Shishi

Kerberos 5 implementation for the GNU system
for version 0.0.8, 2 October 2003

Simon Josefsson

This manual is last updated 2 October 2003 for version 0.0.8 of Shishi.

Table of Contents

1 Introduction

Shishi implements the Kerberos 5 network security system.

1.1 Getting Started

This manual documents the Shishi application and library programming interface. All commands, functions and data types provided by Shishi are explained.

The reader is assumed to possess basic familiarity with network security and the Kerberos 5 security system.

This manual can be used in several ways. If read from the beginning to the end, it gives a good introduction into the library and how it can be used in an application. Forward references are included where necessary. Later on, the manual can be used as a reference manual to get just the information needed about any particular interface of the library. Experienced programmers might want to start looking at the examples at the end of the manual, and then only read up those parts of the interface which are unclear.

1.2 Features and Status

Shishi might have a couple of advantages over other packages doing a similar job.

It's Free Software
: Anybody can use, modify, and redistribute it under the terms of the GNU General Public License (see Appendix D [Copying], page 143).

It's thread-safe
: The library uses no global variables.

It's internationalized
: It handles non-ASCII username and passwords and user visible strings used in the library (error messages) can be translated into the users' language.

It's portable
: It should work on all Unix like operating systems, including Windows.

Shishi is far from feature complete, it is not even a full RFC 1510 implementation yet. However, some basic functionality is implemented. A few implemented feature are mentioned below.

- Initial authentication (AS) from raw key or password. This step is typically used to acquire a ticket granting ticket and, less commonly, a server ticket.
- Subsequent authentication (TGS). This step is typically used to acquire a server ticket, by authenticating yourself using the ticket granting ticket.
- Client-Server authentication (AP). This step is used by clients and servers to prove to each other who they are, using negotiated tickets.
- Integrity protected communication (SAFE). This step is used by clients and servers to exchange integrity protected data with each other. The key is typically agreed on using the Client-Server authentication step.

- Ticket cache, supporting multiple principals and realms. As tickets have a life time of typically several hours, they are managed in disk files. There can be multiple ticket caches, and each ticket cache can store tickets for multiple clients (users), servers, encryption types, etc. Functionality is provided for locating the proper ticket for every use.
- Most standard cryptographic primitives. The believed most secure algorithms are supported (see Section 1.4 [Cryptographic Overview], page 4).
- Telnet client and server. This is used to remotely login to other machines, after authenticating yourself with a ticket.
- PAM module. This is used to login locally on a machine.
- KDC addresses located using DNS SRV RRs.
- Modularized low-level crypto interface. Currently Nettle and Libgcrypt are supported. If you wish to add support for another low-level cryptographic library, you only have to implement a few APIs to DES, AES, MD5, SHA1, HMAC, etc, look at '`lib/nettle.c`' or '`lib/libgcrypt.c`' as a starting pointer.

The following table summarize what the current objectives are (i.e., the todo list) and an estimate on how long it will take to implement the feature. If you like to start working on anything, please let me know so work duplication can be avoided.

- Pre-authentication support (week).
- Cross-realm support (week).
- PKINIT (use libksba, weeks)
- Finish GSSAPI support via GSSLib (weeks) Shishi will not support GSSLib natively, but a separate project "GSSLib" is under way to produce a generic GSS implementation, and it will use Shishi to implement the Kerberos 5 mechanism.
- Port to cyclone (cyclone need to mature first)
- Modularize ASN.1 library so it can be replaced (days). Almost done, all ASN.1 functionality is found in lib/asn1.c, although the interface is rather libtasn1 centric.
- KDC (initiated, weeks)
- Set/Change password protocol (weeks?)
- Port applications to use Shishi (indefinite)
- Improve documentation
- Improve internationalization
- Add AP-REQ replay cache (week).
- Study benefits by introducing a PA-TGS-REP. This would provide mutual authentication of the KDC in a way that is easier to analyze. Currently the mutual authentication property is only implicit from successful decryption of the KDC-REP and the 4 byte nonce.
- GUI applet for managing tickets.
- Authorization library (months?) The shishi_authorized_p() is not a good solution, better would be to have a generic and flexible authorization library. Possibly based on S-EXP's in tickets? Should support non-Kerberos uses as well, of course.

1.3 Overview

This section describes RFC 1510 from a protocol point of view[1].

Kerberos provides a means of verifying the identities of principals, (e.g., a workstation user or a network server) on an open (unprotected) network. This is accomplished without relying on authentication by the host operating system, without basing trust on host addresses, without requiring physical security of all the hosts on the network, and under the assumption that packets traveling along the network can be read, modified, and inserted at will. (Note, however, that many applications use Kerberos' functions only upon the initiation of a stream-based network connection, and assume the absence of any "hijackers" who might subvert such a connection. Such use implicitly trusts the host addresses involved.) Kerberos performs authentication under these conditions as a trusted third- party authentication service by using conventional cryptography, i.e., shared secret key. (shared secret key - Secret and private are often used interchangeably in the literature. In our usage, it takes two (or more) to share a secret, thus a shared DES key is a secret key. Something is only private when no one but its owner knows it. Thus, in public key cryptosystems, one has a public and a private key.)

The authentication process proceeds as follows: A client sends a request to the authentication server (AS) requesting "credentials" for a given server. The AS responds with these credentials, encrypted in the client's key. The credentials consist of 1) a "ticket" for the server and 2) a temporary encryption key (often called a "session key"). The client transmits the ticket (which contains the client's identity and a copy of the session key, all encrypted in the server's key) to the server. The session key (now shared by the client and server) is used to authenticate the client, and may optionally be used to authenticate the server. It may also be used to encrypt further communication between the two parties or to exchange a separate sub-session key to be used to encrypt further communication.

The implementation consists of one or more authentication servers running on physically secure hosts. The authentication servers maintain a database of principals (i.e., users and servers) and their secret keys. Code libraries provide encryption and implement the Kerberos protocol. In order to add authentication to its transactions, a typical network application adds one or two calls to the Kerberos library, which results in the transmission of the necessary messages to achieve authentication.

The Kerberos protocol consists of several sub-protocols (or exchanges). There are two methods by which a client can ask a Kerberos server for credentials. In the first approach, the client sends a cleartext request for a ticket for the desired server to the AS. The reply is sent encrypted in the client's secret key. Usually this request is for a ticket-granting ticket (TGT) which can later be used with the ticket-granting server (TGS). In the second method, the client sends a request to the TGS. The client sends the TGT to the TGS in the same manner as if it were contacting any other application server which requires Kerberos credentials. The reply is encrypted in the session key from the TGT.

Once obtained, credentials may be used to verify the identity of the principals in a transaction, to ensure the integrity of messages exchanged between them, or to preserve

[1] The text is a lightly adapted version of the introduction section from RFC 1510 by J. Kohl and C. Neuman, September 1993, unclear copyrights, but presumably owned by The Internet Society.

privacy of the messages. The application is free to choose whatever protection may be necessary.

To verify the identities of the principals in a transaction, the client transmits the ticket to the server. Since the ticket is sent "in the clear" (parts of it are encrypted, but this encryption doesn't thwart replay) and might be intercepted and reused by an attacker, additional information is sent to prove that the message was originated by the principal to whom the ticket was issued. This information (called the authenticator) is encrypted in the session key, and includes a timestamp. The timestamp proves that the message was recently generated and is not a replay. Encrypting the authenticator in the session key proves that it was generated by a party possessing the session key. Since no one except the requesting principal and the server know the session key (it is never sent over the network in the clear) this guarantees the identity of the client.

The integrity of the messages exchanged between principals can also be guaranteed using the session key (passed in the ticket and contained in the credentials). This approach provides detection of both replay attacks and message stream modification attacks. It is accomplished by generating and transmitting a collision-proof checksum (elsewhere called a hash or digest function) of the client's message, keyed with the session key. Privacy and integrity of the messages exchanged between principals can be secured by encrypting the data to be passed using the session key passed in the ticket, and contained in the credentials.

1.4 Cryptographic Overview

Shishi implements several of the standard cryptographic primitives. In this section we give the names of the supported encryption suites, and some notes about them, and their associated checksum suite.

Statements such as "it is weak" should be read as meaning that there is no credible security analysis of the mechanism available, and/or that should an attack be published publicly, few people would likely be surprised. Also keep in mind that the key size mentioned is the actual key size, not the effective key space as far as a brute force attack is concerned.

`NULL`

> `NULL` is a dummy encryption suite for debugging. Encryption and decryption are identity functions. No integrity protection. It is weak. It is associated with the `NULL` checksum.

`arcfour-hmac`
`arcfour-hmac-exp`

> `arcfour-hmac-*` are a proprietary stream cipher with 56 bit (`arcfour-hmac-exp`) or 128 bit (`arcfour-hmac`) keys, used in a proprietary way described in an expired IETF draft '`draft-brezak-win2k-krb-rc4-hmac-04.txt`'. Deriving keys from passwords is supported, and is done by computing a message digest (MD4) of a 16-bit Unicode representation of the ASCII password, with no salt. Data is integrity protected with a keyed hash (HMAC-MD5), where the key is derived from the base key in a creative way. It is weak. It is associated with the `arcfour-hmac-md5` checksum.

`des-cbc-crc`
: `des-cbc-crc` is DES encryption and decryption with 56 bit keys and 8 byte blocks in CBC mode, using the key as IV. The keys can be derived from passwords by an obscure application specific algorithm. Data is integrity protected with an unkeyed but encrypted `CRC32`-like checksum. It is weak. It is associated with the `rsa-md5-des` checksum.

`des-cbc-md4`
: `des-cbc-md4` is DES encryption and decryption with 56 bit keys and 8 byte blocks in CBC mode, using a zero IV. The keys can be derived from passwords by an obscure application specific algorithm. Data is integrity protected with an unkeyed but encrypted MD4 hash. It is weak. It is associated with the `rsa-md4-des` checksum.

`des-cbc-md5`
: `des-cbc-md5` is DES encryption and decryption with 56 bit keys and 8 byte blocks in CBC mode, using a zero IV. The keys can be derived from passwords by an obscure application specific algorithm. Data is integrity protected with an unkeyed but encrypted MD5 hash. It is weak. It is associated with the `rsa-md5-des` checksum. This is the strongest RFC 1510 interoperable encryption mechanism.

`des3-cbc-sha1-kd`
: `des3-cbc-sha1-kd` is DES encryption and decryption with three 56 bit keys (effective key size 112 bits) and 8 byte blocks in CBC mode. The keys can be derived from passwords by a algorithm based on the paper "A Better Key Schedule For DES-like Ciphers"[2] by Uri Blumenthal and Steven M. Bellovin (it is not clear if the algorithm, and the way it is used, is used by any other protocols, although it seems unlikely). Data is integrity protected with a keyed SHA1 hash in HMAC mode. It has no security proof, but is assumed to provide adequate security in the sense that knowledge on how to crack it is not known to the public. Note that the key derivation function is not widely used outside of Kerberos, hence not widely studied. It is associated with the `hmac-sha1-des3-kd` checksum.

`aes128-cts-hmac-sha1-96`
`aes256-cts-hmac-sha1-96`
: `aes128-cts-hmac-sha1-96` and `aes256-cts-hmac-sha1-96` is AES encryption and decryption with 128 bit and 256 bit key, respectively, and 16 byte blocks in CBC mode with Cipher Text Stealing. Cipher Text Stealing means data length of encrypted data is preserved (pure CBC add up to 7 pad characters). The keys can be derived from passwords with RSA Laboratories PKCS#5 Password Based Key Derivation Function 2[3], which is allegedly provably secure in a random oracle model. Data is integrity protected with a keyed SHA1 hash, in HMAC mode, truncated to 96 bits. There is no security proof, but the schemes are assumed to provide adequate security in the sense that knowledge

[2] `http://www.research.att.com/~smb/papers/ides.pdf`

[3] `http://www.rsasecurity.com/rsalabs/pkcs/pkcs-5/`

on how to crack them is not known to the public. Note that AES has yet to receive the test of time, and the CBC variation used is not widely standardized (hence not widely studied). It is associated with the `hmac-sha1-96-aes128` and `hmac-sha1-96-aes256` checksums, respectively.

The protocol do not include any way to negotiate which checksum mechanisms to use, so in most cases the associated checksum will be used. However, checksum mechanisms can be used with other encryption mechanisms, as long as they are compatible in terms of key format etc. Here are the names of the supported checksum mechanisms, with some notes on their status and the compatible encryption mechanisms. They are ordered by increased security as perceived by the author.

`NULL`

`NULL` is a dummy checksum suite for debugging. It provides no integrity. It is weak. It is compatible with the `NULL` encryption mechanism.

`arcfour-hmac-md5`

`arcfour-hmac-md5` is a keyed HMAC-MD5 checksum computed on a MD5 message digest, in turn computed on a four byte message type indicator concatenated with the application data. (The `arcfour` designation is thus somewhat misleading, but since this checksum mechanism is described in the same document as the `arcfour` encryption mechanisms, it is not a completely unnatural designation.) It is weak. It is compatible with all encryption mechanisms.

`rsa-md4`

`rsa-md4` is a unkeyed MD4 hash computed over the message. Since it is unkeyed, it is in general a weak checksum, however applications can, with care, use it non-weak ways (e.g., by including the hash in other messages that are encrypted or checksummed). It is compatible with all encryption mechanisms.

`rsa-md4-des`

`rsa-md4-des` is a DES CBC encryption of one block of random data and a unkeyed MD4 hash computed over the random data and the message to integrity protect. The key used is derived from the base protocol key by XOR with a constant. It is weak. It is compatible with the `des-cbc-crc`, `des-cbc-md4`, `des-cbc-md5` encryption mechanisms.

`rsa-md5`

`rsa-md5` is a unkeyed MD5 hash computed over the message. Since it is unkeyed, it is in general a weak checksum, however applications can, with care, use it non-weak ways (e.g., by including the hash in other messages that are encrypted or checksummed). It is compatible with all encryption mechanisms.

`rsa-md5-des`

`rsa-md5-des` is a DES CBC encryption of one block of random data and a unkeyed MD5 hash computed over the random data and the message to integrity protect. The key used is derived from the base protocol key by XOR with a constant. It is weak. It is compatible with the `des-cbc-crc`, `des-cbc-md4`, `des-cbc-md5` encryption mechanisms.

`hmac-sha1-des3-kd`
: `hmac-sha1-des3-kd` is a keyed SHA1 hash in HMAC mode computed over the message. The key is derived from the base protocol by the simplified key derivation function (similar to the password key derivation functions of `des3-cbc-sha1-kd`). It has no security proof, but is assumed to provide good security, if the key derivation function is good. It is compatible with the `des3-cbc-sha1-kd` encryption mechanism.

`hmac-sha1-96-aes128`
`hmac-sha1-96-aes256`
: `hmac-sha1-96-aes*` are keyed SHA1 hashes in HMAC mode computed over the message and then truncated to 96 bits. The key is derived from the base protocol by the simplified key derivation function (similar to the password key derivation functions of `des3-cbc-sha1-kd`). It has no security proof, but is assumed to provide good security, if the key derivation function is good. It is compatible with the `aes*-cts-hmac-sha1-96` encryption mechanisms.

Several of the cipher suites have long names that can be hard to memorize. For your convenience, the following short-hand aliases exists.

`arcfour`
: Alias for `arcfour-hmac`.

`des-crc`
: Alias for `des-cbc-crc`.

`des-md4`
: Alias for `des-cbc-md4`.

`des-md5`
`des`
: Alias for `des-cbc-md5`.

`des3`
`3des`
: Alias for `des3-cbc-sha1-kd`.

`aes128`
: Alias for `aes128-cts-hmac-sha1-96`.

`aes`
`aes256`
: Alias for `aes256-cts-hmac-sha1-96`.

1.5 Supported Platforms

Shishi has at some point in time been tested on the following platforms. Online build reports for each platforms and Shishi version is available at `http://josefsson.org/autobuild/`.

1. Debian GNU/Linux 3.0 (Woody)

   GCC 2.95.4 and GNU Make. This is the main development platform. `alphaev67-unknown-linux-gnu`, `alphaev6-unknown-linux-gnu`, `arm-unknown-linux-gnu`, `armv4l-unknown-linux-gnu`, `hppa-unknown-linux-gnu`, `hppa64-unknown-linux-gnu`, `i686-pc-linux-gnu`, `ia64-unknown-linux-gnu`, `m68k-unknown-linux-gnu`, `mips-unknown-linux-gnu`, `mipsel-unknown-linux-gnu`, `powerpc-unknown-linux-gnu`, `s390-ibm-linux-gnu`, `sparc-unknown-linux-gnu`, `sparc64-unknown-linux-gnu`.
2. Debian GNU/Linux 2.1

   GCC 2.95.4 and GNU Make. `armv4l-unknown-linux-gnu`.
3. Tru64 UNIX

   Tru64 UNIX C compiler and Tru64 Make. `alphaev67-dec-osf5.1`, `alphaev68-dec-osf5.1`.
4. SuSE Linux 7.1

   GCC 2.96 and GNU Make. `alphaev6-unknown-linux-gnu`, `alphaev67-unknown-linux-gnu`.
5. SuSE Linux 7.2a

   GCC 3.0 and GNU Make. `ia64-unknown-linux-gnu`.
6. SuSE Linux

   GCC 3.2.2 and GNU Make. `x86_64-unknown-linux-gnu` (AMD64 Opteron "Melody").
7. RedHat Linux 7.2

   GCC 2.96 and GNU Make. `alphaev6-unknown-linux-gnu`, `alphaev67-unknown-linux-gnu`, `ia64-unknown-linux-gnu`.
8. RedHat Linux 8.0

   GCC 3.2 and GNU Make. `i686-pc-linux-gnu`.
9. RedHat Advanced Server 2.1

   GCC 2.96 and GNU Make. `i686-pc-linux-gnu`.
10. Slackware Linux 8.0.01

    GCC 2.95.3 and GNU Make. `i686-pc-linux-gnu`.
11. Mandrake Linux 9.0

    GCC 3.2 and GNU Make. `i686-pc-linux-gnu`.
12. IRIX 6.5

    MIPS C compiler, IRIX Make. `mips-sgi-irix6.5`.
13. AIX 4.3.2

    IBM C for AIX compiler, AIX Make. `rs6000-ibm-aix4.3.2.0`.
14. HP-UX 11

    HP-UX C compiler and HP Make. `ia64-hp-hpux11.22`, `hppa2.0w-hp-hpux11.11`.
15. SUN Solaris 2.8

    Sun WorkShop Compiler C 6.0 and SUN Make. `sparc-sun-solaris2.8`.

16. NetBSD 1.6

    GCC 2.95.3 and GNU Make. `alpha-unknown-netbsd1.6`, `i386-unknown-netbsdelf1.6`.

17. OpenBSD 3.1 and 3.2

    GCC 2.95.3 and GNU Make. `alpha-unknown-openbsd3.1`, `i386-unknown-openbsd3.1`.

18. FreeBSD 4.7 and 4.8

    GCC 2.95.4 and GNU Make. `alpha-unknown-freebsd4.7`, `alpha-unknown-freebsd4.8`, `i386-unknown-freebsd4.7`, `i386-unknown-freebsd4.8`.

19. MacOS X 10.2 Server Edition

    GCC 3.1 and GNU Make. `powerpc-apple-darwin6.5`.

If you use Shishi on, or port Shishi to, a new platform please report it to the author (see Section 1.7 [Bug Reports], page 9).

1.6 Downloading and Installing

The package can be downloaded from several places, including `http://josefsson.org/shishi/releases/`. The latest version is stored in a file, e.g., '`shishi-0.0.42.tar.gz`' where the '`0.0.42`' indicate the highest version number.

The package is then extracted, configured and built like many other packages that use Autoconf. For detailed information on configuring and building it, refer to the '`INSTALL`' file that is part of the distribution archive.

Here is an example terminal session that download, configure, build and install the package. You will need a few basic tools, such as '`sh`', '`make`' and '`cc`'.

```
$ wget -q http://josefsson.org/shishi/releases/shishi-0.0.4.tar.gz
$ tar xfz shishi-0.0.4.tar.gz
$ cd shishi-0.0.4/
$ ./configure
...
$ make
...
$ make install
...
```

After this you should be prepared to continue with the user, administration or programming manual, depending on how you want to use Shishi.

1.7 Bug Reports

If you think you have found a bug in Shishi, please investigate it and report it.

- Please make sure that the bug is really in Shishi, and preferably also check that it hasn't already been fixed in the latest version.
- You have to send us a test case that makes it possible for us to reproduce the bug.

- You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way. Make sure that the bug report includes all information you would need to fix this kind of bug for someone else.

Please make an effort to produce a self-contained report, with something definite that can be tested or debugged. Vague queries or piecemeal messages are difficult to act on and don't help the development effort.

If your bug report is good, we will do our best to help you to get a corrected version of the software; if the bug report is poor, we won't do anything about it (apart from asking you to send better bug reports).

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please also send a note.

Send your bug report to:

'`bug-shishi@josefsson.org`'

1.8 Contributing

If you want to submit a patch for inclusion – from solve a typo you discovered, up to adding support for a new feature – you should submit it as a bug report (see Section 1.7 [Bug Reports], page 9). There are some things that you can do to increase the chances for it to be included in the official package.

Unless your patch is very small (say, under 10 lines) we require that you assign the copyright of your work to the Free Software Foundation. This is to protect the freedom of the project. If you have not already signed papers, we will send you the necessary information when you submit your contribution.

For contributions that doesn't consist of actual programming code, the only guidelines are common sense. Use it.

For code contributions, a number of style guides will help you:

- Coding Style. Follow the GNU Standards document (see ⟨undefined⟩ [top], page ⟨undefined⟩).

  If you normally code using another coding standard, there is no problem, but you should use '`indent`' to reformat the code (see ⟨undefined⟩ [top], page ⟨undefined⟩) before submitting your work.
- Use the unified diff format '`diff -u`'.
- Return errors. The only valid reason for ever aborting the execution of the program is due to memory allocation errors, but for that you should call '`xalloc_die`' to allow the application to recover if it wants to.
- Design with thread safety in mind. Don't use global variables. Don't even write to per-handle global variables unless the documented behaviour of the function you write is to write to the per-handle global variable.
- Avoid using the C math library. It causes problems for embedded implementations, and in most situations it is very easy to avoid using it.
- Document your functions. Use comments before each function headers, that, if properly formatted, are extracted into Texinfo manuals and GTK-DOC web pages.

- Supply a ChangeLog and NEWS entries, where appropriate.

2 User Manual

Usually Shishi interacts with you to get some initial authentication information like a password, and then contacts a server to receive a so called ticket granting ticket. From now on, you rarely interacts with Shishi directly. Applications that needs security services instruct the Shishi library to use the ticket granting ticket to get new tickets for various servers. An example could be if you log on to a host remotely via '`telnet`'. The host usually requires authentication before permitting you in. The '`telnet`' client uses the ticket granting ticket to get a ticket for the server, and then use this ticket to authenticate you against the server (typically the server is also authenticated to you). You perform the initial authentication by typing `shishi` at the prompt. Sometimes it is necessary to supply options telling Shishi what your principal name (user name in the Kerberos realm) or realm is. In the example, I specify the client name `simon@JOSEFSSON.ORG`.

```
$ shishi simon@JOSEFSSON.ORG
Enter password for 'simon@JOSEFSSON.ORG':
simon@JOSEFSSON.ORG:
Authtime:       Fri Aug 15 04:44:49 2003
Endtime:        Fri Aug 15 05:01:29 2003
Server:         krbtgt/JOSEFSSON.ORG key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   INITIAL (512)
$
```

As you can see, Shishi also prints a short description of the ticket received.

A logical next step is to display all tickets you have received (by the way, the tickets are usually stored as text in '`~/.shishi/tickets`'). This is achieved by typing `shishi --list`.

```
$ shishi --list
Tickets in `/home/jas/.shishi/tickets':

jas@JOSEFSSON.ORG:
Authtime:       Fri Aug 15 04:49:46 2003
Endtime:        Fri Aug 15 05:06:26 2003
Server:         krbtgt/JOSEFSSON.ORG key des-cbc-md5 (3)
Ticket key:     des-cbc-md5 (3) protected by des-cbc-md5 (3)
Ticket flags:   INITIAL (512)

jas@JOSEFSSON.ORG:
Authtime:       Fri Aug 15 04:49:46 2003
Starttime:      Fri Aug 15 04:49:49 2003
Endtime:        Fri Aug 15 05:06:26 2003
Server:         host/latte.josefsson.org key des-cbc-md5 (3)
Ticket key:     des-cbc-md5 (3) protected by des-cbc-md5 (3)

2 tickets found.
$
```

As you can see, I had a ticket for the server '`host/latte.josefsson.org`' which was generated by '`telnet`':ing to that host.

If, for some reason, you want to manually get a ticket for a specific server, you can use the `shishi --server-name` command. Normally, however, the application that uses Shishi will take care of getting a ticket for the appropriate server, so you normally wouldn't need this command.

```
$ shishi --server-name=user/billg --encryption-type=des-cbc-md4
jas@JOSEFSSON.ORG:
Authtime:       Fri Aug 15 04:49:46 2003
Starttime:      Fri Aug 15 04:54:33 2003
Endtime:        Fri Aug 15 05:06:26 2003
Server:         user/billg key des-cbc-md4 (2)
Ticket key:     des-cbc-md4 (2) protected by des-cbc-md5 (3)
$
```

As you can see, I acquired a ticket for '`user/billg`' with a '`des-cbc-md4`' (see Section 1.4 [Cryptographic Overview], page 4) encryption key specified with the '`--encryption-type`' parameter.

To wrap up this introduction, lets see how you can remove tickets. You may want to do this if you leave your terminal for lunch or similar, and don't want someone to be able to copy the file and then use your credentials. Note that this only destroy the tickets locally, it does not contact any server and tell it that these credentials are no longer valid. So if someone stole your ticket file, you must contact your administrator and have them reset your account, simply using this parameter is not sufficient.

```
$ shishi --server-name=imap/latte.josefsson.org --destroy
1 ticket removed.
$ shishi --server-name=foobar --destroy
No tickets removed.
$ shishi --destroy
3 tickets removed.
$
```

Since the '`--server-name`' parameter takes a long to type, it is possible to type the server name directly, after the client name. The following example demonstrate a AS-REQ followed by a TGS-REQ for a specific server (assuming you did not have any tickets from the start).

```
$ src/shishi simon@latte.josefsson.org imap/latte.josefsson.org
Enter password for 'simon@latte.josefsson.org':
simon@latte.josefsson.org:
Acquired:       Wed Aug 27 17:21:06 2003
Expires:        Wed Aug 27 17:37:46 2003
Server:         imap/latte.josefsson.org key aes256-cts-hmac-sha1-96 (18)
Ticket key:     aes256-cts-hmac-sha1-96 (18) protected by aes256-cts-hmac-sha1-96 (18)
Ticket flags:   FORWARDED PROXIABLE (12)
$
```

Refer to the reference manual for all available parameters (see Section 4.2 [Parameters for shishi], page 22). The rest of this section contains description of more specialized usage modes that can be ignored by most users.

2.1 Proxiable and Proxy Tickets

At times it may be necessary for a principal to allow a service to perform an operation on its behalf. The service must be able to take on the identity of the client, but only for a particular purpose. A principal can allow a service to take on the principal's identity for a particular purpose by granting it a proxy.

The process of granting a proxy using the proxy and proxiable flags is used to provide credentials for use with specific services. Though conceptually also a proxy, users wishing to delegate their identity in a form usable for all purpose MUST use the ticket forwarding mechanism described in the next section to forward a ticket-granting ticket.

The PROXIABLE flag in a ticket is normally only interpreted by the ticket-granting service. It can be ignored by application servers. When set, this flag tells the ticket-granting server that it is OK to issue a new ticket (but not a ticket-granting ticket) with a different network address based on this ticket. This flag is set if requested by the client on initial authentication. By default, the client will request that it be set when requesting a ticket-granting ticket, and reset when requesting any other ticket.

This flag allows a client to pass a proxy to a server to perform a remote request on its behalf (e.g. a print service client can give the print server a proxy to access the client's files on a particular file server in order to satisfy a print request).

In order to complicate the use of stolen credentials, Kerberos tickets are usually valid from only those network addresses specifically included in the ticket[4]. When granting a proxy, the client MUST specify the new network address from which the proxy is to be used, or indicate that the proxy is to be issued for use from any address.

