AFS Programmer's Reference: *Authentication Server* Interface

Transarc Corporation

April 12, 1993

FS-00-D166

Contents

1 Overview . . . 1
- 1.1 Introduction . . . 1
- 1.2 Scope of this Document . . . 1
- 1.3 Document Layout . . . 2
  - 1.3.1 Related Documents . . . 2

2 *Authentication Server* Architecture . . . 4
- 2.1 Encryption Key . . . 4
  - 2.1.1 Overview . . . 4
  - 2.1.2 Mutual Authentication . . . 4
  - 2.1.3 System Design . . . 6
- 2.2 Assumptions . . . 6
  - 2.2.1 Introduction . . . 6
  - 2.2.2 Physical Security . . . 6
  - 2.2.3 Time Synchrony . . . 7
  - 2.2.4 Passwords . . . 7
- 2.3 Security . . . 8
  - 2.3.1 Introduction . . . 8
  - 2.3.2 Security through Encryption . . . 8
  - 2.3.3 Methods of Attack . . . 9

3 *Authentication Server* Interface . . . 12
- 3.1 Introduction . . . 12
- 3.2 Constants . . . 12
  - 3.2.1 Interface Function Opcodes . . . 13
  - 3.2.2 Constants . . . 13
  - 3.2.3 Miscellaneous . . . 14
- 3.3 Structures . . . 15
  - 3.3.1 `struct ka_CBS` . . . 15
  - 3.3.2 `struct ka_BBS` . . . 15
  - 3.3.3 `struct EncryptionKey` . . . 16
  - 3.3.4 `struct kaident` . . . 16

3.3.5 `struct kaentryinfo` 16
3.3.6 `struct kasstats` 17
3.3.7 `struct kadstats` 18
3.3.8 `struct ka_kcInfo` 19
3.3.9 `struct ka_debugInfo` 19
3.4 User-Level Library Interface 20
3.5 RPC Function Calls 22
3.5.1 KAA_Authenticate 23
3.5.2 KAA_ChangePassword 23
3.5.3 KAT_GetTicket 23
3.5.4 KAM_SetPassword 24
3.5.5 KAM_SetFields 24
3.5.6 KAM_CreateUser 25
3.5.7 KAM_DeleteUser 25
3.5.8 KAM_GetEntry 26
3.5.9 KAM_ListEntry 26
3.5.10 KAM_GetStats 26
3.5.11 KAM_Debug 27
3.5.12 KAM_GetPassword 27
3.5.13 KAM_GetRandomKey 27
3.6 User-Level Library Functions 28
3.6.1 ka_UserAuthenticateGeneral 28
3.6.2 ka_UserReadPassword 29
3.7 Cell Related Functions 29
3.7.1 ka_CellConfig 29
3.7.2 ka_LocalCell 29
3.7.3 ka_ExpandCell 30
3.7.4 ka_CellToRealm 30
3.8 Miscellaneous Client-Side Functions 31
3.8.1 ka_Andrew_StringToKey 31
3.8.2 StringToKey 31
3.8.3 ka_StringToKey 31
3.8.4 ka_ReadPassword 32
3.8.5 ka_ParseLoginName 32
3.8.6 ka_Init 33
3.8.7 ka_ExplicitCell 33
3.8.8 ka_GetServers 33
3.8.9 ka_GetSecurity 34
3.8.10 ka_SingleServerConn 34
3.8.11 ka_AuthServerConn 34
3.8.12 ka_Authenticate 35

3.8.13 ka_ChangePassword 35
3.8.14 ka_GetToken 36
3.8.15 ka_GetAdminToken 36
3.8.16 ka_GetAuthToken 36
3.8.17 ka_GetServerToken 37
3.9 Command Line Interface 37

Index . . . . **i**

Chapter 1

Overview

1.1 Introduction

The Authentication Server, or AuthServer, is a program that runs on one or more machines in a distributed workstation environment. In such an environment, it is common for a collection of server processes to provide facilities to the network community that are not available to individual workstations. In a large community, a formal mechanism to confidently identify clients to servers and vice versa is necessary. The AuthServer mediates this identification process. It provides information to the server that allows it to name its client. Conversely, the client can be confident it is in communication with the desired server and not with a Trojan Horse. This identification is not perfect but can be relied upon given certain assumptions about the security of the network, the integrity of the human administrators, and the cryptographic effort that might be expended to forge an identification.

1.2 Scope of this Document

This paper describes the design and structure of the AFS Authentication Server. The scope of this work is to provide readers with a sufficiently detailed description of the AuthServer so that they may construct client applications that call the server's RPC interface.

1.3 Document Layout

The second chapter discusses various aspects of the Auth Server's architecture. The basis of authentication in this system is a *shared secret* known by both parties in a communication. This secret is usually called a *key* or sometimes an *Encryption Key*. After this is discussed, the security issue is considered and assumptions made. Chapter three discusses the Auth Server's API in detail.

1.3.1 Related Documents

This document is a member of a documentation suite providing programmer level specifications for the operations of the various AFS servers and agents and the interfaces they export, as well as the underlying RPC system they use to communicate. The full suite of related AFS specification documents is listed below:

- *AFS Programmer's Reference: Architectural Overview*: This paper provides an architectural overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and, examining their interactions.
- *AFS Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.
- *AFS Programmer's Reference: BOS Server Interface*: This paper describes the "nanny" service which assists in the administrability of the AFS environment.
- *AFS Programmer's Reference: File Server/Cache Manager Interface*: This document describes the workings and interfaces of the two primary AFS agents: the *File Server* and *Cache Manager*. The *File Server* provides a centralized disk repository for sets of files, regulating access to them. End users sitting on client machines rely on the *Cache Manager* agent, running in their kernel, to act as their agent in accessing the files stored on *File Server* machines, making those files appear as if they were really housed locally.
- *AFS Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for providing two-way mappings between printable user names and their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users suitable for placement on ACLs.

Authentication Server Interface

- *AFS Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.

In addition to these papers, the AFS product is delivered with its own user, administrator, installation, and command reference documents.

Chapter 2

Authentication Server **Architecture**

This chapter presents the general architecture of the Authentication Server. First, mutual authentication is discussed in detail. Following this, some assumptions such as physical security and passwords are discussed. Finally, the security issue is discussed.

