwise.
*/
61
int init_comm(p_state)
    prin_state  *p_state;   /* state of the principal(input) */
{
    struct servent  *server;  /* server descriptor */
    int  on = 1;
    int  result;

    if((server = getservbyname(SERVER_NAME, "tcp")) < 0)
    {
        fprintf(fio_log, "getservbyname failure\n");
        exit(FAILURE);
    }

    if((p_state->stream_sock = socket(AF_INET,
                                      SOCK_STREAM, 0)) < 0)
    {
        fprintf(fio_log, "socket call failure\n");
        exit(FAILURE);
    }

    p_state->stream_addr.sin_family = AF_INET;
    p_state->stream_addr.sin_addr.s_addr = INADDR_ANY;
    p_state->stream_addr.sin_port = htons(server->s_port);


    (void)setsockopt(p_state->stream_sock,
        SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on) );

    if((result = bind(p_state->stream_sock,
            (struct sockaddr *)&p_state->stream_addr,
            sizeof(struct sockaddr))) < 0)
    {
        fprintf(fio_log, "bind call failure\n");
```

Example 6-8: (continued)

```
      exit(FAILURE);
   }

   (void) listen(p_state->stream_sock, 5);

   return(OK);
}

/* init_files opens for access all of the files that the
   server needs in order to run.  init_files exits if an
   error occurs.  Otherwise, it returns, OK.
*/
62
init_files()
{
   int  result;
   struct stat  statistics;

   if((fio_log = fopen(log_file, "a+")) == (FILE *)NULL)
   {
      fprintf(stdout, "LOG file will not open\n");
      exit(FAILURE);
   }

   if((result = stat(acl_file, &statistics)) == -1)
   {
      fprintf(fio_log, "ACL file does not exist\n");
      exit(FAILURE);
   }
   return(OK);
}

usage()
{
   fprintf(stdout, "server -a acl_file -l log_file
      -s srvtab_file -p primary_name -t tkt_file");

}

/* Both the low- and high-level servers begin executing in
   main.  The server code is designed to receive allow
   clients to access the server's access control list file.
   All communication between the clients and the server is
   Kerberos-authenticated.
*/
63
main(argc, argv)
   int argc; /* number of arguments to the server (input) */
   char **argv;/* the arguments to the server (input) */
{
   extern char  *optarg;       /* the name of a switch */
   extern int  optind, getopt();   /* argument routines */
   int  c;                     /* the switch name */
   prin_state   p_state;       /* state of the principal */
   assoc_state  a_state;       /* an association descriptor */

64
   strcpy(acl_file, "./acl_file");
   strcpy(log_file, "./srv_log_file");
   strcpy(p_state.primary_name, SERVER_NAME);
   p_state.tkt_file[0] = '\0';
   p_state.srvtab_file[0] = '\0';
```

Example 6-8: (continued)

```
   while ((c = getopt(argc, argv, "a:l:s:p:t:")) != EOF)
   {
      switch(c) {
65
      case 'a':
      {
         strcpy(acl_file, optarg);
         break;
      }
      case 'l':
      {
         strcpy(log_file, optarg);
         break;
      }
      case 's':
      {
         strcpy(p_state.srvtab_file, optarg);
         break;
      }
      case 'p':
      {
         strcpy(p_state.primary_name, optarg);
         break;
      }
      case 't':
      {
         strcpy(p_state.tkt_file, optarg);
         break;
      }
      default:
      {
         usage();
      }
      }
   }

66
   init_files();

67
   init_comm(&p_state);

68
   init_krb(&p_state);

   for(;;)
   {
69
      if(begin_assoc(&p_state, &a_state) == FAILURE)
      {
         fprintf(fio_log, "Begin association error\n");
         exit(FAILURE);
      }


70
      if(service_assoc(&p_state, &a_state) == FAILURE)
      {
         fprintf(fio_log, "Service association error\n");
         exit(FAILURE);
      }
```

Example 6-8: (continued)

```
71
        end_assoc(&a_state);
    }
}
```

6.2.12 The client.h File

The `client.h` file is one of four files that contain code for the high-level and low-level clients. Example 6-9 consists of `client.h`:

Example 6-9: The client.h Routine

```
/* client.h includes the structure definitions used by both
   the high- and low-level clients.
*/

#include "all.h"

#define SERVER_NAME"krb_sample_cli"

/* The command structure stores the action performed by
   the command as well as the principal name the action
   should be performed upon.
 */
72
struct command {
   char        action[80];/* command action */
   char        principal[MAX_K_NAME_SZ]; /* entire principal name */
   char        primary_name[ANAME_SZ];/* primary name of principal */
   char        instance[INST_SZ];/* instance of the principal */
   char        realm[REALM_SZ];/* realm of the principal */
};
typedef struct command command;

