KERBEROS PROTOCOL TUTORIAL

This tutorial was written by Fulvio Ricciardi and is reprinted here with his permission. Mr. Ricciardi works at the National Institute of Nuclear Physics in Lecce, Italy. He is also the author of the Linux project zeroshell.net, where he originally published this tutorial. Thank you Mr. Ricciardi!

Document version: 1.0.3 (11/27/2007)
Author: Fulvio Ricciardi (Fulvio.Ricciardi@le.infn.it)
INFN - the National Institute of Nuclear Physics
Computing and Network Services - LECCE, Italy

Kerberos is an authentication protocol for trusted hosts on untrusted networks.

1 Introduction
2 Aims
3 Definitions of components and terms
3.1 Realm
3.2 Principal
3.3 Ticket
3.4 Encryption
3.4.1 Encryption type
3.4.2 Encryption key
3.4.3 Salt
3.4.4 Key Version Number (kvno)
3.5 Key Distribution Center (KDC)
3.5.1 Database
3.5.2 Authentication Server (AS)
3.5.3 Ticket Granting Server (TGS)
3.6 Session Key
3.7 Authenticator
3.8 Replay Cache
3.9 Credential Cache
4 Kerberos Operation
4.1 Authentication Server Request (AS_REQ)
4.2 Authentication Server Reply (AS_REP)
4.3 Ticket Granting Server Request (TGS_REQ)
4.4 Ticket Granting Server Replay (TGS_REP)
4.5 Application Server Request (AP_REQ)
4.6 Pre-Authentication [4.6 Application Server Replay (AP_REP) missing]
5 Tickets in-depth
5.1 Initial tickets
5.2 Renewable tickets
5.3 Forwardable tickets
6 Cross Authentication
6.1 Direct trust relationships
6.2 Transitive trust relationships
6.3 Hierarchical trust relationship

1 Introduction

The Kerberos protocol is designed to provide reliable authentication over open and insecure networks where communications between the hosts belonging to it may be intercepted. However, one should be aware that Kerberos does not provide any guarantees if the computers being used are vulnerable: the authentication servers, application servers (imap, pop, smtp, telnet, ftp, ssh , AFS, lpr, ...) and clients must be kept constantly updated so that the authenticity of the requesting users and service providers can be guaranteed.

The above points justify the sentence: **"Kerberos is an authentication protocol for trusted hosts on untrusted networks"**. By way of example, and to reiterate the concept: Kerberos' strategies are useless if someone who obtains privileged access to a server, can copy the file containing the secret key. Indeed, the intruder will put this key on another machine, and will only have to obtain a simple spoof DNS or IP address for that server to appear to clients as the authentic server.

2 Aims

Before describing the elements that make up the Kerberos authentication system and looking at its operation, some of the aims the protocol wishes to achieve are listed below:

- The user's password must never travel over the network;
- The user's password must never be stored in any form on the client machine: it must be immediately discarded after being used;
- The user's password should never be stored in an unencrypted form even in the authentication server database;
- The user is asked to enter a password only once per work session. Therefore users can transparently access all the services they are authorized for without having to re-enter the password during this session. This characteristic is known as **Single Sign-On**;
- Authentication information management is centralized and resides on the authentication server. The application servers must not contain the authentication information for their users. This is essential for obtaining the following results:
  1. The administrator can disable the account of some user by acting on the single database of authentication information and not on the several application servers providing the various services;
  2. When a user changes its password, it is changed for all services at the same time;
  3. There is no redundancy of authentication information which would otherwise have to be safeguarded in various places;
- Not only do the users have to demonstrate that they are who they say, but, when requested, the application servers must prove their authenticity to the client as well. This characteristic is known as **Mutual authentication**;
- Following the completion of authentication and authorization, the client and server must be able to establish an encrypted connection, if required. For this purpose, Kerberos provides support for the generation and exchange of an encrypting key to be used to encrypt data.