2.1 Encryption Key

2.1.1 Overview

The basis of authentication in this system is a *shared secret* known by both parties in a communication. This secret is usually called a *key* or sometimes an *Encryption Key* or *password*. A simple protocol can be used by each side to determine whether the other truly does know the correct key without actually sending the key across the network. The *Data Encryption Standard (DES)* is used to provide this determination, although other key-based encryption systems could be used. The AuthServer provides the client, directly, and the server, indirectly, with the shared secret which they use to identify each other.

2.1.2 Mutual Authentication

To deliver keys to the client and server, the AuthServer must use an untrustworthy communication channel. Because of this fundamental problem, each identity, including both clients and servers, must be registered with the AuthServer through secure means. Generally, this involves a face-to-face meeting with a new user and a system administrator to establish a user name and initial password. Such a registered user is called a principal identity or just a *principal*. The user's password is used to generate an encryption key which is used in all further transactions; the password itself is discarded. The user should change his password periodically and can do so without the intervention of a system administrator.

The key, generated from the user's password and stored in the AuthServer, is then used to *mutually authenticate* any client-server pair. The basic algorithm is quite simple. The client calls the AuthServer with the name of the server it wishes to contact and a random number which has been encrypted with the client's key. The AuthServer decrypts the packet with the client's key, which it knows, to obtain the random number. It then looks up the key of the server and creates a *ticket* encrypted with the server key and containing the client's name and a *session key*. Then the AuthServer creates a packet for return to the client which contains the random number, the session key, and the ticket prepared for the desired server all encrypted with the client's key. This packet contains two copies of the session key: one encrypted with the client's key and one encrypted two times: first with the server's key then with the client's key. This session key will be the secret shared by the client and server when they establish communications.

The client decrypts the packet returned from the AuthServer and checks the random number to see that it matches the one sent. If the number matches, it can be sure the AuthServer is genuine and really knows the client's key. The client saves the session key and delivers the ticket, which it can not interpret, to the server when the connection is opened. The server decrypts the ticket and verifies that the user name it contains is allowed to use its service. The session key can be used to encrypt communication between the two. The server knows who this user is because the name was in the ticket encrypted by the AuthServer. The client knows it is in communication with the real server since the server was able to obtain the session key from the the ticket created by the AuthServer. The parties can work in privacy, both confident of the identity of the other. This is *secure mutually authenticated communication.*

In practice a slightly more complicated system is employed for creating server tickets. The process of getting a ticket for a server is divided into two steps. The first involves contacting the AuthServer using the protocol outlined above. The AuthServer produces a ticket, called the AuthTicket, which is given to an intermediate server called the Ticket Granting Service (TGS). The second step is to contact the TGS, using the AuthTicket as proof of authentication, to obtain a ticket for the desired server. This second step may be repeated many times during a login session to get tickets for various servers as they are needed. The advantage of dividing the process this way is that the user's password is used only for the first step. Since the AuthTicket has a lifetime of a few hours, it is a smaller security risk to keep it in memory for the duration of a login session than to keep a password for the same period.

2.1.3 System Design

The design of this system borrows heavily from the work Needham and Schroeder[7] did on using encryption in an insecure network environment. Many ideas have been adapted from their work, especially the format of the tickets and much of their terminology.

A good deal of effort has gone into making the system compatible with Kerberos[6]. In this paper we refer to *cells* which are broadly equivalent to the *realms* of Kerberos. We have mostly adopted the name *ticket*, but earlier descriptions of the Andrew system referred to them as *tokens*. The term tokens is used here to refer to a structure containing a ticket and the associated session key.

2.2 Assumptions

2.2.1 Introduction

The security and reliability of the AuthServer depends on several assumptions about its environment. Making these assumptions explicit should help prevent security violations or other problems caused by inadvertently changing some aspect of the system on which the AuthServer relies. In this section we cover all the parts of the system that are unusual or critical to correct operation.

2.2.2 Physical Security

It is critical to the secure operation of the AuthServer that the computer on which it runs is a *physically secure machine*. There are many ways to compromise the integrity of the AuthServer if access to the hardware is possible. Many of these are difficult to combat or even detect. Providing physical security is an established technique with well known advantages and well understood weaknesses.

To obtain the many advantages of multiprocessing, the AuthServer is usually run as a distributed process on several machines. Each of these machines must be physically secure. The connection between the AuthServer's machines and the rest of the system, including both servers and clients, does not have to be secure. The protocols used to establish and maintain communication between the AuthServer and clients and between clients and servers attempt to protect against all types of attacks on the network. In addition, the physical security of neither the client nor other server machines is depended upon, as they may be owned and operated by individuals or groups throughout the user community.

2.2.3 Time Synchrony

The protocols depend upon a degree of *time synchrony* to detect stale or replayed messages which may be used in an attack on system security. Because perfect synchronization is unrealistic, this protocol assumes that clocks on different machines are within fifteen minutes of each other. This requirement is only important for changing passwords, because this is an operation with a side-effect. This synchronization limit means that a user who changes his or her password twice within fifteen minutes could have it changed back by an attacker replaying the first change request.

2.2.4 Passwords

The security of the system critically depends on the security of user passwords. Most users do not keep their password's safe or choose appropriate passwords. This is usually the largest source of security problems in any system. These problems, however, are limited to the compromise of individual users, although individuals with access to sensitive data need to be careful about their passwords.

The problem of managing server passwords is much more critical. The ability to reboot a server and have the server's password entered without human intervention is important for large systems that need to provide a high level of availability. To accomplish this, server passwords are stored on the local, physically secure disk. This approach is a reasonable interim solution but one which has many potential risks. The AuthServer simply assumes that all passwords are safe.

Another problem is the possibility of a *Trojan Horse* replacing the standard login program. Because fetching the login program over the network usually precedes the authentication process, there is no protection from having a rogue file server supply a copy of login that steals passwords. The same general problem applies to booting a workstation from a network server. These problems are not addressed by the AuthServer and are assumed to be controlled in other ways.

