

# The Kerberos Authentication Protocol

Sven Groot (0024821)

June 23, 2005

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	Goals . . . . .	3
2.2	Basic design . . . . .	5
<b>3</b>	<b>Kerberos in detail</b>	<b>5</b>
3.1	The Authentication Service (AS) Exchange . . . . .	6
3.2	The Ticket Granting Service (TGS) Exchange . . . . .	8
3.3	The Client/Server (CS) Exchange . . . . .	9
3.4	Additional exchanges . . . . .	10
3.5	Tickets and Authenticators . . . . .	10
<b>4</b>	<b>Kerberos in the real world</b>	<b>13</b>
4.1	MIT . . . . .	13
4.1.1	Core implementation features . . . . .	13
4.1.2	Client usage . . . . .	14
4.2	Microsoft Windows . . . . .	14
4.2.1	Kerberos Components . . . . .	15
4.2.2	Incompatibilities . . . . .	16

## 1 Introduction

In May of 1983, a five-year research program was started at the Michigan Institute of Technology to explore how to use computers in the curriculum. This project was called *Project Athena*, and it received major funding from IBM and DEC. Project Athena led to the development of many things now taken for granted in Information Technology. This included things such as a graphical windowing system (that gave birth to the X Window system for Unix) and the idea of networking personal computers, which was unusual at that time. In 1999, it was estimated that 96% of MIT undergraduate and 94% of graduate students have Project Athena accounts, and that on a typical day, some 6000 different users use the system to access their files and various software packages.

Because the existing system of a time-shared mainframe would allow users to access their personal files regardless of the workstation they were using, as well as the constrained resources of personal computers (it was impossible to place all used applications on local hard drives, nor could every computer be equipped with for instance a printer), Project Athena would use dedicated application servers that would provide for instance file, print or e-mail services to the rest of the network.

Of course, the realisation of a networked system used by so many means some pretty strong security needs to be in place. It was needed for both clients (users) and servers to authenticate to each other. This led to the development of the Kerberos authentication protocol.

Kerberos, developed internally by MIT, was first released to the public in version 4, which gained wide community adoption. Some revisions were incorporated which resulted in the current version, Kerberos V5. Where the operational and technical description in this document is concerned, it will always refer to Kerberos V5, unless otherwise noted.

## 2 Overview

In this section, we look at the high level goals of Kerberos, as well as the basic designs and some of the more fine-grained problems that it attempts to solve.

### 2.1 Goals

Kerberos was designed to be the authentication protocol for Project Athena. Because of the environment that Project Athena provided, several goal arose that had to be met. Instead of dumb terminals attached to a time-shared machine, Project Athena used networked personal computers, and users typically had full control over these computers. As such, it could not be assumed that these client computers were secure, and it could also not be assumed that the network itself was secure. Instead, it was assumed that for instance network transmissions could be monitored by untrusted parties (it makes one wonder what the network topology of Project Athena was; in todays Internet it is logical to assume your traffic can be monitored since it must pass numerous routers before reaching its destination, but in a typical Local Area Network with star topology such monitoring would involve breaking a cable and putting a listening device on it, or hacking into a switch or hub; however, considering Athena was

developed mid-eighties, it is not inconceivable that a ring topology was used, or perhaps the Athena network was already routed. Whatever the case, the resulting design does make it very applicable to a routed network such as the Internet where transmissions are inherently insecure).

Because users would need to be able to use services provided by other systems in the network, and because they would not want anybody else to use these services in their name (this is especially true for things like access to their personal files or e-mail), they would need to be authenticated. This authentication must be achieved without relying on local workstation logon, since it is not assumed that the workstations themselves are secure. Instead, users must authenticate themselves to the network. In doing so, they must not send any information over the network that would allow any other (malicious) user, who is eaves-dropping on the network stream, to falsely authenticate themselves as other users using that information. Although I keep saying users here, the same applies if servers need to authenticate themselves to another server, or indeed if any *principle*, e.g. workstation user or network server, needs to authenticate to another principle. This authentication will typically occur by means of a password.