The PROXY flag is set in a ticket by the TGS when it issues a proxy ticket. Application servers MAY check this flag and at their option they MAY require additional authentication from the agent presenting the proxy in order to provide an audit trail.

Here is how you would acquire a PROXY ticket for the service ‘`imap/latte.josefsson.org`’:

```
$ shishi jas@JOSEFSSON.ORG imap/latte.josefsson.org --proxy
Enter password for ‘jas@JOSEFSSON.ORG’:
libshishi: warning: KDC bug: Reply encrypted using wrong key.
jas@JOSEFSSON.ORG:
Authtime:       Mon Sep  8 20:02:35 2003
Starttime:      Mon Sep  8 20:02:36 2003
Endtime:        Tue Sep  9 04:02:35 2003
Server:         imap/latte.josefsson.org key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   PROXY (16)
$
```

As you noticed, this asked for your password. The reason is that proxy tickets must be acquired using a proxiable ticket granting ticket, which was not present. If you often need to get proxy tickets, you may acquire a proxiable ticket granting ticket from the start:

```
$ shishi --proxiable
Enter password for ‘jas@JOSEFSSON.ORG’:
jas@JOSEFSSON.ORG:
Authtime:       Mon Sep  8 20:04:27 2003
Endtime:        Tue Sep  9 04:04:27 2003
Server:         krbtgt/JOSEFSSON.ORG key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   PROXIABLE INITIAL (520)
```

Then you should be able to acquire proxy tickets based on that ticket granting ticket, as follows:

```
$ shishi jas@JOSEFSSON.ORG imap/latte.josefsson.org --proxy
libshishi: warning: KDC bug: Reply encrypted using wrong key.
jas@JOSEFSSON.ORG:
Authtime:       Mon Sep  8 20:04:27 2003
Starttime:      Mon Sep  8 20:04:32 2003
Endtime:        Tue Sep  9 04:04:27 2003
Server:         imap/latte.josefsson.org key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   PROXY (16)
$
```

2.2 Forwardable and Forwarded Tickets

Authentication forwarding is an instance of a proxy where the service that is granted is complete use of the client's identity. An example where it might be used is when a user logs in to a remote system and wants authentication to work from that system as if the login were local.

The FORWARDABLE flag in a ticket is normally only interpreted by the ticket-granting service. It can be ignored by application servers. The FORWARDABLE flag has an interpretation similar to that of the PROXIABLE flag, except ticket-granting tickets may also be issued with different network addresses. This flag is reset by default, but users MAY request that it be set by setting the FORWARDABLE option in the AS request when they request their initial ticket-granting ticket.

This flag allows for authentication forwarding without requiring the user to enter a password again. If the flag is not set, then authentication forwarding is not permitted, but the same result can still be achieved if the user engages in the AS exchange specifying the requested network addresses and supplies a password.

The FORWARDED flag is set by the TGS when a client presents a ticket with the FORWARDABLE flag set and requests a forwarded ticket by specifying the FORWARDED KDC option and supplying a set of addresses for the new ticket. It is also set in all tickets issued based on tickets with the FORWARDED flag set. Application servers may choose to process FORWARDED tickets differently than non-FORWARDED tickets.

If addressless tickets are forwarded from one system to another, clients SHOULD still use this option to obtain a new TGT in order to have different session keys on the different systems.

Here is how you would acquire a FORWARDED ticket for the service '`host/latte.josefsson.org`':

```
$ shishi jas@JOSEFSSON.ORG host/latte.josefsson.org --forwarded
Enter password for ‘jas@JOSEFSSON.ORG’:
libshishi: warning: KDC bug: Reply encrypted using wrong key.
jas@JOSEFSSON.ORG:
Authtime:       Mon Sep  8 20:07:11 2003
Starttime:      Mon Sep  8 20:07:12 2003
Endtime:        Tue Sep  9 04:07:11 2003
Server:         host/latte.josefsson.org key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   FORWARDED (4)
$
```

As you noticed, this asked for your password. The reason is that forwarded tickets must be acquired using a forwardable ticket granting ticket, which was not present. If you often need to get forwarded tickets, you may acquire a forwardable ticket granting ticket from the start:

```
$ shishi --forwardable
Enter password for ‘jas@JOSEFSSON.ORG’:
jas@JOSEFSSON.ORG:
Authtime:       Mon Sep  8 20:08:53 2003
Endtime:        Tue Sep  9 04:08:53 2003
Server:         krbtgt/JOSEFSSON.ORG key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   FORWARDABLE INITIAL (514)
$
```

Then you should be able to acquire forwarded tickets based on that ticket granting ticket, as follows:

```
$ shishi jas@JOSEFSSON.ORG host/latte.josefsson.org --forwarded
libshishi: warning: KDC bug: Reply encrypted using wrong key.
jas@JOSEFSSON.ORG:
Authtime:       Mon Sep  8 20:08:53 2003
Starttime:      Mon Sep  8 20:08:57 2003
Endtime:        Tue Sep  9 04:08:53 2003
Server:         host/latte.josefsson.org key des3-cbc-sha1-kd (16)
Ticket key:     des3-cbc-sha1-kd (16) protected by des3-cbc-sha1-kd (16)
Ticket flags:   FORWARDED (4)
$
```

3 Administration Manual

This section describe how you get the KDC server up and running to answer queries from clients.

First you must create a user database. Currently this is rather simplistic, and the database only contains cryptographic keys. Use the '`shishi --string-to-key`' command to generate keys, and store them in the '`shishid.keys`' file. The file path is '`/usr/local/etc/shishid.keys`' by default, although you can use '`shishid -k`' to specify another location.

Create a random key for the Kerberos Ticket Granting Service for your realm:

```
$ shishi --string-to-key --random \
krbtgt/latte.josefsson.org@latte.josefsson.org | \
tee /usr/local/etc/shishid.keys
-----BEGIN SHISHI KEY-----
Keytype: 18 (aes256-cts-hmac-sha1-96)
Principal: krbtgt/latte.josefsson.org
Realm: latte.josefsson.org

oconxMTf59B5bvTylY+KE4mchA/gtmYI2Qok+48tnSM=
-----END SHISHI KEY-----
$
```

Create a key for a user from a specified password:

```
$ shishi --string-to-key=fnord \
simon@latte.josefsson.org | tee --append \
/usr/local/etc/shishid.keys
-----BEGIN SHISHI KEY-----
Keytype: 18 (aes256-cts-hmac-sha1-96)
Principal: simon
Realm: latte.josefsson.org

c1rqwvYwuDFrABvqWVq9bWUsQWg/xbErsIUmLN+3lYM=
-----END SHISHI KEY-----
$
```

There is nothing special with a ticket granting key, you could have created it based on a password similar to the user key. However, please keep in mind that passwords typically have little entropy.

Finally, create a random key for a service:

```
$ shishi --string-to-key --random \
imap/latte.josefsson.org@latte.josefsson.org | \
tee --append /usr/local/etc/shishid.keys
-----BEGIN SHISHI KEY-----
Keytype: 18 (aes256-cts-hmac-sha1-96)
Principal: imap/latte.josefsson.org
Realm: latte.josefsson.org

ts2v0QHWyW9FyXbWtCvLPqdEc60qPq5Yvat3p82rp5c=
-----END SHISHI KEY-----
$
```

You are now ready to start the KDC. Refer to the reference manual for available parameters (see Section 4.3 [Parameters for shishid], page 24).

```
$ shishid
```

Then you can use '`shishi`' as usual to acquire tickets (see Chapter 2 [User Manual], page 12). The following example demonstrate a AS-REQ for '`krbtgt/latte.josefsson.org`' followed by a TGS-REQ for '`imap/latte.josefsson.org`'.

```
$ shishi simon@latte.josefsson.org imap/latte.josefsson.org
Enter password for 'simon@latte.josefsson.org':
simon@latte.josefsson.org:
Acquired:       Wed Aug 27 17:16:37 2003
Expires:        Wed Aug 27 17:33:17 2003
Server:         imap/latte.josefsson.org key aes256-cts-hmac-sha1-96 (18)
Ticket key:     aes256-cts-hmac-sha1-96 (18) protected by aes256-cts-hmac-sha1-96 (18)
Ticket flags:   FORWARDED PROXIABLE (12)
$
```

4 Reference Manual

This chapter describes in high detail all parameters, configuration file verbs, etc.

4.1 Configuration file

The valid configuration file tokens are described here. The user configuration file is typically located in '`~/.shishi/shishi.conf`' (compare '`shishi --configuration-file`') and the system configuration is typicall located in '`/usr/local/etc/shishi.conf`'. All tokens are valid in both files, and have the same meaning. However, as the system file is supposed to apply to all users on a system, it would not make sense to use some tokens in both files. For example, the '`default-principal`' is rarely useful in a system configuration file.

4.1.1 'default-realm'

Specify the default realm, by default the hostname of the host is used. E.g.,

```
default-realm JOSEFSSON.ORG
```

4.1.2 'default-principal'

Specify the default principal, by default the login username is used. E.g.,

```
default-principal jas
```

4.1.3 'client-kdc-etypes'

Specify which encryption types client asks server to respond in during AS/TGS exchanges. List valid encryption types, in preference order. Supported algorithms include aes256-cts-hmac-sha1-96, aes128-cts-hmac-sha1-96, des3-cbc-sha1-kd, des-cbc-md5, des-cbc-md4, des-cbc-crc and null. This option also indicates which encryption types are accepted by the client when receiving the response. Note that the preference order is not cryptographically protected, so a man in the middle can modify the order without being detected. Thus, only specify encryption types you trust completely here. The default only includes aes256-cts-hmac-sha1-96, as suggested by RFC1510bis. E.g.,

```
client-kdc-etypes=aes256-cts-hmac-sha1-96 des3-cbc-sha1-kd des-cbc-md5
```

4.1.4 'verbose', 'verbose-asn1', 'verbose-noice', 'verbose-crypto'

Enable verbose library messages. E.g.,

```
verbose
verbose-noice
```

4.1.5 'realm-kdc'

Specify KDC addresses for realms. Value is 'REALM,KDCADDRESS[/PROTOCOL][,KDCADDRESS[/PROTOCOL]...

KDCADDRESS is the hostname or IP address of KDC.

Optional PROTOCOL is udp for UDP, tcp for TCP, and TLS for TLS connections. By default UDP is tried first, and TCP used as a fallback if the KRB_ERR_RESPONSE_TOO_BIG error is received.

If not specified, Shishi tries to locate the KDC using SRV RRs, which is recommended. This option should normally only be used during experiments, or to access badly maintained realms.

```
realm-kdc=JOSEFSSON.ORG,ristretto.josefsson.org
```

4.1.6 'server-realm'

Specify realm for servers. Value is 'REALM,SERVERREGEXP[,SERVERREGEXP...]'.

SERVERREGEXP is a regular expression matching servers in the realm. The first match is used. E.g.,

```
server-realm=JOSEFSSON.ORG,.josefsson.org
```

Note: currently not used.

4.1.7 'kdc-timeout', 'kdc-retries'

How long shishi waits for a response from a KDC before continuing to next KDC for realm. The default is 5 seconds. E.g.,

```
kdc-timeout=10
```

How many times shishi sends a request to a KDC before giving up. The default is 3 times. E.g.,

```
kdc-retries=5
```

4.1.8 'stringprocess'

How username and passwords entered from the terminal, or taken from the command line, are processed.

"none": no processing is used.

"stringprep": convert from locale charset to UTF-8 and process using experimental RFC 1510 stringprep profile.

It can also be a string indicating a character set supported by iconv via libstringprep, in which case data is converted from locale charset into the indicated character set. E.g., UTF-8, ISO-8859-1, KOI-8, EBCDIC-IS-FRISS are supported on GNU systems. On some systems you can use "locale -m" to list available character sets. By default, the "none" setting is used which is consistent with RFC 1510 that is silent on the issue. In practice, however, converting to UTF-8 improves interoperability.

E.g.,

```
stringprocess=UTF-8
```

4.1.9 'ticket-life'

Specify default ticket life time.

The string can be in almost any common format. It can contain month names, time zones, 'am' and 'pm', 'yesterday', 'ago', 'next', etc. Refer to the "Date input formats" in the GNU CoreUtils package for entire story (see section "Date input formats" in *GNU CoreUtils*). As an extra feature, if the resulting string you specify has expired within the last 24 hours, an extra day is added to it. This allows you to specify "17:00" to always mean the next 17:00, even if your system clock happens to be 17:30.

The default is 8 hours.

E.g.,

```
#ticket-life=8 hours
#ticket-life=1 day
ticket-life=17:00
```

4.1.10 'renew-life'

Specify how long a renewable ticket should remain renewable.

See ticket-life for the syntax. The extra feature that handles negative values within the last 2 hours is not active here.

The default is 7 days.

E.g.,

```
#renew-life=1 week
#renew-life=friday 17:00
renew-life=sunday
```

4.2 Parameters for shishi

If no command is given, Shishi try to make sure you have a ticket granting ticket for the default realm, and then display it.

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
Usage: shishi [OPTION...] [CLIENT [SERVER]] [OPTION...]
  or:  shishi [OPTION...] --list [CLIENT [SERVER]]
  or:  shishi [OPTION...] --destroy [CLIENT [SERVER]]
  or:  shishi [OPTION...] --string-to-key [CLIENT] [OPTION...]
  or:  shishi [OPTION...]
Shishi -- A Kerberos 5 implementation

      --client-name=NAME     Client name. Default is login username.
  -d, --destroy              Destroy tickets in local cache, subject to
                             --client-name and --server-name limiting.
  -e, --endtime=STRING       Specify when ticket validity should expire.  The
                             time syntax may be relative (to the start time),
```

```
                            such as "20 hours", or absolute, such as
                            "2001-02-03 04:05:06 CET". The default is 8 hours
                            after the start time.
 -E, --encryption-type=ETYPE,[ETYPE...]
                            Encryption types to use.  ETYPE is either
                            registered name or integer.
     --force-as             Force AS mode. Default is to use TGS iff a TGT is
                            found.
     --force-tgs            Force TGS mode. Default is to use TGS iff a TGT is
                            found.
     --forwardable          Get a forwardable ticket, i.e., one that can be
                            used to get forwarded tickets.
     --forwarded            Get a forwarded ticket.
 -l, --list                 List tickets in local cache, subject to
                            --server-name limiting.
     --proxiable            Get a proxiable ticket, i.e., one that can be used
                            to get proxy tickets.
     --proxy                Get a proxy ticket.
     --realm=REALM          Realm of server. Default is DNS domain of local
                            host. For AS, this also indicates realm of client.
     --renew-till=STRING    Specify renewable life of ticket.  Implies
                            --renewable.  Accepts same time syntax as
                            --endtime.  If --renewable is specified, the
                            default is 1 week after the start time.
     --renewable            Get a renewable ticket.
 -R, --renew                Renew ticket.  Use --server-name to specify
                            ticket, default is the most recent renewable
                            ticket granting ticket for the default realm.
     --server=[FAMILY:]ADDRESS:SERVICE/TYPE
                            Send all requests to HOST instead of using normal
                            logic to locate KDC addresses (discouraged).
     --server-name=NAME     Server name. Default is "krbtgt/REALM" where REALM
                            is server realm (see --realm).
 -s, --starttime=STRING     Specify when ticket should start to be valid.
                            Accepts same time syntax as --endtime. The default
                            is to become valid immediately.
     --ticket-granter=NAME  Service name in ticket to use for authenticating
                            request. Only for TGS. Defaults to
                            "krbtgt/REALM@REALM" where REALM is server realm
                            (see --realm).

Options for low-level cryptography (CRYPTO-OPTIONS):
     --client-name=NAME     Username. Default is login name.
     --key-version=INTEGER  Version number of key. Default is 0.
     --parameter=STRING     String-to-key parameter. This data is specific for
                            each encryption algorithm and rarely needed.
     --random               Generate key from random data.
```

```
      --realm=REALM          Realm of principal. Defaults to DNS domain of
                             local host.
      --salt=SALT            Salt to use for --string-to-key. Defaults to
                             concatenation of realm and (unwrapped) client
                             name.
      --string-to-key[=[PASSWORD]]
                             Convert password into Kerberos key.  Note that
                             --client-name, --realm, and --salt influence the
                             generated key.

Other options:
      --configuration-file=FILE   Read user configuration from file.  Default
                             is ~/.shishi/config.
  -c, --ticket-file=FILE     Read tickets from FILE. Default is
                             $HOME/.shishi/tickets.
  -o, --library-options=STRING   Parse STRING as a configuration file
                             statement.
  -q, --quiet, --silent      Don't produce any output.
      --system-configuration-file=FILE
                             Read system wide configuration from file.  Default
                             is /usr/local/etc/shishi.conf.
      --ticket-write-file=FILE   Write tickets to FILE.  Default is to write
                             them back to ticket file.
  -v, --verbose              Produce verbose output.  Use multiple times to
                             increase amount of verbose output.
  CLIENT                     Set client name and realm from NAME.  The
                             --client-name and --realm parameters can be used
                             to override part of NAME.
  SERVER                     Set server name and realm from NAME.  The
                             --server-name and --server-realm parameters can be
                             used to override part of SERVER.

  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

4.3 Parameters for shishid

If no parameters are specified, '`shishid`' listens on the defaults interfaces and answers incoming requests using the keys in the default key file.

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
  -c, --configuration-file=FILE   Read configuration from file.  Default is
                             /usr/local/etc/shishi.conf.
  -k, --key-file=FILE        Read keys from file.  Default is
                             /usr/local/etc/shishid.keys.
```

```
-l, --listen=[FAMILY:]ADDRESS:SERVICE/TYPE,...
                           What to listen on. Family is "IPv4" or "IPv6", if
                           absent the family is decided by
                           gethostbyname(ADDRESS). An address of "*"
                           indicates all addresses on the local host. The
                           default is "IPv4:*:kerberos/udp,
                           IPv4:*:kerberos/tcp, IPv6:*:kerberos/udp,
                           IPv6:*:kerberos/tcp".
-q, -s, --quiet, --silent  Don't produce any output.
-u, --setuid=NAME          After binding socket, set user identity.
-v, --verbose              Produce verbose output.
-?, --help                 Give this help list
    --usage                Give a short usage message
-V, --version              Print program version
```

5 Programming Manual

This chapter describes all the publicly available functions in the library.

5.1 Preparation

To use 'Libshishi', you have to perform some changes to your sources and the build system. The necessary changes are small and explained in the following sections. At the end of this chapter, it is described how the library is initialized, and how the requirements of the library are verified.

A faster way to find out how to adapt your application for use with 'Libshishi' may be to look at the examples at the end of this manual (see Section 5.15 [Examples], page 122).

5.1.1 Header

All interfaces (data types and functions) of the library are defined in the header file 'shishi.h'. You must include this in all programs using the library, either directly or through some other header file, like this:

```
#include <shishi.h>
```

The name space of 'Libshishi' is `shishi_*` for function names, `Shishi*` for data types and `SHISHI_*` for other symbols. In addition the same name prefixes with one prepended underscore are reserved for internal use and should never be used by an application.

5.1.2 Initialization

'Libshishi' must be initialized before it can be used. The library is initialized by calling `shishi_init` (see Section 5.2 [Initialization Functions], page 29). The resources allocated by the initialization process can be released if the application no longer has a need to call 'Libshishi' functions, this is done by calling `shishi_done`.

In order to take advantage of the internationalisation features in 'Libshishi', such as translated error messages, the application must set the current locale using `setlocale` before initializing 'Libshishi'.

5.1.3 Version Check

It is often desirable to check that the version of 'Libshishi' used is indeed one which fits all requirements. Even with binary compatibility new features may have been introduced but due to problem with the dynamic linker an old version is actually used. So you may want to check that the version is okay right after program startup.

`const char *` **shishi_check_version** (`const char *` *req_version*) [Function]

req_version: version string to compare with, or NULL

Check that the the version of the library is at minimum the one given as a string in `req_version`.

the actual version string of the library; NULL if the condition is not met. If *NULL* is passed to this function no check is done and only the version string is returned. It is a pretty good idea to run this function as soon as possible, because it may also intializes some subsystems. In a multithreaded environment if should be called before any more threads are created.

The normal way to use the function is to put something similar to the following early in your `main`:

```
if (!shishi_check_version (SHISHI_VERSION))
  {
    printf ("shishi_check_version failed:\n"
            "Header file incompatible with shared library.\n");
    exit(1);
  }
```

5.1.4 Building the source

If you want to compile a source file including the 'shishi.h' header file, you must make sure that the compiler can find it in the directory hierarchy. This is accomplished by adding the path to the directory in which the header file is located to the compilers include file search path (via the '`-I`' option).

However, the path to the include file is determined at the time the source is configured. To solve this problem, 'Libshishi' uses the external package `pkg-config` that knows the path to the include file and other configuration options. The options that need to be added to the compiler invocation at compile time are output by the '`--cflags`' option to `pkg-config shishi`. The following example shows how it can be used at the command line:

```
gcc -c foo.c `pkg-config shishi --cflags`
```

Adding the output of '`pkg-config shishi --cflags`' to the compilers command line will ensure that the compiler can find the 'Libshishi' header file.

A similar problem occurs when linking the program with the library. Again, the compiler has to find the library files. For this to work, the path to the library files has to be added to the library search path (via the '`-L`' option). For this, the option '`--libs`' to `pkg-config shishi` can be used. For convenience, this option also outputs all other options that are required to link the program with the 'Libshishi' libararies (in particular, the '`-lshishi`' option). The example shows how to link '`foo.o`' with the 'Libshishi' library to a program `foo`.

```
gcc -o foo foo.o `pkg-config shishi --libs`
```

Of course you can also combine both examples to a single command by specifying both options to `pkg-config`:

```
gcc -o foo foo.c `pkg-config shishi --cflags --libs`
```

5.1.5 Autoconf tests

If you work on a project that uses Autoconf (see ⟨undefined⟩ [top], page ⟨undefined⟩) to help find installed libraries, the suggestions in the previous section are not the entire

story. There are a few methods to detect and incorporate Shishi into your Autoconf based package. The preferred approach, is to use Libtool in your project, and use the normal Autoconf header file and library tests.

5.1.5.1 Autoconf test via 'pkg-config'

If your audience is a typical GNU/Linux desktop, you can often assume they have the 'pkg-config' tool installed, in which you can use its Autoconf M4 macro to find and set up your package for use with Shishi. The following illustrate this scenario.

```
AC_ARG_ENABLE(kerberos_v5,
AC_HELP_STRING([--disable-kerberos_v5],
                        [don't use the KERBEROS_V5 mechanism]),
kerberos_v5=$enableval)
if test "$kerberos_v5" != "no" ; then
PKG_CHECK_MODULES(SHISHI, shishi >= 0.0.0,
[kerberos_v5=yes],
                         [kerberos_v5=no])
if test "$kerberos_v5" != "yes" ; then
kerberos_v5=no
AC_MSG_WARN([shishi not found, disabling Kerberos 5])
else
kerberos_v5=yes
AC_DEFINE(USE_KERBEROS_V5, 1,
                           [Define to 1 if you want Kerberos 5.])
fi
fi
AC_MSG_CHECKING([if Kerberos 5 should be used])
AC_MSG_RESULT($kerberos_v5)
```

5.1.5.2 Standalone Autoconf test using Libtool

If your package uses Libtool(see ⟨undefined⟩ [top], page ⟨undefined⟩), you can use the normal Autoconf tests to find the Shishi library and rely on the Libtool dependency tracking to include the proper dependency libraries (e.g., Libidn). The following illustrate this scenario.

```
AC_CHECK_HEADER(shishi.h,
AC_CHECK_LIB(shishi, shishi_check_version,
[kerberos5=yes AC_SUBST(SHISHI_LIBS, -lshishi)],
kerberos5=no),
kerberos5=no)
AC_ARG_ENABLE(kerberos5,
AC_HELP_STRING([--disable-kerberos5],
                        [disable Kerberos 5 unconditionally]),
kerberos5=$enableval)
if test "$kerberos5" != "no" ; then
AC_DEFINE(USE_KERBEROS_V5, 1,
```