2.3 Security

2.3.1 Introduction

The basic authentication problem has two parts: establishing identity and defeating attempts to forge identities. The latter almost totally dictates the details of the solution. The job of establishing the identity of the user is simply a matter of determining whether or not a user knows the password associated with a principal. This could be as simple as having the user sit down at a terminal and type in his name and password which is then sent over the network to the AuthServer. To provide security in a potentially hostile environment, various counter-measures must be taken to prevent a user's password from being revealed and to prevent one user from masquerading as another. The basic source of security loopholes comes from the fact that both the local workstations and the network that connects them to the AuthServer are easily accessible to an attacker. However, these weaknesses are an intrinsic part of a distributed workstation environment so the design of the protocols takes them into account.

2.3.2 Security through Encryption

Security problems are primarily addressed by encrypting sensitive messages with DES. The choice of key and the details of the message contents vary according to the situation. The DES algorithm has several features that are important to its use in authentication [8].

- It has a relatively compact, eight byte key that can be formed from several convenient sources such as passwords or random numbers.
- Data of any length can be encrypted and decrypted given the key, but it is very difficult to do either without the key. The data must be padded to a multiple of eight bytes. The standard method of encrypting a sequence of data longer than eight bytes is called *cipher block chaining.* This method uses some data from the encryption of the previous eight byte chunk to encrypt the current chunk. This means that a change to a message will affect the translation of all subsequent data in the message.
- The key must be kept hidden.

The basic paradigm for achieving mutual authentication involves exchanging messages that can be recognized as correct when decrypted using the shared secret as the key.

The properties of DES are such that without knowledge of the key it is not feasible to create a properly encrypted message or to decrypt a message. The choice of the message contents results from a consideration of the various methods of attack that the system must protect itself against.

2.3.3 Methods of Attack

The attacks that the design attempts to cope with fall into these categories:

- eavesdropping - This attack assumes that someone can listen in on a conversation and obtain private information. It includes both intercepting keys or passwords, which would be very serious and simple snooping on the contents of personal files.
- tampering - In this type of attack, data is modified, either in transit or by intercepting a packet, altered, and reinjected into the network.
- replay - This is reinsertion of old packets into the network in an attempt to deceive the recipient. The advantage of this approach is that the attacker does not need full knowledge of the contents of the packets. It is conceptually similar to tampering except that it assumes that the original packet was received or that the delay between interception and reinjection is relatively long.
- server misrepresentation - This is a specific case of a Trojan Horse where an attacker pretends to be a server. Mutual authentication is important in this case.
- cryptographic attack - This is an attempt to decrypt the encryption algorithm. Apart from using a encryption algorithm blessed by the National Security Agency (DES), little attempt is made to worry about this.

Eavesdropping implies that anything sent over the network can be read. This means that sensitive data must be encrypted when sent over the network. This primarily affects keys.

Encryptions can be used to detect tampering. The data to be protected must be encrypted even though the data itself may not be sensitive. If an encrypted message is tampered with, the decryption process will scramble it in ways impossible for the attacker to predict. With suitable consistency checks on the decrypted message, tampering can be detected and such messages rejected. By using encryption to prevent tampering, the recipient can assume that the data in a message is internally consistent.

Replay attacks can be defeated in one of two ways. Either by including a timestamp that causes old messages to be rejected or by including an identifier specified by the recipient in an earlier message. The advantage of timestamps is that they are easy to use; a single test by the server can detect a replayed message. The disadvantage is that the inevitable skew between the clocks of different machines and the routing delays imposed by the network puts a lower bound on the delay that can be detected by rejecting messages with “old” timestamps. Because many systems contain clock skews of several minutes, this test cannot be made very rigorous.

Using a handshaking algorithm that exchanges identifiers has the advantage that the identifiers are only used once. If the identifier is encountered in a replayed message after the original transaction has completed, the message will always be rejected. The problem is that the protocol for exchanging identifiers is more complex. The basic scenario is as follows:

client: $\{\text{r\_1} = \text{random}()\}^{key} -- >$
$< -- \{\text{r\_1}+1, \text{r\_2} = \text{random}()\}^{key}$ **:server**

At this point the client knows that the server is real, but the server is still not sure about the client. Since the transaction is initiated by the client it gets the first information back. It takes one more message

client: $\{\text{r\_2}+1\}^{key} -- >$

to convince the server and complete the mutual authentication.

An additional safeguard against replay is to test the source of each packet against the host the previous packet came from and make sure that all packets are from the same host. This makes hijacking an ongoing connection more difficult. There are two problems with using the host address:

- The hardware that inserts the host address may be altered to fake any desired return address.
- The user may intentionally change hosts from time to time which limits the strictness of this test to the lifetime of a single connection.

The problem of server misrepresentation is resolved whenever mutual authentication happens before sensitive data is exchanged. Alternatively, if sensitive data is encrypted, and assuming incorrect decryption can be detected with consistency checks, illegitimate servers can be rejected without authentication. In this case the encrypted data transfer accomplishes the authentication.

Protection from cryptographic attack is beyond the scope of this mechanism. It is implicitly assumed that anyone prepared to do a serious cryptographic attack on DES encrypted data in this system can have whatever he or she can get. Any data so sensitive that this might be a problem should be subject to additional safety measures. No such data should be stored in the Andrew File System.

Several simple design rules should serve to make such attacks more difficult. The goal of these rules is to not give away any more information than necessary.

- Keep messages short.
- Make the contents of an encrypted message as difficult to predict as possible.
- Put the most random data at the front of the message so that the cipher block chaining will randomize the encrypted text as much as possible.
- Do not repeat an encrypted message more often that strictly necessary.
- Do not provide "free" samples of encrypted text.

Chapter 3

Authentication Server **Interface**

3.1 Introduction

This chapter documents the API for the Authentication Server facility, as defined by the *kauth.rg Rxgen* interface file and the *kaserver.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer are included.

3.2 Constants

This section covers the basic constant definitions of interest to the AuthServer application programmer. These definitions appear in the *kaserver.h* file which is automatically generated from *kauth.rg*, the RPC interface definition file. See Chapter 6 of the *AFS Programmer's Reference: Specification for the Rx Remote Procedure Call Facility* for an example of an RPC interface definition file.

The subsections describe constants in the following categories:

- Interface function opcodes
- Constants
- Micellaneous

3.2.1 Interface Function Opcodes

These constants, appearing in the *kauth.rg Rxgen* interface file for the AuthServer, define the opcodes for the RPC routines. Every *Rx* call to this interface contains this opcode. The thread dispatcher uses it to select the proper code at the server site to carry out the call.