Of course, in a large network with many users and many servers, it is infeasible for every server to have the passwords on file for all users. Not only would this incur a large storage overhead, imagine having to change your password at all those servers. So there must be a central location where this authentication occurs.

Users will also not appreciate it if they have to authenticate themselves every time they need to access a service, regardless of whether they have authenticated to that particular service before.

And lastly, needed for instance when sending confidential data to a service, it should also be possible for a user to authenticate the service, so as to prevent the data from falling into the wrong hands.

When taking all this into consideration, it is possible to identify the following requirements goals for the design of Kerberos.

- Principals must not be able to impersonate other principals (i.e. principals must be authenticated)
- The authentication secret (i.e. password) used by the principals must not be transmitted in clear text
- It must not be necessary for all principals to be able to authenticate all other principals by themselves.
- Principals should only need to authenticate themselves once (in the case of a user, this is typically when they logon to a workstation).
- It must not be possible for principals to use (*replay*) the authentication data sent across the network by another principal to falsely authenticate as that principle.
- The system must allow for mutual authentication of principals.

## 2.2 Basic design

Kerberos endeavours to meet the requirements listed above by using a trusted third party authentication system. In this system, the network has one *authentication server*, officially called the *Key Distribution Center* (KDC), which knows all the secret keys (passwords) used by all the principals of the network. In this system, not only the users, but also the servers, have such a secret key.

The KDC distributes *tickets* to clients, together with a temporary encryption key, called the *session key*. A client then sends the ticket to a server to prove its identity and establish the session key, and an *authenticator* to prove that they are in fact the principal whom that session key was assigned to. The authenticator is used to prevent another principal from replaying the ticket, as per the requirements listed above.

In order to meet the requirement that the secret key is never sent across the network, instead various parts of messages are encrypted using the secret key. The ticket is encrypted using the secret key of the server it is for, so that only that server can read the information in the ticket. The session key is encrypted using the client's secret key when it is sent to the client. The ticket also contains a copy of the session key. By encrypting the authenticator using the session key, the following is achieved: the server knows that the client is the principal it claims to be, because otherwise it could not know the session key since that was encrypted using the client's key in the KDC's response. The server can now also send a response back to the client, and encrypt it using the session key. This proves that the server is who it says it is, since it can only know the session key if it could decrypt the ticket.

Typically, a principal will ask the KDC for credentials once, at which time it gets a special ticket for a *ticket granting service*, which can then be used to get tickets for other services without having to supply credentials again.

Kerberos tickets are valid only for a limited time, their *lifetime*. After the lifetime ends, a ticket expires and a new ticket must be acquired for future communication with the service that the ticket was for. The ticket-granting ticket can usually be renewed for a limited time after it has expired, so the users don't need to re-enter their credentials (as a user of Lijbrandt Telecom Internet, I can personally attest to how incredibly annoying having to periodically re-authenticate is; in the case of Lijbrandt, this used to be necessary every twelve hour; fortunately this requirement was recently removed).

Kerberos operates in *realms*. A separate installation comprises its own realm, and each realm has its own KDC. It's not uncommon for such realms to be named after Internet Domain names. Kerberos enables principals to authenticate across realms if the administrators of the realms have agreed on a shared secret.

## 3 Kerberos in detail

When authenticating using Kerberos a series of messages is exchanged between principals and the authentication server, as well as between the principals themselves ( the client and server). Tickets must be obtained from the authentication server and then exchanged between the client and server to perform authentication. Will not look at these messages in greater detail.

### 3.1 The Authentication Service (AS) Exchange

The first exchange of messages is an exchange between a client and the Authentication Service (KDC). The exchange is used to obtain a ticket and session key for use with a server when no credentials were previously obtained. This exchange is typically initiated by a client, usually at the start of a login session, to obtain credentials for the Ticket Granting Service, which can then be used to obtain tickets for other servers. In the case of a server whose credentials cannot or may not be negotiated using the TGS, this exchange is also used to obtain a ticket for such a server.

This exchange consists of two messages, the first being a request from the client to the KDC in which it specifies the credentials it wants and also certain options. The second is the reply from the KDC containing the ticket and the session key to use. The clients secret key (usually derived from a password) is used for encryption.