```
  [Define to 1 if you want Kerberos 5.])
else
AC_MSG_WARN([Shishi not found, disabling Kerberos 5])
fi
AC_MSG_CHECKING([if Kerberos 5 should be used])
AC_MSG_RESULT($kerberos5)
```

5.1.5.3 Standalone Autoconf test

If your package does not use Libtool, as well as detecting the Shishi library as in the previous case, you must also detect whatever dependencies Shishi requires to work (e.g., libidn). Since the dependencies are in a state of flux, we do not provide an example and we do not recommend this approach, unless you are experienced developer.

5.2 Initialization Functions

`Shishi * shishi ( void)` [Function]

Initializes the Shishi library, and set up, using `shishi_set_outputtype()`, the library so that future warnings and informational messages are printed to stderr. If this function fails, it may print diagnostic errors to stderr.

Returns Shishi library handle, or *NULL* on error.

`Shishi * shishi_server ( void)` [Function]

Initializes the Shishi library, and set up, using `shishi_set_outputtype()`, the library so that future warnings and informational messages are printed to the syslog. If this function fails, it may print diagnostic errors to the syslog.

Returns Shishi library handle, or *NULL* on error.

`void shishi_done (Shishi * handle)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

Deallocates the shishi library handle. The handle must not be used in any calls to shishi functions after this.

If there is a default tkts, it is written to the default tkts file (call `shishi_tkts_default_file_set()` to change the default tkts file). If you do not wish to write the default tkts file, close the default tkts with shishi_tkts_done(handle, NULL) before calling this function.

`int shishi_init (Shishi ** handle)` [Function]

handle: pointer to handle to be created.

Create a Shishi library handle, using `shishi()`, and read the system configuration file, user configuration file and user tickets from their default locations. The paths to the system configuration file is decided at compile time, and is $sysconfdir/shishi.conf. The user configuration file is $HOME/.shishi/config, and the user ticket file is $HOME/.shishi/ticket.

The handle is allocated regardless of return values, except for SHISHI_HANDLE_ERROR which indicates a problem allocating the handle. (The other error conditions comes from reading the files.)

Returns SHISHI_OK iff successful.

`int` **shishi_init_with_paths** (`Shishi ** handle, const char * tktsfile, const char * systemcfgfile, const char * usercfgfile`) [Function]

handle: pointer to handle to be created.

tktsfile: Filename of ticket file, or NULL.

systemcfgfile: Filename of system configuration, or NULL.

usercfgfile: Filename of user configuration, or NULL.

Create a Shishi library handle, using `shishi()`, and read the system configuration file, user configuration file, and user tickets from the specified locations. If any of `usercfgfile` or `systemcfgfile` is NULL, the file is read from its default location, which for the system configuration file is decided at compile time, and is $sysconfdir/shishi.conf, and for the user configuration file is $HOME/.shishi/config. If the ticket file is NULL, a ticket file is not read at all.

The handle is allocated regardless of return values, except for SHISHI_HANDLE_ERROR which indicates a problem allocating the handle. (The other error conditions comes from reading the files.)

Returns SHISHI_OK iff successful.

`int` **shishi_init_server** (`Shishi ** handle`) [Function]

handle: pointer to handle to be created.

Create a Shishi library handle, using `shishi_server()`, and read the system configuration file. The paths to the system configuration file is decided at compile time, and is $sysconfdir/shishi.conf.

The handle is allocated regardless of return values, except for SHISHI_HANDLE_ERROR which indicates a problem allocating the handle. (The other error conditions comes from reading the file.)

Returns SHISHI_OK iff successful.

`int` **shishi_init_server_with_paths** (`Shishi ** handle, const char * systemcfgfile`) [Function]

handle: pointer to handle to be created.

systemcfgfile: Filename of system configuration, or NULL.

Create a Shishi library handle, using `shishi_server()`, and read the system configuration file from specified location. The paths to the system configuration file is decided at compile time, and is $sysconfdir/shishi.conf. The handle is allocated regardless of return values, except for SHISHI_HANDLE_ERROR which indicates a problem allocating the handle. (The other error conditions comes from reading the file.)

Returns SHISHI_OK iff successful.

`int` **shishi_cfg** (`Shishi * handle, char * option`) [Function]

handle: Shishi library handle create by `shishi_init()`.

option: string with shishi library option.

Configure shishi library with given option.

Returns SHISHI_OK if option was valid.

`int` **shishi_cfg_from_file** (`Shishi * handle, const char * cfg`) [Function]

handle: Shishi library handle create by `shishi_init()`.

cfg: filename to read configuration from.

Configure shishi library using configuration file.

Returns SHISHI_OK iff succesful.

`int` **shishi_cfg_print** (`Shishi * handle, FILE * fh`) [Function]

handle: Shishi library handle create by `shishi_init()`.

fh: file descriptor opened for writing.

Print library configuration status, mostly for debugging purposes.

Returns SHISHI_OK.

`const char *` **shishi_cfg_default_systemfile** (`Shishi * handle`) [Function]

handle: Shishi library handle create by `shishi_init()`.

Return system configuration filename.

`const char *` **shishi_cfg_default_userdirectory** (`Shishi * handle`) [Function]

handle: Shishi library handle create by `shishi_init()`.

Return directory with configuration files etc.

`const char *` **shishi_cfg_default_userfile** (`Shishi * handle`) [Function]

handle: Shishi library handle create by `shishi_init()`.

Return user configuration filename.

`int` **shishi_cfg_clientkdcetype** (`Shishi * handle, int32_t ** etypes`) [Function]

handle: Shishi library handle create by `shishi_init()`.

etypes: output array with encryption types.

Set the etypes variable to the array of preferred client etypes.

Return the number of encryption types in the array, 0 means none.

`int` **shishi_cfg_clientkdcetype_set** (`Shishi * handle, char * value`) [Function]

handle: Shishi library handle create by `shishi_init()`.

value: string with encryption types.

Set the "client-kdc-etypes" configuration option from given string. The string contains encryption types (integer or names) separated by comma or whitespace, e.g. "aes256-cts-hmac-sha1-96 des3-cbc-sha1-kd des-cbc-md5".

Return SHISHI_OK iff successful.

`int` **shishi_cfg_authorizationtype_set** (`Shishi * handle, char * value`) [Function]

handle: Shishi library handle create by `shishi_init()`.

value: string with authorization types.

Set the "authorization-types" configuration option from given string. The string contains authorization types (integer or names) separated by comma or whitespace, e.g. "basic k5login".

Return SHISHI_OK iff successful.

5.3 Ticket Set Functions

A "ticket set" is, as the name implies, a collection of tickets. Functions are provided to read tickets from file into a ticket set, to query number of tickets in the set, to extract a given ticket from the set, to search the ticket set for tickets matching certain criterium, to write the ticket set to a file, etc. High level functions for performing a initial authentication (see Section 5.7 [AS Functions], page 67) or subsequent authentication (see Section 5.8 [TGS Functions], page 72) and storing the new ticket in the ticket set are also provided.

To manipulate each individual ticket, See Section 5.6 [Ticket Functions], page 60. For low-level ASN.1 manipulation see See Section 5.9 [Ticket (ASN.1) Functions], page 76.

`char * shishi_tkts_default_file_guess ( void)` [Function]

Guesses the default ticket filename; it is $HOME/.shishi/tickets.

Returns default tkts filename as a string that has to be deallocated with `free()` by the caller.

`const char * shishi_tkts_default_file (Shishi * handle)` [Function]

handle: Shishi library handle create by `shishi_init()`.

Returns the default ticket set filename used in the library. (Not a copy of it, so don't modify or deallocate it.)

`void shishi_tkts_default_file_set (Shishi * handle, const char * tktsfile)` [Function]

handle: Shishi library handle create by `shishi_init()`.

tktsfile: string with new default tkts file name, or NULL to reset to default.

Set the default ticket set filename used in the library. The string is copied into the library, so you can dispose of the variable immediately after calling this function.

`Shishi_tkts * shishi_tkts_default (Shishi * handle)` [Function]

handle: Shishi library handle create by `shishi_init()`.

Return the handle global ticket set.

`int shishi_tkts (Shishi * handle, Shishi_tkts ** tkts)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

tkts: output pointer to newly allocated tkts handle.

Returns *SHISHI_OK* iff successful.

`void shishi_tkts_done (Shishi_tkts ** tkts)` [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

Deallocates all resources associated with ticket set. The ticket set handle must not be used in calls to other shishi_tkts_*() functions after this.

`int` **shishi_tkts_size** (`Shishi_tkts * tkts`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

Returns number of tickets stored in ticket set.

`Shishi_tkt * shishi_tkts_nth` (`Shishi_tkts * tkts`, `int ticketno`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

ticketno: integer indicating requested ticket in ticket set.

Returns a ticket handle to the ticketno:th ticket in the ticket set, or NULL if ticket set is invalid or ticketno is out of bounds. The first ticket is ticketno 0, the second ticketno 1, and so on.

`int shishi_tkts_remove` (`Shishi_tkts * tkts`, `int ticketno`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

ticketno: ticket number of ticket in the set to remove. The first ticket is ticket number 0.

Returns SHISHI_OK if succesful or if ticketno larger than size of ticket set.

`int shishi_tkts_add` (`Shishi_tkts * tkts`, `Shishi_tkt * tkt`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

tkt: ticket to be added to ticket set.

Returns SHISHI_OK iff succesful.

`int shishi_tkts_new` (`Shishi_tkts * tkts`, `Shishi_asn1 ticket`, `Shishi_asn1 enckdcreppart`, `Shishi_asn1 kdcrep`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

ticket: input ticket variable.

enckdcreppart: input ticket detail variable.

kdcrep: input KDC-REP variable.

Allocate a new ticket and add it to the ticket set.

Returns SHISHI_OK iff succesful.

`int shishi_tkts_read` (`Shishi_tkts * tkts`, `FILE * fh`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

fh: file descriptor to read from.

Read tickets from file descriptor and add them to the ticket set.

Returns SHISHI_OK iff succesful.

`int shishi_tkts_from_file` (`Shishi_tkts * tkts`, `const char * filename`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

filename: filename to read tickets from.

Read tickets from file and add them to the ticket set.

Returns SHISHI_OK iff succesful.

`int` **shishi_tkts_write** (`Shishi_tkts * tkts, FILE * fh`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

fh: file descriptor to write tickets to.

Write tickets in set to file descriptor.

Returns SHISHI_OK iff succesful.

`int` **shishi_tkts_expire** (`Shishi_tkts * tkts`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

Remove expired tickets from ticket set.

Returns SHISHI_OK iff succesful.

`int` **shishi_tkts_to_file** (`Shishi_tkts * tkts, const char * filename`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

filename: filename to write tickets to.

Write tickets in set to file.

Returns SHISHI_OK iff succesful.

`int` **shishi_tkts_print_for_service** (`Shishi_tkts * tkts, FILE * fh, const char * service`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

fh: file descriptor to print to.

service: service to limit tickets printed to, or NULL.

Print description of tickets for specified service to file descriptor. If service is NULL, all tickets are printed.

Returns SHISHI_OK iff succesful.

`int` **shishi_tkts_print** (`Shishi_tkts * tkts, FILE * fh`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

fh: file descriptor to print to.

Print description of all tickets to file descriptor.

Returns SHISHI_OK iff succesful.

`int` **shishi_tkt_match_p** (`Shishi_tkt * tkt, Shishi_tkts_hint * hint`) [Function]

tkt: ticket to test hints on.

hint: structure with characteristics of ticket to be found.

Returns 0 iff ticket fails to match given criteria.

`Shishi_tkt *` **shishi_tkts_find** (`Shishi_tkts * tkts, Shishi_tkts_hint * hint`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

hint: structure with characteristics of ticket to be found.

Search the ticketset sequentially (from ticket number 0 through all tickets in the set) for a ticket that fits the given characteristics. If a ticket is found, the hint->startpos field is updated to point to the next ticket in the set, so this function can be called repeatedly with the same hint argument in order to find all tickets matching a certain criterium. Note that if tickets are added to, or removed from, the ticketset during a query with the same hint argument, the hint->startpos field must be updated appropriately.

```
Shishi_tkts_hint hint;
Shishi_tkt tkt;
...
memset(hint, 0, sizeof(hint));
hint.server = "imap/mail.example.org";
tkt = shishi_tkts_find (shishi_tkts_default(handle), hint);
if (!tkt)
printf("No ticket found...\n");
else
...do something with ticket
```

Returns a ticket if found, or NULL if no further matching tickets could be found.

`Shishi_tkt *` **shishi_tkts_find_for_clientserver** (`Shishi_tkts * tkts, const char * client, const char * server`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

client: client name to find ticket for.

server: server name to find ticket for.

Short-hand function for searching the ticket set for a ticket for the given client and server. See `shishi_tkts_find()`.

Returns a ticket if found, or NULL.

`Shishi_tkt *` **shishi_tkts_find_for_server** (`Shishi_tkts * tkts, const char * server`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

server: server name to find ticket for.

Short-hand function for searching the ticket set for a ticket for the given server using the default client principal. See `shishi_tkts_find_for_clientserver()` and `shishi_tkts_find()`.

Returns a ticket if found, or NULL.

`Shishi_tkt *` **shishi_tkts_get_tgt** (`Shishi_tkts * tkts, Shishi_tkts_hint * hint`) [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

hint: structure with characteristics of ticket to begot.

Get a ticket granting ticket (TGT) suitable for acquiring ticket matching the hint. I.e., get a TGT for the server realm in the hint structure (hint->serverrealm), or the default realm if the serverrealm field is NULL. Can result in AS exchange.

Currently this function do not implement cross realm logic.

This function is used by `shishi_tkts_get()`, which is probably what you really want to use unless you have special needs.

Returns a ticket granting ticket if successful, or NULL if this function is unable to acquire on.

`Shishi_tkt * shishi_tkts_get_tgs (Shishi_tkts * tkts, Shishi_tkts_hint * hint, Shishi_tkt * tgt)` [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

hint: structure with characteristics of ticket to begot.

tgt: ticket granting ticket to use.

Get a ticket via TGS exchange using specified ticket granting ticket.

This function is used by `shishi_tkts_get()`, which is probably what you really want to use unless you have special needs.

Returns a ticket if successful, or NULL if this function is unable to acquire on.

`Shishi_tkt * shishi_tkts_get (Shishi_tkts * tkts, Shishi_tkts_hint * hint)` [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

hint: structure with characteristics of ticket to begot.

Get a ticket matching given characteristics. This function first looks in the ticket set for the ticket, then tries to find a suitable TGT, possibly via an AS exchange, using `shishi_tkts_get_tgt()`, and then use that TGT in a TGS exchange to get the ticket.

Currently this function do not implement cross realm logic.

Returns a ticket if found, or NULL if this function is unable to get the ticket.

`Shishi_tkt * shishi_tkts_get_for_clientserver (Shishi_tkts * tkts, const char * client, const char * server)` [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

client: client name to get ticket for.

server: server name to get ticket for.

Short-hand function for getting a ticket for the given client and server. See `shishi_tkts_get()`.

Returns a ticket if found, or NULL.

`Shishi_tkt * shishi_tkts_get_for_server (Shishi_tkts * tkts, const char * server)` [Function]

tkts: ticket set handle as allocated by `shishi_tkts()`.

server: server name to get ticket for.

Short-hand function for getting a ticket for the given server and the default principal client. See `shishi_tkts_get()`.

Returns a ticket if found, or NULL.

5.4 AP-REQ and AP-REP Functions

The "AP-REQ" and "AP-REP" are ASN.1 structures used by application client and servers to prove to each other who they are. The structures contain auxilliary information, together with an authenticator (see Section 5.11 [Authenticator Functions], page 91) which is the real cryptographic proof. The following illustrates the AP-REQ and AP-REP ASN.1 structures.

```
AP-REQ          ::= [APPLICATION 14] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (14),
        ap-options      [2] APOptions,
        ticket          [3] Ticket,
        authenticator   [4] EncryptedData {Authenticator,
                                { keyuse-pa-TGSReq-authenticator
                                  | keyuse-APReq-authenticator }}
}

AP-REP          ::= [APPLICATION 15] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (15),
        enc-part        [2] EncryptedData {EncAPRepPart,
                                { keyuse-EncAPRepPart }}
}

EncAPRepPart    ::= [APPLICATION 27] SEQUENCE {
        ctime           [0] KerberosTime,
        cusec           [1] Microseconds,
        subkey          [2] EncryptionKey OPTIONAL,
        seq-number      [3] UInt32 OPTIONAL
}
```

`int` **shishi_ap** (`Shishi *` *handle*, `Shishi_ap **` *ap*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ap: pointer to new structure that holds information about AP exchange

Create a new AP exchange.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_nosubkey** (`Shishi *` *handle*, `Shishi_ap **` *ap*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ap: pointer to new structure that holds information about AP exchange

Create a new AP exchange without subkey in authenticator.

Returns SHISHI_OK iff successful.

`void` **shishi_ap_done** (`Shishi_ap *` *ap*) [Function]

ap: structure that holds information about AP exchange

Deallocate resources associated with AP exchange. This should be called by the application when it no longer need to utilize the AP exchange handle.

`int` **shishi_ap_set_tktoptions** (`Shishi_ap * ap, Shishi_tkt * tkt, int options`) [Function]

ap: structure that holds information about AP exchange

tkt: ticket to set in AP.

options: AP-REQ options to set in AP.

Set the ticket (see `shishi_ap_tkt_set()`) and set the AP-REQ apoptions (see `shishi_apreq_options_set()`).

Returns SHISHI_OK iff successful.

`int` **shishi_ap_set_tktoptionsdata** (`Shishi_ap * ap, Shishi_tkt * tkt, int options, const char * data, size_t len`) [Function]

ap: structure that holds information about AP exchange

tkt: ticket to set in AP.

options: AP-REQ options to set in AP.

data: input array with data to checksum in Authenticator.

len: length of input array with data to checksum in Authenticator.

Set the ticket (see `shishi_ap_tkt_set()`) and set the AP-REQ apoptions (see `shishi_apreq_options_set()`) and set the Authenticator checksum data.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_set_tktoptionsasn1usage** (`Shishi_ap * ap, Shishi_tkt * tkt, int options, Shishi_asn1 node, char * field, int authenticatorcksumkeyusage, int authenticatorkeyusage`) [Function]

ap: structure that holds information about AP exchange

tkt: ticket to set in AP.

options: AP-REQ options to set in AP.

node: input ASN.1 structure to store as authenticator checksum data.

field: field in ASN.1 structure to use.

authenticatorcksumkeyusage: key usage for checksum in authenticator.

authenticatorkeyusage: key usage for authenticator.

Set ticket, options and authenticator checksum data using `shishi_ap_set_tktoptionsdata()`. The authenticator checksum data is the DER encoding of the ASN.1 field provided.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_tktoptions** (`Shishi * handle, Shishi_ap ** ap, Shishi_tkt * tkt, int options`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ap: pointer to new structure that holds information about AP exchange

tkt: ticket to set in newly created AP.

options: AP-REQ options to set in newly created AP.

Create a new AP exchange using `shishi_ap()`, and set the ticket and AP-REQ apoptions using `shishi_ap_set_tktoption()`.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_tktoptionsdata** (`Shishi * handle, Shishi_ap ** ap, Shishi_tkt * tkt, int options, const char * data, size_t len`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ap: pointer to new structure that holds information about AP exchange

tkt: ticket to set in newly created AP.

options: AP-REQ options to set in newly created AP.

data: input array with data to checksum in Authenticator.

len: length of input array with data to checksum in Authenticator.

Create a new AP exchange using `shishi_ap()`, and set the ticket, AP-REQ apoptions and the Authenticator checksum data using `shishi_ap_set_tktoptionsdata()`.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_tktoptionsasn1usage** (`Shishi * handle, Shishi_ap ** ap, Shishi_tkt * tkt, int options, Shishi_asn1 node, char * field, int authenticatorcksumkeyusage, int authenticatorkeyusage`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ap: pointer to new structure that holds information about AP exchange

tkt: ticket to set in newly created AP.

options: AP-REQ options to set in newly created AP.

node: input ASN.1 structure to store as authenticator checksum data.

field: field in ASN.1 structure to use.

authenticatorcksumkeyusage: key usage for checksum in authenticator.

authenticatorkeyusage: key usage for authenticator.

Create a new AP exchange using `shishi_ap()`, and set ticket, options and authenticator checksum data from the DER encoding of the ASN.1 field using `shishi_ap_set_tktoptionsasn1usage()`.

Returns SHISHI_OK iff successful.

`Shishi_tkt *` **shishi_ap_tkt** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Returns the ticket from the AP exchange, or NULL if not yet set or an error occured.

`void` **shishi_ap_tkt_set** (`Shishi_ap * ap, Shishi_tkt * tkt`) [Function]

ap: structure that holds information about AP exchange

tkt: ticket to store in AP.

Set the Ticket in the AP exchange.

`int` **shishi_ap_authenticator_cksumdata** (`Shishi_ap * ap, char * out, size_t * len`) [Function]

ap: structure that holds information about AP exchange

out: output array that holds authenticator checksum data.

len: on input, maximum length of output array that holds authenticator checksum data, on output actual length of output array that holds authenticator checksum data.

Returns SHISHI_OK if successful, or SHISHI_TOO_SMALL_BUFFER if buffer provided was too small.

`void` **shishi_ap_authenticator_cksumdata_set** (`Shishi_ap * ap, const char * authenticatorcksumdata, size_t authenticatorcksumdatalen`) [Function]

ap: structure that holds information about AP exchange

authenticatorcksumdata: input array with authenticator checksum data to use in AP.

authenticatorcksumdatalen: length of input array with authenticator checksum data to use in AP.

Set the Authenticator Checksum Data in the AP exchange.

`int` **shishi_ap_authenticator_cksumtype** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Get the Authenticator Checksum Type in the AP exchange.

Return the authenticator checksum type.

`void` **shishi_ap_authenticator_cksumtype_set** (`Shishi_ap * ap, int cksumtype`) [Function]

ap: structure that holds information about AP exchange

cksumtype: authenticator checksum type to set in AP.

Set the Authenticator Checksum Type in the AP exchange.

`Shishi_asn1` **shishi_ap_authenticator** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Returns the Authenticator from the AP exchange, or NULL if not yet set or an error occured.

`void` **shishi_ap_authenticator_set** (`Shishi_ap * ap, Shishi_asn1 authenticator`) [Function]

ap: structure that holds information about AP exchange

authenticator: authenticator to store in AP.

Set the Authenticator in the AP exchange.

`Shishi_asn1` **shishi_ap_req** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Returns the AP-REQ from the AP exchange, or NULL if not yet set or an error occured.

`void` **shishi_ap_req_set** (`Shishi_ap * ap, Shishi_asn1 apreq`) [Function]

ap: structure that holds information about AP exchange

apreq: apreq to store in AP.

Set the AP-REQ in the AP exchange.

`int` **shishi_ap_req_der** (`Shishi_ap * ap, char ** out, size_t * outlen`) [Function]

ap: structure that holds information about AP exchange

out: pointer to output array with der encoding of AP-REQ.

outlen: pointer to length of output array with der encoding of AP-REQ.

Build AP-REQ using `shishi_ap_req_build()` and DER encode it. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_req_der_set** (`Shishi_ap * ap, char * der, size_t derlen`) [Function]

ap: structure that holds information about AP exchange

der: input array with DER encoded AP-REQ.

derlen: length of input array with DER encoded AP-REQ.

DER decode AP-REQ and set it AP exchange. If decoding fails, the AP-REQ in the AP exchange is lost.

Returns SHISHI_OK.

`int` **shishi_ap_req_build** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Checksum data in authenticator and add ticket and authenticator to AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_req_process_keyusage** (`Shishi_ap * ap, Shishi_key * key, int32_t keyusage`) [Function]

ap: structure that holds information about AP exchange

key: cryptographic key used to decrypt ticket in AP-REQ.

keyusage: key usage to use during decryption, for normal AP-REQ's this is normally SHISHI_KEYUSAGE_APREQ_AUTHENTICATOR, for AP-REQ's part of TGS-REQ's, this is normally SHISHI_KEYUSAGE_TGSREQ_APREQ_AUTHENTICATOR.

Decrypt ticket in AP-REQ using supplied key and decrypt Authenticator in AP-REQ using key in decrypted ticket, and on success set the Ticket and Authenticator fields in the AP exchange.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_req_process** (`Shishi_ap * ap, Shishi_key * key`) [Function]

ap: structure that holds information about AP exchange

key: cryptographic key used to decrypt ticket in AP-REQ.

Decrypt ticket in AP-REQ using supplied key and decrypt Authenticator in AP-REQ using key in decrypted ticket, and on success set the Ticket and Authenticator fields in the AP exchange.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_req_asn1** (`Shishi_ap * ap, Shishi_asn1 * apreq`) [Function]

ap: structure that holds information about AP exchange

apreq: output AP-REQ variable.

Build AP-REQ using `shishi_ap_req_build()` and return it.

Returns SHISHI_OK iff successful.

`Shishi_key * shishi_ap_key (Shishi_ap * ap)` [Function]

ap: structure that holds information about AP exchange

Extract the application key from AP. If subkeys are used, it is taken from the Authenticator, otherwise the session key is used.

Return application key from AP.

`Shishi_asn1 shishi_ap_rep (Shishi_ap * ap)` [Function]

ap: structure that holds information about AP exchange

Returns the AP-REP from the AP exchange, or NULL if not yet set or an error occured.

`void shishi_ap_rep_set (Shishi_ap * ap, Shishi_asn1 aprep)` [Function]

ap: structure that holds information about AP exchange

aprep: aprep to store in AP.

Set the AP-REP in the AP exchange.

`int shishi_ap_rep_der (Shishi_ap * ap, char ** out, size_t * outlen)` [Function]

ap: structure that holds information about AP exchange

out: output array with newly allocated DER encoding of AP-REP.

outlen: length of output array with DER encoding of AP-REP.

Build AP-REP using `shishi_ap_rep_build()` and DER encode it. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int shishi_ap_rep_der_set (Shishi_ap * ap, char * der, size_t derlen)` [Function]

ap: structure that holds information about AP exchange

der: input array with DER encoded AP-REP.

derlen: length of input array with DER encoded AP-REP.

DER decode AP-REP and set it AP exchange. If decoding fails, the AP-REP in the AP exchange remains.

Returns SHISHI_OK.

`int shishi_ap_rep_build (Shishi_ap * ap)` [Function]

ap: structure that holds information about AP exchange

Checksum data in authenticator and add ticket and authenticator to AP-REP.

Returns SHISHI_OK iff successful.

`int shishi_ap_rep_asn1 (Shishi_ap * ap, Shishi_asn1 * aprep)` [Function]

ap: structure that holds information about AP exchange

aprep: output AP-REP variable.

Build AP-REP using `shishi_ap_rep_build()` and return it.

Returns SHISHI_OK iff successful.

`int` **shishi_ap_rep_verify** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Verify AP-REP compared to Authenticator.

Returns SHISHI_OK, SHISHI_APREP_VERIFY_FAILED or an error.

`int` **shishi_ap_rep_verify_der** (`Shishi_ap * ap, char * der, size_t derlen`) [Function]

ap: structure that holds information about AP exchange

der: input array with DER encoded AP-REP.

derlen: length of input array with DER encoded AP-REP.

DER decode AP-REP and set it in AP exchange using `shishi_ap_rep_der_set()` and verify it using `shishi_ap_rep_verify()`.

Returns SHISHI_OK, SHISHI_APREP_VERIFY_FAILED or an error.

`int` **shishi_ap_rep_verify_asn1** (`Shishi_ap * ap, Shishi_asn1 aprep`) [Function]

ap: structure that holds information about AP exchange

aprep: input AP-REP.

Set the AP-REP in the AP exchange using `shishi_ap_rep_set()` and verify it using `shishi_ap_rep_verify()`.

Returns SHISHI_OK, SHISHI_APREP_VERIFY_FAILED or an error.

`Shishi_asn1` **shishi_ap_encapreppart** (`Shishi_ap * ap`) [Function]

ap: structure that holds information about AP exchange

Returns the EncAPREPPart from the AP exchange, or NULL if not yet set or an error occured.

`void` **shishi_ap_encapreppart_set** (`Shishi_ap * ap, Shishi_asn1 encapreppart`) [Function]

ap: structure that holds information about AP exchange

encapreppart: EncAPRepPart to store in AP.

Set the EncAPRepPart in the AP exchange.

`const char *` **shishi_ap_option2string** (`Shishi_apoptions option`) [Function]

option: enumerated AP-Option type, see Shishi_apoptions.

Convert AP-Option type to AP-Option name string. Note that `option` must be just one of the AP-Option types, it cannot be an binary ORed indicating several AP-Options.

Returns static string with name of AP-Option that must not be deallocated, or "unknown" if AP-Option was not understood.

`Shishi_apoptions` **shishi_ap_string2option** (`const char * str`) [Function]

str: zero terminated character array with name of AP-Option, e.g. "use-session-key".

Convert AP-Option name to AP-Option type.

Returns enumerated type member corresponding to AP-Option, or 0 if string was not understood.

`Shishi_asn1` **shishi_apreq** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new AP-REQ, populated with some default values.

Returns the AP-REQ or NULL on failure.

`int` **shishi_apreq_print** (`Shishi * handle, FILE * fh, Shishi_asn1 apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

apreq: AP-REQ to print.

Print ASCII armored DER encoding of AP-REQ to file.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_save** (`Shishi * handle, FILE * fh, Shishi_asn1 apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

apreq: AP-REQ to save.

Save DER encoding of AP-REQ to file.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_to_file** (`Shishi * handle, Shishi_asn1 apreq, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write AP-REQ to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

apreq: output variable with newly allocated AP-REQ.

Read ASCII armored DER encoded AP-REQ from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

apreq: output variable with newly allocated AP-REQ.

Read DER encoded AP-REQ from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_from_file** (`Shishi * handle, Shishi_asn1 * apreq, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: output variable with newly allocated AP-REQ.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read AP-REQ from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_set_authenticator** (`Shishi * handle, Shishi_asn1 apreq, int32_t etype, const char * buf, size_t buflen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ to add authenticator field to.

etype: encryption type used to encrypt authenticator.

buf: input array with encrypted authenticator.

buflen: size of input array with encrypted authenticator.

Set the encrypted authenticator field in the AP-REP. The encrypted data is usually created by calling `shishi_encrypt()` on the DER encoded authenticator. To save time, you may want to use `shishi_apreq_add_authenticator()` instead, which calculates the encrypted data and calls this function in one step.

`int` **shishi_apreq_add_authenticator** (`Shishi * handle, Shishi_asn1 apreq, Shishi_key * key, int keyusage, Shishi_asn1 authenticator`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ to add authenticator field to.

key: key to to use for encryption.

keyusage: kerberos key usage value to use in encryption.

authenticator: authenticator as allocated by `shishi_authenticator()`.

Encrypts DER encoded authenticator using key and store it in the AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_set_ticket** (`Shishi * handle, Shishi_asn1 apreq, Shishi_asn1 ticket`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ to add ticket field to.

ticket: input ticket to copy into AP-REQ ticket field.

Copy ticket into AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_options** (`Shishi * handle, Shishi_asn1 apreq, int * flags`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ to get options from.

flags: Output integer containing options from AP-REQ.

Extract the AP-Options from AP-REQ into output integer.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_use_session_key_p** (`Shishi * handle, Shishi_asn1 apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ as allocated by `shishi_apreq()`.

Return non-0 iff the "Use session key" option is set in the AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_mutual_required_p** (`Shishi * handle, Shishi_asn1 apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ as allocated by `shishi_apreq()`.

Return non-0 iff the "Mutual required" option is set in the AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_options_set** (`Shishi * handle, Shishi_asn1 apreq, int options`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ as allocated by `shishi_apreq()`.

options: Options to set in AP-REQ.

Set the AP-Options in AP-REQ to indicate integer.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_options_add** (`Shishi * handle, Shishi_asn1 apreq, int option`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ as allocated by `shishi_apreq()`.

option: Options to add in AP-REQ.

Add the AP-Options in AP-REQ. Options not set in input parameter `option` are preserved in the AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_options_remove** (`Shishi * handle, Shishi_asn1 apreq, int option`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ as allocated by `shishi_apreq()`.

option: Options to remove from AP-REQ.

Remove the AP-Options from AP-REQ. Options not set in input parameter `option` are preserved in the AP-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_get_authenticator_etype** (`Shishi * handle, Shishi_asn1 apreq, int32_t * etype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

etype: output variable that holds the value.

Extract KDC-REP.enc-part.etype.

Returns SHISHI_OK iff successful.

`int` **shishi_apreq_get_ticket** (`Shishi * handle, Shishi_asn1 apreq, Shishi_asn1 * ticket`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

apreq: AP-REQ variable to get ticket from.

ticket: output variable to hold extracted ticket.

Extract ticket from AP-REQ.

Returns SHISHI_OK iff successful.

`Shishi_asn1` **shishi_aprep** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new AP-REP, populated with some default values.

Returns the authenticator or NULL on failure.

`int` **shishi_aprep_print** (`Shishi * handle, FILE * fh, Shishi_asn1 aprep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

aprep: AP-REP to print.

Print ASCII armored DER encoding of AP-REP to file.

Returns SHISHI_OK iff successful.

`int` **shishi_aprep_save** (`Shishi * handle, FILE * fh, Shishi_asn1 aprep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

aprep: AP-REP to save.

Save DER encoding of AP-REP to file.

Returns SHISHI_OK iff successful.

`int` **shishi_aprep_to_file** (`Shishi * handle, Shishi_asn1 aprep, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

aprep: AP-REP to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write AP-REP to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_aprep_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * aprep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

aprep: output variable with newly allocated AP-REP.

Read ASCII armored DER encoded AP-REP from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_aprep_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * aprep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

aprep: output variable with newly allocated AP-REP.

Read DER encoded AP-REP from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_aprep_from_file** (`Shishi * handle, Shishi_asn1 * aprep, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

aprep: output variable with newly allocated AP-REP.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read AP-REP from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_aprep_get_enc_part_etype** (`Shishi * handle, Shishi_asn1 aprep, int32_t * etype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

aprep: AP-REP variable to get value from.

etype: output variable that holds the value.

Extract AP-REP.enc-part.etype.

Returns SHISHI_OK iff successful.

`Shishi_asn1` **shishi_encapreppart** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new EncAPRepPart, populated with some default values. It uses the current time as returned by the system for the ctime and cusec fields.

Returns the encapreppart or NULL on failure.

`int` **shishi_encapreppart_print** (`Shishi * handle, FILE * fh, Shishi_asn1 encapreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

encapreppart: EncAPRepPart to print.

Print ASCII armored DER encoding of EncAPRepPart to file.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_save** (`Shishi * handle, FILE * fh, Shishi_asn1 encapreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

encapreppart: EncAPRepPart to save.

Save DER encoding of EncAPRepPart to file.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_to_file** (`Shishi * handle, Shishi_asn1 encapreppart, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write EncAPRepPart to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * encapreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

encapreppart: output variable with newly allocated EncAPRepPart.

Read ASCII armored DER encoded EncAPRepPart from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * encapreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

encapreppart: output variable with newly allocated EncAPRepPart.

Read DER encoded EncAPRepPart from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_from_file** (`Shishi * handle, Shishi_asn1 * encapreppart, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: output variable with newly allocated EncAPRepPart.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read EncAPRepPart from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_get_key** (`Shishi * handle, Shishi_asn1 encapreppart, int32_t * keytype, char * keyvalue, size_t * keyvalue_len`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: input EncAPRepPart variable.

keytype: output variable that holds key type.

keyvalue: output array with key.

keyvalue_len: on input, maximum size of output array with key, on output, holds the actual size of output array with key.

Extract the subkey from the encrypted AP-REP part.

Returns SHISHI_OK iff succesful.

`int` **shishi_encapreppart_ctime** (`Shishi * handle, Shishi_asn1 encapreppart, char ** ctime`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart as allocated by `shishi_encapreppart()`.

ctime: newly allocated zero-terminated character array with client time.

Extract client time from EncAPRepPart.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_ctime_set** (`Shishi * handle, Shishi_asn1 encapreppart, char * ctime`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart as allocated by `shishi_encapreppart()`.

ctime: string with generalized time value to store in EncAPRepPart.

Store client time in EncAPRepPart.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_cusec_get** (`Shishi * handle, Shishi_asn1 encapreppart, int * cusec`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart as allocated by `shishi_encapreppart()`.

cusec: output integer with client microseconds field.

Extract client microseconds field from EncAPRepPart.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_cusec_set** (`Shishi * handle, Shishi_asn1 encapreppart, int cusec`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart as allocated by `shishi_encapreppart()`.

cusec: client microseconds to set in authenticator, 0-999999.

Set the cusec field in the Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_seqnumber_get** (`Shishi * handle, Shishi_asn1 encapreppart, uint32_t * seqnumber`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart as allocated by `shishi_encapreppart()`.

seqnumber: output integer with sequence number field.

Extract sequence number field from EncAPRepPart.

Returns SHISHI_OK iff successful.

`int` **shishi_encapreppart_time_copy** (`Shishi * handle, Shishi_asn1 encapreppart, Shishi_asn1 authenticator`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encapreppart: EncAPRepPart as allocated by `shishi_encapreppart()`.

authenticator: Authenticator to copy time fields from.

Copy time fields from Authenticator into EncAPRepPart.

Returns SHISHI_OK iff successful.

5.5 SAFE and PRIV Functions

The "KRB-SAFE" is an ASN.1 structure used by application client and servers to exchange integrity protected data. The integrity protection is keyed, usually with a key agreed on via the AP exchange (see Section 5.4 [AP-REQ and AP-REP Functions], page 37). The following illustrates the KRB-SAFE ASN.1 structure.