The functions are described in section 3.5.

Name	*Value*	*Description*
AUTHENTICATE_OLD	1	Opcode for *KAA_Authenticate_old()*
CHANGEPASSWORD	2	Opcode for *KAA_ChangePassword()*
GETTICKET_OLD	3	Opcode for *KAT_GetTicket_old()*
SETPASSWORD	4	Opcode for *KAM_SetPassword()*
SETFIELDS	5	Opcode for *KAM_SetFields()*
CREATEUSER	6	Opcode for *KAM_CreateUser()*
DELETEUSER	7	Opcode for *KAM_DeleteUser()*
GETENTRY	8	Opcode for *KAM_GetEntry()*
LISTENTRY	9	Opcode for *KAM_ListEntry()*
GETSTATS	10	Opcode for *KAM_GetStats()*
DEBUG	11	Opcode for *KAM_Debug()*
GETPASSWORD	12	Opcode for *KAM_GetPassword()*
GETRANDOMKEY	13	Opcode for *KAM_GetRandomKey()*
AUTHENTICATE	21	Opcode for *KAA_Authenticate()*
GETTICKET	23	Opcode for *KAT_GetTicket()*

3.2.2 Constants

The following constants are required to properly use the AuthServer RPC interface, both to provide values and to interpret information returned by the calls.

Name	*Value*	*Description*
`MAXKAKVNO`	127	The key version number must fit in a byte.

The next six constants are flags. NOTE: zero is an illegal value.

Name	*Value*	*Description*
`KAFNORMAL`	0x001	Set for all user entries.

If the normal flag is off, then one of the two flags below must be set.

Name	*Value*	*Description*
`KAFFREE`	0x002	Set if in free list.
`KAFOLDKEYS`	0x010	Set if entry used to store old keys.

Otherwise one of the following may be set to define the usage of the misc field.

Name	*Value*	*Description*
`KAFSPECIAL`	0x100	Set if special AuthServer principal.
`KAFASSOCROOT`	0x200	Set if root of associate tree.
`KAFASSOC`	0x400	Set if entry is an associate.

The following flags define special properties for normal users and are settable using *SetFields()*.

Name	*Value*	*Description*
`KAFADMIN`	0x004	An administrator.
`KAFNOTGS`	0x008	Don't allow principal to get or use TGT.
`KAFNOSEAL`	0x020	Don't allow principal as server in GetTicket.
`KAFNOCPW`	0x040	Don't allow principal to change its own key.
`KAFNEWASSOC`	0x080	Allow user to create associates.

3.2.3 Miscellaneous

This section lists miscellaneous constants used in the structures described in the following sections.

To make future revisions easy to accommodate they are assigned a major and minor version number. Major version changes will require recompilation because the structures have changed size. Minor version changes will be more or less upward compatible.

Name	*Value*	*Description*
KAMAJORVERSION	5	
KAMINORVERSION	1	
NEVERDATE	037777777777	Initial value used in date fields.
KADEBUGKCINFOSIZE	25	This returns information about the state of the server for debugging problems remotely.

3.3 Structures

This section describes the major exported *Authentication Server* data structures of interest to application programmers. These structures are returned by server RPC interface routines.

3.3.1 struct ka_CBS

This structure is used to specify input byte sequences in routines such as *ChangePassword()* and *GetTicket()*. Since the byte sequence is not NULL terminated, the number of bytes in the sequence must be included in the structure.

Fields

long SeqLen - Number of bytes in character string.

char *SeqBody - A character string, not NULL terminated.

3.3.2 struct ka_BBS

This structure is used to specify where a called function, such as *ChangePassword()* and *GetTicket()*, should put a reply byte sequence and the maximum number of bytes allowed in the reply. The called function specifies the number of bytes being returned.

Fields

long MaxSeqLen - Maximum number of bytes allowed in character string.

long SeqLen - Actual number of bytes in character string.

char *SeqBody - Character string, not NULL terminated.

3.3.3 `struct EncryptionKey`

This structure defines an encryption key that is bit level compatible with DES and ktc_encryptionKey, but that will have to be cast to the appropriate type in calls.

Fields

char key[**8**] - 16 hex digits.

3.3.4 `struct kaident`

This structure returns name and instance strings.

Fields

`char name[MAXKANAMELEN]` - The users name, NULL terminated, at most 63 characters in name.

`char instance[MAXKANAMELEN]` - The group name, NULL terminated, at most 63 characters in group.

3.3.5 `struct kaentryinfo`

This structure is used to return the information stored in an entry in the authentication database. For example, the calling sequence to *GetEntry()* specifies a pointer to one of these. NOTE: `Date` is a synonym for `unsigned long`.

Fields

`long minor_version` - The minor version number.

`long flags` - Holds flags described in section 3.2.2.

`Date user_expiration` - User registration good until then.

`Date modification_time` - Time of last update.

`struct kaident modification_user` - User name and instance last modified.

`Date change_password_time` - Time user changed own password.

`long max_ticket_lifetime` - Maximum lifetime for tickets.

`long key_version` - Version number of this key.

`EncryptionKey key` - The key to use.

`unsigned long keyCheckSum` - Crypto-cksum of key.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.3.6 `struct kasstats`

These are (static) statistics kept in the database header. This structure can be examined via a call to *GetStats()*.

Fields

`long minor_version` - The minor version number.

`long allocs` - Total number of calls to AllocBlock.

`long frees` - Total number of calls to FreeBlock.

`long cpws` - Number of user change password commands.

`long reserved1` - Not used.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.3.7 struct kadstats

These are dynamic statistics kept in each AuthServer process. The current values can be examined via a call to *GetStats()*.

Fields

`long minor_version` - The minor version number.

`long host` - Server, in NW byte order.

`Date start_time` - Time statistics were last cleared.

`long hashTableUtilization` - Use of non-empty hash table entries in parts per 10,000.

`declare_stat (Authenticate)` - Count of requests and aborts for each RPC.

`declare_stat (ChangePassword)` - Count of requests and aborts for each RPC.

`declare_stat (GetTicket)` - Count of requests and aborts for each RPC.

`declare_stat (CreateUser)` - Count of requests and aborts for each RPC.

`declare_stat (SetPassword)` - Count of requests and aborts for each RPC.

`declare_stat (SetFields)` - Count of requests and aborts for each RPC.