/* The prin_state structure stores the client state associated
   only with the running client.
*/
73
struct prin_state {
   char        primary_name[SNAME_SZ];/* primary name of client */
   char        instance[INST_SZ];/* instance name of client */
   char        realm[REALM_SZ];/* realm name of client */
   char        version[KRB_SENDAUTH_VLEN];  /* version of client */

   char        srvtab_file[FILE_NAME_SZ]; /* srvtab file of client */
   char        tkt_file[FILE_NAME_SZ];    /* ticket file of client */
};
typedef struct prin_state prin_state;

/* The assoc_state structure stores the client state
   related to an association with a server.
*/
74
struct assoc_state {
   int         comm_sock;   /* socket used to talk with server. */
   char        version[KRB_SENDAUTH_VLEN];  /*version of server */

   struct sockaddr_in  l_addr;   /* address of the client */
   struct sockaddr_in  f_addr;   /* address of the server */

   char        f_primary_name[SNAME_SZ]; /* primary name of server */
```

Example 6-9: (continued)

```
   char        f_instance[INST_SZ]; /* instance name of the server */
   char        f_realm[REALM_SZ];   /* realm name of the server */

   CREDENTIALS  cred;   /* cred structure for the server */
};
typedef struct assoc_state assoc_state;
```

6.2.13 The l_client.c File

The `l_client.c` file is one of four files that contain code for both the high-level and low-level clients. Example 6-10 consists of `l_client.c`:

Example 6-10: The l_client.c Routine

```
/* l_client.c contains client code that is only used by
   the low-level client.
*/

#include "client.h"

char   log_file[FILE_NAME_SZ];       /* name of log file */
extern FILE  *fio_log;    /* file descriptor of log file */
char   comm_file[FILE_NAME_SZ];  /* name of command file */
extern FILE  *fio_comm; /*file descriptor of command file*/
char   out_file[FILE_NAME_SZ];    /* name of output file */
extern FILE  *fio_out;  /* file descriptor of output file*/

/* authen_assoc attempts to authenticate the association
   between the low-level client and the low-level server
   for the client.  It returns FAILURE if the authentication
   fails and OK, otherwise.
*/
authen_assoc(p_state, a_state)
   assoc_state  *a_state; /* association descriptor (input)*/
   prin_state   *p_state; /* state of the principal(input) */
{
   char  auth_init_msg[MAX_MSG_SZ]; /* sendmutual message */
   KTEXT_ST  krb_txt;  /* contains the krb_mk_req message */
   struct timeval  time;   /* current time */
   struct timezone zone;   /* time zone */
   MSG_DAT  msg_data; /* sendmutual message sent by server*/
   Key_schedule schedule; /* key schedule for session key */

   char return_buf[MAX_MSG_SZ]; /* result message to send */
   int  return_len;/* length of the above */
   int  return_val;/* value to return to caller  */
   u_long  checksum;/* used in krb_mk_req message */
   int   length;
   int   result;

75
   if(gettimeofday(&time, &zone))
   {
      fprintf(fio_log, "bad gettimeofday call.\n");
      return(FAILURE);
   }

   checksum = (u_long)time.tv_usec;

76
   if((result = krb_mk_req(&krb_txt, a_state->f_primary_name,
```

Example 6-10: (continued)

```
      a_state->f_instance, a_state->f_realm,
      checksum)) != KSUCCESS)
   {
      fprintf(fio_log, "Bad mk_req call, %d\n", result);
      return(FAILURE);
   }

77
   if((length = write_msg(a_state->comm_sock, krb_txt.dat,
      krb_txt.length)) == FAILURE)
   {
      fprintf(fio_log, "Bad write_msg call.\n");
      return(FAILURE);
   }

78
   if((krb_txt.length = read_msg(a_state->comm_sock,
         krb_txt.dat, sizeof(auth_init_msg))) == FAILURE)
   {
      fprintf(fio_log, "Bad read_msg call.\n");
      return(FAILURE);
   }

   return_val = FAILURE;

79
   if((result = krb_get_cred(a_state->f_primary_name,
       a_state->f_instance,
       a_state->f_realm, &a_state->cred)) != GC_OK)
   {
      fprintf(fio_log, "Bad krb_get_cred call.\n");
      make_return_msg(return_buf, &return_len, FAILURE);
   }
   else if(des_key_sched(a_state->cred.session, schedule))
80

   {
      fprintf(fio_log, "Bad des_key_schedule call.\n");
      make_return_msg(return_buf, &return_len, FAILURE);
   }
   else if((result = krb_recvmutual(KOPT_NORDWR,
               (int)NULL, checksum,
               &krb_txt, &msg_data, &a_state->cred,
               schedule, &a_state->l_addr,
               &a_state->f_addr)) != KSUCCESS)
81
   {
      fprintf(fio_log, "Bad krb_recvmutual call,
               %s.\n", result);
      make_return_msg(return_buf, &return_len, FAILURE);
   }
   else
   {
      return_val = OK;
82
      make_return_msg(return_buf, &return_len, OK);
   }

83
   if((result = make_authen_msg(a_state->cred.session,
      &a_state->f_addr, &a_state->l_addr,
      return_buf, &return_len)) != OK)
   {
```

Example 6-10: (continued)