3 Definition of the components and terms

This section provides the definition of the objects and terms knowledge of which is essential for the correct understanding of the Kerberos protocol. Since many definitions are based on others, wherever possible I have tried to put them in order, so that the meaning of a term is not used before defining it. However, it may be necessary to read this section twice to fully understand the meaning of the terms.

3.1 Realm

The term realm indicates an authentication administrative domain. Its intention is to establish the boundaries within which an authentication server has the authority to authenticate a user, host or service. This does not mean that the authentication between a user and a service that they need belong to the same realm. If the two objects are part of different realms, and there is a trust relationship between them, then the authentication can take place. This characteristic known as Cross-Authentication will be described below.

Typically, a realm corresponds to a DNS domain and only those objects belonging to the authentication server of that realm.

The name of the realm is a character string. It is recommended that it be the same as the name of the DNS domain of the organization to which the machines belong. Although it is possible to use lowercase characters, it is customary to use only uppercase characters. For example, if the domain name of the organization is example.com, then the Kerberos realm is EXAMPLE.COM. Although it is possible to use the same name as a DNS domain, it is recommended that a distinction be made between the two spaces by using different cases. In this way, it is possible to establish trust relationships between two realms, one called EXAMPLE.COM and the other called example.com. If the organization is also a subdomain of a larger domain example.com, it is appropriate that the related Kerberos realm is EXAMPLE.COM.

3.2 Principal

A principal is the name used to refer to the entries in the authentication server database. A principal is associated with each user, host or service of a given realm. A principal in Kerberos 5 is of the following type:

component1/component2/...component[n]@REALM

However, in practice a maximum of two components are used. For an entry referring to a user the principal is the following type:

Name[/Instance]@REALM

The instance is optional and is normally used to better qualify the type of user. For example administrator users normally have the admin instance. The following are examples of principals referred to users:

pippo@EXAMPLE.COM admin/admin@EXAMPLE.COM pluto/admin@EXAMPLE.COM ...

If, instead, the entries refer to services, the principals assume the following form:

Service/Hostname@REALM

The first component is the name of the service, for example imap, AFS, ftp. Often it is the word host which is used to indicate generic access to the machine (telnet, rsh, ssh). The second component is the complete hostname (FQDN) of the machine providing the requested service. It is important that this component exactly matches (in lower case letters) the DNS reverse resolution of the application server's IP address. The following are valid examples of principals referred to services:

imap/mail.example.com@EXAMPLE.COM
host/foo.example.com@EXAMPLE.COM
afs/cellname.com@EXAMPLE.COM

It should be noted that the last case is an exception because the second component is not a hostname but the name of the AFS cell that the principal authenticates. In an AFS network, the principal AFS/cellname@REALM will be associated to a key used to encrypt the TGS of cellname. Ticket-granting tickets will be described later.

In Kerberos 4 there can never be more than two components, and they are separated by the character "." instead of /. While for hostnames in the principals of type host/hostname.domain.com@REALM there is no difference between Kerberos 4 and Kerberos 5, for those principals which have a slash in the name, the Kerberos 4 version will be different. For example, the principal afs/cellname@REALM will be translated into afs.cellname@REALM in Kerberos 4.

pippo@EXAMPLE.COM pluto.admin@EXAMPLE.COM afs/cellname@EXAMPLE.COM

3.3 Ticket

A ticket is something a client presents to an application server to demonstrate the authenticity of its identity. Tickets are issued by the authentication server and are encrypted using the server key of the application server and the service providing the ticket, and can be read only by the application server. The ticket allows the server to verify the identity of the person to whom the ticket was issued. The ticket contains:

- the name of the server for which it is intended;
- the name of the client to which it was issued;
- the IP address of the client to which it was issued (optional, in Kerberos 5 this field is optional and may also be unknown or set to zero, or indicate a different address);
- the date and time issued (timestamp);
- the ticket lifetime;
- the session key that the client and server will share during the session.

Each ticket has an application lifetime (typically 10 hours). This is essential since the authentication server no longer has any control over an already issued ticket. Even if it prevents users from using the ticket, if they already possess it. This is the reason for limiting the lifetime of the tickets in order to limit any abuse over time.