```
KRB-SAFE        ::= [APPLICATION 20] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (20),
        safe-body       [2] KRB-SAFE-BODY,
        cksum           [3] Checksum
}

KRB-SAFE-BODY   ::= SEQUENCE {
        user-data       [0] OCTET STRING,
        timestamp       [1] KerberosTime OPTIONAL,
        usec            [2] Microseconds OPTIONAL,
        seq-number      [3] UInt32 OPTIONAL,
        s-address       [4] HostAddress,
        r-address       [5] HostAddress OPTIONAL
}
```

`int` **shishi_safe** (`Shishi * handle, Shishi_safe ** safe`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: pointer to new structure that holds information about SAFE exchange

Create a new SAFE exchange.

Returns SHISHI_OK iff successful.

`void` **shishi_safe_done** (`Shishi_safe * safe`) [Function]

safe: structure that holds information about SAFE exchange

Deallocate resources associated with SAFE exchange. This should be called by the application when it no longer need to utilize the SAFE exchange handle.

`Shishi_key *` **shishi_safe_key** (`Shishi_safe * safe`) [Function]

safe: structure that holds information about SAFE exchange

Returns the key used in the SAFE exchange, or NULL if not yet set or an error occured.

`void` **shishi_safe_key_set** (`Shishi_safe * safe, Shishi_key * key`) [Function]

safe: structure that holds information about SAFE exchange

key: key to store in SAFE.

Set the Key in the SAFE exchange.

`Shishi_asn1` **shishi_safe_safe** (`Shishi_safe * safe`) [Function]

safe: structure that holds information about SAFE exchange

Returns the ASN.1 safe in the SAFE exchange, or NULL if not yet set or an error occured.

`void` **shishi_safe_safe_set** (`Shishi_safe * safe, Shishi_asn1 asn1safe`) [Function]

safe: structure that holds information about SAFE exchange

asn1safe: KRB-SAFE to store in SAFE exchange.

Set the KRB-SAFE in the SAFE exchange.

`int` **shishi_safe_safe_der** (`Shishi_safe * safe, char ** out, size_t * outlen`) [Function]

safe: safe as allocated by `shishi_safe()`.

out: output array with newly allocated DER encoding of SAFE.

outlen: length of output array with DER encoding of SAFE.

DER encode SAFE structure. Typically `shishi_safe_build()` is used to build the SAFE structure first. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_safe_der_set** (`Shishi_safe * safe, char * der, size_t derlen`) [Function]

safe: safe as allocated by `shishi_safe()`.

der: input array with DER encoded KRB-SAFE.

derlen: length of input array with DER encoded KRB-SAFE.

DER decode KRB-SAFE and set it SAFE exchange. If decoding fails, the KRB-SAFE in the SAFE exchange remains.

Returns SHISHI_OK.

`int` **shishi_safe_print** (`Shishi * handle, FILE * fh, Shishi_asn1 safe`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

safe: SAFE to print.

Print ASCII armored DER encoding of SAFE to file.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_save** (`Shishi * handle, FILE * fh, Shishi_asn1 safe`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

safe: SAFE to save.

Save DER encoding of SAFE to file.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_to_file** (`Shishi * handle, Shishi_asn1 safe, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: SAFE to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write SAFE to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * safe`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

safe: output variable with newly allocated SAFE.

Read ASCII armored DER encoded SAFE from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * safe`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

safe: output variable with newly allocated SAFE.

Read DER encoded SAFE from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_from_file** (`Shishi * handle, Shishi_asn1 * safe, int filetype, const char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: output variable with newly allocated SAFE.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read SAFE from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_cksum** (`Shishi * handle, Shishi_asn1 safe, int32_t * cksumtype, char ** cksum, size_t * cksumlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: safe as allocated by `shishi_safe()`.

cksumtype: output checksum type.

cksum: output array with newly allocated checksum data from SAFE.

cksumlen: output size of output checksum data buffer.

Read checksum value from KRB-SAFE. `cksum` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_set_cksum** (`Shishi * handle, Shishi_asn1 safe, int32_t cksumtype, const char * cksum, size_t cksumlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: safe as allocated by `shishi_safe()`.

cksumtype: input checksum type to store in SAFE.

cksum: input checksum data to store in SAFE.

cksumlen: size of input checksum data to store in SAFE.

Store checksum value in SAFE. A checksum is usually created by calling `shishi_checksum()` on some application specific data using the key from the ticket that is being used. To save time, you may want to use `shishi_safe_build()` instead, which calculates the checksum and calls this function in one step.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_user_data** (`Shishi * handle, Shishi_asn1 safe, char ** userdata, size_t * userdatalen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: safe as allocated by `shishi_safe()`.

userdata: output array with newly allocated user data from KRB-SAFE.

userdatalen: output size of output user data buffer.

Read user data value from KRB-SAFE. `userdata` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_set_user_data** (`Shishi *` *handle*, `Shishi_asn1` *safe*, `const char *` *userdata*, `size_t` *userdatalen*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

safe: safe as allocated by `shishi_safe()`.

userdata: input user application to store in SAFE.

userdatalen: size of input user application to store in SAFE.

Set the application data in SAFE.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_build** (`Shishi_safe *` *safe*, `Shishi_key *` *key*) [Function]

safe: safe as allocated by `shishi_safe()`.

key: key for session, used to compute checksum.

Build checksum and set it in KRB-SAFE. Note that this follows RFC 1510bis and is incompatible with RFC 1510, although presumably few implementations use the RFC1510 algorithm.

Returns SHISHI_OK iff successful.

`int` **shishi_safe_verify** (`Shishi_safe *` *safe*, `Shishi_key *` *key*) [Function]

safe: safe as allocated by `shishi_safe()`.

key: key for session, used to verify checksum.

Verify checksum in KRB-SAFE. Note that this follows RFC 1510bis and is incompatible with RFC 1510, although presumably few implementations use the RFC1510 algorithm.

Returns SHISHI_OK iff successful, SHISHI_SAFE_BAD_KEYTYPE if an incompatible key type is used, or SHISHI_SAFE_VERIFY_FAILED if the actual verification failed.

The "KRB-PRIV" is an ASN.1 structure used by application client and servers to exchange confidential data. The confidentiality is keyed, usually with a key agreed on via the AP exchange (see Section 5.4 [AP-REQ and AP-REP Functions], page 37). The following illustrates the KRB-PRIV ASN.1 structure.

```
KRB-PRIV        ::= [APPLICATION 21] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (21),
                        -- NOTE: there is no [2] tag
        enc-part        [3] EncryptedData -- EncKrbPrivPart
}

EncKrbPrivPart  ::= [APPLICATION 28] SEQUENCE {
        user-data       [0] OCTET STRING,
        timestamp       [1] KerberosTime OPTIONAL,
        usec            [2] Microseconds OPTIONAL,
        seq-number      [3] UInt32 OPTIONAL,
        s-address       [4] HostAddress -- sender's addr --,
        r-address       [5] HostAddress OPTIONAL -- recip's addr
}
```

`int` **shishi_priv** (`Shishi * handle, Shishi_priv ** priv`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

priv: pointer to new structure that holds information about PRIV exchange

Create a new PRIV exchange.

Returns SHISHI_OK iff successful.

`void` **shishi_priv_done** (`Shishi_priv * priv`) [Function]

priv: structure that holds information about PRIV exchange

Deallocate resources associated with PRIV exchange. This should be called by the application when it no longer need to utilize the PRIV exchange handle.

`Shishi_key *` **shishi_priv_key** (`Shishi_priv * priv`) [Function]

priv: structure that holds information about PRIV exchange

Returns the key used in the PRIV exchange, or NULL if not yet set or an error occured.

`void` **shishi_priv_key_set** (`Shishi_priv * priv, Shishi_key * key`) [Function]

priv: structure that holds information about PRIV exchange

key: key to store in PRIV.

Set the Key in the PRIV exchange.

`Shishi_asn1` **shishi_priv_priv** (`Shishi_priv * priv`) [Function]

priv: structure that holds information about PRIV exchange

Returns the ASN.1 priv in the PRIV exchange, or NULL if not yet set or an error occured.

`void` **shishi_priv_priv_set** (`Shishi_priv * priv, Shishi_asn1 asn1priv`) [Function]

priv: structure that holds information about PRIV exchange

asn1priv: KRB-PRIV to store in PRIV exchange.

Set the KRB-PRIV in the PRIV exchange.

`int` **shishi_priv_priv_der** (`Shishi_priv * priv, char ** out, size_t * outlen`) [Function]

priv: priv as allocated by `shishi_priv()`.

out: output array with newly allocated DER encoding of PRIV.

outlen: length of output array with DER encoding of PRIV.

DER encode PRIV structure. Typically `shishi_priv_build()` is used to build the PRIV structure first. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_priv_der_set** (`Shishi_priv * priv, char * der, size_t derlen`) [Function]

priv: priv as allocated by `shishi_priv()`.

der: input array with DER encoded KRB-PRIV.

derlen: length of input array with DER encoded KRB-PRIV.

DER decode KRB-PRIV and set it PRIV exchange. If decoding fails, the KRB-PRIV in the PRIV exchange remains.

Returns SHISHI_OK.

`Shishi_asn1` **shishi_priv_encprivpart** (`Shishi_priv * priv`) [Function]

priv: structure that holds information about PRIV exchange

Returns the ASN.1 encprivpart in the PRIV exchange, or NULL if not yet set or an error occured.

`void` **shishi_priv_encprivpart_set** (`Shishi_priv * priv, Shishi_asn1 asn1encprivpart`) [Function]

priv: structure that holds information about PRIV exchange

asn1encprivpart: ENCPRIVPART to store in PRIV exchange.

Set the ENCPRIVPART in the PRIV exchange.

`int` **shishi_priv_encprivpart_der** (`Shishi_priv * priv, char ** out, size_t * outlen`) [Function]

priv: priv as allocated by `shishi_priv()`.

out: output array with newly allocated DER encoding of ENCPRIVPART.

outlen: length of output array with DER encoding of ENCPRIVPART.

DER encode ENCPRIVPART structure. Typically `shishi_encprivpart_build()` is used to build the ENCPRIVPART structure first. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_encprivpart_der_set** (`Shishi_priv * priv, char * der, size_t derlen`) [Function]

priv: priv as allocated by `shishi_priv()`.

der: input array with DER encoded ENCPRIVPART.

derlen: length of input array with DER encoded ENCPRIVPART.

DER decode ENCPRIVPART and set it PRIV exchange. If decoding fails, the ENCPRIVPART in the PRIV exchange remains.

Returns SHISHI_OK.

`int` **shishi_priv_print** (`Shishi * handle, FILE * fh, Shishi_asn1 priv`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

priv: PRIV to print.

Print ASCII armored DER encoding of PRIV to file.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_save** (`Shishi * handle, FILE * fh, Shishi_asn1 priv`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

priv: PRIV to save.

Save DER encoding of PRIV to file.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_to_file** (`Shishi * handle, Shishi_asn1 priv, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

priv: PRIV to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write PRIV to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * priv`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

priv: output variable with newly allocated PRIV.

Read ASCII armored DER encoded PRIV from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * priv`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

priv: output variable with newly allocated PRIV.

Read DER encoded PRIV from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_from_file** (`Shishi * handle, Shishi_asn1 * priv, int filetype, const char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

priv: output variable with newly allocated PRIV.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read PRIV from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_enc_part_etype** (`Shishi * handle, Shishi_asn1 priv, int32_t * etype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

priv: PRIV variable to get value from.

etype: output variable that holds the value.

Extract PRIV.enc-part.etype.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_set_enc_part** (`Shishi * handle, Shishi_asn1 priv, int32_t etype, const char * encpart, size_t encpartlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

priv: priv as allocated by `shishi_priv()`.

etype: input encryption type to store in PRIV.

encpart: input encrypted data to store in PRIV.

encpartlen: size of input encrypted data to store in PRIV.

Store encrypted data in PRIV. The encrypted data is usually created by calling `shishi_encrypt()` on some application specific data using the key from the ticket that is being used. To save time, you may want to use `shishi_priv_build()` instead, which encryptes the data and calls this function in one step.

Returns SHISHI_OK iff successful.

`int` **shishi_encprivpart_user_data** (`Shishi * handle, Shishi_asn1 encprivpart, char ** userdata, size_t * userdatalen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encprivpart: encprivpart as allocated by `shishi_priv()`.

userdata: output array with newly allocated user data from KRB-PRIV.

userdatalen: output size of output user data buffer.

Read user data value from KRB-PRIV. `userdata` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_encprivpart_set_user_data** (`Shishi * handle, Shishi_asn1 encprivpart, const char * userdata, size_t userdatalen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

encprivpart: encprivpart as allocated by `shishi_priv()`.

userdata: input user application to store in PRIV.

userdatalen: size of input user application to store in PRIV.

Set the application data in PRIV.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_build** (`Shishi_priv * priv, Shishi_key * key`) [Function]

priv: priv as allocated by `shishi_priv()`.

key: key for session, used to encrypt data.

Build checksum and set it in KRB-PRIV. Note that this follows RFC 1510bis and is incompatible with RFC 1510, although presumably few implementations use the RFC1510 algorithm.

Returns SHISHI_OK iff successful.

`int` **shishi_priv_process** (`Shishi_priv * priv, Shishi_key * key`) [Function]

priv: priv as allocated by `shishi_priv()`.

key: key to use to decrypt EncPrivPart.

Decrypt encrypted data in KRB-PRIV and set the EncPrivPart in the PRIV exchange.

Returns SHISHI_OK iff successful, SHISHI_PRIV_BAD_KEYTYPE if an incompatible key type is used, or SHISHI_CRYPTO_ERROR if the actual decryption failed.

5.6 Ticket Functions

`int` **shishi_tkt** (`Shishi * handle, Shishi_tkt ** tkt`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

tkt: output variable with newly allocated ticket.

Create a new ticket handle.

Returns SHISHI_OK iff successful.

`Shishi_tkt *` **shishi_tkt2** (`Shishi * handle, Shishi_asn1 ticket, Shishi_asn1 enckdcreppart, Shishi_asn1 kdcrep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: input variable with ticket.

enckdcreppart: input variable with auxilliary ticket information.

kdcrep: input variable with KDC-REP ticket information.

Create a new ticket handle.

Returns new ticket handle, or *NULL* on error.

`void` **shishi_tkt_done** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Deallocate resources associated with ticket. The ticket must not be used again after this call.

`Shishi_asn1` **shishi_tkt_ticket** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Returns actual ticket.

`Shishi_asn1` **shishi_tkt_enckdcreppart** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Returns auxilliary ticket information.

`void` **shishi_tkt_enckdcreppart_set** (`Shishi_tkt * tkt, Shishi_asn1 enckdcreppart`) [Function]

enckdcreppart: EncKDCRepPart to store in Ticket.

Set the EncKDCRepPart in the Ticket.

`Shishi_asn1` **shishi_tkt_kdcrep** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Returns KDC-REP information.

`Shishi_asn1` **shishi_tkt_encticketpart** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Returns EncTicketPart information.

`void` **shishi_tkt_encticketpart_set** (`Shishi_tkt * tkt, Shishi_asn1 encticketpart`) [Function]

tkt: input variable with ticket info.

encticketpart: encticketpart to store in ticket.

Set the EncTicketPart in the Ticket.

`Shishi_key *` **shishi_tkt_key** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Returns key extracted from enckdcreppart.

`int` **shishi_tkt_key_set** (`Shishi_tkt * tkt, Shishi_key * key`) [Function]

tkt: input variable with ticket info.

key: key to store in ticket.

Set the key in the EncTicketPart.

Returns SHISHI_OK iff successful.

`int` **shishi_tkt_client** (`Shishi_tkt * tkt, char * client, size_t * clientlen`) [Function]

tkt: input variable with ticket info.

client: output buffer that holds client name of ticket.

clientlen: on input, maximum size of output buffer, on output, actual size of output buffer.

Returns client principal of ticket.

`int` **shishi_tkt_client_p** (`Shishi_tkt * tkt, const char * client`) [Function]

tkt: input variable with ticket info.

client: client name of ticket.

Determine if ticket is for specified client.

Returns non-0 iff ticket is for specified client.

`int` **shishi_tkt_cnamerealm_p** (`Shishi_tkt * tkt, const char * client`) [Function]

tkt: input variable with ticket info.

client: principal name (client name and realm) of ticket.

Determine if ticket is for specified client principal.

Returns non-0 iff ticket is for specified client principal.

`int` **shishi_tkt_realm** (`Shishi_tkt * tkt, char ** realm, size_t * realmlen`) [Function]

tkt: input variable with ticket info.

realm: pointer to newly allocated character array with realm name.

realmlen: length of newly allocated character array with realm name.

Extract realm of server in ticket.

Returns SHISHI_OK iff successful.

`int` **shishi_tkt_server_p** (`Shishi_tkt * tkt, const char * server`) [Function]

tkt: input variable with ticket info.

server: server name of ticket.

Determine if ticket is for specified server.

Returns non-0 iff ticket is for specified server.

`int` **shishi_tkt_flags** (`Shishi_tkt * tkt, int * flags`) [Function]

tkt: input variable with ticket info.

flags: pointer to output integer with flags.

Extract flags in ticket.

Returns SHISHI_OK iff successful.

`int` **shishi_tkt_flags_set** (`Shishi_tkt * tkt, int flags`) [Function]

tkt: input variable with ticket info.

flags: integer with flags to store in ticket.

Set flags in ticket. Note that this reset any already existing flags.

Returns SHISHI_OK iff successful.

`int` **shishi_tkt_forwardable_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is forwardable.

The FORWARDABLE flag in a ticket is normally only interpreted by the ticket-granting service. It can be ignored by application servers. The FORWARDABLE flag has an interpretation similar to that of the PROXIABLE flag, except ticket-granting tickets may also be issued with different network addresses. This flag is reset by default, but users MAY request that it be set by setting the FORWARDABLE option in the AS request when they request their initial ticket-granting ticket.

Returns non-0 iff forwardable flag is set in ticket.

`int` **shishi_tkt_forwarded_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is forwarded.

The FORWARDED flag is set by the TGS when a client presents a ticket with the FORWARDABLE flag set and requests a forwarded ticket by specifying the FORWARDED KDC option and supplying a set of addresses for the new ticket. It is also set in all tickets issued based on tickets with the FORWARDED flag set. Application servers may choose to process FORWARDED tickets differently than non-FORWARDED tickets.

Returns non-0 iff forwarded flag is set in ticket.

`int` **shishi_tkt_proxiable_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is proxiable.

The PROXIABLE flag in a ticket is normally only interpreted by the ticket-granting service. It can be ignored by application servers. When set, this flag tells the ticket-granting server that it is OK to issue a new ticket (but not a ticket-granting ticket) with a different network address based on this ticket. This flag is set if requested by the client on initial authentication. By default, the client will request that it be set when requesting a ticket-granting ticket, and reset when requesting any other ticket.

Returns non-0 iff proxiable flag is set in ticket.

`int` **shishi_tkt_proxy_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is proxy ticket.

The PROXY flag is set in a ticket by the TGS when it issues a proxy ticket. Application servers MAY check this flag and at their option they MAY require additional authentication from the agent presenting the proxy in order to provide an audit trail.

Returns non-0 iff proxy flag is set in ticket.

`int` **shishi_tkt_may_postdate_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket may be used to grant postdated tickets.

The MAY-POSTDATE flag in a ticket is normally only interpreted by the ticket-granting service. It can be ignored by application servers. This flag MUST be set in a ticket-granting ticket in order to issue a postdated ticket based on the presented ticket. It is reset by default; it MAY be requested by a client by setting the ALLOW-POSTDATE option in the KRB_AS_REQ message. This flag does not allow a client to obtain a postdated ticket-granting ticket; postdated ticket-granting tickets can only by obtained by requesting the postdating in the KRB_AS_REQ message. The life (endtime-starttime) of a postdated ticket will be the remaining life of the ticket-granting ticket at the time of the request, unless the RENEWABLE option is also set, in which case it can be the full life (endtime-starttime) of the ticket-granting ticket. The KDC MAY limit how far in the future a ticket may be postdated.

Returns non-0 iff may-postdate flag is set in ticket.

`int` **shishi_tkt_postdated_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is postdated.

The POSTDATED flag indicates that a ticket has been postdated. The application server can check the authtime field in the ticket to see when the original authentication occurred. Some services MAY choose to reject postdated tickets, or they may only accept them within a certain period after the original authentication. When the KDC issues a POSTDATED ticket, it will also be marked as INVALID, so that the application client MUST present the ticket to the KDC to be validated before use.

Returns non-0 iff postdated flag is set in ticket.

`int` **shishi_tkt_invalid_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is invalid.

The INVALID flag indicates that a ticket is invalid. Application servers MUST reject tickets which have this flag set. A postdated ticket will be issued in this form. Invalid tickets MUST be validated by the KDC before use, by presenting them to the KDC in a TGS request with the VALIDATE option specified. The KDC will only validate tickets after their starttime has passed. The validation is required so that postdated tickets which have been stolen before their starttime can be rendered permanently invalid (through a hot-list mechanism).

Returns non-0 iff invalid flag is set in ticket.

`int` **shishi_tkt_renewable_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is renewable.

The RENEWABLE flag in a ticket is normally only interpreted by the ticket-granting service (discussed below in section 3.3). It can usually be ignored by application servers. However, some particularly careful application servers MAY disallow renewable tickets.

Returns non-0 iff renewable flag is set in ticket.

`int` **shishi_tkt_initial_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket was issued using AS exchange.

The INITIAL flag indicates that a ticket was issued using the AS protocol, rather than issued based on a ticket-granting ticket. Application servers that want to require the demonstrated knowledge of a client's secret key (e.g. a password-changing program) can insist that this flag be set in any tickets they accept, and thus be assured that the client's key was recently presented to the application client.

Returns non-0 iff initial flag is set in ticket.

`int` **shishi_tkt_pre_authent_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket was pre-authenticated.

The PRE-AUTHENT and HW-AUTHENT flags provide additional information about the initial authentication, regardless of whether the current ticket was issued directly (in which case INITIAL will also be set) or issued on the basis of a ticket-granting ticket (in which case the INITIAL flag is clear, but the PRE-AUTHENT and HW-AUTHENT flags are carried forward from the ticket-granting ticket).

Returns non-0 iff pre-authent flag is set in ticket.

`int` **shishi_tkt_hw_authent_p** (`Shishi_tkt *` *tkt*) [Function]

tkt: input variable with ticket info.

Determine if ticket is authenticated using a hardware token.

The PRE-AUTHENT and HW-AUTHENT flags provide additional information about the initial authentication, regardless of whether the current ticket was issued directly (in which case INITIAL will also be set) or issued on the basis of a ticket-granting ticket (in which case the INITIAL flag is clear, but the PRE-AUTHENT and HW-AUTHENT flags are carried forward from the ticket-granting ticket).

Returns non-0 iff hw-authent flag is set in ticket.

`int` **shishi_tkt_transited_policy_checked_p** (`Shishi_tkt *` *tkt*) [Function]

tkt: input variable with ticket info.

Determine if ticket has been policy checked for transit.

In Kerberos, the application server is ultimately responsible for accepting or rejecting authentication and SHOULD check that only suitably trusted KDCs are relied upon to authenticate a principal. The transited field in the ticket identifies which realms (and thus which KDCs) were involved in the authentication process and an application server would normally check this field. If any of these are untrusted to authenticate the indicated client principal (probably determined by a realm-based policy), the authentication attempt MUST be rejected. The presence of trusted KDCs in this list does not provide any guarantee; an untrusted KDC may have fabricated the list.

While the end server ultimately decides whether authentication is valid, the KDC for the end server's realm MAY apply a realm specific policy for validating the transited field and accepting credentials for cross-realm authentication. When the KDC applies such checks and accepts such cross-realm authentication it will set the TRANSITED-POLICY-CHECKED flag in the service tickets it issues based on the cross-realm TGT. A client MAY request that the KDCs not check the transited field by setting the DISABLE-TRANSITED-CHECK flag. KDCs are encouraged but not required to honor this flag.

Application servers MUST either do the transited-realm checks themselves, or reject cross-realm tickets without TRANSITED-POLICY- CHECKED set.

Returns non-0 iff transited-policy-checked flag is set in ticket.

`int` **shishi_tkt_ok_as_delegate_p** (`Shishi_tkt *` *tkt*) [Function]

tkt: input variable with ticket info.

Determine if ticket is ok as delegated ticket.

The copy of the ticket flags in the encrypted part of the KDC reply may have the OK-AS-DELEGATE flag set to indicates to the client that the server specified in the ticket has been determined by policy of the realm to be a suitable recipient of delegation. A client can use the presence of this flag to help it make a decision whether to delegate credentials (either grant a proxy or a forwarded ticket- granting ticket) to this server. It is acceptable to ignore the value of this flag. When setting this flag, an administrator should consider the security and placement of the server on which the service will run, as well as whether the service requires the use of delegated credentials.

Returns non-0 iff ok-as-delegate flag is set in ticket.

`int` **shishi_tkt_keytype** (`Shishi_tkt *` *`tkt`*`, int32_t *` *`etype`*`)` [Function]

tkt: input variable with ticket info.

etype: pointer to encryption type that is set, see Shishi_etype.

Extract encryption type of key in ticket (really EncKDCRepPart).

Returns SHISHI_OK iff successful.

`int` **shishi_tkt_keytype_p** (`Shishi_tkt *` *`tkt`*`, int32_t` *`etype`*`)` [Function]

tkt: input variable with ticket info.

etype: encryption type, see Shishi_etype.

Determine if key in ticket (really EncKDCRepPart) is of specified key type (really encryption type).

Returns non-0 iff key in ticket is of specified encryption type.

`time_t` **shishi_tkt_lastreqc** (`Shishi_tkt *` *`tkt`*`, Shishi_lrtype` *`lrtype`*`)` [Function]

tkt: input variable with ticket info.

lrtype: lastreq type to extract, see Shishi_lrtype. E.g., SHISHI_LRTYPE_LAST_REQUEST.

Extract C time corresponding to given lastreq type field in the ticket.

Returns C time interpretation of the specified lastreq field, or (time_t) -1.

`time_t` **shishi_tkt_authctime** (`Shishi_tkt *` *`tkt`*`)` [Function]

tkt: input variable with ticket info.

Extract C time corresponding to the authtime field. The field holds the time when the original authentication took place that later resulted in this ticket.

Returns C time interpretation of the endtime in ticket.

`time_t` **shishi_tkt_startctime** (`Shishi_tkt *` *`tkt`*`)` [Function]

tkt: input variable with ticket info.

Extract C time corresponding to the starttime field. The field holds the time where the ticket start to be valid (typically in the past).

Returns C time interpretation of the endtime in ticket.

`time_t` **shishi_tkt_endctime** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Extract C time corresponding to the endtime field. The field holds the time where the ticket stop being valid.

Returns C time interpretation of the endtime in ticket.

`time_t` **shishi_tkt_renew_tillc** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Extract C time corresponding to the renew-till field. The field holds the time where the ticket stop being valid for renewal.

Returns C time interpretation of the renew-till in ticket.

`int` **shishi_tkt_valid_at_time_p** (`Shishi_tkt * tkt, time_t now`) [Function]

tkt: input variable with ticket info.

now: time to check for.

Determine if ticket is valid at a specific point in time.

Returns non-0 iff ticket is valid (not expired and after starttime) at specified time.

`int` **shishi_tkt_valid_now_p** (`Shishi_tkt * tkt`) [Function]

tkt: input variable with ticket info.

Determine if ticket is valid now.

Returns 0 iff ticket is invalid (expired or not yet valid).

`void` **shishi_tkt_lastreq_pretty_print** (`Shishi_tkt * tkt, FILE * fh`) [Function]

tkt: input variable with ticket info.

fh: file handle open for writing.

Print a human readable representation of the various lastreq fields in the ticket (really EncKDCRepPart).

`void` **shishi_tkt_pretty_print** (`Shishi_tkt * tkt, FILE * fh`) [Function]

tkt: input variable with ticket info.

fh: file handle open for writing.

Print a human readable representation of a ticket to file handle.

5.7 AS Functions

The Authentication Service (AS) is used to get an initial ticket using e.g. your password. The following illustrates the AS-REQ and AS-REP ASN.1 structures.