The first message, sent from the client to the KDC, is the `KRB_AS_REQ` message. It consists of the following components (note: fields enclosed in `[...]` are optional).

$$\begin{aligned} KRB\_AS\_REQ &= \\ &\{pvno, msgtype, [pdata], reqbody\} \\ \\ reqbody &= \\ &\{kdcoptions, [cname], realm, [sname], [from], till, \\ &[rtime], nonce, [etype], [addresses...]\} \end{aligned}$$

The fields in this message are:

**pvno** The protocol version number.

**msgtype** The type of the message. The type will be `KRB_AS_REQ`.

**pdata** This optional field contains pre-authentication data. It is usually left out of a `KRB_AS_REQ` message. It can however be used for certain extensions of the Kerberos protocol, or it can contain information needed by the KDC to select the key to use when generating or decrypting the response. This latter use facilitates the use of smartcards.

**kdcoptions** Specifies which options the client wants to be set on the ticket, and other options concerning the behaviour of the KDC. Among others, these include whether a ticket will be renewable, proxiabile or forwardable; whether it should be postdating or allow postdating of other tickets (for more more information about these options see section 3.5).

**cname** The name of the client.

**sname** The name of the server.

**realm** The realm of both the client and the server

**from** The time at which a postdated ticket becomes valid.

**till** The requested expiration time for the ticket.

**rtill** The requested renew-till time for the ticket.

**nonce** A unique identifier identifying the exchange. It could be a randomly generated number, or the current time.

**etype** The desired encryption algorithm to be used for the response.

**addresses** The network addresses from which the requested ticket will be valid. If a proxy is requested, this need not include the network address of the client.

The client presents this message to the server, in unencrypted form, identifying itself, and providing the server for which the ticket is meant (typically the Ticket-Granting Server) as well as the nonce that prevents replay.

The authentication server, upon receiving this message, will look up the client and server in its database, and get the secret key for both of them. It will also generate the random *session key*. It prepares the ticket using the options specified in the `KRB_AS_REQ` message and encrypts it using the specified encryption type (DES by default). Not all of the options need to be honoured exactly. For instance the expiration time can be the requested time or the current time plus the default expiration timeout, whichever comes first. It will then send a message to the client, the `KRB_AS_REP` message.

$$\begin{aligned} KRB\_AS\_REP = & \\ \{pvno, msgtype, [pdata], crealm, cname, \{ticket\}_{K_s}, \{encpart\}_{K_c}\} & \\ & \\ & encpart = \\ \{key, lastreq, nonce, keyexpiration, flags, authtime, [starttime], & \\ endtime, [renewtill], srealm, sname, [caddr]\} & \end{aligned}$$

The fields in this message are:

**pvno, msgtype and pdata** These are the same as for the `KRB_AS_REQ` message. The msgtype parameter value will be `KRB_AS_REP`.

**crealm, cname, srealm and sname** The realm and name of the client and server's principal identities.

**ticket** The new ticket, encrypted using the server's secret key. Tickets are described in detail in section 3.5.

**key** The session key as described above.

**lastreq** The time of the last request by the principal. This information is sometimes displayed to the user so they can determine if illicit use has been made by their account.

**nonce** The nonce from the request message.

**keyexpiration** Specifies information about the expiration of the client's secret key (*not* the session key). If the secret key is about to expire (for instance due to the password expiring), the client can take due action.

The remaining fields are duplicates of those in the ticket (see section 3.5) and are provided so that the client can verify if the issued ticket is useful for the client's purposes.

After the client receives the response, it will verify it, and can use the ticket to contact whatever server it wished to authenticate to, as described in section 3.3.

## 3.2 The Ticket Granting Service (TGS) Exchange

As mentioned previously, the AS Exchange is used primarily to get a ticket for a special server, the *Ticket Granting Server* (TGS), which can be used to obtain additional tickets without having to use the client's secret key anywhere in the process. This protects the secrecy of the client's secret, and makes Kerberos more transparent to the user. If this had not been done, then any time a ticket needs to be requested the client's secret key would be needed. This would mean that either the user has to enter their password every time (which is a nuisance), or the client's workstation needs to cache the secret key. Since Kerberos does not assume that the workstations are secure, this compromises the security of the secret key. Therefore, this additional ticket exchange is used.