```
      return(FAILURE);
   }

84
   if(write_msg(a_state->comm_sock, return_buf,
        return_len) != OK)
   {
      fprintf(fio_log, "Bad write_msg call.\n");
      return(FAILURE);
   }

   return(return_val);
}


/* init_krb initializes the Kerberos libraries for the
   low-level client.  If the low-level client is successful,
   init_krb returns OK. Otherwise, the client exits.
 */
85
init_krb(p_state)
   prin_state  *p_state; /* state of the principal(input) */
{
   char         hostname[HOST_NAME_SZ];/* name of the local host */
   char         *srvtab_file;  /* pointer to the srvtab file name */
   char         *tkt_file;     /* pointer to the ticket file name */
   char         *char_ptr;
   int          result;

86
   if((result = gethostname(hostname, sizeof(hostname))) < 0)
   {
      fprintf(fio_log, "gethostname failure\n");
      exit(FAILURE);
   }
87
   char_ptr = krb_get_phost(hostname);
   strcpy(p_state->instance, char_ptr);

88
   krb_get_lrealm(p_state->realm, 0);

   if(p_state->srvtab_file[0] != '\0')
      srvtab_file = p_state->srvtab_file;
   else
      srvtab_file = (char *)NULL;

   if(p_state->tkt_file[0] != '\0')
      tkt_file = p_state->tkt_file;
   else
      tkt_file = (char *)NULL;

89
   if((result = krb_svc_init(p_state->primary_name,
       p_state->instance,
       (char *)NULL, (int) TKT_LIFETIME, srvtab_file,
       tkt_file)) != KSUCCESS)
   {
      fprintf(fio_log, "krb_svc_init failure\n");
      exit(FAILURE);
   }

}
```

Example 6-10: (continued)

```
/* init_files opens for access all of the files that the
   low-level client needs to run.  init_files exits if an
   error occurs.  Otherwise, it returns, OK.
*/
90
init_files()
{
   int result;

   if((fio_log = fopen(log_file, "a+")) == (FILE *)NULL)
   {
      fprintf(stderr, "LOG file will not open\n");
      exit(FAILURE);
   }

   if((fio_comm = fopen(comm_file, "r")) == (FILE *)NULL)
   {
      fprintf(fio_log, "COMM file will not initialize\n");
      exit(FAILURE);
   }

   if((fio_out = fopen(out_file, "a+")) == (FILE *)NULL)
   {
      fprintf(fio_log, "OUTPUT file will not initialize\n");
      exit(FAILURE);
   }
}

/* usage prints a usage message */
usage()
{
   fprintf(stdout, "client -c command_file -o output_file
      -l log_file -s srvtab_file -t tkt_file -p primary_name");

}

/* read_args reads all of the arguments to the low-level
   client command.
*/
91
read_args(argc, argv, p_state)
   int argc;  /* number of arguments to the server (input) */
   char **argv;     /* the arguments to the server (input) */
   prin_state  *p_state;  /* state of the principal(input) */
{
   extern char  *optarg;   /* the argument to a switch */
   extern int   optind, getopt();  /* argument routine */
   int c;                          /* switch character */

92
   strcpy(comm_file, "./comm_file");
   strcpy(log_file, "./cli_log_file");
   strcpy(out_file, "./cli_out_file");
   strcpy(p_state->primary_name, SERVER_NAME);
   p_state->tkt_file[0] = '\0';
   p_state->srvtab_file[0] = '\0';

   while ((c = getopt(argc, argv, "c:o:l:s:p:t:")) != EOF)
   {
93
      switch(c) {
      case 'c':
```

Example 6-10: (continued)

```
        {
           strcpy(comm_file, optarg);
           break;
        }
        case 'o':
        {
           strcpy(out_file, optarg);
           break;
        }
        case 'l':
        {
           strcpy(log_file, optarg);
           break;
        }
        case 's':
        {
           strcpy(p_state->srvtab_file, optarg);
           break;
        }
        case 't':
        {
           strcpy(p_state->tkt_file, optarg);
           break;
        }
        case 'p':
        {
           strcpy(p_state->primary_name, optarg);
           break;
        }
        default:
        {
           usage();
        }
        }
    }
}
```

6.2.14 The h_client.c File

The `h_client.c` file is one of four files that contain code for both the high-level and low-level clients. Example 6-11 consists of `h_client.c`:

Example 6-11: The h_client.c Routine

```
/* h_client.c contains client code that is only used by
   the high-level client.
*/

#include "client.h"

extern FILE  *fio_log;/* file descriptor of log file */
extern FILE  *fio_comm;/* file descriptor of command file*/
extern FILE  *fio_out;/* file descriptor of output file*/

/* authen_assoc attempts to authenticate the association
   between the high-level client and the high-level server
   for the client.  It returns FAILURE if the authentication
   fails and OK, otherwise.
*/
94
authen_assoc(p_state, a_state)
```

Example 6-11: (continued)