`declare_stat (DeleteUser)` - Count of requests and aborts for each RPC.

`declare_stat (GetEntry)` - Count of requests and aborts for each RPC.

`declare_stat (ListEntry)` - Count of requests and aborts for each RPC.

`declare_stat (GetStats)` - Count of requests and aborts for each RPC.

`declare_stat (GetPassword)` - Count of requests and aborts for each RPC.

`declare_stat (GetRandomKey)` - Count of requests and aborts for each RPC.

`declare_stat (Debug)` - Count of requests and aborts for each RPC.

`declare_stat (UAuthenticate)` - Count of requests and aborts for each RPC.

`declare_stat (UGetTicket)` - Count of requests and aborts for each RPC.

`long string_checks` - Errors detected in name and instance strings.

`long reserved1` - Not used.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.3.8 struct ka_kcInfo

This structure describes an entry in the key cache in the AuthServer. This information is visible as part of the `ka_debugInfo` structure.

Fields

`Date used` - When key used last.

`long kvno` - Version number of this key.

`char primary` - Determines whether this key is due to be superseded.

`char keycksum` - The checksum for this key.

`char principal[64]` - Stores user name and group.

3.3.9 struct ka_debugInfo

This structure describes the state of the AuthServer. The information is visible via a call to *Debug()*.

Fields

`long minorVersion` - The minor version number.

`long host` - Server in NW byte order.

`Date startTime` - Time server was started.

`int noAuth` - Running with authentication off.

`Date lastTrans` - Time of last transaction.

`char lastOperation[16]` - Name of last operation.

`char lastAuth[256]` - Last principal to authenticate.

`char lastUAuth[256]` - Last principal to authenticate via UDP.

`char lastTGS[256]` - Last principal to call ticket granting service.

`char lastUTGS[256]` - Last principal to call TGS via UDP.

`char lastAdmin[256]` - Last principal to call admin service.

`char lastTGSServer[256]` - Last server for which a ticket was requested.

`char lastUTGSServer[256]` - Last server for which a ticket was requested via UDP.

`Date nextAutoCPW` - When server password expires (probably NEVERDATE).

`int updatesRemaining` - Update necessary for next AutoCPW.

`Date dbHeaderRead` - Time cheader structure was last read in.

`long dbVersion` - Minor version number.

`long dbFreePtr` - Pointer to first free entry in the database file.

`long dbEofPtr` - Pointer to first free byte in the database file.

`long dbKvnoPtr` - Pointer to first key version number in the database.

`long dbSpecialKeysVersion` - Latest key version number.

`long cheader_lock` - Lock for the key cache header.

`long keycache_lock` - Lock for the key cache.

`long kcVersion` - Minor version number.

`int kcSize` - Size of key cache.

`int kcUsed` - Number of entries in key cache.

`struct ka_kcInfo kcInfo[KADEBUGKCINFOSIZE]` - Describes the keys in the key cache.

`long reserved1` - Not used.

`long reserved2` - Not used.

`long reserved3` - Not used.

`long reserved4` - Not used.

3.4 User-Level Library Interface

The authentication functions described in the following sections are made visible via *libkauth.a*. This library in turn depends on two additional libraries: *libauth.a* and *librxkad.a*. This section lists the routines that are visible in these libraries. The programmer should use the high level routines available in *libkauth.a*.

```
libkauth.a
        KAA_Authenticate
        KAA_Authenticate_old
        KAA_ChangePassword
        KAM_CreateUser
        KAM_Debug
        KAM_DeleteUser
        KAM_GetEntry
        KAM_GetPassword
```

```
        KAM_GetRandomKey
        KAM_GetStats
        KAM_ListEntry
        KAM_SetFields
        KAM_SetPassword
        KAT_GetTicket
        KAT_GetTicket_old
        Andrew_StringToKey
        StringToKey
        ka_Init
        ka_ParseLoginName
        ka_ReadPassword
        ka_StringToKey
        debugCell
        ka_AuthServerConn
        ka_Authenticate
        ka_ChangePassword
        ka_ExplicitCell
        ka_GetSecurity
        ka_GetServers
        ka_GetToken
        ka_SingleServerConn
        ka_GetAdminToken
        ka_GetAuthToken
        ka_GetServerToken
        ka_CellConfig
        ka_CellToRealm
        ka_ExpandCell
        ka_LocalCell
        ka_UserAuthenticate
        ka_UserAuthenticateGeneral
        ka_UserReadPassword

libauth.a

        afsconf_CellApply
        afsconf_Check
        afsconf_Close
        afsconf_CloseInternal
        afsconf_GetCellInfo
        afsconf_GetLocalCell
        afsconf_Open
        afsconf_OpenInternal
        afsconf_Reopen
        afsconf_Touch
        afsconf_AddKey
        afsconf_DeleteKey
        afsconf_GetKey
        afsconf_GetKeys
        afsconf_GetLatestKey
        afsconf_IntGetKeys
        afsconf_ResetKeys
        afsconf_AddUser
        afsconf_DeleteUser
```

```
        afsconf_GetNoAuthFlag
        afsconf_GetNthUser
        afsconf_SetNoAuthFlag
        afsconf_SuperUser
        afsconf_SetCellInfo
        afsconf_CheckAuth
        afsconf_ClientAuth
        afsconf_ClientAuthSecure
        afsconf_ServerAuth

librxkad.a

        rxkad_AllocCID
        rxkad_NewClientSecurityObject
        rxkad_client_init
        rxkad_GetServerInfo
        rxkad_NewServerSecurityObject
        rxkad_CheckPacket
        rxkad_CksumChallengeResponse
        rxkad_Close
        rxkad_DeriveXORInfo
        rxkad_DestroyConnection
        rxkad_GetStats
        rxkad_NewConnection
        rxkad_PreparePacket
        rxkad_SetLevel
        rxkad_SetupEndpoint
        ktohl
        tkt_CheckTimes
        tkt_DecodeTicket
        tkt_MakeTicket
        fc_cbc_encrypt
        fc_ecb_encrypt
        fc_keysched
        fcrypt_init
        rxkad_DecryptPacket
        rxkad_EncryptPacket
        rxkad_crypt_init
```

3.5 RPC Function Calls

This section describes the Remote Procedure Calls used between the AuthServer and clients. Each function in this section has an assigned *opcode* number, see section 3.2.1. This is the low-level numerical identifier for the function, and appears in the set of network packets constructed for the RPC call.