The TGS itself needs to have the access to all the secret keys. Although it does not encrypt the reply using the client's secret key, it still needs to encrypt the ticket using the server's secret key. Therefore, the TGS is often the same physical system as the KDC.

The TGS Exchange is much like the AS Exchange. Here too, two messages are sent, one from the client to the TGS that requests a ticket, and one from the TGS. These two messages are `KRB_TGS_REQ` and `KRB_TGS_REP`. These two messages are in structure very similar to the `KRB_AS_REQ` and `KRB_AS_REP` messages. In fact, in the official standard for the Kerberos protocol (see [1]), both these messages are described to be instances of the `KRB_KDC_REQ` and `KRB_KDC_REP` messages respectively. As such, I will not list all fields of these messages here, but instead refer to section 3.1 and point out only the differences in the messages.

The first, and perhaps most important, difference is that in the TGS Exchange the encryption is not done using the client's secret key, as was mentioned before. Instead, the `encpart` section of the response is encrypted using the session key that is a part of the ticket-granting ticket, which was previously distributed in the AS Exchange. Alternatively, a sub-session key can be used that is specified in the authenticator for the message.

The `padata` field is used in this case to hold an authentication header, which consists of the ticket-granting ticket (TGT), and an authenticator (see section 3.3).

Once the TGS has received the message, it will verify that it is valid and that the TGT held within is valid and has not expired. If all checks out, the reply is sent, once again much like was done in the AS Exchange. Several pieces of data, such as the client's name and addresses are by default copied from the TGT, although these can be different for forwarded or proxy tickets (see TODO).

Once the client has received the reply, it can once more check if the ticket is usable, and then use it to contact the destination server.

### 3.3 The Client/Server (CS) Exchange

The client, through all these previous exchanges, had really only one goal in mind. It wanted to contact a server. Unfortunately, this server had decided it needed authentication, causing the client to first go through the first two exchanges at least once. Of course, once a ticket has been obtained for a server, and it has not expired yet, the AS and TGS Exchange can be skipped and the client can immediately move forward to the last exchange, the Client/Server Exchange.

In this exchange, only a single message is required, one from the client to the server. Optionally, the server can also send a reply message which will confirm the reception of the ticket and authenticate the server to the client as well.

The client, after having acquired a ticket for use with this server, generates the `KRB_AP_REQ` message, sometimes also called the *authentication header*. This message has the following format:

$$KRB\_AP\_REQ = \{pvno, msgtype, apoptions, ticket, authenticator\}$$

The fields in this message are:

**pvno and msgtype** These fields are the same as previously described. `msgtype` is `KRB_AP_REQ`.

**apoptions** This field contains options that affect the way the AP request is processed. Using this field, the client can indicate that the ticket the client is presenting to a server is encrypted using the session key from the server's TGT, instead of the server's secret key. As a second option the client can indicate that it requires mutual authentication, in which case the server is required to send a response to the request.

**ticket** This is the ticket obtained from the TGS or the KDC that authenticates the client to the server. It is encrypted using the server's secret key.

**authenticator** The authenticator for the request is generated using the current system time, the client's name and some optional values. The authenticator can include a subsession key that will be used for further communication between the client and the server. The authenticator is encrypted using the session key that is specified in the ticket.

More precise definitions of the ticket and authenticator are given in section 3.5.

Once the `KRB_AP_REQ` message is received by the server, it is checked for validity. The message is valid if the ticket contained in it can be decrypted using the server's secret key, and if the authenticator can be decrypted using the session key contained in the ticket. The name and realm of the client specified in the ticket are validated against the ones specified in the authenticator. Of course, the ticket may not be expired, and the authenticator's time may not vary too much from the server's local time. Within the allowed time skew, a server must remember all the authenticators it received to prevent replay attacks. If the message checks out, the server is ensured of the client's identity.