```
   assoc_state  *a_state;  /* association descriptor (input) */
   prin_state   *p_state;  /* state of the principal(input) */
{
   KTEXT_ST  ticket; /*ticket authenticator pair sent to server*/
   struct  timeval  time; /* current time */
   struct  timezone zone; /* time zone */
   MSG_DAT  msg_data;  /* message sent by server to client */

   Key_schedule  schedule;  /* key schedule for session key */
   u_long  checksum;        /* used in krb_sendauth message */
   char return_buf[MAX_MSG_SZ];   /* result message to send */
   int  return_len;   /* length of the above */
   int  return_val;   /* value to return to caller */
   int  result;

   if(gettimeofday(&time, &zone))
   {
      fprintf(fio_log, "bad gettimeofday call.\n");
      return(FAILURE);
   }

   checksum = (u_long)time.tv_usec;

95
   if((result = krb_sendauth((long)KOPT_DO_MUTUAL,
       a_state->comm_sock,
       &ticket, a_state->f_primary_name, a_state->f_instance,
       a_state->f_realm, checksum, &msg_data,
       &a_state->cred, schedule, &a_state->l_addr,
       &a_state->f_addr, "first_v")) != KSUCCESS)
   {
      fprintf(fio_log, "krb_sendauth failure, %d\n", result);
      return_val = FAILURE;
      make_return_msg(return_buf, &return_len, FAILURE);
   }
   else
   {
      return_val = OK;
96
      make_return_msg(return_buf, &return_len, OK);
   }

97
   if((result = make_authen_msg(a_state->cred.session,
      &a_state->f_addr, &a_state->l_addr,
      return_buf, &return_len)) != OK)
   {
      return(FAILURE);
   }

98
   if(write_msg(a_state->comm_sock, return_buf,
        return_len) != OK)
   {
      fprintf(fio_log, "Bad write_msg call.\n");
      return(FAILURE);
   }

   return(return_val);
}


/* init_krb initializes the Kerberos libraries for the
```

Example 6-11: (continued)

```
   high-level client.   If the high-level client is
   successfull, init_krb returns OK.  Otherwise, the
   client exits.
*/
99
init_krb(p_state)
   prin_state  *p_state; /* state of the principal(input) */
{
   int          result;

100
   fprintf(fio_out,"Primary Name:");
   if(fgets(p_state->primary_name, ANAME_SZ, fio_comm) == NULL)
   {
      return(FAILURE);
   }
   p_state->primary_name[strlen(p_state->primary_name) - 1] = '\0';

   fprintf(fio_out,"Instance:");
   if(fgets(p_state->instance, INST_SZ, fio_comm) == NULL)
   {
      return(FAILURE);
   }
   p_state->instance[strlen(p_state->instance) - 1] = '\0';

   fprintf(fio_out,"Realm:");
   if(fgets(p_state->realm, REALM_SZ, fio_comm) == NULL)
   {
      return(FAILURE);
   }
   p_state->realm[strlen(p_state->realm) - 1] = '\0';

101
   if(p_state->tkt_file[0] != '\0')
      krb_set_tkt_string(p_state->tkt_file);

102
   if((result = krb_get_pw_in_tkt(p_state->primary_name,
       p_state->instance,
       p_state->realm, "krbtgt", p_state->realm,
       (int) TKT_LIFETIME, (char *)NULL)) != KSUCCESS)
   {
      fprintf(fio_log, "krb_get_pw_in_tkt failure\n");
      exit(FAILURE);
   }
}

/* init_files opens for access all of the files that
   the high-level client needs in order to run.
   init_files exits if an error occurs.  Otherwise,
   it returns, OK.
*/
103
init_files()
{
   fio_log = stderr;
   fio_out = stdout;
   fio_comm = stdin;
}

/* usage prints a usage message */
usage()
{
```

Example 6-11: (continued)

```
   fprintf(stdout, "client  -t tkt_file\n");
}


/* read_args reads all of the arguments to the high-level
   client command.
*/
104
read_args(argc, argv, p_state)
   int argc; /* number of arguments to the server (input) */
   char **argv;    /* the arguments to the server (input) */
   prin_state  *p_state; /* state of the principal(input) */
{
   extern char *optarg;        /* argument to a switch */
   extern int  optind, getopt();   /* argument routine */
   int c;                          /* switch character */

105
   p_state->tkt_file[0] = '\0';

   while ((c = getopt(argc, argv, "t:")) != EOF)
   {
106
      switch(c) {
      case 't':
      {
         strcpy(p_state->tkt_file, optarg);
         break;
      }
      default:
      {
         usage();
      }
      }
   }

   return(OK);
}
```

6.2.15 The client.c File

The `client.c` file is one of four files that contain code for both the high-level and low-level clients. Example 6-12 consists of `client.c`:

Example 6-12: The client.c Routine

```
/* client.c contains the code that forms the basis of the
   low- and high-level clients.
*/
#include "client.h"