Tickets contain a set of time information and flags which characterize their behavior, but we will go into this later. We will discuss tickets and their operation later.

3.4 Encryption

As you can see Kerberos often needs to encrypt and decrypt the messages (tickets and authenticators) passing between the various participants in the authentication. It is important to note that Kerberos uses only symmetrical key encryption (in other words the same key is used to encrypt and decrypt). Certain projects (e.g. pkinit) are active for introducing a public key system in order to obtain the initial user authentication through the presentation of a private key corresponding to a certified public key, but since there is no standard we'll skip this discussion for now.

3.4.1 Encryption type

Kerberos 4 implements a single type of encryption which is DES at 56 bits. The weakness of this encryption plus other protocol vulnerabilities have made Kerberos 4 obsolete. Version 5 of Kerberos, however, does not predetermine the number or type of encryption methodologies supported. It is the task of each specific implementation to support and best negotiate the various types of encryption. However, this flexibility and expandability of the protocol has accentuated interoperability problems between the various implementations of Kerberos 5. In order for clients and application and authentication servers using different implementations to interoperate, they must have at least one encryption type in common. The difficulty related to the interoperability between implementations of Kerberos 5 and the one present in the Active Directory of Windows is a classic example of this. Indeed, Windows Active Directory supports a limited number of encryptions and only had DES at 56 bits in common with Unix. This required keeping the latter enabled, despite the risks being well known, if interoperability had to be guaranteed. The problem was subsequently solved with version 1.3 of MIT Kerberos 5. This version introduced RC4-HMAC support, which is also present in Windows and is more secure than DES. Among the supported encryptions (but not by Windows) the triple DES (3DES) and newer AES128 and AES256 are worth mentioning.

3.4.2 Encryption key

As stated above, one of the aims of the Kerberos protocol is to prevent the user's password from being stored in its unencrypted form, even in the authentication server database. Considering that each encryption algorithm uses its own key length, it is clear that it is not to be feasible to use a different password of a fixed size for each encryption method expected, the encryption key will be generated by combining the passwords. For these reasons the string2key function has been introduced, which transforms an unencryption key suitable for the type of encryption to be used. This function is called each time a user changes password or enters it for authentication. The string2key is called a hash function, meaning it generates a fixed length output starting from the password which generated it (unless by brute force). Famous hashing algorithms are MD5 and CRC32.

3.4.3 Salt

In Kerberos 5, unlike version 4, the concept of password salt has been introduced. This is a string to be concatenated to the unencrypted password before applying the string2key function to obtain the key. Kerberos 5 uses the same salt

$$K_{User} = string2key\ (\ P_{User} + "principal@REALM-CORP"\ )$$

K_{User} is the encryption key of the user pippo and P_{User} is the unencrypted password of the user.

This type of salt has the following advantages:

- two principals belonging to the same realm and having the same unencrypted password still have different keys. For example, imagine an administrator having a personal for owner key work (suppose: ZOMPELCOM) and one for administrator (suppose: admin@ZOMPELCOM). It is very likely that this user does not use the same password for both accounts, but if it was the case, the security of the well-known keys and the related keys are different.
- the user has an account in the domain (suppose: pippo@ZOMPELCOM) and also in the realm (suppose: pippo@ZOMPELCOM). Even if the password is the same for both realms, thanks to the presence of the salt, a possible compromise of an account in one realm, will not automatically cause the other to be compromised.

A null salt can be configured for compatibility with Kerberos 4. Vice versa, for compatibility with AFS, it is possible to configure a salt which is not the complete name of the principal, but simply the name of the cell.

Having discussed the concepts of encryption type, string2key and salt, it is possible to check the accuracy of the following observation: in order that there is interoperability between the various Kerberos implementations, it is not sufficient to negotiate a common type of encryption, but the same types of string2key and salt need to be used as well.