Usually, the client will send its initial request (the request it needed to authenticate for) together with the `KRB_AP_REQ` message. If this is the case, the

server doesn't need to respond to this message, but can after the client has been authenticated immediately carry out the action corresponding to the request. The client can however have opted to use mutual authentication, by setting the appropriate option in the message. If this is the case, the server needs to respond to, which it will do with a `KRB_AP_REP` message. This message has the following format:

$$KRB\_AP\_REP = \{pvno, msgtype, \{ctime, cusec, [subkey], seqnumber\}_{K_{c,s}}\}$$

The fields in this message are:

**pvno and msgtype** These fields are the same as previously described. msg-type is `KRB_AP_REP`.

**ctime** This is the current time of the client. This value must be the client's time as provided by the authenticator.

**cusec** This is the microsecond part of the client time. This value is also extracted from the authenticator.

**subkey** An optional sub-session key that is used to encrypt the data of this specific application session. If this key is absent, the sub-session key in the authenticator is used, otherwise the session key from the ticket.

The response proves the identity of the server to the client. This is achieved because the `ctime`, `cusec` and `subkey` messages are encrypted using the session key specified in the ticket. This proves the server has the session key, and that is only possible if it could decrypt the ticket, which was encrypted using the server's secret key.

### 3.4 Additional exchanges

Kerberos defines a number of additional message exchanges that can be used for purposes other than authentication. I will mention these here briefly.

The first of these is the `KRB_SAFE` exchange. The `KRB_SAFE` message is used by clients to prevent a message from being tampered with. The data that is sent in the message is signed using a checksum, usually computed using a hash algorithm, such as MD5 or SHA1. The checksum is keyed using an encryption key, usually the last sub-session key or the session key.

The second is `KRB_PRIV`, which is used to ensure confidentiality of a message. The message data is encrypted using an encryption key, usually the last sub-session key or the session key. The message also includes a timestamp or sequence number to prevent replay attacks.

The last of these exchanges is `KRB_CRED`, which is used to send credentials from one host to another. The ticket, along with encrypted data containing the session keys and other required data is sent in the `KRB_CRED` message.

### 3.5 Tickets and Authenticators

It is now time to look in detail at the key players of the Kerberos protocol, the tickets and the authenticators. First we focus on tickets. A ticket helps a client authenticate to a service. Tickets have the following format:

$$\begin{aligned}
\textit{Ticket} &= \\
&\{ \textit{tktvno}, \textit{realm}, \textit{sname}, \{ \textit{encpart} \}_{K_s} \} \\
&\textit{encpart} = \\
&\{ \textit{flags}, \textit{key}, \textit{crealm}, \textit{cname}, \textit{transited}, \textit{authtime}, [\textit{starttime}], \\
&\quad \textit{endtime}, [\textit{renewtill}], [\textit{caddr}], [\textit{authorizationdata}] \}
\end{aligned}$$

The fields in the ticket are:

- tktvno** The version number for the ticket format. For Kerberos V5, this field has value 5.
- realm** The realm that issued the ticket. This is also the realm for the server the ticket is for.
- sname** The name of the server the ticket is for.
- flags** This field specifies various options that are used or requested when the ticket was issued. The following flags are defined:
- FORWARDABLE** This flag, which is normally only meant for the TGS, means that it is allowed for the TGS to issue another Ticket-Granting Ticket based on the presented ticket, but with a different network address.
  - FORWARDED** This flag indicates that the ticket has been forwarded or was granted based on a ticket-granting ticket that was forwarded.
  - PROXIABLE** Again normally meant only for the ticket-granting service, the proxiable flag has identical meaning to the forwardable flag, except that only non-TGT tickets may be issued with a different network address.
  - PROXY** Indicates that the ticket is a proxy.
  - MAY-POSTDATE** This flag is also normally only for the TGS, and means that a postdated ticket may be issued based on this TGT.
  - POSTDATED** Indicates that the ticket is postdated.
  - INVALID** The ticket is invalid, and must be validated by the KDC before use. Application servers should reject tickets with this flag set.
  - RENEWABLE** The ticket can be renewed by the TGS until the renewtill date is reached.
  - INITIAL** The ticket was issued using the AS exchange, and not using the TGS exchange.
  - PRE-AUTHENT** Indicates that the client was authenticated by the KDC before a ticket was issued.
  - HW-AUTHENT** Indicates that the initial authentication of the client involved hardware that only the specified client could have, like a smartcard.
- key** The session key used to encrypt the authenticator and the response from the server (if any) during the CS exchange.