3.5.1 KAA_Authenticate — Authenticate user password

```
proc KAA_Authenticate(IN kaname name, IN kaname instance, IN Date start_time, IN Date end_time,
                      IN struct ka_CBS *request, INOUT struct ka_BBS *answer) = 21;
```

Description

Calling this routine invokes the authentication protocol described in chapter 2. It uses the `name` and `instance` to look up the user's key in the authentication database. (NOTE: The value of `instance` is always NULL.) The key allows the `request` to be decrypted and, if it is properly formed, a ticket is created. The ticket and newly invented session key are assembled into the `answer`, encrypted with the key, and returned. This ticket is referred to as the AuthTicket. The lifetime of the ticket is specified by `start_time` and `end_time`, although the actual expiration time may be earlier if the requested interval exceeds the `max_ticket_lifetime` field of the user's AuthServer entry. This request does not modify the database.

The function *KAA_Authenticate_old()* is provided for backward compatibility with old kauth.

3.5.2 KAA_ChangePassword — Change user password

```
proc KAA_ChangePassword(IN kaname name, IN kaname instance, IN struct ka_CBS *arequest,
                        INOUT struct ka_BBS *oanswer) = 2;
```

Description

This call invokes the protocol for changing passwords described above. The encryption key used for both the *request* and *answer* sequences is the old one. The new key takes effect as soon as the lock associated with this operation is released. This request modifies the authentication database, but instead of updating the modification data a separate field called `change_password_time` is set.

3.5.3 KAT_GetTicket — Ask kaserver for a ticket for some other service

```
proc KAT_GetTicket(IN long kvno, IN kaname auth_domain, IN struct ka_CBS *aticket,
                   IN kaname name, IN kaname instance, IN struct ka_CBS *atimes,
                   INOUT struct ka_BBS *oanswer) = 23;
```

Description

This call requires an RPC connection encrypted with the AuthTicket. As long as that ticket is valid, the `name` and `instance` of a server are used to create a ticket and associated session key for that server. (NOTE: The value of `instance` is always NULL.) This operation does not modify the database.

The routine *ka_GetToken()* performs the same function and may be easier for the application writer to use.

The function *KAT_GetTicket_old()* is provided for backward compatibility with old kauth.

3.5.4 KAM_SetPassword — Initialize a password

```
proc KAM_SetPassword(IN kaname name, IN kaname instance, IN long kvno,
                     IN EncryptionKey password) = 4;
```

Description

The key and key version number of the user are set to the provided values. This call requires an RPC connection encrypted with an AdminTicket.

3.5.5 KAM_SetFields — Reset values in user database entry

```
proc KAM_SetFields(IN kaname name, IN kaname instance, IN long flags,
                   IN Date user_expiration, IN long max_ticket_lifetime, IN long maxAssociates,
                   IN long spare1, IN long spare2) = 5;
```

Description

This function alters the miscellaneous parameters associated with a user. The `flags` field can be set to one of three values.

- Normal - This is the default state: a regular user.
- Admin - This user is privileged and can modify the authentication database.
- Inactive - This makes the entry a placeholder. The user is not deleted but authentication attempts will fail.

The `user_expiration` is the time after which attempts to authenticate as this user will fail. The `max_ticket_lifetime` can be set to limit the lifetime of an authentication ticket created for a user. This call requires an RPC connection encrypted with an AdminTicket.

3.5.6 KAM_CreateUser — Enter a user in the database

proc **KAM_CreateUser**(`IN kaname name, IN kaname instance, IN EncryptionKey password) = 6;`

Description

This function adds a user to the authentication database. The key version number will be zero. The user's flags and maximum ticket lifetime will be set to default values. The registration will not have an expiration time. The modification data is set. This call requires an RPC connection encrypted with an AdminTicket.

3.5.7 KAM_DeleteUser — Delete a user from the database

proc **KAM_DeleteUser** `(IN kaname name, IN kaname instance) = 7;`

Description

This function removes a user from the authentication database. It requires an RPC connection encrypted with an AdminTicket.

3.5.8 KAM_GetEntry — Return a user entry in the database

```
proc KAM_GetEntry(IN kaname name, IN kaname instance, IN long major_version,
                  OUT struct kaentryinfo *entry) = 8;
```

Description

This function returns information about an entry in the authentication database. If the major version number does not match that in use by the server, the call returns an error code. This request does not modify the database. This call requires an RPC connection encrypted with an AdminTicket.

3.5.9 KAM_ListEntry — List entries in sequence

```
proc KAM_ListEntry(IN long previous_index,
                   OUT long *index, OUT long *count, OUT kaident *name) = 9;
```

Description

This function provides a way to step through all the entries in the database. The first call should be made with `previous_index` set to zero. The function returns `count`, which is an estimate of the number of entries remaining to be returned, and `index`, which should be passed in as `previous_index` on the next call. Each call that returns a non-zero `index` also returns a structure `kaident`, which gives the name and instance of an entry. A negative `count` or a non-zero return code indicates that an error occurred. A zero `index` means there were no more entries. A zero `count` means the last entry has been returned. This call does not modify the database and requires an RPC connection encrypted with an AdminTicket.

3.5.10 KAM_GetStats — Returns kasstats and kadstats

```
proc KAM_GetStats(IN long major_version, OUT long *admin_accounts,
                  OUT struct kasstats *statics, OUT struct kadstats *dynamics) = 10;
```

Description

This function returns statistics about the AuthServer and its database. If the `major_version` does not match that used by the server, the call returns an error code. The database is not modified. This call requires an RPC connection encrypted with an AdminTicket.

3.5.11 KAM_Debug — Return ka_debugInfo

`proc` **KAM_Debug**`(IN long major_version, IN int checkDB, OUT struct ka_debugInfo *info) = 11;`

Description

This function returns information about the authentication database, the key cache, and the state of the AuthServer.

3.5.12 KAM_GetPassword — Return a users password

`proc` **KAM_GetPassword** `(IN kaname name, OUT EncryptionKey *password) = 12;`

Description

This function returns a password from an entry in the authentication database. This could be either a user password (by specifying a user name) or a server password (by specifying a service name, e.g., "afs").