It is also important to note that, in explaining the concepts of string2key and salt, reference was only made to encryption keys made from the user's password. In the case of users, the server in fact has a new idea, even if it is even a secret to the secret password (who would enter it?), but a key which, once [illegible] the user's password, is used to encrypt the messages addressed to the principal, protecting them from the server.

3.4.4 Key Version Number (kvno)

When a user changes a password or an administrator updates the secret key for an application server, this change is logged by advancing a counter. The current value of this counter identifies the key version, is known as the Key Version Number or more briefly kvno.

3.6 Key Distribution Center (KDC)

We have spoken generically about the authentication server. Since it is the fundamental object involved in the authentication of users and services, we will now take a more in depth look at it in the section dedicated to protocol operation.

The authentication server is a Kerberos environment, based on the ticket distribution function for access to the services, it is called Key Distribution Center or more briefly KDC. Since it represents the trusted third party, it is very delicate whether to place it in a single physical server (it often coincides with a single process) or whether to distribute it on more servers.

Note: it is possible to make the server redundant within the realm in a Master/Slave (MIT and Heimdal) or Multimaster structure (Windows Active Directory). How to obtain redundancy will not be discussed here.

3.5.1 Database

The database is the container for entries associated with users and services. We refer to an entry as a principal. It is the owner of the entry (principal) to be aware of the secret key (or keys) associated with it.

The principal is uniquely identified by a string of the form:

- the principal name (primary + instance)
- the encryption key (the secret key)
- the maximum validity duration for tickets associated to the principal
- the maximum renewable duration associated to the principal (only in Kerberos 5)
- the attributes associated to the principal (for example the expiration of the account)
- the password expiration date
- the password expiration date for the principal (only in Kerberos 5 will not be used)

In order to make it more difficult to steal the keys present in the database, the implementations encrypt the database using a key known only to the KDC master towards the slaves. The encryption using this key, which it is necessary to allow to read the database, is known as master key.

3.5.2 Authentication Server (AS)

The Authentication Server is the part of the KDC which replies to the initial authentication requests. Replying to the client, which the user or just authenticated, must enter the password, in response to an authentication request, the AS returns a special ticket known as the TGT.

3.5.3 Ticket Granting Server (TGS)

The Ticket Granting Server (TGS) is the part of the KDC responsible for the issuance of tickets for access to the services. The TGS can be considered as an application server (given that to access it it is necessary to present the TGT) which provides the tickets to other servers.

3.6 Session Key

Every time one the users authenticates, share a secret with the KDC. This secret is known as the session key and will session changes.

However, the session key is not used to encrypt the messages exchanged between the client and the KDC. The session key is used to encrypt the messages exchanged between the client and the TGS. The session key is generated by the KDC and is encrypted with the secret key of the client (the password) and with the secret key of the TGS. The client receives the session key and the TGT. The client can decrypt the session key with its own secret key (the password) and can then use it to encrypt the messages addressed to the TGS. The TGS receives the session key and the TGT. The TGS can decrypt the session key with its own secret key and can then use it to decrypt the messages addressed to the client. The session key is used to encrypt the messages exchanged between the client and the TGS. The session key is also used to encrypt the messages addressed to the client. The session key is also used to encrypt the messages addressed to

Now each of the previous phases is described in greater detail with reference to Kerberos 5, but omitting out the differences with version 4. Nevertheless, it should be borne in mind that the Kerberos protocol is rather complicated and this document is not intended as a guide for those who wish to know the exact operating details (in any case, these are already written up in RFC1510). The discussion below has been left intentionally abstract, but sufficient for those who examine the KDC logs to understand the various authentication transitions and any problems which occur.