- crealm** The realm of the client.
- cname** The name of the client that this ticket was issued to.
- transited** This indicates which realms were involved in authenticating the client in cross-realm operation.
- authtime** The initial authentication time. It's the time the initial ticket on which this ticket is based was issued. This field is also included in the KRB\_AS\_REP message.
- starttime** If present, it indicates the time that the ticket becomes valid.
- endtime** The time at which the ticket expires. Servers will refuse tickets with an endtime in the past. Note that servers may also refuse a ticket that has not yet expired, because they place their own limits on the age of a ticket.
- renewtill** When the RENEWABLE flag is specified, this field indicates the maximum endtime of a renewed ticket.
- caddr** Indicates the addresses from which the ticket can be used. If none are specified, the ticket can be used from any address. The presence of these addresses makes it harder for an attacker to use stolen credentials. This is not failsafe however. The client's physical workstation could be compromised, or attackers could spoof the network address of the client.
- authorizationdata** This field is used to pass authorization data from the principal on whose behalf a ticket was issued to the application service. This field is not used by Kerberos itself, it can be used for various purposes. For instance a proxy that is valid only for a specific purpose can be issued.

Renewable tickets are used because clients may want to hold tickets that are valid for a long time, without unnecessarily increasing the risk of the credentials being stolen. If instead it would use shortlived tickets, the client's secret key would need to be available. If this were cached on the client, it would impose an even greater risk, as one of the assumptions of Kerberos is that clients are not secure. In this case, renewable tickets can be used. A renewable ticket has two expiration times: the real expiration time of the individual ticket, and the latest possible expiration time for renewed tickets. An application will, before the ticket expires, present a renewable ticket to the KDC, specifying the RENEW option in the request, which will issue a new ticket which is identical to the one presented except for the session key and expiration time. The KDC can keep a list of tickets reported stolen to prevent renewal of such stolen tickets.

Proxiable tickets can be used when it is necessary for a client to allow a service to perform an operation on its behalf. The service will "pretend" to be the client, but must be able to do so only for a specific purpose. A client can use a proxiable ticket to request a proxy ticket with the network address(es) of the principal that will impersonate the client. This proxy ticket is then distributed to the service, which uses it to perform the operation.

When a service needs complete use of a client's identity, a forwardable ticket is used. Similar to the proxy case, the client requests a forwarded ticket and passes it to the service. This is used for instance when logging in to a remote

system where the remote login must function as if it were local. Because of this flag, authentication can be forwarded without the user entering its password again.

What remains then is the Authenticator. The authenticator is a vital part, because without it the server cannot verify that the client the ticket is received from is the actual client mentioned in the ticket. The authenticator has the following format:

$$\text{Authenticator} = \{ \text{authenticatorvno}, \text{crealm}, \text{cname}, [\text{cksum}], \text{cusec}, \text{ctime}, [\text{subkey}], [\text{seqnumber}], [\text{authorizationdata}] \}_{K_{c,s}}$$

The fields in the Authenticator are:

**authenticatorvno** The version number of the authenticator format. This is 5 for Kerberos V5.

**crealm and cname** These fields are the same as for the ticket.

**cksum** A checksum of the application data included with a KRB\_AP\_REQ message.

**cusec** The microsecond part of the client's timestamp.

**ctime** The client's time.

**subkey** The sub-session key to be used to protect the specific application session. If omitted, the session key for the ticket is used.

**seqnumber** The initial sequence number for KRB\_PRIV or KRB\_SAFE messages.

**authorizationdata** This field is the same as for the ticket.

The Authenticator is encrypted using the session key, which the client can only know if it has the appropriate secret key for the client whom the ticket was issued to.

## 4 Kerberos in the real world

### 4.1 MIT

#### 4.1.1 Core implementation features

The base implementation of Kerberos is provided by MIT itself. This implementation, written in C, is the one used by Project Athena itself, but also in other environments.