FILE *fio_log;     /* file descriptor for the log file */
FILE *fio_comm;    /* file descriptor of the command file */
FILE *fio_out;     /* file descriptor of the ouput file */

int  line_num = 0; /* the number of the line in the command
                      file being read */

/* make_command creates the "on-the-wire" version of the
   command described in comm_todo.  The command is placed
   in msg.  make_command returns OK if the command is made
```

Example 6-12: (continued)

```
   correctly.
*/
make_command(comm_todo, msg, msg_length)
   command  *comm_todo;     /* command to convert (input) */
   char  *msg; /* command in "on-the-wire" format (output)*/
   int   *msg_length;     /* length of the above (output) */
{
   char         *action;   /* the action described by the command */
   char         *principal;       /* the principal in the command */
   u_char  *comm_length;
   u_long  long_val;

   comm_length = (u_char *)msg;

   action = (char *)comm_length + sizeof(u_long);
   strcpy(action, comm_todo->action);

   if(!strcmp(action, "end"))
   {
      long_val = (action + strlen(action) + 1) -
                 (msg + sizeof(u_long));
      *msg_length = (int)long_val + sizeof(u_long);
      long_val = htonl(long_val);
      bcopy((char *)&long_val, (char *)comm_length,
            sizeof(u_long));

      return(OK);
   }

   principal = action + strlen(action) + 1;
   strcpy(principal, comm_todo->principal);

   long_val = (principal + strlen(principal) + 1) -
              (msg + sizeof(u_long));
   *msg_length = (int)long_val + sizeof(u_long);
   long_val = htonl(long_val);
   bcopy((char *)&long_val, (char *)comm_length,
         sizeof(u_long));

   return(OK);
}

/* end_assoc reinitializes an a_state structure after an
   association ends.  It returns OK if successful, FAILURE
   otherwise.
 */
end_assoc(a_state)
   assoc_state  *a_state; /* association descriptor (input) */
{
   close(a_state->comm_sock);
   a_state->version[0] = '\0';

   if((a_state->comm_sock = socket(AF_INET,
                                   SOCK_STREAM, 0)) < 0)
   {
      fprintf(fio_log, "socket call failure\n");
      return(FAILURE);
   }

   return(OK);
}

/* read_comm_file reads a command from the command file
```

Example 6-12: (continued)

```
   and converts it to a form that can be stored in a
   command structure.  read_comm_file returns OK if it
   succeeds, FAILURE otherwise.
*/
107
int read_comm_file(comm)
   command  *comm; /* command read from the file (output) */
{
   char         line[MAX_COMM_LINE_SZ];/* line read from the file */
   int  line_pos;   /* the current position in line */
   int  action_pos; /* position of the action in line */
   int  prin_pos;   /* position of the principal in line */
   int  result;

   line_num++;

   comm->action[0] = '\0';
   comm->principal[0] = '\0';
   comm->primary_name[0] = '\0';
   comm->instance[0] = '\0';
   comm->realm[0] = '\0';

   if(fgets(line, (int)MAX_COMM_LINE_SZ, fio_comm) == NULL)
   {
      return(FAILURE);
   }

   for(line_pos = 0; line[line_pos] ==
       '\t' || line[line_pos] == ' ';
       line_pos++);

   for(action_pos = 0; line[line_pos] != '\t' &&
       line[line_pos] != ' '
       && line[line_pos] != '\0' && line[line_pos] != '\n';
       line_pos++, action_pos++)
   {
      comm->action[action_pos] = line[line_pos];
   }

   comm->action[action_pos] = '\0';

   if(!strcmp(comm->action, "end"))
   {
      return(OK);
   }

   for(; line[line_pos] == '\t' || line[line_pos] ==
         ' '; line_pos++);

   for(prin_pos = 0; line[line_pos] !=
       '\t' && line[line_pos] != ' '
       && line[line_pos] != '\0' && line[line_pos] != '\n';
       line_pos++, prin_pos++)
   {
      comm->principal[prin_pos] = line[line_pos];
   }

   comm->principal[prin_pos] = '\0';

108
   if((result = kname_parse(comm->primary_name, comm->instance,
      comm->realm, comm->principal)) != KSUCCESS)
   {
```

Example 6-12: (continued)

```
        fprintf(fio_log, "Line %d: Bad principal name\n",line_num);
        return(FAILURE);
    }
    return(OK);
}

/* service_assoc reads commands from the command file and
   sends the commands to the server described by
   association, a_state.
   service_assoc returns OK if the association has ended,
   FAILURE if there was an error, or STOP if the command
   file has no more commands.
*/
109
service_assoc(p_state, a_state)
    assoc_state  *a_state; /* association descriptor (input) */
    prin_state   *p_state; /* state of the principal(input)  */
{
    char safe_msg[MAX_MSG_SZ];  /* the message to send  */
    int  safe_msg_len;          /* length of the above */
    char return_buf[MAX_MSG_SZ];  /* the return message */
    int  return_len;        /* length of the above */
    char msg[MAX_MSG_SZ];   /* the answer sent */
    int  msg_len;           /* length of the above */
    command comm_todo;      /* the command to send */
    char status[MAX_COMM_LINE_SZ];  /* status of the answer */
    int  result;

    while(1)
    {
110
        if((result = read_comm_file(&comm_todo)) == FAILURE)
        {
            return(STOP);
        }