Note: the subsequent paragraphs enclose unencrypted data in round brackets (), and encrypted data in curly brackets {}: (x, y, z) means that x, y, z are unencrypted; { x, y, z }K indicates that x, y, z are encrypted all together using the symmetrical key K. It is also important to note that the order in which the components are listed in a packet has nothing to do with the real order found in the actual messages (UDP or TCP). This discussion is very abstract. Should you wish further details, please refer to RFC1510 having a good background on the descriptive protocol ASN.1

4.1 Authentication Server Request (AS_REQ)

In this phase, known as the initial authentication request, the client (kinit) asks the KDC (more specifically the AS) for a ticket granting ticket. The request is composed in the following way and looks like this:

$$AS_REQ = (Principal_{client} , Principal_{service} , IP_list , Lifetime)$$

where: $Principal_{client}$ is the principal associated with the user seeking authentication (e.g. pippo@EXAMPLE.COM); $Principal_{service}$ is the principal associated to the service the ticket is being asked for and thus is the string "krbtgt/REALM@REALM" (see note*); IP_list is a list of IP addresses that indicate the host where it is possible to use the ticket which will be issued (see note **); $Lifetime$ is the maximum validity time (requested) for the ticket to be issued.

***Note** *: it may seem superfluous to add the $Principal_{service}$ to the initial authentication request since this would be constantly set to the TGS principal i.e. krbtgt/REALM@REALM. However, this is not the case, indeed, a user planning to use just one service during a work session, would not use the Single Sign-on, and may ask the AS directly for the ticket for this service. This is skipping the subsequent request to the TGS. From an operational standpoint (kinit 1.3.6) the following command is sufficient:*

```
kinit -S imap/mbox.example.com@EXAMPLE.COM pippo@EXAMPLE.COM
```

***Note** **: IP_list may also be null. In this case the corresponding ticket can be used by any machine. This resolves the problem of those clients under NAT, since their request would arrive at the service with a source address different from that of the requesting user, but the same as the router making the NAT. Instead, in the case of machines with more than one network card, we should contain the IP addresses of all the cards in fact it would be difficult to predict beforehand the network which would be used for the service would be contacted.*

4.2 Authentication Server Reply (AS_REP)

When the user's previous request arrives, the AS checks whether $Principal_{client}$ and $Principal_{service}$ exist in the KDC database: if at least one of the two does not exist an error message is sent to the client, otherwise the Authentication Server processes the reply as follows:

- It randomly creates a session key which will be the secret shared between the client and the TGS. Let's say SK_{TGS};
- It creates the Ticket Granting Ticket putting it inside the requesting user's principal, the service principal (it is generally krbtgt/REALM@REALM, but read the note* for the previous paragraph), the IP address list (these first three pieces of information are copied as they arrive by the AS_REQ packet), date and time (of the KDC) in timestamp format, lifetime (see note*) and lastly the session key. SK_{TGS}; the Ticket Granting Ticket thus appears as follows:

$$TGT = (Principal_{client} , krbtgt/REALM@REALM , IP_list , Timestamp , Lifetime , SK_{TGS})$$

- It generates and sends the reply containing: the ticket created previously, encrypted using the secret key for the service (let's call it K_{TGS}); the service principal, timestamp, lifetime and session key all encrypted using the secret key for the user requesting the service (let's call it K_{user}). In summary:

$$AS_REP = (Principal_{service} , Timestamp , Lifetime , SK_{TGS})K_{user} \{ TGT \}K_{TGS}$$

It can be seen that in this message contains sensitive information (Principal_service, Timestamp, Lifetime and session key). But this is not the case: since the information present in the TGT is encrypted using the secret key for the server, it cannot be read by the client and needs to be sent to the appropriate server along with the service request. Instead, the user receives the reply with all sensitive material encrypted using his own secret key (obtained from the password provided by the client when the kinit command was launched) and can decrypt it only if the password he provides is the same as that stored in the database. If the decryption will be successful and thus the session key can be extracted and with of the message encrypted by the KDC using the secret key of the user stored in the database. If the user is really who he/she says, and has thus entered the correct password, the decrypting operation will be successful and thus the session key can be extracted and with the TGT (which remains encrypted) stored in the user's credential cache.