The MIT implementation provides several runtime libraries which can be linked to by applications. An application can selectively choose which libraries to use, but the core library is probably used by all applications.

The core library contains the base functions that handle assembling and disassembling the network messages. This includes the ASN.1 encoding and decoding that is used to convert to the network encoding. It also includes routines to verify the correct sequence of messages.

The encryption routines are called by the core library. Since Kerberos V5 is capable of dealing with different types of encryption systems, these functions are called through a table that can be set up by a specific encryption system implementation. The MIT implementation provides a number of default implementations, and initializes the table with those values.

The checksum routines are handled in much the same way, they are called through a table where custom routines can be plugged in. The table indicates the properties of the different checksum algorithms, such as whether it's keyed and collision proof. MIT provides four checksum algorithms: CRC-32, DES, MD4 and MD5. Certain weaknesses were recently identified in the MD5 algorithm, and it's not unlikely to be replaced or deprecated for future versions of the Kerberos protocol.

The routines for storing credentials in a cache can also be extended. The default credential cache is specified through an environment variable, allowing client's to switch caches as needed. Two cache implementations are provided, one based on Unix file descriptors, and one based on C stdio.

Servers store their secret keys in a key table. MIT provides a C stdio based implementation for the key table.

A separate database library is included to handle all access to the KDC's principal database, both by the KDC itself and administrative programs. MIT provided an implementation that uses the UNIX dbm database. This database is not locked, so separate locking code is included.

The MIT implementation is targeted mainly at UNIX, but it is designed so that specific operating system features are only accessed in contained libraries, so that porting it to another operating system should at least be possible.

#### 4.1.2 Client usage

When all goes well, a client will not notice it is using Kerberos. The process of obtaining a TGT is usually automated with logging in to the client and client applications such as e-mail clients or printer or file sharing client will automate the retrieval of the appropriate tickets. The TGT can be acquired manually using *kinit* if needed. A server using Kerberos runs *klogind* which will allow a user to login remotely to that machine when presented with a valid ticket.

The client login process will convert from a password to the client's secret key, so that the client needn't remember the actual binary secret key. This conversion can be seeded with an additional string, often the client's realm name, to make sure that the secret key differs in different realms even when the client uses the same password.

## 4.2 Microsoft Windows

Historically, the authentication protocol on Microsoft Windows networks has been based on the LAN Manager, which was first introduced for OS/2 in 1987, and was later adapted for use with Windows for Workgroups. When Windows NT came around, the authentication protocol used was the NT LAN Manager Challenge/Response Protocol, or simply known as NTLM. NTLM is similar in operation to the HTTP Digest authentication scheme defined in [6], with one crucial difference: instead of using the password itself when creating the

response, a hash of the password, known as the NTLM Hash, is used. This eliminates the need to store passwords on the server using reversible encryption.

Unfortunately, the NTLM protocol was far from secure. The hashes it used were weak (although not as weak as the easily breakable LAN Manager hashes), and the protocol was vulnerable to all kinds of forwarding and replay attacks. Although some of these concerns were alleviated with the NTLMv2 protocol, which was introduced with Windows NT4 SP4, it remains a far from ideal solution. On top of that, NTLM is poorly documented; what little can be found on the Internet on how it works is usually based on reverse engineering the protocol, not official documentation from Microsoft itself.

Recognizing the shortcomings of NTLM, Microsoft decided to switch to Kerberos V5 with Windows 2000. This change did not stand alone, as Microsoft switched a lot of the system over to open standards. The proprietary flat domain model of NT4 was abandoned for the LDAP based Active Directory, NetBIOS was (partly, some areas of the system still need it) abandoned for DNS name resolution, and so forth.

#### 4.2.1 Kerberos Components

As is common in Kerberos implementations, Windows combines the Authentication Service and the Ticket Granting Service in the KDC. The KDC runs as a domain service on every Domain Controller in a Windows 2000 or Windows Server 2003 domain. This service is, not surprisingly, called the Kerberos Key Distribution Center service, and runs in the process context of the Local Security Authority (LSA) sub system (*lsass.exe*). Any Domain Controller can accept and process authentication requests directed for the domain's KDC. The KDC uses a security principal whose secret key is used to encrypt the Ticket Granting Tickets. This principal is called *krbtgt*, a special account that is created for every new domain, that cannot be deleted, and whose password is changed on a regular basis. The name of the Windows domain is the name of the Kerberos realm used.