111
        if((result = make_command(&comm_todo, safe_msg,
            &safe_msg_len))!= OK)
        {
            fprintf(fio_log, "Line: %d, Error formatting command\n",
                    line_num);
            continue;
        }

112
        if((result = make_authen_msg(a_state->cred.session,
            &a_state->f_addr, &a_state->l_addr,
            safe_msg, &safe_msg_len)) == FAILURE)
        {
            fprintf(fio_log, "Line: %d, Error creating
                    authenticated command\n", line_num);
            continue;
        }

113
        if(write_msg(a_state->comm_sock, safe_msg,
                     safe_msg_len) != OK)
        {
            fprintf(fio_log, "Bad write_msg call.\n");
            return(FAILURE);
        }
```

Example 6-12: (continued)

114
```
      if((return_len = read_msg(a_state->comm_sock, return_buf,
         sizeof(return_buf))) == FAILURE)
      {
         fprintf(fio_log, "Bad read_msg call.\n");
         return(FAILURE);
      }
```

115
```
      if((result = read_authen_msg(
         a_state->cred.session,
         &a_state->f_addr, &a_state->l_addr,
         return_buf, return_len, msg, &msg_len)) != OK)
      {
         fprintf(fio_log, "Intruder alert, bad safe msg.\n");
         continue;
      }
```

116
```
      if((result = read_return_msg(msg, msg_len, status)) != OK)
      {
         fprintf(fio_log, "Bad return message\n");
         continue;
      }
```

117
```
      fprintf(fio_out, "Line: %d, '%s %s' %s\n", line_num,
         comm_todo.action, comm_todo.principal, status);
      fflush(fio_out);
```

118
```
      if(!strcmp(comm_todo.action, "end"))
      {
         break;
      }
   }

   return(OK);
}

/* begin_assoc attempts to begin communicating with a server
   as well as authenticate the server and make sure the client
   is authorized to talk to the server.  begin_assoc returns
   FAILURE if an error occurs, OK if an association has been
   established, or STOP if there are no more commands in the
   command file.
*/
```
119
```
int begin_assoc(p_state, a_state)
   assoc_state *a_state;  /* association descriptor (input) */
   prin_state  *p_state;  /* state of the principal(input)  */
{
   command  comm;      /* the command read from the file */
   int  no_assoc = 1;     /* 1 if no associaction exists */
   struct servent *server;   /* server entry in /etc/services */
   struct hostent *foreign_host; /*name of server's local host*/

   char  primary_name[ANAME_SZ];    /* server's primary name */
   char  instance_name[INST_SZ];    /* server's instance */
   char  realm_name[REALM_SZ];      /* server's realm */
   char  return_buf[MAX_MSG_SZ];    /* the return message */
   char  return_len;                /* length of the above */
   char  msg[MAX_MSG_SZ];           /* the answer returned */
```

Example 6-12: (continued)

```
   char  msg_len;                    /* length of the above */
   char  status[MAX_COMM_LINE_SZ]; /* status of the answer */
   int   result;
   int   length;

120
   while(no_assoc)
   {
121
      if((result = read_comm_file(&comm)) == FAILURE)
      {
         return(STOP);
      }

122
      if(strcmp(comm.action, "begin"))
      {
         fprintf(fio_log, "An association must begin with
                 the begin command\n");
         continue;
      }

123
      if((foreign_host =gethostbyname(comm.instance)) <
         (struct hostent *)0)
      {
         fprintf(fio_log, "Line %d: gethostbyname failure\n",
                 line_num);
         continue;
      }

      bcopy((char *)foreign_host->h_addr,
            (char *)&a_state->f_addr.sin_addr, sizeof(long));

      a_state->f_addr.sin_family = AF_INET;

124
      if((server = getservbyname(comm.primary_name, "tcp")) <
         (struct servent *) 0)
      {
         fprintf(fio_log, "Line %d: getservbyname failure\n",
                 line_num);
         continue;
      }


      a_state->f_addr.sin_port = htons(server->s_port);

125
      if((result = connect(a_state->comm_sock,
         (struct sockaddr *)&a_state->f_addr,
         sizeof(struct sockaddr))) < 0)
      {
         fprintf(fio_log, "Line %d: connect failure\n",
                 line_num);
         continue;
      }

      length = sizeof(struct sockaddr_in);

126
      if((result = getsockname(a_state->comm_sock,
         (struct sockaddr *)&a_state->l_addr, &length)) < 0 ||
```

Example 6-12: (continued)

```
            length != sizeof(struct sockaddr_in))
        {
           fprintf(fio_log, "Line %d: getsockname failure\n",
                   line_num);
           continue;
        }