***Note** *: the actual lifetime, i.e. which is put in the ticket is the lowest of the following values: the lifetime requested by the client, the one contained in the user's principal and that in the service principal. Actually in terms of implementation another limit can be set from the configuration of the KDC and applied to any ticket.*

4.3 Ticket Granting Server Request (TGS_REQ)

At this point, the user who has already proved to be who he/she says (thus in his/her credential cache there is a TGT and session key SK_{TGS} and wants to access the service but does not yet have a suitable ticket, sends a request (TGS_REQ) to the Ticket Granting Service constructing it as follows:

- Create an authenticator with the user principal, client machine timestamp and encrypt everything with the session key shared with the TGS (i.e. TGS, i.e.):

$$Authenticator = (Principal_{client} , Timestamp)SK_{TGS}$$

- Create a request packet containing: the service principal for which the ticket is needed and lifetime uncrypted; the Ticket Granting Ticket which is already encrypted with the key of the TGS; and the authenticator just created. In summary:

$$TGS_REQ = (Principal_{service} , Lifetime , Authenticator) \{ TGT \}K_{TGS}$$

4.4 Ticket Granting Server Reply (TGS_REP)

When the previous request arrives, the TGS first verifies that the principal of the requested service ($Principal_{service}$) exists in the KDC database. If it exists, it opens the TGT using the key for krbtgt/REAM@REALM and extracts the session key (SK_{TGS}) which it uses to decrypt the authenticator. For the service ticket to be issued it checks that the following conditions have a positive outcome:

- the TGT has not expired;
- The principal present in the authenticator matches the one present in the TGT;
  - The authenticator is not present in the replay cache and has not expired;
    - If IP_list is Not null it checks that the source IP address of the request packet (TGS_REQ) is one of those contained in the list.

The previously checked conditions prove that the TGT really belongs to the user who made the request and therefore the TGS starts to process the reply as follows:

- It randomly creates a session key which will be the secret shared between the client and the service. Let's say $SK_{service}$;
- It creates the service ticket putting inside the requesting user's principal, the service principal, the IP address list, date and time (of the KDC) in timestamp format, the lifetime (see note* below) and lastly the session key $SK_{service}$. Known as $Principal_{service}$, the service principal, timestamp, lifetime and new session key all encrypted using the session key extracted from TGT. In summary:

$$Service_Ticket = (Principal_{client} , IP_list , Timestamp , Lifetime , SK_{service})K_{TGS}$$

- It sends the authenticator contained in the previously created packet encrypted using the decoding secret key (i.e. SK_{TGS}) and the service ticket (which is encrypted with the secret key of the requested service and extracts the session key $SK_{service}$ which it uses to decrypt the authenticator. To establish that the requesting user is authentic and thus grant the new ticket is:

$$TGS_REP = (Principal_{client} , Service_Ticket)SK_{TGS}$$

When the client receives the reply, having in the credential cache the session key (SK_{TGS}), can decrypt the reply and extract the session key ($SK_{service}$) which it uses to ask the application server for access to the resource via an AP_REQ message. It should be borne in mind that, unlike the previous messages where the KDC was involved, the AP_REQ is not standard, but varies depending on the application. Thus, the application programmer has the job of establishing the strategy with which the client will use its credentials to prove its identity to the server. However, we can consider the following strategy by way of example:

4.5 Application Request (AP_REQ)

The client, having the credentials to access the service (i.e. the ticket and related session key), can ask the application server for access to the resource via an AP_REQ message. It should be borne in mind that, unlike the previous messages where the KDC was involved, the AP_REQ is not standard, but varies depending on the application. Thus, the application programmer has the job of establishing the strategy with which the client will use its credentials to prove its identity to the server. However, we can consider the following strategy by way of example:

- The client creates an authenticator containing the user principal and timestamp and encrypts everything with the session key $SK_{service}$ that it shares with the application server, i.e.:

$$Authenticator = (Principal_{client} , Timestamp)SK_{service}$$

- It creates a request packet containing the service ticket ($Service_Ticket$), which is encrypted with its secret key and the authenticator just created. In summary:

$$AP_REQ = Authenticator \{ Service_Ticket \}K_{service}$$

When the previous request arrives, the application server opens the ticket using the secret key for the requested service and extracts the session key ($SK_{service}$) which it uses to decrypt the authenticator. To establish that the requesting user is authentic and thus grant access to the service, the server verifies the following conditions:

- the ticket has not expired;
- The principal present in the authenticator matches the one present in the ticket;
- The authenticator is not present in the replay cache and has not expired;
- If IP_list (extracted from the ticket) is not null it checks that the source IP address of the request packet is one of those contained in the list;

Note: the previous strategy is very similar to the one used by the Ticket Granting Server to check the authenticity of the user requesting a service ticket. But this should not be surprising, since we have already explained that the TGS can be considered as an application server whose service is to provide tickets to those who prove their identity with a TGT.

4.6 Pre-Authentication

As seen in the description of the Authentication Server Reply (AS_REP), before distributing a ticket the KDC simply checks that the principal of the requesting user and service exists in the KDC database. If then, particularly if it involves a TGT, it is even easier, because krbtgt/REALM@REALM certainly exists and thus it is sufficient to know that a user's principal exists to be able to obtain a TGT with a simple initial authentication request. Obviously, this TGT, if the request comes from an illegitimate user, cannot be used because they do not know the password and cannot obtain the session key not containing a valid authenticator. However, this ticket, obtained in such a easy way can undergo a brute-force attack in an attempt to guess the long-term key for the selected ticket's intended for. Obviously, guessing the secret of a service is not any easy thing even with current processing powers, however, with Kerberos 5, a pre-authentication concept has been introduced to reinforce security. Thus if the KDC policies (configurable) request pre-authentication for an initial client request, the Authentication Server replies with an error packet indicating the need to pre-authenticate. The client, in light of the error, asks the user to enter the password and resubmit the request but this time adding the timestamp encrypted with the user long term key, which, as we know, is obtained by applying the string2key to the unencrypted password after having added the salt, if there is one. This time, the KDC, since it knows the secret key of the user, attempts to decrypt the timestamp present in the request and if it is successful and the timestamp is in line, i.e. included within the established tolerance, it decides that the requesting user is authentic and the authentication process continues normally.

It is important to note that pre-authentication is a KDC policy and thus the protocol does not necessarily require it. In terms of implementation, MIT Kerberos 5 and Heimdal have pre-authentication disabled by default, while Kerberos within Windows Active Directory and the AFS kaserver (which is a pre-authenticated Kerberos 4) request it.

5 Tickets in further detail

As mentioned earlier, now that the operation of the KDC and messages between the hosts involved in the authentication have been discussed, we can now turn to the tickets. These, depending on whether they have attributes (also called flags) or not, behave in a certain manner. I have listed the most important types of tickets below, and even if not completely correct given that I are talking about a protocol, I will refer (just enough to make things clear) to version 1.3.6 of MIT Kerberos 5.

5.1 Initial tickets

An initial ticket is what is obtained directly from AS, i.e. when users must authenticate by entering the password. From here, it may be deduced that the TGT is always an initial ticket. On the other hand, the service tickets are distributed by the TGS upon presentation of a TGT and thus are not initial tickets. However, there is an exception to this rule, in order to guarantee that the user enter only the password once, some Kerberos applications may request that the service ticket obtained in this case be an initial ticket despite not being a TGT. Is requested from the AS instead of the TGS and is thus an initial ticket. In operational terms, the user pippo, wishing to obtain a ticket which is initial (thus without using the TGT) for an imap service on the machine mbox.example.com uses the command:

```
[pippo@client01 pippo]$ kinit -S imap/mbox.example.com@EXAMPLE.COM pippo@EXAMPLE.COM
Password for pippo@EXAMPLE.COM:
[pippo@client01 pippo]$
[pippo@client01 pippo]$ klist -f
Ticket cache: FILE:/tmp/krb5cc_500
Default principal: pippo@EXAMPLE.COM

Valid starting     Expires            Service principal
01/27/05 14:28:59  01/28/05 14:28:39  imap/mbox.example.com@EXAMPLE.COM
        Flags: I

Kerberos 4 ticket cache: /tmp/tkt500
klist: You have no tickets cached
```

The presence of the flag I should be a clear indication that it is an initial ticket.