3.5.13 KAM_GetRandomKey — Return a random legal DES key

`proc` **KAM_GetRandomKey** `(OUT EncryptionKey *password) = 13;`

Description

This function returns a random DES key and is preferred over a calling routine just inventing a key. It returns a properly formatted 8 byte DES key, sets the parity properly in each byte of the key, and avoids the 5 "bad" keys.

3.6 User-Level Library Functions

This section contains specifications for the remaining routines that are visible in *libkauth.a*. These routines are the higher level interface that is most frequently used by management applications.

3.6.1 ka_UserAuthenticateGeneral — Authenticate a user

```
long ka_UserAuthenticateGeneral(long flags,
                                char *name,
                                char *instance,
                                char *realm,
                                char *password,
                                Date lifetime,
                                long spare1,
                                long spare2,
                                char **reasonP)
```

Description

This function returns 0 if successful. `name`, `instance` and `realm` can be gotten from a principal by calling *ka_ParseLoginName()*. `reason` will contain a textual description of the failure in case of error `spare2` must be 0.

This routine replaces *ka_UserAuthenticate()*, which is retained in the library for backward compatibility.

3.6.2 ka_UserReadPassword — Read a user password

```
long ka_UserReadPassword (char *prompt,
                          char *password,
                          int plen,
                          char **reasonP)
```

Description

This routine does a non-echo read of a password entered from a prompt on a standard tty. Error messages are returned in `reasonP`.

3.7 Cell Related Functions

The following routines are used by management applications to manage cell information.

3.7.1 ka_CellConfig — Access the cell configuration directory

```
int ka_CellConfig(char *dir)
```

Description

This function uses `dir` to find the specified *CellServDB* file, otherwise it uses */usr/vice/etc/CellServDB* as the default.

3.7.2 ka_LocalCell — Return the local cell name

```
char *ka_LocalCell()
```

Description

This function returns 0 as an error, otherwise it returns the local cell name.

3.7.3 ka_ExpandCell — Expands short cell name

```
int ka_ExpandCell(char *cell,
                  char *fullCell,
                  int *alocal)
```

Description

`cell` and `fullCell` must be preallocated or else NULL. `cell` is a prefix of a cell name. If `cell` is 0, the name of the local cell will be used. `fullCell` and `alocal` are OUT parameters. `fullCell` is the expanded cell name which matches the prefix. `alocal` indicates whether the returned cell name is the name of the local cell. If a cell name is not returned then the function returns `KANOCELL`, `KANOCELLS` or 0.

3.7.4 ka_CellToRealm — Convert cell name to upper case

```
int ka_CellToRealm(char *cell,
                   char *realm,
                   int *local)
```

Description

This function converts a cell name into a Kerberos realm name. It actually calls *ka_ExpandCell()* and converts the returned cell name into upper case.

Like *kaExpandCell()*, if `cell` is 0 it uses the local cell name and sets `local` to 1. Otherwise it uses the prefix found in `cell` to determine the full cell name. If the cell name is not local then `local` is set to 0. The expanded cell name is returned in `realm`. If a cell name is not returned then the function returns `KANOCELL`, `KANOCELLS` or 0.

3.8 Miscellaneous Client-Side Functions

This section contains descriptions of a variety of client functions.

The first three functions are called from management applications to manage passwords and DES keys. They accept a password string as input and convert it via a one-way encryption algorithm to a DES encryption key. In all three `str`, `cell` and `key` must be pre-allocated and null-terminated. *strlen(cell)* must be less than `MAXKTCREALMLEN`.

3.8.1 ka_Andrew_StringToKey — Convert password to DES key

```
static void ka_AndrewStringToKey(char *str,
                                 char *cell,
                                 struct ktc_encryptionKey *key)
```

Description

This function is compatible with the original Andrew authentication service password database.

3.8.2 StringToKey — Convert password to DES key

```
static void StringToKey(char *str,
                        char *cell,
                        struct ktc_encryptionKey *key)
```

3.8.3 ka_StringToKey — Convert password to DES key

```
int ka_StringToKey(char *str
                   char *cell,
                   struct ktc_encryptionKey *key)
```

3.8.4 ka_ReadPassword — Read password, convert to DES key

```
long ka_ReadPassword(char *prompt,
                     int verify,
                     char *cell,
                     struct ktc_encryptionKey *key)
```

Description

This function prints out a prompt and reads a string from the terminal, turning off echoing. If `verify` is requested it requests that the string be entered again and the two strings are compared. The string is then converted to a DES encryption key. A non-zero return indicates an error while reading in the password string.

3.8.5 ka_ParseLoginName — Parse user-entered login name

```
long ka_ParseLoginName(char *login,
                       char name[MAXKTCNAMELEN],
                       char inst[MAXKTCNAMELEN],
                       char cell[MAXKTCREALMLEN])
```

Description

This routine parses a string that might be entered by a user from the terminal. It defines a syntax that allows a user to specify his or her identity in terms of his or her name, instance and cell with a single string. These three output strings must be allocated by the caller to their maximum length. The syntax is very simple: the first dot ('.') separates the name from the instance and the first atsign ('@') begins the cell name. A backslash ('\') can be used to quote these special characters. A backslash followed by an octal digit (zero through seven) introduces a three digit octal number which is interpreted as the ascii value of a single character. A non-zero return indicates an argument error.

Error Codes

`KABADARGUMENT` No output parameters were specified.

`KABADNAME` A component of the user name was too long or a cell was specified but the cell parameter was null.

3.8.6 ka_Init — Initialize a client to make calls to the AuthServer

```
long ka_Init(int flags)
```

Client side applications call this function to initialize error tables and connect to the correct *CellServDB* file. The argument is not used but reserved for future use.

3.8.7 ka_ExplicitCell — Copy list of servers to debugging cell

```
void ka_ExplicitCell(char *cell,
                     long serverList[])
```

This function copies the specified list of servers in `serverList` into a specially known cell named "explicit". The cell can then be used to debug experimental servers.

3.8.8 ka_GetServers — Return a list of AuthServers

```
int ka_GetServers(char *cell,
                  struct afsconf_cell *cellinfo)
```

This function returns a list of Authentication Servers in the cell named in `cell`.