127
        strcpy(a_state->f_primary_name, comm.primary_name);
        strcpy(a_state->f_instance, comm.instance);
        strcpy(a_state->f_realm, comm.realm);

128
        if((result = authen_assoc(p_state, a_state)) != OK)
        {
           fprintf(fio_log, "authen_assoc error\n");
           continue;
        }

129
        if((return_len = read_msg(a_state->comm_sock, return_buf,
            sizeof(return_buf))) == FAILURE)
        {
           fprintf(fio_log, "Bad read_msg call.\n");
           return(FAILURE);
        }

130
        if((result = read_authen_msg(
            a_state->cred.session,
            &a_state->f_addr, &a_state->l_addr,
            return_buf, return_len, msg, &msg_len)) != OK)
        {
           fprintf(fio_log, "Intruder alert, bad safe msg.\n");
           continue;
        }

131
        if((result = read_return_msg(msg, msg_len, status)) != OK)
        {
           fprintf(fio_log, "Bad return message\n");
           continue;
        }

        if(!strcmp("FAILURE", status))
        {
           fprintf(fio_log, "The Client is not authorized to
                   talk to the server.\n");
           continue;
        }

        fprintf(fio_out, "Line: %d, '%s %s' %s\n", line_num,
                comm.action, comm.principal, "OK");
        fflush(fio_out);

        no_assoc = 0;
     }

     return(OK);
}

/* init_comm initializes the socket that the client will use
   to communicate with the server.  The client exits if an
```

Example 6-12: (continued)

```
   error occurs and returns OK otherwise.
*/
132
int init_comm(a_state)
   assoc_state *a_state; /*an association descriptor (input)*/
{
   if((a_state->comm_sock = socket(AF_INET,
                                   SOCK_STREAM, 0)) < 0)
   {
      fprintf(fio_log, "socket call failure\n");
      exit(FAILURE);
   }
   return(OK);
}

/* Both the low- and high-level clients begin executing in
   main.  The client code is designed to read commands from
   a command file and either begin communicating with a
   server or send that command to a server for processing,
   depending on the type of command.  All communication
   between the client and the server is Kerberos-authenticated.
*/
133
main(argc, argv)
   int argc; /* number of arguments to the server (input) */
   char **argv; /* arguments to the server (input) */
{
   prin_state   p_state;   /* state of the principal */
   assoc_state  a_state;   /* an association descriptor */
   int  status;

134
   read_args(argc, argv, &p_state);

135
   init_files();

136
   init_comm(&a_state);

137
   init_krb(&p_state);

   for(;;)
   {

138
      if((status = begin_assoc(&p_state, &a_state)) == FAILURE)
      {
         fprintf(fio_log, "Begin association error\n");
         exit(FAILURE);
      }
      else if (status == STOP)
      {
         fprintf(fio_log, "Normal exit\n");
         exit(OK);
      }

139
      if((status = service_assoc(&p_state, &a_state)) == FAILURE)
      {
         fprintf(fio_log, "Service association error\n");
         exit(FAILURE);
```

Example 6-12: (continued)

```
        }
        else if (status == STOP)
        {
           fprintf(fio_log, "Normal exit\n");
           exit(OK);
        }

140
        if(end_assoc(&a_state) == FAILURE)
        {
           fprintf(fio_log, "End association error\n");
           exit(FAILURE);
        }
    }
}
```

Glossary

Authentication

The process of determining the identity of an entity such as a user, application, or a host. Currently, ULTRIX Kerberos supports the authentication of commonly networked applications, such as `passwd`, `su`, `named`, `rlogin`, and `auditd`. That is, Kerberos currently authenticates applications to each other across machine boundaries in a distributed network.

Authenticator

The authenticator prevents a hostile user from replaying a stolen ticket. It is used only once, and is built by the client. It expires after approximately five minutes.

The authenticator contains the name of the client, its workstation IP address, and the current workstation time, all encrypted with the session key. After building the authenticator, the client sends it with the service ticket to the requested service.

Bindmaster

An alias for the machine that runs the master `named`.

BIND/Hesiod client

Any system that uses the BIND/Hesiod service to resolve host names and addresses. In an UPGRADE or ENHANCED security level environment, all BIND/Hesiod clients must convert to BIND/Hesiod slaves and run a Kerberos-authenticated `named` daemon.

BIND/Hesiod primary server

The server that loads the BIND/Hesiod database from a file on disk. It distributes the master BIND/Hesiod database to BIND/Hesiod secondaries and answers queries. See also *Kerberos master.*

BIND/Hesiod slave

A Kerberos client can be a BIND/Hesiod slave which answers name and address queries and runs a Kerberos-authenticated `named`.

BSD security level

The lowest security level. The `named` daemon is Kerberos-authenticated at this level to guarantee that all Hesiod distributed information comes from a correct, specified source.

The BSD distributed environment verifies that the BIND/Hesiod primary and secondary servers are accesible and working correctly. At this level, the `/etc/passwd` file distributes passwords in a network. This security level is required before making progressive transitions to the UPGRADE and ENHANCED security levels.

See also *UPGRADE security level* and *ENHANCED security level.*

Cleartext password

An unencrypted password.

Client application

An application that requests that an application on another machine be performed on its behalf. See also *Kerberos client* and *Server application.*

Distributed authentication

The major function of Kerberos as it is now implemented with ULTRIX; it enables the identification of entities across a network. Network connections between Kerberos and ULTRIX are socket-based and implemented through the TCP/IP standard Internet protocol suite. See also *Kerberos.*

ENHANCED security level

The highest security level. At this level, the Kerberos master and the BIND/Hesiod primary server limit login access to only those in the local `passwd` and `auth` files. To prevent distributed lookups on the master, set the `auth` and `passwd` variables to local in the `/etc/svc.conf` file on the Kerberos master-BIND/Hesiod primary.

See also *BSD security level* and *UPGRADE security level.*

Instance

The second part of a 3-part unique name for a Kerberos principal. It distinguishes among variations of the primary name. The unique name for a Kerberos principal is expressed as:

```
name.instance@realm
```

The following example shows two different Kerberos principals that share the same Primary name, but have different instances:

```
rlogin.venus
rlogin.mars
```

In the example above, there is no realm name.

IP

Internet Protocol. The Internet standard protocol that defines the Internet datagram as the unit of information passed across the Internet.

IP datagram

Basic unit of information passed across the Internet. It contains a source and destination address with the data.

Kerberos

An authentication service offered with ULTRIX. Currently, it authenticates applications to each other across machine boundaries in a distributed network of shared applications on different workstations. It serves as a single point of "trust" in a local area network (LAN).

Kerberos client

A principal that has been authenticated with Kerberos.

Kerberos master

The system on which the master Kerberos database resides. It can run the Kerberos-authenticated `named` daemon. The Kerberos master can also be the *BIND/Hesiod primary server* that loads the BIND/Hesiod database from a file on disk.

Mutual authentication

With Kerberos, there is mutual authentication between applications "X" and "Y" because X trusts Kerberos to give application Y only enough information to authenticate itself as X to Y. This enables each to know that the other is not a false representation from a hostile user.

Primary name

The first part of a 3-part unique name for a Kerberos principal. It is the name of the client or service. The unique name for a Kerberos principal is expressed as:

```
name.instance@realm
```

For example, in the following principal name for `rlogin`, the primary name is `rlogin`:

```
rlogin.venus
```

In the example above, there is no realm name.

Principal

Any communicating entity in Kerberos.

Private key

A large number derived from a client (that is, principal) password. Each key is known only to Kerberos and to the client it belongs to. This enables Kerberos to create a ticket that convinces one principal that another principal is really who it claims to be. To make session keys, Kerberos uses the Data Encryption Standard (DES) encryption library.

Realm

The third part of a 3-part unique name for a Kerberos principal. It is the name of a group of machines, such as those on a Local Area Network (LAN). Each LAN is located in a separate realm, and each LAN contains a separate Kerberos master.

The unique name for a Kerberos principal is expressed as:

```
name.instance@realm
```

For example, if the realm were located within the Digital Equipment Corporation, then the Kerberos principal name for `rlogin` on machine `venus` would be:

```
rlogin.venus@dec.com
```

Session key

A temporary private key. See also *Private key.*

Server application

An application requested by a client application. See also *Client application.*

TCP

Transmission Control Protocol. The Internet standard transport level protocol that provides full duplex stream data flow between applications.

Ticket

A ticket enables a principal to authenticate to another principal. The `kerberos` daemon provides a principal with a ticket.

Ticket-granting ticket

After a principal is Kerberos-authenticated, it recieves a ticket-granting ticket which grants permission to receive various service tickets. The ticket-granting ticket can be reused later during the workstation session.

Although ULTRIX Kerberos does not currently support user-level authentication, the ticket-granting ticket is designed to enable a user to supply a password only once, at the start of a workstation session.

Time client

The system that runs `timed` as a client. See also *time master.*

Time master

The system that runs the NTP (network time protocol) daemon `ntpd`, for time synchronization over a wide area network. It also runs the Berkeley `timed` as master to distribute time to all workstations in the network. See also *Time client.*

ULTKERB400 software subset

This software subset must be installed on the Kerberos master and on all other systems that are going to run the Kerberos-authenticated `named` daemon on a slave server. The kernel does not need to be rebuilt after this subset is added.

UPGRADE security level

This security level converts the BSD-style passwords into `auth`-style passwords. Users who do not upgrade their passwords at this level cannot log in at the ENHANCED level, unless the superuser runs `passwd` for them.

See also *BSD security level* and *ENHANCED security level.*

Index

E

G

H

I

K

L

M

N

P

R

S

T

U

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to Kerberos
AA-PBKVA-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

_____

What do you like best about this manual? _____

_____

_____

What do you like least about this manual? _____

_____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

Do Not Tear – Fold Here and Tape

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST–CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–2/Z04
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

Do Not Tear – Fold Here

Cut
Along
Dotted
Line