5.2 Renewable tickets

A renewable ticket can be resubmitted to the KDC for renewal, i.e. it is reassigned the entire lifetime. Obviously, the KDC will honour the renewal request only if the tick[illegible] Center database). Being able to renew a ticket combines the necessity of having short duration tickets for security reasons, with not having to re-enter the password for long periods, for example, imagine a job which must be processed for days and must not [illegible] human intervention. In the following example pippo asks for a ticket which lasts for a maximum of one hour but is renewable for 8 days:

```
kinit -l 1h -r 8d pippo
Password for pippo@EXAMPLE.COM:
[pippo@client01 pippo]$
[pippo@client01 pippo]$ klist -f
Ticket cache: FILE:/tmp/krb5cc_500
Default principal: pippo@EXAMPLE.COM

Valid starting     Expires            Service principal
01/27/05 15:35:14  01/27/05 16:34:54  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/03/05 15:35:14, Fla[illegible]

Kerberos 4 ticket cache: /tmp/tkt500
klist: You have no tickets cached
```

while for pippo to renew his ticket without re-entering the password

```
[pippo@client01 pippo]$ kinit -R
[pippo@client01 pippo]$
[pippo@client01 pippo]$ klist -f
Ticket cache: FILE:/tmp/krb5cc_500
Default principal: pippo@EXAMPLE.COM

Valid starting     Expires            Service principal
01/27/05 15:47:52  01/27/05 16:47:32  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/03/05 15:35:14, Flags: RIT

Kerberos 4 ticket cache: /tmp/tkt500
klist: You have no tickets cached
```

5.3 Forwardable tickets

Let's suppose we have a work session on a machine with the related TGT and wish to login from it onto another machine, keeping the ticket. Forwardable tickets are the solution to this problem. A ticket forwarded from one host to another is in itself [illegible] forwardable, thus once authenticated it is possible to access the login on all the desired machines without having to re-enter any password.

To obtain the same result without Kerberos, it would be necessary to use much less secure methods such as rsh or public key authentication with ssh. However, this latter method may be impracticable on systems where the user home directories a[illegible] re on network filesystems (e.g. NFS or AFS) since the private key (which should be private) would go over the network.

6 Cross Authentication

We've already mentioned the possibility for a user belonging to a certain realm to authenticate and access the services of another realm. This characteristic known as cross-authentication is based on the assumption that there is a trust relationship betw[illegible] een the realms [illegible] We will look at cross authentication, [illegible] breaking down the trust relationships into direct, transitive and hierarchical.

6.1 Direct trust relationships

This type of trust relationship is elem[illegible] and is the basis of cross-authentication and is used to construct the other two types of relationships we will look at later. It occurs when the KDC of realm B has direct trust in [illegible] the KDC of realm A, thus allowing the users of the [illegible] latter realm to access its resources. From a practical point of view, a direct trust relationship is obtained by having the two involved KDCs share a key (the keys become two if a bi-directional trust is desired). To do this [illegible] a concept of a remote Ticket Granting Ticket is [illegible] introduced which, in the example of the two realms A and B, assumes the form krbtgt/B@A and is added to both the KDCs with the same key. This key is the secret which will guarantee the trust between the [illegible] two realms. Obviously, to make it bi-directional (i.e. that A also [illegible] trusts B), it is necessary to [illegible]

[illegible] then Kerberos 5 will support adjacent realms (hierarchically) having a trust [illegible] presence of appropriate keys) and will automatically construct (without the need for capaths) the transitive authentication paths. However, administrators can alter this automatic mechanism (for example [illegible] ration.

[illegible] relationship (naturally this assumed trust must be supported by the [illegible] for reasons of efficiency) by forcing the capaths in the client configu[illegible]

© The MIT Kerberos Consortium 2007. All Rights Reserved. Terms & Conditions | Privacy Policy