```
-- Request --

AS-REQ          ::= KDC-REQ {10}

KDC-REQ {INTEGER:tagnum}        ::= [APPLICATION tagnum] SEQUENCE {
        pvno            [1] INTEGER (5) -- first tag is [1], not [0] --,
```

```
        msg-type        [2] INTEGER (tagnum),
        padata          [3] SEQUENCE OF PA-DATA OPTIONAL,
        req-body        [4] KDC-REQ-BODY
}

KDC-REQ-BODY    ::= SEQUENCE {
        kdc-options             [0] KDCOptions,
        cname                   [1] PrincipalName OPTIONAL
                                    -- Used only in AS-REQ --,
        realm                   [2] Realm
                                    -- Server's realm
                                    -- Also client's in AS-REQ --,
        sname                   [3] PrincipalName OPTIONAL,
        from                    [4] KerberosTime OPTIONAL,
        till                    [5] KerberosTime,
        rtime                   [6] KerberosTime OPTIONAL,
        nonce                   [7] UInt32,
        etype                   [8] SEQUENCE OF Int32 -- EncryptionType
                                    -- in preference order --,
        addresses               [9] HostAddresses OPTIONAL,
        enc-authorization-data  [10] EncryptedData {
                                        AuthorizationData,
                                        { keyuse-TGSReqAuthData-sesskey
                                          | keyuse-TGSReqAuthData-subkey }
                                     } OPTIONAL,
        additional-tickets      [11] SEQUENCE OF Ticket OPTIONAL
}

-- Reply --

AS-REP          ::= KDC-REP {11, EncASRepPart, {keyuse-EncASRepPart}}

KDC-REP {INTEGER:tagnum,
         TypeToEncrypt,
         UInt32:KeyUsages}      ::= [APPLICATION tagnum] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (tagnum),
        padata          [2] SEQUENCE OF PA-DATA OPTIONAL,
        crealm          [3] Realm,
        cname           [4] PrincipalName,
        ticket          [5] Ticket,
        enc-part        [6] EncryptedData {TypeToEncrypt, KeyUsages}
}

EncASRepPart    ::= [APPLICATION 25] EncKDCRepPart

EncKDCRepPart   ::= SEQUENCE {
```

```
        key             [0] EncryptionKey,
        last-req        [1] LastReq,
        nonce           [2] UInt32,
        key-expiration  [3] KerberosTime OPTIONAL,
        flags           [4] TicketFlags,
        authtime        [5] KerberosTime,
        starttime       [6] KerberosTime OPTIONAL,
        endtime         [7] KerberosTime,
        renew-till      [8] KerberosTime OPTIONAL,
        srealm          [9] Realm,
        sname           [10] PrincipalName,
        caddr           [11] HostAddresses OPTIONAL
}
```

`int` **shishi_as** (`Shishi *` *handle*, `Shishi_as **` *as*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

as: holds pointer to newly allocate Shishi_as structure.

Allocate a new AS exchange variable.

Returns SHISHI_OK iff successful.

`void` **shishi_as_done** (`Shishi_as *` *as*) [Function]

as: structure that holds information about AS exchange

Deallocate resources associated with AS exchange. This should be called by the application when it no longer need to utilize the AS exchange handle.

`Shishi_asn1` **shishi_as_req** (`Shishi_as *` *as*) [Function]

as: structure that holds information about AS exchange

Returns the generated AS-REQ packet from the AS exchange, or NULL if not yet set or an error occured.

`int` **shishi_as_req_build** (`Shishi_as *` *as*) [Function]

as: structure that holds information about AS exchange

Possibly remove unset fields (e.g., rtime).

Returns SHISHI_OK iff successful.

`void` **shishi_as_req_set** (`Shishi_as *` *as*, `Shishi_asn1` *asreq*) [Function]

as: structure that holds information about AS exchange

asreq: asreq to store in AS.

Set the AS-REQ in the AS exchange.

`int` **shishi_as_req_der** (`Shishi_as *` *as*, `char **` *out*, `size_t *` *outlen*) [Function]

as: structure that holds information about AS exchange

out: output array with newly allocated DER encoding of AS-REQ.

outlen: length of output array with DER encoding of AS-REQ.

DER encode AS-REQ. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_as_req_der_set** (`Shishi_as * as, char * der, size_t derlen`) [Function]

as: structure that holds information about AS exchange

der: input array with DER encoded AP-REQ.

derlen: length of input array with DER encoded AP-REQ.

DER decode AS-REQ and set it AS exchange. If decoding fails, the AS-REQ in the AS exchange remains.

Returns SHISHI_OK.

`Shishi_asn1` **shishi_as_rep** (`Shishi_as * as`) [Function]

as: structure that holds information about AS exchange

Returns the received AS-REP packet from the AS exchange, or NULL if not yet set or an error occured.

`int` **shishi_as_rep_process** (`Shishi_as * as, Shishi_key * key, const char * password`) [Function]

as: structure that holds information about AS exchange

key: user's key, used to encrypt the encrypted part of the AS-REP.

password: user's password, used if key is NULL.

Process new AS-REP and set ticket. The key is used to decrypt the AP-REP. If both key and password is NULL, the user is queried for it.

Returns SHISHI_OK iff successful.

`int` **shishi_as_rep_build** (`Shishi_as * as, Shishi_key * key`) [Function]

as: structure that holds information about AS exchange

key: user's key, used to encrypt the encrypted part of the AS-REP.

Build AS-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_as_rep_der** (`Shishi_as * as, char ** out, size_t * outlen`) [Function]

as: structure that holds information about AS exchange

out: output array with newly allocated DER encoding of AS-REP.

outlen: length of output array with DER encoding of AS-REP.

DER encode AS-REP. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`void` **shishi_as_rep_set** (`Shishi_as * as, Shishi_asn1 asrep`) [Function]

as: structure that holds information about AS exchange

asrep: asrep to store in AS.

Set the AS-REP in the AS exchange.

`int` **shishi_as_rep_der_set** (`Shishi_as * as, char * der, size_t derlen`) [Function]

as: structure that holds information about AS exchange

der: input array with DER encoded AP-REP.

derlen: length of input array with DER encoded AP-REP.

DER decode AS-REP and set it AS exchange. If decoding fails, the AS-REP in the AS exchange remains.

Returns SHISHI_OK.

`Shishi_asn1` **shishi_as_krberror** (`Shishi_as * as`) [Function]

as: structure that holds information about AS exchange

Returns the received KRB-ERROR packet from the AS exchange, or NULL if not yet set or an error occured.

`int` **shishi_as_krberror_der** (`Shishi_as * as, char ** out, size_t * outlen`) [Function]

as: structure that holds information about AS exchange

out: output array with newly allocated DER encoding of KRB-ERROR.

outlen: length of output array with DER encoding of KRB-ERROR.

DER encode KRB-ERROR. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`void` **shishi_as_krberror_set** (`Shishi_as * as, Shishi_asn1 krberror`) [Function]

as: structure that holds information about AS exchange

krberror: krberror to store in AS.

Set the KRB-ERROR in the AS exchange.

`Shishi_tkt *` **shishi_as_tkt** (`Shishi_as * as`) [Function]

as: structure that holds information about AS exchange

Returns the newly aquired tkt from the AS exchange, or NULL if not yet set or an error occured.

`void` **shishi_as_tkt_set** (`Shishi_as * as, Shishi_tkt * tkt`) [Function]

as: structure that holds information about AS exchange

tkt: tkt to store in AS.

Set the Tkt in the AS exchange.

`int` **shishi_as_sendrecv** (`Shishi_as * as`) [Function]

as: structure that holds information about AS exchange

Send AS-REQ and receive AS-REP or KRB-ERROR. This is the initial authentication, usually used to acquire a Ticket Granting Ticket.

Returns SHISHI_OK iff successful.

5.8 TGS Functions

The Ticket Granting Service (TGS) is used to get subsequent tickets, authenticated by other tickets (so called ticket granting tickets). The following illustrates the TGS-REQ and TGS-REP ASN.1 structures.

```
-- Request --

TGS-REQ         ::= KDC-REQ {12}

KDC-REQ {INTEGER:tagnum}        ::= [APPLICATION tagnum] SEQUENCE {
        pvno            [1] INTEGER (5) -- first tag is [1], not [0] --,
        msg-type        [2] INTEGER (tagnum),
        padata          [3] SEQUENCE OF PA-DATA OPTIONAL,
        req-body        [4] KDC-REQ-BODY
}

KDC-REQ-BODY    ::= SEQUENCE {
        kdc-options             [0] KDCOptions,
        cname                   [1] PrincipalName OPTIONAL
                                    -- Used only in AS-REQ --,
        realm                   [2] Realm
                                    -- Server's realm
                                    -- Also client's in AS-REQ --,
        sname                   [3] PrincipalName OPTIONAL,
        from                    [4] KerberosTime OPTIONAL,
        till                    [5] KerberosTime,
        rtime                   [6] KerberosTime OPTIONAL,
        nonce                   [7] UInt32,
        etype                   [8] SEQUENCE OF Int32 -- EncryptionType
                                    -- in preference order --,
        addresses               [9] HostAddresses OPTIONAL,
        enc-authorization-data  [10] EncryptedData {
                                        AuthorizationData,
                                        { keyuse-TGSReqAuthData-sesskey
                                          | keyuse-TGSReqAuthData-subkey }
                                     } OPTIONAL,
        additional-tickets      [11] SEQUENCE OF Ticket OPTIONAL
}

-- Reply --

TGS-REP         ::= KDC-REP {13, EncTGSRepPart,
                        { keyuse-EncTGSRepPart-sesskey
                          | keyuse-EncTGSRepPart-subkey }}

KDC-REP {INTEGER:tagnum,
```

```
         TypeToEncrypt,
         UInt32:KeyUsages}      ::= [APPLICATION tagnum] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (tagnum),
        padata          [2] SEQUENCE OF PA-DATA OPTIONAL,
        crealm          [3] Realm,
        cname           [4] PrincipalName,
        ticket          [5] Ticket,
        enc-part        [6] EncryptedData {TypeToEncrypt, KeyUsages}
}

EncTGSRepPart   ::= [APPLICATION 26] EncKDCRepPart

EncKDCRepPart   ::= SEQUENCE {
        key             [0] EncryptionKey,
        last-req        [1] LastReq,
        nonce           [2] UInt32,
        key-expiration  [3] KerberosTime OPTIONAL,
        flags           [4] TicketFlags,
        authtime        [5] KerberosTime,
        starttime       [6] KerberosTime OPTIONAL,
        endtime         [7] KerberosTime,
        renew-till      [8] KerberosTime OPTIONAL,
        srealm          [9] Realm,
        sname           [10] PrincipalName,
        caddr           [11] HostAddresses OPTIONAL
}
```

`int` **shishi_tgs** (`Shishi *` *handle*, `Shishi_tgs **` *tgs*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

tgs: holds pointer to newly allocate Shishi_tgs structure.

Allocate a new TGS exchange variable.

Returns SHISHI_OK iff successful.

`void` **shishi_tgs_done** (`Shishi_tgs *` *tgs*) [Function]

Deallocate resources associated with AS exchange. This should be called by the application when it no longer need to utilize the AS exchange handle.

`Shishi_tkt *` **shishi_tgs_tgtkt** (`Shishi_tgs *` *tgs*) [Function]

tgs: structure that holds information about TGS exchange

Returns the ticket-granting-ticket used in the TGS exchange, or NULL if not yet set or an error occured.

`void` **shishi_tgs_tgtkt_set** (`Shishi_tgs *` *tgs*, `Shishi_tkt *` *tgtkt*) [Function]

tgs: structure that holds information about TGS exchange

tgtkt: ticket granting ticket to store in TGS.

Set the Ticket in the TGS exchange.

`Shishi_ap * shishi_tgs_ap (Shishi_tgs * tgs)` [Function]

tgs: structure that holds information about TGS exchange

Returns the AP exchange (part of TGS-REQ) from the TGS exchange, or NULL if not yet set or an error occured.

`Shishi_asn1 shishi_tgs_req (Shishi_tgs * tgs)` [Function]

tgs: structure that holds information about TGS exchange

Returns the generated TGS-REQ from the TGS exchange, or NULL if not yet set or an error occured.

`void shishi_tgs_req_set (Shishi_tgs * tgs, Shishi_asn1 tgsreq)` [Function]

tgs: structure that holds information about TGS exchange

tgsreq: tgsreq to store in TGS.

Set the TGS-REQ in the TGS exchange.

`int shishi_tgs_req_der (Shishi_tgs * tgs, char ** out, size_t * outlen)` [Function]

tgs: structure that holds information about TGS exchange

out: output array with newly allocated DER encoding of TGS-REQ.

outlen: length of output array with DER encoding of TGS-REQ.

DER encode TGS-REQ. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int shishi_tgs_req_der_set (Shishi_tgs * tgs, char * der, size_t derlen)` [Function]

tgs: structure that holds information about TGS exchange

der: input array with DER encoded AP-REQ.

derlen: length of input array with DER encoded AP-REQ.

DER decode TGS-REQ and set it TGS exchange. If decoding fails, the TGS-REQ in the TGS exchange remains.

Returns SHISHI_OK.

`int shishi_tgs_req_process (Shishi_tgs * tgs)` [Function]

tgs: structure that holds information about TGS exchange

Process new TGS-REQ and set ticket. The key to decrypt the TGS-REQ is taken from the EncKDCReqPart of the TGS tgticket.

Returns SHISHI_OK iff successful.

`int shishi_tgs_req_build (Shishi_tgs * tgs)` [Function]

tgs: structure that holds information about TGS exchange

Checksum data in authenticator and add ticket and authenticator to TGS-REQ.

Returns SHISHI_OK iff successful.

`Shishi_asn1` **shishi_tgs_rep** (`Shishi_tgs * tgs`) [Function]

tgs: structure that holds information about TGS exchange

Returns the received TGS-REP from the TGS exchange, or NULL if not yet set or an error occured.

`int` **shishi_tgs_rep_der** (`Shishi_tgs * tgs, char ** out, size_t * outlen`) [Function]

tgs: structure that holds information about TGS exchange

out: output array with newly allocated DER encoding of TGS-REP.

outlen: length of output array with DER encoding of TGS-REP.

DER encode TGS-REP. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`int` **shishi_tgs_rep_process** (`Shishi_tgs * tgs`) [Function]

tgs: structure that holds information about TGS exchange

Process new TGS-REP and set ticket. The key to decrypt the TGS-REP is taken from the EncKDCRepPart of the TGS tgticket.

Returns SHISHI_OK iff successful.

`int` **shishi_tgs_rep_build** (`Shishi_tgs * tgs, int keyusage, Shishi_key * key`) [Function]

tgs: structure that holds information about TGS exchange

keyusage: keyusage integer.

key: user's key, used to encrypt the encrypted part of the TGS-REP.

Build TGS-REP.

Returns SHISHI_OK iff successful.

`Shishi_asn1` **shishi_tgs_krberror** (`Shishi_tgs * tgs`) [Function]

tgs: structure that holds information about TGS exchange

Returns the received TGS-REP from the TGS exchange, or NULL if not yet set or an error occured.

`int` **shishi_tgs_krberror_der** (`Shishi_tgs * tgs, char ** out, size_t * outlen`) [Function]

tgs: structure that holds information about TGS exchange

out: output array with newly allocated DER encoding of KRB-ERROR.

outlen: length of output array with DER encoding of KRB-ERROR.

DER encode KRB-ERROR. `out` is allocated by this function, and it is the responsibility of caller to deallocate it.

Returns SHISHI_OK iff successful.

`void` **shishi_tgs_krberror_set** (`Shishi_tgs * tgs, Shishi_asn1 krberror`) [Function]

tgs: structure that holds information about TGS exchange

krberror: krberror to store in TGS.

Set the KRB-ERROR in the TGS exchange.

`Shishi_tkt * shishi_tgs_tkt (Shishi_tgs * tgs)` [Function]

tgs: structure that holds information about TGS exchange

Returns the newly aquired ticket from the TGS exchange, or NULL if not yet set or an error occured.

`void shishi_tgs_tkt_set (Shishi_tgs * tgs, Shishi_tkt * tkt)` [Function]

tgs: structure that holds information about TGS exchange

tkt: ticket to store in TGS.

Set the Ticket in the TGS exchange.

`int shishi_tgs_sendrecv (Shishi_tgs * tgs)` [Function]

tgs: structure that holds information about TGS exchange

Send TGS-REQ and receive TGS-REP or KRB-ERROR. This is the subsequent authentication, usually used to acquire server tickets.

Returns SHISHI_OK iff successful.

`int shishi_tgs_set_server (Shishi_tgs * tgs, const char * server)` [Function]

tgs: structure that holds information about TGS exchange

server: indicates the server to acquire ticket for.

Set the server in the TGS-REQ.

Returns SHISHI_OK iff successful.

`int shishi_tgs_set_realm (Shishi_tgs * tgs, const char * realm)` [Function]

tgs: structure that holds information about TGS exchange

realm: indicates the realm to acquire ticket for.

Set the server in the TGS-REQ.

Returns SHISHI_OK iff successful.

`int shishi_tgs_set_realmserver (Shishi_tgs * tgs, const char * realm, const char * server)` [Function]

tgs: structure that holds information about TGS exchange

realm: indicates the realm to acquire ticket for.

server: indicates the server to acquire ticket for.

Set the realm and server in the TGS-REQ.

Returns SHISHI_OK iff successful.

5.9 Ticket (ASN.1) Functions

`int shishi_ticket_realm_get (Shishi * handle, Shishi_asn1 ticket, char ** realm, size_t * realmlen)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: input variable with ticket info.

realm: output array with newly allocated name of realm in ticket.

realmlen: size of output array.

Extract realm from ticket.

Returns SHISHI_OK iff successful.

`int` **shishi_ticket_realm_set** (`Shishi * handle, Shishi_asn1 ticket, const char * realm`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: input variable with ticket info.

realm: input array with name of realm.

Set the realm field in the Ticket.

Returns SHISHI_OK iff successful.

`int` **shishi_ticket_sname_set** (`Shishi * handle, Shishi_asn1 ticket, Shishi_name_type name_type, char * sname[]`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: Ticket variable to set server name field in.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

Set the server name field in the Ticket.

Returns SHISHI_OK iff successful.

`int` **shishi_ticket_get_enc_part_etype** (`Shishi * handle, Shishi_asn1 ticket, int32_t * etype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: Ticket variable to get value from.

etype: output variable that holds the value.

Extract Ticket.enc-part.etype.

Returns SHISHI_OK iff successful.

`int` **shishi_ticket_set_enc_part** (`Shishi * handle, Shishi_asn1 ticket, int etype, int kvno, char * buf, size_t buflen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: Ticket to add enc-part field to.

etype: encryption type used to encrypt enc-part.

kvno: key version number.

buf: input array with encrypted enc-part.

buflen: size of input array with encrypted enc-part.

Set the encrypted enc-part field in the Ticket. The encrypted data is usually created by calling `shishi_encrypt()` on the DER encoded enc-part. To save time, you may want to use `shishi_ticket_add_enc_part()` instead, which calculates the encrypted data and calls this function in one step.

Returns SHISHI_OK iff successful.

`int` **shishi_ticket_add_enc_part** (`Shishi * handle, Shishi_asn1 ticket, Shishi_key * key, Shishi_asn1 encticketpart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

ticket: Ticket to add enc-part field to.

key: key used to encrypt enc-part.

encticketpart: EncTicketPart to add.
Encrypts DER encoded EncTicketPart using key and stores it in the Ticket.
Returns SHISHI_OK iff successful.

5.10 AS/TGS Functions

The Authentication Service (AS) is used to get an initial ticket using e.g. your password. The Ticket Granting Service (TGS) is used to get subsequent tickets using other tickets. Protocol wise the procedures are very similar, which is the reason they are described together. The following illustrates the AS-REQ, TGS-REQ and AS-REP, TGS-REP ASN.1 structures. Most of the functions use the mnemonic "KDC" instead of either AS or TGS, which means the function operates on both AS and TGS types. Only where the distinction between AS and TGS is important are the AS and TGS names used. Remember, these are low-level functions, and normal applications will likely be satisfied with the AS (see Section 5.7 [AS Functions], page 67) and TGS (see Section 5.8 [TGS Functions], page 72) interfaces, or the even more high-level Ticket Set (see Section 5.3 [Ticket Set Functions], page 32) interface.