3.8.9 ka_GetSecurity — Return a security object

```
long ka_GetSecurity(int service,
                    struct ktc_token *token,
                    struct rx_securityClass **scP,
                    int *siP)
```

This function returns a security object appropriate for the `service` passed in as an argument.

Error Codes

`KABADARGUMENT` `service` was not recognized.

`KARXFAIL` Failed to get security object.

3.8.10 ka_SingleServerConn — Establish connection to one server

```
long ka_SingleServerConn(char *cell,
                         char *server,
                         int service,
                         struct ktc_token *token,
                         struct ubik_client **conn)
```

This function establishes a connection to only one Authentication Server. Inputs to the function are

- `cell` - the cell name
- `server` - the server for which the connection is being requested
- `service` - the service required
- `token` - a token may or may not be needed depending on the service.

The connection is returned in `conn`.

3.8.11 ka_AuthServerConn — Establish connection to all AuthServers

```
long ka_AuthServerConn(char *cell,
                       int service,
                       struct ktc_token *token,
                       struct ubik_client **conn)
```

Description

This function gets connections to all the Authentication Servers in a cell. Inputs to the function are

- `cell` - the cell name
- `service` - the service required
- `token` - a token may or may not be needed depending on the service.

The connection is returned in `conn`.

3.8.12 ka_Authenticate — Return a ticket for the ticket-granting service

```
long ka_Authenticate(char *name,
                     char *instance,
                     struct ubik_client *conn,
                     int service,
                     struct ktc_encryptionKey *key,
                     Date start,end,
                     struct ktc_token *token)
```

This function is the interface to the AuthServer RPC routine *KAA_Authenticate()*. It formats the request packet, calls the encryption routine on the answer, calls *KAA_Authenticate()* and decrypts the response. The response is checked for correctness and its contents are copied into the token. A non-zero return indicates either bad key parity, a timeout, a bad packet or a server error.

3.8.13 ka_ChangePassword — Change user password in the database

```
long ka_ChangePassword(char *name,
                       char *instance,
                       struct ubik_client *conn,
                       struct ktc_encryptionKey *oldkey,
                       struct ktc_encryptionKey *newkey)
```

3.8.14 ka_GetToken — Return a ticket for the named service

```
long ka_GetToken(char *name,
                 char *instance,
                 struct ubik_client *conn,
                 Date start, end,
                 struct ktc_token *auth_token,
                 char *auth_domain,
                 struct ktc_token *token)
```

This is a low-level function used by *ka_GetServerToken()*. It assumes that a connection to the AuthServer has already been established. The desired ticket lifetime is specified by `start` and `end`.

3.8.15 ka_GetAdminToken — Return an admin token

```
long ka_GetAdminToken(char *name,
                      char *instance,
                      char *cell,
                      struct ktc_encryptionKey *key,
                      long lifetime,
                      struct ktc_token *token,
                      int new)
```

Description

`key` contains the key constructed from the user's password and `lifetime` indicates how long the ticket will be valid (in seconds). `token` is used to pass in a token and to return one. If `token` is NULL, a token is not returned. If `new` is set to 1, then the function should get a new token if necessary.

3.8.16 ka_GetAuthToken — Return an authentication token

```
long ka_GetAuthToken(char *name,
                     char *instance,
                     char *cell,
                     struct ktc_encryptionKey *key;
                     long lifetime)
```

Description

This function returns a ticket to administer for a particular principle specified by `name` and `instance`. `key` contains the key constructed from the user's password and `lifetime` indicates how long the key will be valid (in seconds).

3.8.17 ka_GetServerToken — Return a token for a server

```
long ka_GetServerToken(char *name,
                       char *instance;
                       char *cell;
                       Date lifetime;
                       struct ktc_token *token;
```

This function sets up the necessary connections etc. and then calls *ka_GetToken()*.

3.9 Command Line Interface

Another interface exported by the *Authentication Server* is the set of command line switches it accepts. Using these switches, many server parameters and actions can be set. Under normal conditions, the *Authentication Server* process is started up by the *BOS Server* on that machine, as described in *AFS Programmer's Reference: BOS Server Interface*. So, in order to utilize any combination of these command-line options, the system administrator must define the *Authentication Server* bnode in such a way that these parameters are properly included.

A description of the set of currently-supported command line switches can be found in the *AFS Command Reference Manual*.

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[4] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, RFC 1094, March 1989.

[5] Sun Microsystems, Inc. *Design and Implementation of the Sun Network File System*, USENIX Summer Conference Proceedings, June 1985.

[6] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[7] R. Needham, M. Schroeder. *Using Encryption for Authentication in Large Networks of Computers*, CACM, 21(1978)993-9.

[8] S. Matyas, C. Meyer. *Generation, Distribution and Installation of Cryptographic Keys*, IBM Sys Jr, 17(1978)126-37.

Index

function *ka_Andrew_StringToKey()*, 31
function *ka_Authenticate()*, 35
function *ka_AuthServerConn()*, 34
function *ka_CellConfig()*, 29
function *ka_CellToRealm()*, 30
function *ka_ChangePassword()*, 35
function *ka_ExpandCell()*, 30
function *ka_ExplicitCell()*, 33
function *ka_GetAdminToken()*, 36
function *ka_GetAuthToken()*, 36
function *ka_GetSecurity()*, 34
function *ka_GetServers()*, 33
function *ka_GetServerToken()*, 37
function *ka_GetToken()*, 36
function *ka_Init()*, 33
function *ka_LocalCell()*, 29
function *ka_ParseLoginName()*, 32
function *ka_ReadPassword()*, 32
function *ka_SingleServerConn()*, 34
function *ka_StringToKey()*, 31
function *ka_UserAUthenticateGeneral()*, 28
function *ka_UserReadPassword()*, 29
function *KAA_Authenticate()*, 23
function *KAA_ChangePassword()*, 23
function *KAM_CreateUser()*, 25
function *KAM_Debug()*, 27
function *KAM_DeleteUser ()*, 25
function *KAM_GetEntry()*, 26
function *KAM_GetPassword ()*, 27
function *KAM_GetRandomKey ()*, 27
function *KAM_GetStats()*, 26
function *KAM_ListEntry()*, 26
function *KAM_SetFields()*, 24
function *KAM_SetPassword()*, 24
function *KAT_GetTicket()*, 23
function *StringToKey()*, 31