The KDC uses the domain's Active Directory as its account database. Active Directory is a database based on Microsoft Jet which is kept on every Domain Controller, so it is available to every instance of the KDC. Every replica of the Active Directory is writable, and changes are propagated between the different replicas using a proprietary multi-master replication protocol over a secure channel. Actual storage of the account data in the Active Directory is handled by the Directory System Agent (DSA), a protected process that runs in the LSA. Access to account data is protected by Access Control Lists, just like every other protected object in a Windows system, and clients can never access the data directly, only through the Active Directory Service Interface (ADSI). A principal's secret key is protected so that it is not readable by anyone except the principal itself, not even to Administrators. Only trusted processes running in the LSA can modify the secret key. Further encryption is employed to prevent an attacker with access to the system's backups from doing an off-line attack.

Kerberos services are provided to systems running Windows as a Security Support Provider, or SSP. Windows also includes the SSP for NTLM, and both are loaded by the LSA at system boot. Whichever one ends up being used depends on the capabilities of the computer on the other side. The Kerberos SSP is preferred; the NTLM SSP is only used when Kerberos is not available.

After interactive logon is complete, the Kerberos SSP can be accessed by other processes on the system running in the user's security context through the use of a Security Support Provider Interface (SSPI), which is a protocol-agnostic part of the Win32 Application Programming Interface (API) that allows applications to access an SSP without knowing the details of the protocol being used.

Credentials received by the KDC are stored in an area of volatile memory that is protected by the LSA. This credential cache is never paged to disk, and is cleared when the system shuts down. Besides the credentials, a hashed version of the users password is also kept, used to request a new TGT if the TGT expires during the logon session.

#### **4.2.2 Incompatibilities**

The Windows implementation of Kerberos has some incompatibilities with the Kerberos protocol as defined in [1]. The biggest perhaps is that Windows computers are Active Directory clients, and as such expect certain authorization data belonging to Active Directory, as well as a GlobalPolicy to be present in the Kerberos messages they receive. If this data is missing, Active Directory clients cannot function, which means it is impossible for a Windows client computer to participate in a Kerberos realm that is not an Active Directory domain. Non-Windows clients can however participate in a Windows-based Kerberos realm using the Windows 2000 or Windows Server 2003 Domain Controller as the KDC.

Another extension is the ability to allow an administrator to set the user's password. This was a feature supported by NT4 domains, but not then supported by Kerberos. Some other mechanism could have been defined for setting passwords, but this would have required registering additional ports or allowing a Remote Procedure Call, etc, and the administrative tool would still have needed to negotiate a ticket with Kerberos to access this service in order to send the new password securely.

And finally, Windows uses some extensions to allow public key cryptography to be used in authentication Kerberos client's. This allows the use of smartcards for logins. This work is based on the IETF Internet Draft defined in [7].

## References

- [1] J. Kohl, C. Neuman, *RFC1510: The Kerberos Network Authentication Service (V5)*, Internet Engineering Task Force, September 1993
- [2] J. Kojl, C. Neuman, T. Ts'o, *The Evolution of the Kerberos Authentication Service*, 1991
- [3] B. Bryant, *Designing an Authentication System: a Dialogue in Four Scenes*, 1988
- [4] *Kerberos V5 Administrators Guide*, Michigan Institute of Technology, 2002
- [5] *Windows 2000 Kerberos Authentication*, Microsoft Corporation, 1999
- [6] A. Niemi, J. Arkko, V. Torvinen *RFC3310: Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA)*, Internet Engineering Task Force, September 2002.
- [7] B. Tung, C. Neuman, J. Wray, A. Medvinsky, M. Hur, and J. Trostle, *Public Key Cryptography for Initial Authentication in Kerberos*, Internet Engineering Task Force draft, May 2005