```
-- Request --

AS-REQ          ::= KDC-REQ {10}
TGS-REQ         ::= KDC-REQ {12}

KDC-REQ {INTEGER:tagnum}        ::= [APPLICATION tagnum] SEQUENCE {
        pvno            [1] INTEGER (5) -- first tag is [1], not [0] --,
        msg-type        [2] INTEGER (tagnum),
        padata          [3] SEQUENCE OF PA-DATA OPTIONAL,
        req-body        [4] KDC-REQ-BODY
}

KDC-REQ-BODY    ::= SEQUENCE {
        kdc-options             [0] KDCOptions,
        cname                   [1] PrincipalName OPTIONAL
                                    -- Used only in AS-REQ --,
        realm                   [2] Realm
                                    -- Server's realm
                                    -- Also client's in AS-REQ --,
        sname                   [3] PrincipalName OPTIONAL,
        from                    [4] KerberosTime OPTIONAL,
        till                    [5] KerberosTime,
        rtime                   [6] KerberosTime OPTIONAL,
        nonce                   [7] UInt32,
        etype                   [8] SEQUENCE OF Int32 -- EncryptionType
                                    -- in preference order --,
        addresses               [9] HostAddresses OPTIONAL,
        enc-authorization-data  [10] EncryptedData {
                                        AuthorizationData,
```

```
                                        { keyuse-TGSReqAuthData-sesskey
                                          | keyuse-TGSReqAuthData-subkey }
                                    } OPTIONAL,
        additional-tickets      [11] SEQUENCE OF Ticket OPTIONAL
}

-- Reply --

AS-REP          ::= KDC-REP {11, EncASRepPart, {keyuse-EncASRepPart}}
TGS-REP         ::= KDC-REP {13, EncTGSRepPart,
                        { keyuse-EncTGSRepPart-sesskey
                          | keyuse-EncTGSRepPart-subkey }}

KDC-REP {INTEGER:tagnum,
         TypeToEncrypt,
         UInt32:KeyUsages}      ::= [APPLICATION tagnum] SEQUENCE {
        pvno            [0] INTEGER (5),
        msg-type        [1] INTEGER (tagnum),
        padata          [2] SEQUENCE OF PA-DATA OPTIONAL,
        crealm          [3] Realm,
        cname           [4] PrincipalName,
        ticket          [5] Ticket,
        enc-part        [6] EncryptedData {TypeToEncrypt, KeyUsages}
}

EncASRepPart    ::= [APPLICATION 25] EncKDCRepPart
EncTGSRepPart   ::= [APPLICATION 26] EncKDCRepPart

EncKDCRepPart   ::= SEQUENCE {
        key             [0] EncryptionKey,
        last-req        [1] LastReq,
        nonce           [2] UInt32,
        key-expiration  [3] KerberosTime OPTIONAL,
        flags           [4] TicketFlags,
        authtime        [5] KerberosTime,
        starttime       [6] KerberosTime OPTIONAL,
        endtime         [7] KerberosTime,
        renew-till      [8] KerberosTime OPTIONAL,
        srealm          [9] Realm,
        sname           [10] PrincipalName,
        caddr           [11] HostAddresses OPTIONAL
}
```

int shishi_as_derive_salt (Shishi * *handle*, Shishi_asn1 *asreq*, Shishi_asn1 *asrep*, char * *salt*, size_t * *saltlen*) [Function]

handle: shishi handle as allocated by shishi_init().

asreq: input AS-REQ variable.

asrep: input AS-REP variable.

salt: output array with salt.

saltlen: on input, maximum size of output array with salt, on output, holds actual size of output array with salt.

Derive the salt that should be used when deriving a key via `shishi_string_to_key()` for an AS exchange. Currently this searches for PA-DATA of type SHISHI_PA_PW_SALT in the AS-REP and returns it if found, otherwise the salt is derived from the client name and realm in AS-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_kdc_copy_crealm** (`Shishi *` *`handle`*`, Shishi_asn1` *`kdcrep`*`, Shishi_asn1` *`encticketpart`*`)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP to read crealm from.

encticketpart: EncTicketPart to set crealm in.

Set crealm in KDC-REP to value in EncTicketPart.

Returns SHISHI_OK if successful.

`int` **shishi_as_check_crealm** (`Shishi *` *`handle`*`, Shishi_asn1` *`asreq`*`, Shishi_asn1` *`asrep`*`)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

asreq: AS-REQ to compare realm field in.

asrep: AS-REP to compare realm field in.

Verify that AS-REQ.req-body.realm and AS-REP.crealm fields matches. This is one of the steps that has to be performed when processing a AS-REQ and AS-REP exchange, see `shishi_kdc_process()`.

Returns SHISHI_OK if successful, SHISHI_REALM_MISMATCH if the values differ, or an error code.

`int` **shishi_kdc_copy_cname** (`Shishi *` *`handle`*`, Shishi_asn1` *`kdcrep`*`, Shishi_asn1` *`encticketpart`*`)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REQ to read cname from.

encticketpart: EncTicketPart to set cname in.

Set cname in KDC-REP to value in EncTicketPart.

Returns SHISHI_OK if successful.

`int` **shishi_as_check_cname** (`Shishi *` *`handle`*`, Shishi_asn1` *`asreq`*`, Shishi_asn1` *`asrep`*`)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

asreq: AS-REQ to compare client name field in.

asrep: AS-REP to compare client name field in.

Verify that AS-REQ.req-body.realm and AS-REP.crealm fields matches. This is one of the steps that has to be performed when processing a AS-REQ and AS-REP exchange, see `shishi_kdc_process()`.

Returns SHISHI_OK if successful, SHISHI_CNAME_MISMATCH if the values differ, or an error code.

`int` **shishi_kdc_copy_nonce** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_asn1 enckdcreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to read nonce from.

enckdcreppart: EncKDCRepPart to set nonce in.

Set nonce in EncKDCRepPart to value in KDC-REQ.

Returns SHISHI_OK if successful.

`int` **shishi_kdc_check_nonce** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_asn1 enckdcreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to compare nonce field in.

enckdcreppart: Encrypted KDC-REP part to compare nonce field in.

Verify that KDC-REQ.req-body.nonce and EncKDCRepPart.nonce fields matches. This is one of the steps that has to be performed when processing a KDC-REQ and KDC-REP exchange.

Returns SHISHI_OK if successful, SHISHI_NONCE_LENGTH_MISMATCH if the nonces have different lengths (usually indicates that buggy server truncated nonce to 4 bytes), SHISHI_NONCE_MISMATCH if the values differ, or an error code.

`int` **shishi_tgs_process** (`Shishi * handle, Shishi_asn1 tgsreq, Shishi_asn1 tgsrep, Shishi_asn1 authenticator, Shishi_asn1 oldenckdcreppart, Shishi_asn1 * enckdcreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

tgsreq: input variable that holds the sent KDC-REQ.

tgsrep: input variable that holds the received KDC-REP.

authenticator: input variable with Authenticator from AP-REQ in KDC-REQ.

oldenckdcreppart: input variable with EncKDCRepPart used in request.

enckdcreppart: output variable that holds new EncKDCRepPart.

Process a TGS client exchange and output decrypted EncKDCRepPart which holds details for the new ticket received. This function simply derives the encryption key from the ticket used to construct the TGS request and calls `shishi_kdc_process()`, which see.

Returns SHISHI_OK iff the TGS client exchange was successful.

`int` **shishi_as_process** (`Shishi * handle, Shishi_asn1 asreq, Shishi_asn1 asrep, const char * string, Shishi_asn1 * enckdcreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

asreq: input variable that holds the sent KDC-REQ.

asrep: input variable that holds the received KDC-REP.

string: input variable with zero terminated password.

enckdcreppart: output variable that holds new EncKDCRepPart.

Process an AS client exchange and output decrypted EncKDCRepPart which holds details for the new ticket received. This function simply derives the encryption key from the password and calls `shishi_kdc_process()`, which see.

Returns SHISHI_OK iff the AS client exchange was successful.

`int` **shishi_kdc_process** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_asn1 kdcrep, Shishi_key * key, int keyusage, Shishi_asn1 * enckdcreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: input variable that holds the sent KDC-REQ.

kdcrep: input variable that holds the received KDC-REP.

key: input array with key to decrypt encrypted part of KDC-REP with.

keyusage: kereros key usage value.

enckdcreppart: output variable that holds new EncKDCRepPart.

Process a KDC client exchange and output decrypted EncKDCRepPart which holds details for the new ticket received. Use `shishi_kdcrep_get_ticket()` to extract the ticket. This function verifies the various conditions that must hold if the response is to be considered valid, specifically it compares nonces (`shishi_check_nonces()`) and if the exchange was a AS exchange, it also compares cname and crealm (`shishi_check_cname()` and `shishi_check_crealm()`).

Usually the `shishi_as_process()` and `shishi_tgs_process()` functions should be used instead, since they simplify the decryption key computation.

Returns SHISHI_OK iff the KDC client exchange was successful.

`Shishi_asn1` **shishi_asreq** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new AS-REQ, populated with some default values.

Returns the AS-REQ or NULL on failure.

`Shishi_asn1` **shishi_tgsreq** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new TGS-REQ, populated with some default values.

Returns the TGS-REQ or NULL on failure.

`int` **shishi_kdcreq_print** (`Shishi * handle, FILE * fh, Shishi_asn1 kdcreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

kdcreq: KDC-REQ to print.

Print ASCII armored DER encoding of KDC-REQ to file.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_save** (`Shishi * handle, FILE * fh, Shishi_asn1 kdcreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

kdcreq: KDC-REQ to save.

Print DER encoding of KDC-REQ to file.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_to_file** (`Shishi * handle, Shishi_asn1 kdcreq, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write KDC-REQ to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * kdcreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

kdcreq: output variable with newly allocated KDC-REQ.

Read ASCII armored DER encoded KDC-REQ from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * kdcreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

kdcreq: output variable with newly allocated KDC-REQ.

Read DER encoded KDC-REQ from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_from_file** (`Shishi * handle, Shishi_asn1 * kdcreq, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: output variable with newly allocated KDC-REQ.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read KDC-REQ from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_set_cname** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_name_type name_type, const char * principal`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set client name field in.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

principal: input array with principal name.

Set the client name field in the KDC-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_set_realm** (`Shishi * handle, Shishi_asn1 kdcreq, const char * realm`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set realm field in.

realm: input array with name of realm.

Set the realm field in the KDC-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_set_sname** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_name_type name_type, const char * sname[]`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set server name field in.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

Set the server name field in the KDC-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_etype** (`Shishi * handle, Shishi_asn1 kdcreq, int32_t * etype, int netype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to get etype field from.

etype: output encryption type.

netype: element number to return.

th encryption type from KDC-REQ. The first etype is number 1.

Returns SHISHI_OK iff etype successful set.

`int` **shishi_kdcreq_set_etype** (`Shishi * handle, Shishi_asn1 kdcreq, int32_t * etype, int netype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set etype field in.

etype: input array with encryption types.

netype: number of elements in input array with encryption types.

Set the list of supported or wanted encryption types in the request. The list should be sorted in priority order.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_options** (`Shishi * handle, Shishi_asn1 kdcreq, uint32_t * flags`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set etype field in.

flags: pointer to output integer with flags.

Extract KDC-Options from KDC-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_renewable_p** (`Shishi * handle, Shishi_asn1 kdcreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set etype field in.

Determine if KDC-Option renewable flag is set.

The RENEWABLE option indicates that the ticket to be issued is to have its RENEWABLE flag set. It may only be set on the initial request, or when the ticket-granting ticket on which the request is based is also renewable. If this option is requested, then the rtime field in the request contains the desired absolute expiration time for the ticket.

Returns non-0 iff renewable flag is set in KDC-REQ.

`int` **shishi_kdcreq_options_set** (`Shishi * handle, Shishi_asn1 kdcreq, uint32_t options`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set etype field in.

options: integer with flags to store in KDC-REQ.

Set options in KDC-REQ. Note that this reset any already existing flags.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_options_add** (`Shishi * handle, Shishi_asn1 kdcreq, uint32_t option`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ variable to set etype field in.

option: integer with options to add in KDC-REQ.

Add KDC-Option to KDC-REQ. This preserves all existing options.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_clear_padata** (`Shishi * handle, Shishi_asn1 kdcreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to remove PA-DATA from.

Remove the padata field from KDC-REQ.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_get_padata** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_padata_type padatatype, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to get PA-DATA from.

padatatype: type of PA-DATA, see Shishi_padata_type.

out: output array with newly allocated PA-DATA value.

outlen: size of output array with PA-DATA value.

Get pre authentication data (PA-DATA) from KDC-REQ. Pre authentication data is used to pass various information to KDC, such as in case of a SHISHI_PA_TGS_REQ padatatype the AP-REQ that authenticates the user to get the ticket.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_get_padata_tgs** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_asn1 * apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to get PA-TGS-REQ from.

apreq: Output variable with newly allocated AP-REQ.

Extract TGS pre-authentication data from KDC-REQ. The data is an AP-REQ that authenticates the request. This function call `shishi_kdcreq_get_padata()` with a SHISHI_PA_TGS_REQ padatatype and DER decode the result (if any).

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_add_padata** (`Shishi * handle, Shishi_asn1 kdcreq, int padatatype, const char * data, size_t datalen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to add PA-DATA to.

padatatype: type of PA-DATA, see Shishi_padata_type.

data: input array with PA-DATA value.

datalen: size of input array with PA-DATA value.

Add new pre authentication data (PA-DATA) to KDC-REQ. This is used to pass various information to KDC, such as in case of a SHISHI_PA_TGS_REQ padatatype the AP-REQ that authenticates the user to get the ticket. (But also see `shishi_kdcreq_add_padata_tgs()` which takes an AP-REQ directly.)

Returns SHISHI_OK iff successful.

`int` **shishi_kdcreq_add_padata_tgs** (`Shishi * handle, Shishi_asn1 kdcreq, Shishi_asn1 apreq`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcreq: KDC-REQ to add PA-DATA to.

apreq: AP-REQ to add as PA-DATA.

Add TGS pre-authentication data to KDC-REQ. The data is an AP-REQ that authenticates the request. This functions simply DER encodes the AP-REQ and calls `shishi_kdcreq_add_padata()` with a SHISHI_PA_TGS_REQ padatatype.

Returns SHISHI_OK iff successful.

`Shishi_asn1` **shishi_asrep** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new AS-REP, populated with some default values.

Returns the AS-REP or NULL on failure.

`Shishi_asn1` **shishi_tgsrep** (`Shishi * handle`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new TGS-REP, populated with some default values.

Returns the TGS-REP or NULL on failure.

`int` **shishi_kdcrep_print** (`Shishi * handle, FILE * fh, Shishi_asn1 kdcrep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

kdcrep: KDC-REP to print.

Print ASCII armored DER encoding of KDC-REP to file.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_save** (`Shishi * handle, FILE * fh, Shishi_asn1 kdcrep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

kdcrep: KDC-REP to save.

Print DER encoding of KDC-REP to file.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_to_file** (`Shishi * handle, Shishi_asn1 kdcrep, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write KDC-REP to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_parse** (`Shishi * handle, FILE * fh, Shishi_asn1 * kdcrep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

kdcrep: output variable with newly allocated KDC-REP.

Read ASCII armored DER encoded KDC-REP from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * kdcrep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

kdcrep: output variable with newly allocated KDC-REP.

Read DER encoded KDC-REP from file and populate given variable.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_from_file** (`Shishi * handle, Shishi_asn1 * kdcrep, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: output variable with newly allocated KDC-REP.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read KDC-REP from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_crealm_set** (`Shishi * handle, Shishi_asn1 kdcrep, const char * crealm`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: Kdcrep variable to set realm field in.

crealm: input array with name of realm.

Set the client realm field in the KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_cname_set** (`Shishi * handle, Shishi_asn1 kdcrep, Shishi_name_type name_type, const char * cname[]`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: Kdcrep variable to set server name field in.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

Set the server name field in the KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_client_set** (`Shishi * handle, Shishi_asn1 kdcrep, const char * client`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: Kdcrep variable to set server name field in.

client: zero-terminated string with principal name on RFC 1964 form.

Set the client name field in the KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_get_enc_part_etype** (`Shishi * handle, Shishi_asn1 kdcrep, int32_t * etype`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP variable to get value from.

etype: output variable that holds the value.

Extract KDC-REP.enc-part.etype.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_get_ticket** (`Shishi * handle, Shishi_asn1 kdcrep, Shishi_asn1 * ticket`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP variable to get ticket from.

ticket: output variable to hold extracted ticket.

Extract ticket from KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_set_ticket** (`Shishi * handle, Shishi_asn1 kdcrep, Shishi_asn1 ticket`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP to add ticket field to.

ticket: input ticket to copy into KDC-REP ticket field.

Copy ticket into KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_set_enc_part** (`Shishi * handle, Shishi_asn1 kdcrep, int etype, int kvno, const char * buf, size_t buflen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP to add enc-part field to.

etype: encryption type used to encrypt enc-part.

kvno: key version number.

buf: input array with encrypted enc-part.

buflen: size of input array with encrypted enc-part.

Set the encrypted enc-part field in the KDC-REP. The encrypted data is usually created by calling `shishi_encrypt()` on the DER encoded enc-part. To save time, you may want to use `shishi_kdcrep_add_enc_part()` instead, which calculates the encrypted data and calls this function in one step.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_add_enc_part** (`Shishi * handle, Shishi_asn1 kdcrep, Shishi_key * key, int keyusage, Shishi_asn1 enckdcreppart`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP to add enc-part field to.

key: key used to encrypt enc-part.

keyusage: key usage to use, normally SHISHI_KEYUSAGE_ENCASREPPART, SHISHI_KEYUSAGE_ENCTGSREPPART_SESSION_KEY or SHISHI_KEYUSAGE_ENCTGSREPPA

enckdcreppart: EncKDCRepPart to add.

Encrypts DER encoded EncKDCRepPart using key and stores it in the KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_kdcrep_clear_padata** (`Shishi * handle, Shishi_asn1 kdcrep`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

kdcrep: KDC-REP to remove PA-DATA from.

Remove the padata field from KDC-REP.

Returns SHISHI_OK iff successful.

`int` **shishi_enckdcreppart_get_key** (`Shishi * handle, Shishi_asn1 enckdcreppart, Shishi_key ** key`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: input EncKDCRepPart variable.

key: newly allocated encryption key handle.

Extract the key to use with the ticket sent in the KDC-REP associated with the EndKDCRepPart input variable.

Returns SHISHI_OK iff succesful.

`int` **shishi_enckdcreppart_key_set** (`Shishi * handle, Shishi_asn1 enckdcreppart, Shishi_key * key`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: input EncKDCRepPart variable.

key: key handle with information to store in enckdcreppart.

Set the EncKDCRepPart.key field to key type and value of supplied key.

Returns SHISHI_OK iff succesful.

`int` **shishi_enckdcreppart_nonce_set** (`Shishi * handle, Shishi_asn1 enckdcreppart, uint32_t nonce`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: input EncKDCRepPart variable.

nonce: nonce to set in EncKDCRepPart.

Set the EncKDCRepPart.nonce field.

Returns SHISHI_OK iff succesful.

`int` **shishi_enckdcreppart_flags_set** (`Shishi * handle, Shishi_asn1 enckdcreppart, int flags`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: input EncKDCRepPart variable.

flags: flags to set in EncKDCRepPart.

Set the EncKDCRepPart.flags field.

Returns SHISHI_OK iff succesful.

`int` **shishi_enckdcreppart_populate_encticketpart** (`Shishi *` *handle*, `Shishi_asn1` *enckdcreppart*, `Shishi_asn1` *encticketpart*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: input EncKDCRepPart variable.

encticketpart: input EncTicketPart variable.

Set the flags, authtime, starttime, endtime, renew-till and caddr fields of the EncKDCRepPart to the corresponding values in the EncTicketPart.

Returns SHISHI_OK iff succesful.

`int` **shishi_enckdcreppart_srealm_set** (`Shishi *` *handle*, `Shishi_asn1` *enckdcreppart*, `const char *` *srealm*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: EncKDCRepPart variable to set realm field in.

srealm: input array with name of realm.

Set the server realm field in the EncKDCRepPart.

Returns SHISHI_OK iff successful.

`int` **shishi_enckdcreppart_sname_set** (`Shishi *` *handle*, `Shishi_asn1` *enckdcreppart*, `Shishi_name_type` *name_type*, `char *` *sname*[]) [Function]

handle: shishi handle as allocated by `shishi_init()`.

enckdcreppart: EncKDCRepPart variable to set server name field in.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

Set the server name field in the EncKDCRepPart.

Returns SHISHI_OK iff successful.

5.11 Authenticator Functions

An "Authenticator" is a ASN.1 structure that work as a proof that an entity owns a ticket. It is usually embedded in the AP-REQ structure (see Section 5.4 [AP-REQ and AP-REP Functions], page 37), and you most likely want to use an AP-REQ instead of a Authenticator in normal applications. The following illustrates the Authenticator ASN.1 structure.

```
Authenticator   ::= [APPLICATION 2] SEQUENCE  {
        authenticator-vno       [0] INTEGER (5),
        crealm                  [1] Realm,
        cname                   [2] PrincipalName,
        cksum                   [3] Checksum OPTIONAL,
        cusec                   [4] Microseconds,
        ctime                   [5] KerberosTime,
        subkey                  [6] EncryptionKey OPTIONAL,
        seq-number              [7] UInt32 OPTIONAL,
        authorization-data      [8] AuthorizationData OPTIONAL
}
```

`Shishi_asn1` **shishi_authenticator** (`Shishi *` *handle*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new Authenticator, populated with some default values. It uses the current time as returned by the system for the ctime and cusec fields.

Returns the authenticator or NULL on failure.

`Shishi_asn1` **shishi_authenticator_subkey** (`Shishi *` *handle*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

This function creates a new Authenticator, populated with some default values. It uses the current time as returned by the system for the ctime and cusec fields. It adds a random subkey.

Returns the authenticator or NULL on failure.

`int` **shishi_authenticator_print** (`Shishi *` *handle*, `FILE *` *fh*, `Shishi_asn1` *authenticator*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

authenticator: authenticator as allocated by `shishi_authenticator()`.

Print ASCII armored DER encoding of authenticator to file.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_save** (`Shishi *` *handle*, `FILE *` *fh*, `Shishi_asn1` *authenticator*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for writing.

authenticator: authenticator as allocated by `shishi_authenticator()`.

Save DER encoding of authenticator to file.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_to_file** (`Shishi *` *handle*, `Shishi_asn1` *authenticator*, `int` *filetype*, `char *` *filename*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: Authenticator to save.

filetype: input variable specifying type of file to be written, see Shishi_filetype.

filename: input variable with filename to write to.

Write Authenticator to file in specified TYPE. The file will be truncated if it exists.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_parse** (`Shishi *` *handle*, `FILE *` *fh*, `Shishi_asn1 *` *authenticator*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

authenticator: output variable with newly allocated authenticator.

Read ASCII armored DER encoded authenticator from file and populate given authenticator variable.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_read** (`Shishi * handle, FILE * fh, Shishi_asn1 * authenticator`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

fh: file handle open for reading.

authenticator: output variable with newly allocated authenticator.

Read DER encoded authenticator from file and populate given authenticator variable.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_from_file** (`Shishi * handle, Shishi_asn1 * authenticator, int filetype, char * filename`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: output variable with newly allocated Authenticator.

filetype: input variable specifying type of file to be read, see Shishi_filetype.

filename: input variable with filename to read from.

Read Authenticator from file in specified TYPE.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_set_crealm** (`Shishi * handle, Shishi_asn1 authenticator, const char * crealm`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

crealm: input array with realm.

Set realm field in authenticator to specified value.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_set_cname** (`Shishi * handle, Shishi_asn1 authenticator, Shishi_name_type name_type, const char * cname[]`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

Set principal field in authenticator to specified value.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_client_set** (`Shishi * handle, Shishi_asn1 authenticator, const char * client`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: Authenticator to set client name field in.

client: zero-terminated string with principal name on RFC 1964 form.

Set the client name field in the Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_ctime** (`Shishi * handle, Shishi_asn1 authenticator, char ** ctime`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: Authenticator as allocated by `shishi_authenticator()`.

ctime: newly allocated zero-terminated character array with client time.

Extract client time from Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_ctime_set** (`Shishi * handle, Shishi_asn1 authenticator, char * ctime`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: Authenticator as allocated by `shishi_authenticator()`.

ctime: string with generalized time value to store in Authenticator.

Store client time in Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_cusec_get** (`Shishi * handle, Shishi_asn1 authenticator, int * cusec`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: Authenticator as allocated by `shishi_authenticator()`.

cusec: output integer with client microseconds field.

Extract client microseconds field from Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_cusec_set** (`Shishi * handle, Shishi_asn1 authenticator, int cusec`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

cusec: client microseconds to set in authenticator, 0-999999.

Set the cusec field in the Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_cksum** (`Shishi * handle, Shishi_asn1 authenticator, int32_t * cksumtype, char * cksum, size_t * cksumlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

cksumtype: output checksum type.

cksum: output checksum data from authenticator.

cksumlen: on input, maximum size of output checksum data buffer, on output, actual size of output checksum data buffer.

Read checksum value from authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_set_cksum** (`Shishi * handle, Shishi_asn1 authenticator, int32_t cksumtype, char * cksum, size_t cksumlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

cksumtype: input checksum type to store in authenticator.

cksum: input checksum data to store in authenticator.

cksumlen: size of input checksum data to store in authenticator.

Store checksum value in authenticator. A checksum is usually created by calling `shishi_checksum()` on some application specific data using the key from the ticket that is being used. To save time, you may want to use `shishi_authenticator_add_cksum()` instead, which calculates the checksum and calls this function in one step.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_add_cksum** (`Shishi * handle, Shishi_asn1 authenticator, Shishi_key * key, int keyusage, char * data, size_t datalen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

key: key to to use for encryption.

keyusage: kerberos key usage value to use in encryption.

data: input array with data to calculate checksum on.

datalen: size of input array with data to calculate checksum on.

Calculate checksum for data and store it in the authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_add_cksum_type** (`Shishi * handle, Shishi_asn1 authenticator, Shishi_key * key, int keyusage, int cksumtype, char * data, size_t datalen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

key: key to to use for encryption.

keyusage: kerberos key usage value to use in encryption.

cksumtype: checksum to type to calculate checksum.

data: input array with data to calculate checksum on.

datalen: size of input array with data to calculate checksum on.

Calculate checksum for data and store it in the authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_clear_authorizationdata** (`Shishi * handle, Shishi_asn1 authenticator`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: Authenticator as allocated by `shishi_authenticator()`.

Remove the authorization-data field from Authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_add_authorizationdata** (`Shishi *` *`handle`*`, Shishi_asn1` *`authenticator`*`, int` *`adtype`*`, char *` *`addata`*`, size_t` *`addatalen`*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

adtype: input authorization data type to add.

addata: input authorization data to add.

addatalen: size of input authorization data to add.

Add authorization data to authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_authorizationdata** (`Shishi *` *`handle`*`, Shishi_asn1` *`authenticator`*`, int *` *`adtype`*`, char *` *`addata`*`, size_t *` *`addatalen`*`, int` *`nth`*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

adtype: output authorization data type.

addata: output authorization data.

addatalen: on input, maximum size of output authorization data, on output, actual size of authorization data.

nth: element number of authorization-data to extract.

th authorization data from authenticator. The first field is 1.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_remove_subkey** (`Shishi *` *`handle`*`, Shishi_asn1` *`authenticator`*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

Remove subkey from the authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_get_subkey** (`Shishi *` *`handle`*`, Shishi_asn1` *`authenticator`*`, Shishi_key **` *`subkey`*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

subkey: output newly allocated subkey from authenticator.

Read subkey value from authenticator.

Returns SHISHI_OK if successful or SHISHI_ASN1_NO_ELEMENT if subkey is not present.

`int` **shishi_authenticator_set_subkey** (`Shishi * handle, Shishi_asn1 authenticator, int32_t subkeytype, char * subkey, size_t subkeylen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

subkeytype: input subkey type to store in authenticator.

subkey: input subkey data to store in authenticator.

subkeylen: size of input subkey data to store in authenticator.

Store subkey value in authenticator. A subkey is usually created by calling `shishi_key_random()` using the default encryption type of the key from the ticket that is being used. To save time, you may want to use `shishi_authenticator_add_subkey()` instead, which calculates the subkey and calls this function in one step.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_add_random_subkey** (`Shishi * handle, Shishi_asn1 authenticator`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

Generate random subkey and store it in the authenticator.

Returns SHISHI_OK iff successful.

`int` **shishi_authenticator_add_subkey** (`Shishi * handle, Shishi_asn1 authenticator, Shishi_key * subkey`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

authenticator: authenticator as allocated by `shishi_authenticator()`.

subkey: subkey to add to authenticator.

Store subkey in the authenticator.

Returns SHISHI_OK iff successful.

5.12 Cryptographic Functions

Underneath the high-level functions described earlier, cryptographic operations are happening. If you need to access these cryptographic primitives directly, this section describes the functions available.

Most cryptographic operations need keying material, and cryptographic keys have been isolated into it's own data structure `Shishi_key`. The following illustrates it's contents, but note that you cannot access it's elements directly but must use the accessor functions described below.

```
struct Shishi_key
{
  int type;    /* RFC 1510 encryption integer type */
  char *value; /* Cryptographic key data */
  int version; /* RFC 1510 ``kvno'' */
};
```

All functions that operate on this data structure are described now.

`const char * shishi_key_principal (Shishi_key * key)` [Function]

key: structure that holds key information

Returns the principal owning the key. (Not a copy of it, so don't modify or deallocate it.)

`void shishi_key_principal_set (Shishi_key * key, const char * principal)` [Function]

key: structure that holds key information

principal: string with new principal name.

Set the principal owning the key. The string is copied into the key, so you can dispose of the variable immediately after calling this function.

`const char * shishi_key_realm (Shishi_key * key)` [Function]

key: structure that holds key information

Returns the realm for the principal owning the key. (Not a copy of it, so don't modify or deallocate it.)

`void shishi_key_realm_set (Shishi_key * key, const char * realm)` [Function]

key: structure that holds key information

realm: string with new realm name.

Set the realm for the principal owning the key. The string is copied into the key, so you can dispose of the variable immediately after calling this function.

`int shishi_key_type (Shishi_key * key)` [Function]

key: structure that holds key information

Returns the type of key as an integer as described in the standard.

`void shishi_key_type_set (Shishi_key * key, int32_t type)` [Function]

key: structure that holds key information

type: type to set in key.

Set the type of key in key structure.

`char * shishi_key_value (Shishi_key * key)` [Function]

key: structure that holds key information

Returns the key value as a pointer which is valid throughout the lifetime of the key structure.

`void shishi_key_value_set (Shishi_key * key, const char * value)` [Function]

key: structure that holds key information

value: input array with key data.

Set the key value and length in key structure.

`int shishi_key_version (Shishi_key * key)` [Function]

key: structure that holds key information

Returns the version of key ("kvno").

`void` **shishi_key_version_set** (`Shishi_key * key, int version`) [Function]

key: structure that holds key information

version: new version integer.

Set the version of key ("kvno") in key structure.

`const char *` **shishi_key_name** (`Shishi_key * key`) [Function]

key: structure that holds key information

Calls shishi_cipher_name for key type.

Return name of key.

`size_t` **shishi_key_length** (`Shishi_key * key`) [Function]

key: structure that holds key information

Calls shishi_cipher_keylen for key type.

Returns the length of the key value.

`int` **shishi_key** (`Shishi * handle, Shishi_key ** key`) [Function]

handle: Shishi library handle create by `shishi_init()`.

key: pointer to structure that will hold newly created key information

Create a new Key information structure.

Returns SHISHI_OK iff successful.

`void` **shishi_key_done** (`Shishi_key * key`) [Function]

key: pointer to structure that holds key information.

Deallocates key information structure.

`void` **shishi_key_copy** (`Shishi_key * dstkey, Shishi_key * srckey`) [Function]

dstkey: structure that holds destination key information

srckey: structure that holds source key information

Copies source key into existing allocated destination key.

`int` **shishi_key_from_value** (`Shishi * handle, int32_t type, char * value, Shishi_key ** key`) [Function]

handle: Shishi library handle create by `shishi_init()`.

type: type of key.

value: input array with key value, or NULL.

key: pointer to structure that will hold newly created key information

Create a new Key information structure, and set the key type and key value. KEY contains a newly allocated structure only if this function is successful.

Returns SHISHI_OK iff successful.

`int` **shishi_key_from_base64** (`Shishi * handle, int32_t type, char * value, Shishi_key ** key`) [Function]

handle: Shishi library handle create by `shishi_init()`.

type: type of key.

value: input string with base64 encoded key value, or NULL.

key: pointer to structure that will hold newly created key information

Create a new Key information structure, and set the key type and key value. KEY contains a newly allocated structure only if this function is successful.

Returns SHISHI_INVALID_KEY if the base64 encoded key length doesn't match the key type, and SHISHI_OK on success.

`int` **shishi_key_random** (`Shishi * handle, int32_t type, Shishi_key ** key`) [Function]

handle: Shishi library handle create by `shishi_init()`.

type: type of key.

key: pointer to structure that will hold newly created key information

Create a new Key information structure for the key type and some random data. KEY contains a newly allocated structure only if this function is successful.

Returns SHISHI_OK iff successful.

`int` **shishi_key_from_random** (`Shishi * handle, int32_t type, char * random, size_t randomlen, Shishi_key ** outkey`) [Function]

handle: Shishi library handle create by `shishi_init()`.

type: type of key.

random: random data.

randomlen: length of random data.

outkey: pointer to structure that will hold newly created key information

Create a new Key information structure, and set the key type and key value using `shishi_random_to_key()`. KEY contains a newly allocated structure only if this function is successful.

Returns SHISHI_OK iff successful.

`int` **shishi_key_from_string** (`Shishi * handle, int32_t type, const char * password, size_t passwordlen, const char * salt, size_t saltlen, const char * parameter, Shishi_key ** outkey`) [Function]

handle: Shishi library handle create by `shishi_init()`.

type: type of key.

password: input array containing password.

passwordlen: length of input array containing password.

salt: input array containing salt.

saltlen: length of input array containing salt.

parameter: input array with opaque encryption type specific information.

outkey: pointer to structure that will hold newly created key information

Create a new Key information structure, and set the key type and key value using `shishi_string_to_key()`. KEY contains a newly allocated structure only if this function is successful.

Returns SHISHI_OK iff successful.

Applications that run uninteractively may need keying material. In these cases, the keys are stored in a file, a file that is normally stored on the local host. The file should be protected from unauthorized access. The file is in ASCII format and contains keys as outputed by `shishi_key_print`. All functions that handle these keys sets are described now.

`Shishi_key * shishi_keys_for_serverrealm_in_file (Shishi * handle, const char * filename, const char * server, const char * realm)` [Function]

handle: Shishi library handle create by `shishi_init()`.

filename: file to read keys from.

server: server name to get key for.

realm: realm of server to get key for.

Returns the key for specific server and realm, read from the indicated file, or NULL if no key could be found or an error encountered.

`Shishi_key * shishi_keys_for_server_in_file (Shishi * handle, const char * filename, const char * server)` [Function]

handle: Shishi library handle create by `shishi_init()`.

filename: file to read keys from.

server: server name to get key for.

Returns the key for specific server, read from the indicated file, or NULL if no key could be found or an error encountered.

`Shishi_key * shishi_keys_for_localservicerealm_in_file (Shishi * handle, const char * filename, const char * service, const char * realm)` [Function]

handle: Shishi library handle create by `shishi_init()`.

filename: file to read keys from.

service: service to get key for.

realm: realm of server to get key for, or NULL for default realm.

Returns the key for the server "SERVICE/HOSTNAME`REALM`" (where HOSTNAME is the current system's hostname), read from the default host keys file (see `shishi_hostkeys_default_file()`), or NULL if no key could be found or an error encountered.

The previous functions require that the filename is known. For some applications, servers, it makes sense to provide a system default. These key sets used by server applications are known as "hostkeys". Here are the functions that operate on hostkeys (they are mostly wrappers around generic key sets).

`const char * shishi_hostkeys_default_file (Shishi * handle)` [Function]

handle: Shishi library handle create by `shishi_init()`.

Returns the default host key filename used in the library. (Not a copy of it, so don't modify or deallocate it.)

`void` **shishi_hostkeys_default_file_set** (`Shishi * handle, const char * hostkeysfile`) [Function]

handle: Shishi library handle create by `shishi_init()`.

hostkeysfile: string with new default hostkeys file name, or NULL to reset to default.

Set the default host key filename used in the library. The string is copied into the library, so you can dispose of the variable immediately after calling this function.

`Shishi_key *` **shishi_hostkeys_for_server** (`Shishi * handle, const char * server`) [Function]

handle: Shishi library handle create by `shishi_init()`.

server: server name to get key for

Returns the key for specific server, read from the default host keys file (see `shishi_hostkeys_default_file()`), or NULL if no key could be found or an error encountered.

`Shishi_key *` **shishi_hostkeys_for_serverrealm** (`Shishi * handle, const char * server, const char * realm`) [Function]

handle: Shishi library handle create by `shishi_init()`.

server: server name to get key for

realm: realm of server to get key for.

Returns the key for specific server and realm, read from the default host keys file (see `shishi_hostkeys_default_file()`), or NULL if no key could be found or an error encountered.

`Shishi_key *` **shishi_hostkeys_for_localservicerealm** (`Shishi * handle, const char * service, const char * realm`) [Function]

handle: Shishi library handle create by `shishi_init()`.

service: service to get key for.

realm: realm of server to get key for, or NULL for default realm.

Returns the key for the server "SERVICE/HOSTNAME`REALM`" (where HOSTNAME is the current system's hostname), read from the default host keys file (see `shishi_hostkeys_default_file()`), or NULL if no key could be found or an error encountered.

`Shishi_key *` **shishi_hostkeys_for_localservice** (`Shishi * handle, const char * service`) [Function]

handle: Shishi library handle create by `shishi_init()`.

service: service to get key for.

Returns the key for the server "SERVICE/HOSTNAME" (where HOSTNAME is the current system's hostname), read from the default host keys file (see `shishi_hostkeys_default_file()`), or NULL if no key could be found or an error encountered.

After creating the key structure, it can be used to encrypt and decrypt data, calculate checksum on data etc. All available functions are described now.

`int` **shishi_cipher_supported_p** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return 0 iff cipher is unsupported.

`const char *` **shishi_cipher_name** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return name of encryption type, e.g. `"des3-cbc-sha1-kd"`, as defined in the standards.

`int` **shishi_cipher_blocksize** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return block size for encryption type, as defined in the standards.

`int` **shishi_cipher_minpadsize** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return the minimum pad size for encryption type, as defined in the standards.

`int` **shishi_cipher_confoundersize** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Returns the size of the confounder (random data) for encryption type, as defined in the standards.

`size_t` **shishi_cipher_keylen** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return length of key used for the encryption type, as defined in the standards.

`size_t` **shishi_cipher_randomlen** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return length of random used for the encryption type, as defined in the standards.

`int` **shishi_cipher_defaultcksumtype** (`int32_t` *type*) [Function]
type: encryption type, see Shishi_etype.

Return associated checksum mechanism for the encryption type, as defined in the standards.

`int` **shishi_cipher_parse** (`const char *` *cipher*) [Function]
cipher: name of encryption type, e.g. `"des3-cbc-sha1-kd"`.

Return encryption type corresponding to a string.

`int` **shishi_checksum_supported_p** (`int32_t` *type*) [Function]
type: checksum type, see Shishi_cksumtype.

Return 0 iff checksum is unsupported.

`const char *` **shishi_checksum_name** (`int32_t` *type*) [Function]
type: checksum type, see Shishi_cksumtype.

Return name of checksum type, e.g. `"hmac-sha1-96-aes256"`, as defined in the standards.

`size_t` **shishi_checksum_cksumlen** (`int32_t` *type*) [Function]

type: checksum type, see Shishi_cksumtype.

Return length of checksum used for the checksum type, as defined in the standards.

`int` **shishi_checksum_parse** (`const char *` *checksum*) [Function]

checksum: name of checksum type, e.g. "hmac-sha1-96-aes256".

Return checksum type, see Shishi_cksumtype, corresponding to a string.

`int` **shishi_string_to_key** (`Shishi *` *handle*, `int32_t` *keytype*, `const char *` *password*, `size_t` *passwordlen*, `const char *` *salt*, `size_t` *saltlen*, `const char *` *parameter*, `Shishi_key *` *outkey*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

keytype: cryptographic encryption type, see Shishi_etype.

password: input array with password.

passwordlen: length of input array with password.

salt: input array with salt.

saltlen: length of input array with salt.

parameter: input array with opaque encryption type specific information.

outkey: allocated key handle that will contain new key.

Derive key from a string (password) and salt (commonly concatenation of realm and principal) for specified key type, and set the type and value in the given key to the computed values. The parameter value is specific for each keytype, and can be set if the parameter information is not available.

Returns *SHISHI_OK* iff successful.

`int` **shishi_random_to_key** (`Shishi *` *handle*, `int32_t` *keytype*, `char *` *random*, `size_t` *randomlen*, `Shishi_key *` *outkey*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

keytype: cryptographic encryption type, see Shishi_etype.

random: input array with random data.

randomlen: length of input array with random data.

outkey: allocated key handle that will contain new key.

Derive key from random data for specified key type, and set the type and value in the given key to the computed values.

Returns *SHISHI_OK* iff successful.

`int` **shishi_checksum** (`Shishi *` *handle*, `Shishi_key *` *key*, `int` *keyusage*, `int` *cksumtype*, `const char *` *in*, `size_t` *inlen*, `char **` *out*, `size_t *` *outlen*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to compute checksum with.

keyusage: integer specifying what this key is used for.

cksumtype: the checksum algorithm to use.

in: input array with data to integrity protect.

inlen: size of input array with data to integrity protect.

out: output array with newly allocated integrity protected data.

outlen: output variable with length of output array with checksum.

Integrity protect data using key, possibly altered by supplied key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller.

Returns *SHISHI_OK* iff successful.

`int` **shishi_verify** (`Shishi * handle, Shishi_key * key, int keyusage, int cksumtype, const char * in, size_t inlen, const char * cksum, size_t cksumlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to verify checksum with.

keyusage: integer specifying what this key is used for.

cksumtype: the checksum algorithm to use.

in: input array with data that was integrity protected.

inlen: size of input array with data that was integrity protected.

cksum: input array with alleged checksum of data.

cksumlen: size of input array with alleged checksum of data.

Verify checksum of data using key, possibly altered by supplied key usage. If key usage is 0, no key derivation is used.

Returns *SHISHI_OK* iff successful.

`int` **shishi_encrypt_ivupdate_etype** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * iv, size_t ivlen, char ** ivout, size_t * ivoutlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key is encrypting.

etype: integer specifying what cipher to use.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

ivout: output array with newly allocated updated initialization vector.

ivoutlen: size of output array with updated initialization vector.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypts data as per encryption method using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key

derivation is used. The OUT buffer must be deallocated by the caller. If IVOUT or IVOUTLEN is NULL, the updated IV is not saved anywhere.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_encrypt_iv_etype** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * iv, size_t ivlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key is encrypting.

etype: integer specifying what cipher to use.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypts data as per encryption method using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The next IV is lost, see shishi_encrypt_ivupdate_etype if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_encrypt_etype** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key is encrypting.

etype: integer specifying what cipher to use.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypts data as per encryption method using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The default IV is used, see shishi_encrypt_iv_etype if you need to alter it. The next IV is lost, see shishi_encrypt_ivupdate_etype if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_encrypt_ivupdate** (`Shishi * handle, Shishi_key * key, int keyusage, const char * iv, size_t ivlen, char ** ivout, size_t * ivoutlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key is encrypting.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

ivout: output array with newly allocated updated initialization vector.

ivoutlen: size of output array with updated initialization vector.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypts data using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. If IVOUT or IVOUTLEN is NULL, the updated IV is not saved anywhere.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_encrypt_iv** (`Shishi * handle, Shishi_key * key, int keyusage, const char * iv, size_t ivlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key is encrypting.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypts data using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The next IV is lost, see shishi_encrypt_ivupdate if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_encrypt** (`Shishi * handle, Shishi_key * key, int keyusage, char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key is encrypting.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypts data using specified key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The default IV is used, see shishi_encrypt_iv if you need to alter it. The next IV is lost, see shishi_encrypt_ivupdate if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_decrypt_ivupdate_etype** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * iv, size_t ivlen, char ** ivout, size_t * ivoutlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to decrypt with.

keyusage: integer specifying what this key is decrypting.

etype: integer specifying what cipher to use.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

ivout: output array with newly allocated updated initialization vector.

ivoutlen: size of output array with updated initialization vector.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypts data as per encryption method using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. If IVOUT or IVOUTLEN is NULL, the updated IV is not saved anywhere.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_decrypt_iv_etype** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * iv, size_t ivlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to decrypt with.

keyusage: integer specifying what this key is decrypting.

etype: integer specifying what cipher to use.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypts data as per encryption method using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The next IV is lost, see shishi_decrypt_ivupdate_etype if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_decrypt_etype** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to decrypt with.

keyusage: integer specifying what this key is decrypting.

etype: integer specifying what cipher to use.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypts data as per encryption method using specified key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The default IV is used, see shishi_decrypt_iv_etype if you need to alter it. The next IV is lost, see shishi_decrypt_ivupdate_etype if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_decrypt_ivupdate** (`Shishi * handle, Shishi_key * key, int keyusage, const char * iv, size_t ivlen, char ** ivout, size_t * ivoutlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to decrypt with.

keyusage: integer specifying what this key is decrypting.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

ivout: output array with newly allocated updated initialization vector.

ivoutlen: size of output array with updated initialization vector.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypts data using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. If IVOUT or IVOUTLEN is NULL, the updated IV is not saved anywhere.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_decrypt_iv** (`Shishi * handle, Shishi_key * key, int keyusage, const char * iv, size_t ivlen, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to decrypt with.

keyusage: integer specifying what this key is decrypting.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypts data using specified initialization vector and key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The next IV is lost, see shishi_decrypt_ivupdate_etype if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_decrypt** (`Shishi * handle, Shishi_key * key, int keyusage, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to decrypt with.

keyusage: integer specifying what this key is decrypting.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypts data specified key. The key actually used is derived using the key usage. If key usage is 0, no key derivation is used. The OUT buffer must be deallocated by the caller. The default IV is used, see shishi_decrypt_iv if you need to alter it. The next IV is lost, see shishi_decrypt_ivupdate if you need it.

Note that DECRYPT(ENCRYPT(data)) does not necessarily yield data exactly, some Kerberos encryption types add pad to make the data fit into the block size of the encryption algorithm. Furthermore, the pad is not guaranteed to look in any special way, although existing implementations often pad with the zero byte. This means that you may have to "frame" data, so it is possible to infer the original length after decryption. Compare ASN.1 DER which contains such information.

Returns *SHISHI_OK* iff successful.

`int` **shishi_n_fold** (`Shishi * handle, const char * in, size_t inlen, char * out, size_t outlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt ("M").

out: output array with decrypted data.

outlen: size of output array ("N").

Fold data into a fixed length output array, with the intent to give each input bit approximately equal weight in determining the value of each output bit.

The algorithm is from "A Better Key Schedule For DES-like Ciphers" by Uri Blumenthal and Steven M. Bellovin, <URL:http://www.research.att.com/~smb/papers/ides.pdf>, although the sample vectors provided by the paper are incorrect.

Returns *SHISHI_OK* iff successful.

`int` **shishi_dr** (`Shishi * handle, Shishi_key * key, const char * constant, size_t constantlen, char * derivedrandom, size_t derivedrandomlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: input array with cryptographic key to use.

constant: input array with the constant string.

constantlen: size of input array with the constant string.

derivedrandom: output array with derived random data.

derivedrandomlen: size of output array with derived random data.

Derive "random" data from a key and a constant thusly: DR(KEY, CONSTANT) = TRUNCATE(DERIVEDRANDOMLEN, SHISHI_ENCRYPT(KEY, CONSTANT)).

Returns *SHISHI_OK* iff successful.

`int` **shishi_dk** (`Shishi * handle, Shishi_key * key, const char * constant, size_t constantlen, Shishi_key * derivedkey`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: input cryptographic key to use.

constant: input array with the constant string.

constantlen: size of input array with the constant string.

derivedkey: pointer to derived key (allocated by caller).

DK(KEY, CONSTANT) = SHISHI_RANDOM-TO-KEY(SHISHI_DR(KEY, CONSTANT)).

Returns *SHISHI_OK* iff successful.

`int` **shishi_pbkdf2_sha1** (`Shishi * handle, const char * P, size_t Plen, const char * S, size_t Slen, unsigned int c, unsigned int dkLen, char * DK`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

P: input password, an octet string

Plen: length of password, an octet string

S: input salt, an octet string

Slen: length of salt, an octet string

c: iteration count, a positive integer

dkLen: intended length in octets of the derived key, a positive integer, at most (2^32 - 1) * hLen. The DK array must have room for this many characters.

DK: output derived key, a dkLen-octet string

Derive key using the PBKDF2 defined in PKCS5. PBKDF2 applies a pseudorandom function to derive keys. The length of the derived key is essentially unbounded. (However, the maximum effective search space for the derived key may be limited by the structure of the underlying pseudorandom function, which is this function is always SHA1.)

Returns SHISHI_OK iff successful.

An easier way to use encryption and decryption if your application repeatedly calls, e.g., `shishi_encrypt_ivupdate`, is to use the following functions. They store the key, initialization vector, etc, in a context, and the encryption and decryption operations update the IV within the context automatically.

`Shishi_crypto *` **shishi_crypto** (`Shishi * handle, Shishi_key * key, int keyusage, int32_t etype, const char * iv, size_t ivlen`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: key to encrypt with.

keyusage: integer specifying what this key will encrypt/decrypt.

etype: integer specifying what cipher to use.

iv: input array with initialization vector

ivlen: size of input array with initialization vector.

Initialize a crypto context. This store a key, keyusage, encryption type and initialization vector in a "context", and the caller can then use this context to perform encryption via `shishi_crypto_encrypt()` and decryption via `shishi_crypto_encrypt()` without supplying all those details again. The functions also takes care of propagating the IV between calls.

When the application no longer need to use the context, it should deallocate resources associated with it by calling `shishi_crypto_done()`.

Return a newly allocated crypto context.

`int` **shishi_crypto_encrypt** (`Shishi_crypto * ctx, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

ctx: crypto context as returned by `shishi_crypto()`.

in: input array with data to encrypt.

inlen: size of input array with data to encrypt.

out: output array with newly allocated encrypted data.

outlen: output variable with size of newly allocated output array.

Encrypt data, using information (e.g., key and initialization vector) from context. The IV is updated inside the context after this call.

When the application no longer need to use the context, it should deallocate resources associated with it by calling `shishi_crypto_done()`.

Returns *SHISHI_OK* iff successful.

`int` **shishi_crypto_decrypt** (`Shishi_crypto * ctx, const char * in, size_t inlen, char ** out, size_t * outlen`) [Function]

ctx: crypto context as returned by `shishi_crypto()`.

in: input array with data to decrypt.

inlen: size of input array with data to decrypt.

out: output array with newly allocated decrypted data.

outlen: output variable with size of newly allocated output array.

Decrypt data, using information (e.g., key and initialization vector) from context. The IV is updated inside the context after this call.

When the application no longer need to use the context, it should deallocate resources associated with it by calling `shishi_crypto_done()`.

Returns *SHISHI_OK* iff successful.

`void` **shishi_crypto_close** (`Shishi_crypto * ctx`) [Function]

ctx: crypto context as returned by `shishi_crypto()`.

Deallocate resources associated with the crypto context.

Also included in Shishi is an interface to the really low-level cryptographic primitives. They map directly on the underlying cryptographic library used (e.g., Nettle) and is used internally by Shishi.

`int` **shishi_randomize** (`Shishi *` *handle*, `int` *strong*, `char *` *data*, `size_t` *datalen*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

strong: 0 iff operation should not block, non-0 for very strong randomness.

data: output array to be filled with random data.

datalen: size of output array.

Store cryptographically random data of given size in the provided buffer.

Returns *SHISHI_OK* iff successful.

`int` **shishi_crc** (`Shishi *` *handle*, `const char *` *in*, `size_t` *inlen*, `char *` *out*[*4*]) [Function]

handle: shishi handle as allocated by `shishi_init()`.

in: input character array of data to checksum.

inlen: length of input character array of data to checksum.

Compute checksum of data using CRC32 modified according to RFC 1510. The `out` buffer must be deallocated by the caller.

The modifications compared to standard CRC32 is that no initial and final XOR is performed, and that the output is returned in LSB-first order.

Returns SHISHI_OK iff successful.

`int` **shishi_md4** (`Shishi *` *handle*, `const char *` *in*, `size_t` *inlen*, `char *` *out*[*MD4_DIGEST_SIZE*]) [Function]

handle: shishi handle as allocated by `shishi_init()`.

in: input character array of data to hash.

inlen: length of input character array of data to hash.

Compute hash of data using MD4. The `out` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_md5** (`Shishi *` *handle*, `const char *` *in*, `size_t` *inlen*, `char *` *out*[*MD5_DIGEST_SIZE*]) [Function]

handle: shishi handle as allocated by `shishi_init()`.

in: input character array of data to hash.

inlen: length of input character array of data to hash.

Compute hash of data using MD5. The `out` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_hmac_md5** (`Shishi *` *handle*, `const char *` *key*, `size_t` *keylen*, `const char *` *in*, `size_t` *inlen*, `char *` *outhash*[*MD5_DIGEST_SIZE*]) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: input character array with key to use.

keylen: length of input character array with key to use.

in: input character array of data to hash.

inlen: length of input character array of data to hash.

Compute keyed checksum of data using HMAC-MD5. The `outhash` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_hmac_sha1** (`Shishi * handle, const char * key, size_t keylen, const char * in, size_t inlen, char * outhash[SHA1_DIGEST_SIZE]`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

key: input character array with key to use.

keylen: length of input character array with key to use.

in: input character array of data to hash.

inlen: length of input character array of data to hash.

Compute keyed checksum of data using HMAC-SHA1. The `outhash` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_des_cbc_mac** (`Shishi * handle, const char key[DES_KEY_SIZE], const char iv[DES_BLOCK_SIZE], const char * in, size_t inlen, char * out[DES_BLOCK_SIZE]`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

in: input character array of data to hash.

inlen: length of input character array of data to hash.

Computed keyed checksum of data using DES-CBC-MAC. The `out` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_arcfour** (`Shishi * handle, int decryptp, const char * key, size_t keylen, const char iv[258], char * ivout[258], const char * in, size_t inlen, char ** out`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

decryptp: 0 to indicate encryption, non-0 to indicate decryption.

key: input character array with key to use.

keylen: length of input key array.

in: input character array of data to encrypt/decrypt.

inlen: length of input character array of data to encrypt/decrypt.

out: newly allocated character array with encrypted/decrypted data.

Encrypt or decrypt data (depending on `decryptp`) using ARCFOUR. The `out` buffer must be deallocated by the caller.

The "initialization vector" used here is the concatenation of the sbox and i and j, and is thus always of size 256 + 1 + 1. This is a slight abuse of terminology, and assumes you know what you are doing. Don't use it if you can avoid to.

Returns SHISHI_OK iff successful.

`int` **shishi_des** (`Shishi * handle, int decryptp, const char key[DES_KEY_SIZE], const char iv[DES_BLOCK_SIZE], char * ivout[DES_BLOCK_SIZE], const char * in, size_t inlen, char ** out`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

decryptp: 0 to indicate encryption, non-0 to indicate decryption.

in: input character array of data to encrypt/decrypt.

inlen: length of input character array of data to encrypt/decrypt.

out: newly allocated character array with encrypted/decrypted data.

Encrypt or decrypt data (depending on `decryptp`) using DES in CBC mode. The `out` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_3des** (`Shishi * handle, int decryptp, const char key[DES3_KEY_SIZE], const char iv[DES3_BLOCK_SIZE], char * ivout[DES3_BLOCK_SIZE], const char * in, size_t inlen, char ** out`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

decryptp: 0 to indicate encryption, non-0 to indicate decryption.

in: input character array of data to encrypt/decrypt.

inlen: length of input character array of data to encrypt/decrypt.

out: newly allocated character array with encrypted/decrypted data.

Encrypt or decrypt data (depending on `decryptp`) using 3DES in CBC mode. The `out` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

`int` **shishi_aes_cts** (`Shishi * handle, int decryptp, const char * key, size_t keylen, const char iv[AES_BLOCK_SIZE], char * ivout[AES_BLOCK_SIZE], const char * in, size_t inlen, char ** out`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

decryptp: 0 to indicate encryption, non-0 to indicate decryption.

key: input character array with key to use.

keylen: length of input character array with key to use.

in: input character array of data to encrypt/decrypt.

inlen: length of input character array of data to encrypt/decrypt.

out: newly allocated character array with encrypted/decrypted data.

Encrypt or decrypt data (depending on `decryptp`) using AES in CBC-CTS mode. The length of the key, `keylen`, decide if AES 128 or AES 256 should be used. The `out` buffer must be deallocated by the caller.

Returns SHISHI_OK iff successful.

5.13 Utility Functions

`char * shishi_realm_default_guess ( void)` [Function]

Guesses a realm based on `getdomainname()` (which really is NIS/YP domain, but if it is set it might be a good guess), or if it fails, based on `gethostname()`, or if it fails, the string "could-not-guess-default-realm". Note that the hostname is not trimmed off of the data returned by `gethostname()` to get the domain name and use that as the realm.

Returns guessed realm for host as a string that has to be deallocated with `free()` by the caller.

`const char * shishi_realm_default (Shishi * handle)` [Function]

handle: Shishi library handle create by `shishi_init()`.

Returns the default realm used in the library. (Not a copy of it, so don't modify or deallocate it.)

`void shishi_realm_default_set (Shishi * handle, const char * realm)` [Function]

handle: Shishi library handle create by `shishi_init()`.

realm: string with new default realm name, or NULL to reset to default.

Set the default realm used in the library. The string is copied into the library, so you can dispose of the variable immediately after calling this function.

`char * shishi_realm_for_server_file (Shishi * handle, char * server)` [Function]

handle: Shishi library handle create by `shishi_init()`.

server: hostname to find realm for.

Find Kerberos realm for a host using configuration file.

Returns realm for host, or NULL if not found.

`char * shishi_realm_for_server_dns (Shishi * handle, char * server)` [Function]

handle: Shishi library handle create by `shishi_init()`.

server: hostname to find realm for.

Find Kerberos realm for a host using DNS lookups, according to draft-ietf-krb-wg-krb-dns-locate-03.txt. Since DNS lookups may be spoofed, relying on the realm information may result in a redirection attack. In a single-realm scenario, this only achieves a denial of service, but with cross-realm trust it may redirect you to a compromised realm. For this reason, Shishi prints a warning, suggesting that the user should add the proper 'server-realm' configuration tokens instead.

To illustrate the DNS information used, here is an extract from a zone file for the domain ASDF.COM:

_kerberos.asdf.com. IN TXT "ASDF.COM" _kerberos.mrkserver.asdf.com. IN TXT "MARKETING.ASDF.COM" _kerberos.salesserver.asdf.com. IN TXT "SALES.ASDF.COM"

Let us suppose that in this case, a Kerberos client wishes to use a Kerberized service on the host foo.asdf.com. It would first query:

_kerberos.foo.asdf.com. IN TXT

Finding no match, it would then query:

_kerberos.asdf.com. IN TXT

Returns realm for host, or NULL if not found.

`char * shishi_realm_for_server (Shishi * handle, char * server)` [Function]

handle: Shishi library handle create by `shishi_init()`.

server: hostname to find realm for.

Find Kerberos realm for a host, using various methods. Currently this includes static configuration files (see `shishi_realm_for_server_file()`) and DNS (see `shishi_realm_for_server_dns()`).

Returns realm for host, or NULL if not found.

`char * shishi_principal_default_guess ( void)` [Function]

Guesses a principal using getpwuid(getuid)), or if it fails, the string "user".

Returns guessed default principal for user as a string that has to be deallocated with `free()` by the caller.

`const char * shishi_principal_default (Shishi * handle)` [Function]

handle: Shishi library handle create by `shishi_init()`.

Returns the default principal name used in the library. (Not a copy of it, so don't modify or deallocate it.)

`void shishi_principal_default_set (Shishi * handle, const char * principal)` [Function]

handle: Shishi library handle create by `shishi_init()`.

principal: string with new default principal name, or NULL to reset to default.

Set the default realm used in the library. The string is copied into the library, so you can dispose of the variable immediately after calling this function.

`int shishi_principal_name_set (Shishi * handle, Shishi_asn1 namenode, const char * namefield, Shishi_name_type name_type, const char * name[])` [Function]

handle: shishi handle as allocated by `shishi_init()`.

namenode: ASN.1 structure with principal in `namefield`.

namefield: name of field in namenode containing principal name.

name_type: type of principial, see Shishi_name_type, usually SHISHI_NT_UNKNOWN.

Set the given principal name field to given name.

Returns SHISHI_OK iff successful.

`int shishi_principal_set (Shishi * handle, Shishi_asn1 namenode, const char * namefield, const char * name)` [Function]

handle: shishi handle as allocated by `shishi_init()`.

namenode: ASN.1 structure with principal in `namefield`.

namefield: name of field in namenode containing principal name.

name: zero-terminated string with principal name on RFC 1964 form.

Set principal name field in ASN.1 structure to given name.

Returns SHISHI_OK iff successful.

`int` **shishi_authorization_parse** (`const char * authorization`) [Function]

authorization: name of authorization type, e.g. `"basic"`.

Return authorization type corresponding to a string.

`int` **shishi_authorized_p** (`Shishi * handle, Shishi_tkt * tkt, const char * authzname`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

tkt: input variable with ticket info.

authzname: authorization name.

Simplistic authorization of `authzname` against encrypted client principal name inside ticket. Currently this function only compare the principal name with `authzname` using `strcmp()`.

Returns 1 if authzname is authorized for services by authenticated Kerberos client principal, or 0 otherwise.

5.14 Error Handling

Most functions in 'Libshishi' are returning an error if they fail. For this reason, the application should always catch the error condition and take appropriate measures, for example by releasing the resources and passing the error up to the caller, or by displaying a descriptive message to the user and cancelling the operation.

Some error values do not indicate a system error or an error in the operation, but the result of an operation that failed properly.

5.14.1 Error Values

Errors are returned as an `int`. Except for the SHISHI_OK case, an application should always use the constants instead of their numeric value. Applications are encouraged to use the constants even for SHISHI_OK as it improves readability. Possible values are:

`SHISHI_OK`
: This value indicates success. The value of this error is guaranteed to always be `0` so you may use it in boolean constructs.

`SHISHI_OUTPUTTYPE_STDERR`
: %s%s\n

`SHISHI_OUTPUTTYPE_SYSLOG`
: %s%s

`SHISHI_OUTPUTTYPE_STDERR`
: %s%s\n

`SHISHI_OUTPUTTYPE_SYSLOG`
: %s%s

5.14.2 Error Functions

`const char *` **shishi_strerror** (`int` *err*) [Function]

err: shishi error code

Returns a pointer to a statically allocated string containing a description of the error with the error value `err`. This string can be used to output a diagnostic message to the user.

`const char *` **shishi_error** (`Shishi *` *handle*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

Extract detailed error information string. Note that the memory is managed by the Shishi library, so you must not deallocate the string.

Returns pointer to error information string, that must not be deallocate by caller.

`void` **shishi_error_clear** (`Shishi *` *handle*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

Clear the detailed error information string. See `shishi_error()` for how to access the error string, and `shishi_error_set()` and `shishi_error_printf()` for how to set the error string. This function is mostly for Shishi internal use, but if you develop an extension of Shishi, it may be useful to use the same error handling infrastructure.

`void` **shishi_error_set** (`Shishi *` *handle*, `const char *` *error*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

error: Zero terminated character array containing error description, or NULL to clear the error description string.

Set the detailed error information string to specified string. The string is copied into the Shishi internal structure, so you can deallocate the string passed to this function after the call. This function is mostly for Shishi internal use, but if you develop an extension of Shishi, it may be useful to use the same error handling infrastructure.

`void` **shishi_error_printf** (`Shishi *` *handle*, `const char *` *format*, `...`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

format: printf style format string. ...: print style arguments.

Set the detailed error information string to a printf formatted string. This function is mostly for Shishi internal use, but if you develop an extension of Shishi, it may be useful to use the same error handling infrastructure.

`int` **shishi_outputtype** (`Shishi *` *handle*) [Function]

handle: shishi handle as allocated by `shishi_init()`.

Get the current output type for logging messages.

Return output type (NULL, stderr or syslog) for informational and warning messages.

`void` **shishi_set_outputtype** (`Shishi * handle, int type`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

type: output type.

Set output type (NULL, stderr or syslog) for informational and warning messages.

`void` **shishi_info** (`Shishi * handle, const char * format, ...`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

format: printf style format string. ...: print style arguments.

Print informational message to output as defined in handle.

`void` **shishi_warn** (`Shishi * handle, const char * format, ...`) [Function]

handle: shishi handle as allocated by `shishi_init()`.

format: printf style format string. ...: print style arguments.

Print a warning to output as defined in handle.

5.15 Examples

This section will be extended to contain walk-throughs of example code that demonstrate how 'Shishi' is used to write your own applications that support Kerberos 5. The rest of the current section consists of some crude hints for the example client/server applications that is part of Shishi, taken from an email but saved here for lack of a better place to put it.

There are two programs: 'client' and 'server' in src/.

The client output an AP-REQ, waits for an AP-REP, and then simply reads data from stdin.

The server waits for an AP-REQ, parses it and prints an AP-REP, and then read data from stdin.

Both programs accept a Kerberos server name as the first command line argument. Your KDC must know this server, since the client tries to get a ticket for it (first it gets a ticket granting ticket for the default username), and you must write the key for the server into /usr/local/etc/shishi.keys on the Shishi format, e.g.:

```
-----BEGIN SHISHI KEY-----
Keytype: 16 (des3-cbc-sha1-kd)
Principal: sample/latte.josefsson.org
Realm: JOSEFSSON.ORG

8W0VrQQBpxlACPQEqN91EHxbvFFo2ltt
-----END SHISHI KEY-----
```

You must extract the proper encryption key from the KDC in some way. (This part will be easier when Shishi include a KDC, a basic one isn't far away, give me a week or to.)

The intention is that the data read, after the authentication phase, should be protected using KRB_SAFE (see RFC) but I haven't added this yet.

5.16 Generic Security Service

As an alternative to the native Shishi programming API, it is possible to program Shishi through the Generic Security Services (GSS) API. The advantage of using GSS-API in your security application, instead of the native Shishi API, is that it will be easier to port your application between different Kerberos 5 implementations, and even beyond Kerberos 5 to different security systems, that support GSS-API. In the free software world, however, almost the only widely used security system that supports GSS-API is Kerberos 5, so the last advantage is somewhat academic. But if you are porting applications using GSS-API for other Kerberos 5 implementations, or want a more mature and stable API than the native Shishi API, you may find using Shishi's GSS-API interface compelling. Note that GSS-API only offer basic services, for more advanced uses you must use the native API.

Since the GSS-API is not specific to Shishi, it is distributed independently from Shishi. Further information on the GSS project can be found at `http://josefsson.org/gss/`.

6 Acknowledgements

Shishi uses Libtasn1 by Fabio Fiorina, Libnettle by Niels Mller, Libgcrypt and Libgpg-error by Werner Koch, Libidn by Simon Josefsson, cvs2cl by Karl Fogel, and gdoc by Michael Zucchi.

Several GNU packages simplified development considerably, those packages include Autoconf, Automake, Libtool, Gnulib, Gettext, Indent, CVS, Texinfo, Help2man and Emacs.

Several people reported bugs, sent patches or suggested improvements, see the file THANKS.

This manual borrows text from the Kerberos 5 specification.

Appendix A Criticism of Kerberos

The intention with this section is to discuss various problems with Kerberos 5, so you can form a conscious decision how to deploy and use Shishi correctly in your organization.

```
* No encryption scheme with security proof.

* No standardized API, and GSS mechanism lack important functionality.

* Lack of authorization system.  (krb5_kuserok())

* Host to realm mapping relies on insecure DNS or static configuration files.

* Informational model and user database administration.
```

Appendix B Protocol Extensions

This appendix specifies the non-standard protocol elements implemented by Shishi. By nature of being non-standard, everything described here is experimental. Comments and feedback is appreciated.

B.1 STARTTLS protected KDC exchanges

Shishi is able to "upgrade" TCP communications with the KDC to use the Transport Layer Security (TLS) protocol. The TLS protocol offers integrity and privacy protected exchanges. TLS also offers authentication using username and passwords, X.509 certificates, or OpenPGP certificates. Kerberos 5 claims to offer some of these features, although it is not as rich as the TLS protocol. An inconclusive list of the motivation for using TLS is given below.

- Server authentication of the KDC to the client. In traditional Kerberos 5, KDC authentication is only proved as a side effect that the KDC knows your encryption key (i.e., your password).
- Client authentication against KDC. Kerberos 5 assume the user knows a key (usually in the form of a password). Sometimes external factors make this hard to fulfill. In some situations, users are equipped with smart cards with a RSA authentication key. In others, users have a OpenPGP client on their desktop, with a public OpenPGP key known to the server. In some situations, the policy may be that password authentication may only be done through SRP.
- Kerberos exchanges are privacy protected. Part of many Kerberos packets are transfered without privacy protection (i.e., encryption). That part contains information, such as the client principal name, the server principal name, the encryption types supported by the client, the lifetime of tickets, etc. Revealing such information is, in some threat models, considered a problem. Thus, this enables "anonymity".
- Prevents downgrade attacks affecting encryption types. The encryption type of the ticket in KDC-REQ are sent in the clear in Kerberos 5. This allows an attacker to replace the encryption type with a compromised mechanisms, e.g. 56-bit DES. Since clients in general cannot know the encryption types other servers support, it is difficult for the client to detect if there was a man-in-the-middle or if the remote server simply did not support a stronger mechanism. Clients may chose to refuse 56-bit DES altogether, but in some environments this leads to operational difficulties.
- TLS is well-proved and the protocol is studied by many parties. This is an advantage in network design, where TLS is often already assumed as part of the solution since it is used to protect HTTP, IMAP, SMTP etc. In some threat models, the designer prefer to reduce the number of protocols that can hurt the overall system security if they are compromised.

Other reasons for using TLS exists.

B.1.1 TCP/IP transport with TLS upgrade (STARTTLS)

RFC 1510bis requires Kerberos servers (KDCs) to accept TCP requests. Each request and response is prefixed by a 4 octet integer in network byte order, indicating the length of the packet. The high bit of the length was reserved for future expansion, and servers that do not understand how to interpret a set high bit must return a `KRB-ERROR` with a `KRB_ERR_FIELD_TOOLONG` and close the TCP stream.

The TCP/IP transport with TLS upgrade (STARTTLS) uses this reserved bit as follows. First we define a new extensible typed hole for Kerberos 5 messages, because we used the only reserved bit. It is thus prudent to offer future extensions on our proposal. Secondly we reserve two values in this new typed hole, and described how they are used to implement STARTTLS.

B.1.2 Extensible typed hole based on reserved high bit

When the high bit is set, the remaining 31 bits of the 4 octets are treated as an extensible typed hole, and thus form a 31 bit integer enumerating various extensions. Each of the values indicate a specific extended operation mode, two of which are used and defined here, and the rest are left for others to use. If the KDC do not understand a requested extension, it MUST return a `KRB-ERROR` with a `KRB_ERR_FIELD_TOOLONG` value and close the TCP stream, where the 4 octet length integer MUST have the high bit set, to indicate support for the extensible typed hole construct itself, and the remaining 31 bits indicate the length of the packet, as normal.

Meaning of the 31 lower bits in the 4 octet field, when the high bit is set:

```
0               RESERVED.
1               STARTTLS requested by client.
2               STARTTLS request accepted by server.
3...2147483647  AVAILABLE for registration (via bug-shishi@josefsson.org).
2147483648      RESERVED.
```

B.1.3 STARTTLS requested by client (extension mode 1)

When this is sent by the client, the client is requesting the server to start TLS negotiation on the TCP stream. The client MUST NOT start TLS negotiation immediately. The client should wait for either a KRB-ERROR (sent normally, prefixed by a 4 octet length integer) indicating it does not understand the set high bit, or 4 octet which is to interpreted as an integer in network byte order, where the high bit is set and the remaining 31 bit are interpreted as an integer specifying the "STARTTLS request accepted by server". In the first case, the client infer that the server do not understand (or wish to support) STARTTLS, and can re-try using normal TCP, if unprotected Kerberos 5 exchanges are allowed by client policy. In the latter case, it should invoke TLS negotiation on the stream. If any other data is received, the client MUST close the TCP stream.

B.1.4 STARTTLS request accepted by server (extension mode 2)

This 4 octet message should be sent by the server when it has received the previous 4 octet message. The message is an acknowledgment of the client's request to initiate STARTTLS on the channel. The server MUST then invoke a TLS negotiation.

B.1.5 Proceeding after successful TLS negotiation

If the TLS negotiation ended successfully, possibly also considering client or server policies, the exchange within the TLS protected stream is performed like normal UDP Kerberos 5 exchanges, i.e., there is no TCP 4 octet length field before each packet.

The server MAY consider the authentication performed by the TLS exchange as sufficient to issue Kerberos 5 tickets to the client, without requiring pre-authentication or the like. However, it is not an error to carry out pre-authentication as well. We are currently experimenting with this mode of operation.

B.1.6 Proceeding after failed TLS negotiation

If the TLS negotiation fails, possibly due to client or server policy (e.g., inadequate support of encryption types in TLS, or lack of client or server authentication) the entity that detect the failure should abort or re-try as appropriate, up to local policy.

B.2 Telnet encryption with AES-CCM

This appendix describe how Shishi use the Advanced Encryption Standard (AES) encryption algorithm in Counter with CBC-MAC mode (RFC 3610) with the telnet encryption option (RFC 2946).

B.2.1 Command Names and Codes

```
Encryption Type

      AES_CCM          TBD

Suboption Commands

      AES_CCM_INFO         1
      AES_CCM_INFO_OK      2
      AES_CCM_INFO_BAD     3
```

B.2.2 Command Meanings

```
IAC SB ENCRYPT IS AES_CCM AES_CCM_INFO <M> <L> <nonce> IAC SE
```

The sender of this command select desired M and L parameters, and nonce, as described in RFC 3610, and sends it to the other side of the connection. The parameters and the

nonce are sent in clear text. Only the side of the connection that is WILL ENCRYPT may send the AES_CCM_INFO command.

```
IAC SB ENCRYPT REPLY AES_CCM AES_CCM_INFO_BAD IAC SE
```

The sender of this command reject the parameters received in the AES_CCM_INFO command. Only the side of the connection that is DO ENCRYPT may send the AES_CCM_INFO_BAD command. The command MUST be sent if the nonce field length does not match the selected value for L. The command MAY be sent if the receiver do not accept the parameters for reason such as policy. No capability is provided to negotiate these parameters.

```
IAC SB ENCRYPT REPLY AES_CCM AES_CCM_INFO_OK IAC SE
```

The sender of this command accepts the parameters received in the AES_CCM_INFO command. Only the side of the connection that is DO ENCRYPT may send the AES_CCM_INFO_BAD command. The command MUST NOT be sent if the nonce field length does not match the selected value for L.

B.2.3 Implementation Rules

Once a AES_CCM_INFO_OK command has been received, the WILL ENCRYPT side of the connection should do keyid negotiation using the ENC_KEYID command. Once the keyid negotiation has successfully identified a common keyid, then START and END commands may be sent by the side of the connection that is WILL ENCRYPT. Data will be encrypted using the AES-CCM algorithm, with the negotiated nonce and parameters M and L. After each successful encryption and decryption, the nonce is treated as an integer in network byte order, and incremented by one.

If encryption (decryption) is turned off and back on again, and the same keyid is used when re-starting the encryption (decryption), the intervening clear text must not change the state of the encryption (decryption) machine. In particular, the AES-CCM nonce must not be re-set.

If a START command is sent (received) with a different keyid, the encryption (decryption) machine must be re-initialized immediately following the end of the START command with the new key and the parameters sent (received) in the last AES_CCM_INFO command.

If a new AES_CCM_INFO command is sent (received), and encryption (decryption) is enabled, the encryption (decryption) machine must be re-initialized immediately following the end of the AES_CCM_INFO command with the new nonce and parameters, and the keyid sent (received) in the last START command.

If encryption (decryption) is not enabled when a AES_CCM_INFO command is sent (received), the encryption (decryption) machine must be re- initialized after the next START command, with the keyid sent (received) in that START command, and the nonce and parameters sent (received) in this AES_CCM_INFO command.

At all times MUST each end make sure that a AES-CCM nonce is not used twice under the same encryption key. The rules above help accomplish this in an interoperable way.

B.2.4 Integration with the AUTHENTICATION telnet option

<<This section is slightly complicated. Can't we simplify this?>>

As noted in the telnet ENCRYPTION option specifications, a keyid value of zero indicates the default encryption key, as might be derived from the telnet AUTHENTICATION option. If the default encryption key negotiated as a result of the telnet AUTHENTICATION option contains less than 32 bytes (corresponding to two 128 bit keys), then the AES_CCM option MUST NOT be offered or used as a valid telnet encryption option. Furthermore, depending on policy for key lengths, the AES_CCM option MAY be disabled if the default encryption key contain less than 48 bytes (for two 192 bit keys), or less than 64 bytes (for two 256 bit keys), as well.

The available encrypt key data is divided on two halves, where the first half is used to encrypt data sent from the server (decrypt data received by the client), and the second half is used to encrypt data sent from the client (decrypt data received by the server).

Note that the above algorithm assumes that the AUTHENTICATION mechanism generate keying material suitable for AES-CCM as used in this specification. This is not necessarily true in general, but we specify this behaviour as the default since it is true for most authentication systems in popular use today. New telnet AUTHENTICATION mechanisms may specify alternative methods for determining the keys to be used for this cipher suite in their specification, if the session key negotiated by that authentication mechanism is not a DES key and and where this algorithm may not be safely used.

Kerberos 5 authentication clarification: The key used to encrypt data from the client to the server is taken from the sub-session key in the AP-REQ. The key used to decrypt data from the server to the client is taken from the sub-session key in the AP-REP. If mutual authentication is not negotiated, the key used to encrypt data from the client to the server is taken from the session key in the ticket, and the key used to decrypt data from the server to the client is taken from the sub-session key in the AP-REQ. Leaving the AP-REQ sub-key field empty MUST disable the AES_CCM option.

B.2.5 Security Considerations

The protocol must be properly and securely implemented. For example, an implementation should not be vulnerable to various implementation-specific attacks such as buffer overflows or side-channel analysis.

We wish to repeat the suggestion from RFC 2946, to investigate in a STARTTLS approach for Telnet encryption (and also authentication), when the security level provided by this specification is not adequate.

B.2.5.1 Telnet Encryption Protocol Security Considerations

The security consideration of the Telnet encryption protocol are inherited.

It should be noted that the it is up to the authentication protocol used, if any, to bind the authenticity of the peers to a specific session.

The Telnet encryption protocol does not, in general, protect against possibly malicious downgrading to any mutually acceptable, but not preferred, encryption type. This places a requirement on each peer to only accept encryption types it trust fully. In other words, the Telnet encryption protocol do not guarantee that the strongest mutually acceptable encryption type is always selected.

B.2.5.2 AES-CCM Security Considerations

The integrity and privacy claims are inherited from AES-CCM. In particular, the implementation must make sure a nonce is not used more than once together with the same key.

Furthermore, the encryption key is assumed to be random, i.e., it should not be possible to guess it with probability of success higher than guessing any uniformly selected random key. RFC 1750 gives an overview of issues and recommendations related to randomness.

B.2.6 Acknowledgments

This document is based on the various Telnet Encryption RFCs (RFC 2946, RFC 2947, RFC 2948, RFC 2952 and RFC 2953).

B.3 Kerberized rsh and rlogin

This appendix describe the KCMDV0.2 protocol used in shishi patched version of inetutils. The KCMD protocol was developped by the MIT Kerberos team for kerberized rsh an rlogin programs. Differences between rlogin an rsh will be explained, like those between v0.1 and v0.2 of the protocol for compatibility reasons. It is possible that some parts of this document are not in conformity with original KCMD protocol because there is no official specification about it. However, it seems that shishi implementation is compatible with MIT's one.

B.3.1 Establish connection

First the client should establish a TCP connection with the server. Default ports are 543 (klogin), 544 (kshell), 2105 (eklogin). eklogin is the same as klogin but with encryption. Their is no longer ekshell port because encrypted and normal connection use the same port (kshell). Kshell need a second connection for stderr. The client should send a null terminated string that represent the port of this second connection. Klogin and eklogin does not use a second connection for stderr so the client must send a null byte to the server. Contrary to classic rsh/rlogin, server must not check if the client port is in the range 0-1023.

B.3.2 Kerberos identification

When connections are established, first thing to do is to indicate kerberos authentication must be used. So the client will send a string to indicate it will used kerberos 5. It will call a length-string "strl" the couple (lenght of the string strl, null terminated string strl). Length of the string is an int32 (32bits int) in MSB order (for the network). So the client send this length-string strl :

```
KRB5_SENDAUTH_V1.0
```

After that the client must indicate which version of the protocol it will used by sending this length-string strl :

```
KCMDV0.2
```

It can be V0.1 for older versions. If indentification from client is good, server will send a null byte (0x00). Else if authentication message is wrong, server send byte 0x01, else if protocol version message is wrong server send byte 0x02.

B.3.3 Kerberos authentication

When client is indentified, kerberos authentication can begin. The client must send an AP-REQ to the server. AP-REQ authenticator must have a subkey (only for KCMDV0.2) and a checksum. Authenticator checksum is created on following string :

```
"serverport:""terminaltype""remoteusername"
```

for example :

```
543:linux/38400user
```

remoteusername corresponds to the identity of the client on remote machine.

AP-REQ is sended in der encoded format. The length (int32) of der encoded AP-REQ is sended in network format (MSB), following by the der encoded AP-REQ. If all is correct, server send a null int32 (MSB format but like it is null it is not important). KCMD protocol use mutual authentication, so server must now send and AP-REP : (in32 lenght in MSB of der encoded AP-REP)(der encoded AP-REP).

Now server and client are partially authenticated.

B.3.4 Extended authentication

Client must now send 3 different null terminated strings (without lenght) :

remote user name (user identity on remote machine)

terminal type for rlogin or command for rsh

local user name (user identity on client machine)

example for rsh :

```
"rname\0"
"cat /usr/local/etc/shishi.conf"
"lname\0"
```

Server must verify that checksum in AP-REQ authenticator is correct by computing a new hash like client has done.

Server must verify that principal (in AP-REQ) has right to log in on the remote user account. For the moment shishi only check if remote user name is equal to principal. A more complex authorization code is planned. Look at the end to know how MIT/Heimdal do to check authorization.

If all is correct server send a null byte, else an error message string (null terminated string) is sent. User read the first byte. If it is equal to zero, authentication is correct and is logged on the remote host. Else user can read the error messsage send by the server.

B.3.5 Window size

For rlogin protocol, when authentication is complete, the server can optionnaly send a message to ask for window terminal size of user. Then the user can respond but it is not an obligation.

In KCMDV0.1 server send an urgent TCP message (MSG_OOB) with one byte :

```
TIOCPKT_WINDOW = 0x80
```

In KCMDV0.2 server does not send an urgent message but write on the socket 5 bytes :

```
'\377', '\377', 'o', 'o', TIOCPKT_WINDOW
```

If encryption is enabled (eklogin) server must send this 5 bytes encrypted.

Client can answer in both protocol version with :

```
'\377', '\377', 's', 's', "struct winsize"
```

The winsize structure is filled with corresponding setting to client's terminal. If encryption is enabled this answer must be send encrypted.

B.3.6 End of authentication

The "classic" rsh/rlogin can be used now.

B.3.7 Encryption

Encryption mode is used when a connection with eklogin is established. Encryption with krsh can be used too. Before, there was a specific port for that (ekshell), but now to indicate that encryption must be used with krsh, client must add "-x " before the command when it send it between remote user name and local user name. When the client compute the checksum for AP-REQ authenticator the "- x" must not be included.

Encryption in KCMDV0.2 is not the same as in KCMDV0.1. KCMDV0.1 uses ticket session key as encryption key, and use standard Kerberos encryption functions. This protocol only supports des-cbc-crc, des-cbc-md4, des-cbc-md5 and does not use initialisation vectors.

For example on each encryption/decryption calls, the following prototype kerberos function should be used :

```
kerberos_encrypt (key, keyusage, in, out)  (or decrypt)
```

KCMDV0.2 can be used with all kerberos encryption modes (des, 3des, aes, arcfour) and use AP-REQ authenticator subkey. In opposite to KCMDV0.1 initialisation vectors are used. All encryptions/descryptions must be made using a cryptographic context (for example to use the updated iv, or sbox) :

```
kerberos_init(ctx, iv, key, keyusage)
kerberos_encrypt (ctx, in, out)
```

For both protocols, keyusage id for des-cbc-md5, des-cbc-md4, des-cbc-crc and des3-cbc-sha1 (for KCMDV0.2) :

```
keyusage = 1026
```

For other KCMDV0.2 modes keyusage is different for each encryption/decryption usage. To understand, eklogin use 1 socket. It encrypts data (output 1) to send and decrypts (input 1) received data. Kshell use 2 sockets (1 for transmit data, 1 for stderr). So there are four modes :

```
transmit  : input  1
            output 1

stderr    : input  2
            output 2
```

There is a keyusage for each modes. The keyusage must correspond on client and server side. For example in klogin client input 1 keyusage will be server output 1 keyusage.

I/O	Client	Server
intput 1	1028	1030
output 1	1030	1028
intput 2	1032	1034
output 2	1034	1032

Those keyusages must be used with AES and ARCFOUR modes.

KCMDV0.2 uses IV (initialisation vector). Like for keyusage, client IV must correspond to server IV. IV size is equal to key type, blocksize. All bytes of IV must be initialised to :

I/O	Client	Server
intput 1	0	1
output 1	1	0
intput 2	2	3
output 2	3	2

ARCFOUR mode does not use IV. However, like it is said before, a context must be used to keep the updated sbox.

Normal message with klogin and kshell are sent like that :

```
(int 32 lenght of message in MSB order)
(message)
```

In encrypted mode it is a bit different :

```
(int 32 length of unencrypted message in MSB order)
(encrypted message)
```

In KCMDV0.2 encrypted message is create like that :

```
encrypt (
(int 32 length of message in MSB order)
(message)
)
```

A check on message size can be made in second version of the protocol.

B.3.8 KCMDV0.3

This part only gives possible ways to extend KCMD protocol. Does not take that as must have in KCMD implementation.

Extensions of KCMV0.2 could be made. For example kshell supposes there are no files with name "-x *". I think the same thing can be supposed with terminal name for klogin. So client could add "-x " to terminal type it sends to server to indicate it will use encryption. Like that there will be only one port for klogin/eklogin : 543.

In encrypted mode kshell send command in clear on the network, this could be considered as insecure as user have decided to use encryption. This is not really a problem for klogin because it just sends terminal type.

In encrypted mode, klogin and kshell clients could only send "-x" as command or terminal type. After that encryption is activated, and the client could send terminal type or command encrypted. The server will send the null byte to say that all is correct, or error message in encrypted form.

B.3.9 MIT/Heimdal authorization

This part describes how MIT/Heimdal version check authorization of the user to log in on the remote machine.

Authorization check is made by looking if the file .k5login exists on the account of the remote user. If this file does not exist, remote user name must be the same as principal in AP-REQ to valid authorization. Else if this file exists, check first verify that remote user or root are the owner of .k5login. If it is not the case, the check fails. If it is good, check reads each line of that file and compare each readed name to principal. If principal is found in .k5login, authorization is valid, else user is not allowed to connect on remote host with the specified remote user name (that can be the same as principal).

So someone (for example user "user1") can remote log into "user2" account if .k5login is present in user2 home dir and this file is owned by user2 or root and user1 name is present in this file.

Appendix C Copying This Manual

C.1 GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document,

create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or

distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

C.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix D GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

D.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

D.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions

for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you

indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

D.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘`show w`’ and ‘`show c`’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Concept Index

Function and Data Index

Short Contents

Table of Contents