The MIT Kerberos Administrator's How-to Guide

Protocol, Installation and Single Sign On

By Jean-Yves Migeon

Contents

First part - Introducing Kerberos

In the real world, identification is something we, as human beings, do naturally: through physical appearance, voice patterns, or even scent. It is based on the assumption that those attributes are unique, and that they can be trusted. This ability provides us with the possibility to distinguish one person from another.

However, when put in a situation where we're not able to use those attributes to identify someone, as in a phone call for example, we're left with finding some other means to prove our identitys. We sometimes identify ourselves with what is called a "shared secret", where one party asks the other party to prove his identity through information that is only known by both, like a password.

When we add a computer to this mechanism, with an identification that needs to be provided over a network, things are going a little more complex. Sending this "shared secret", or password, over an unsecured network can be compared to shouting your password in a crowded room.

Many authentication mechanisms were developped during the last decade to solve those problems; Kerberos is one of them. Often seen as an advanced system that offers many more advantages over commonly used setups, such as distributed authentication based on Network Information Server (NIS).

This white paper is intended to introduce, describe, and explain a Kerberos environnement, and how to deploy such system for maximum efficiency with Single Sign On (SSO).

Unix historical authentication and authorization system: NIS

Today, NIS remains the system of choice for network authentication and authorization, were it is used in an environment that consists of a server (containing all the necessary directory services, like */etc/passwd* and */etc/shadow*[1]), and clients, which need this information to allow (or deny) access to certain persons.

NIS (and his counterpart, NIS+) were developed with "central authentication" in mind: administrators have the possibility to create realm accounts, and, with the help of file sharing systems (like NFS), share profiles over an entire network.

NIS is easy to set up and manage, which makes it so popular. However, it remains fundamentaly flawed on the security front. For example, NIS communications are cleartext based. Even unpriviledged users are able to display the content of the passwd database, through yp (yellow pages) commands.

NIS does not provide any kind of Single Sign On mechanisms, the ability to securely store authenticators on the client, preventing the user to re-enter passwords when accessing services (file sharing, intranet, mails, ...).

All in all, on the authentication side, NIS has some flaws that Kerberos tends to solve with his own implementation.

Those flaws are:

- no secure propagation of user authenticators. With some time, anyone using a sniffer is able to get all the cleartext or hashed password propagated on the network. With Kerberos, password never cross the network, even on first login.
- no mutual authentication. In a NIS environment, there's no evidence that the client is a legitimate one, or not. That is, any person capable of faking the server as being a

legitimate NIS client is able to get all the hashed password table stored on the server side.

- no Single Sign On (SSO) possible. In fact, there's no information cached onto the client that allows future use of pre-authentication when the user loged in some time ago. When a user wants to access some kind of resources which requires an authentication, he must reenter his password. One more time, one more risk to get his password eavesdropped.
- When dealing with services that delegate authentication to client, like NFS, a smart client could potentially gain access to any ressources that are shared on the network. That's not something you really want, especially if you're asked to restrict resources' access based on authenticator.
- Because it relies on RPC using dynamically allocated ports, NIS is firewall unfriendly.

These flaws are not necessarly tied to NIS. You can encounter the same ones with any cleartext-based authentication process (for example, LDAP), where no particular security context has been initiated beforehand.

Usually, securing these systems (to avoid password eavesdropping) relies on mechanisms found on lower layers, like SSL or IPsec. While providing a security context, they do not provide users' authentication[2], something Kerberos offers, alongside SSO.

How does Kerberos work?

This part of the article will explain the mechanisms behind Kerberos: ticket exchange principles, Key Distribution Center (termed *KDC*), and authentication mechanisms.

Kerberos was developed with authentication in mind, and not authorization (or accounting). In fact, Kerberos could be compared to some supreme service that tells others: "yes, you can trust me, and this person is the one she claims to be". Nothing more.

A commonly found description for Kerberos is "a secure, single sign on, trusted third party mutual authentication service". It doesn't store any information about UIDs, GIDs, or home's path. In order to propagate this information to hosts, you will eventually need yellow page services: NIS, LDAP, or Samba.

As Kerberos is only dealing with Authentication, it does neither Authorization (the step of granting or denying access to a service based on the user wishing to use it), nor Accounting (account and session management, as well as logging): it delegates those to the services requesting Kerberos' help for user's identification. Anyway, Kerberos being a "service" by itself, it can partially provide such functionalities, but in a very limited range.

Kerberos, being a protocol, has many implementations, developed for different purposes:

- **MIT Kerberos**. The original one; comes from the Project Athena in early 90s. Due to exportation restrictions on cryptography technology, another implementation of Kerberos was developped, in Sweden: Heimdal.
- **Heimdal Kerberos**. Is MIT Kerberos' Swedish counterpart. Heimdal is not restricted by exportation rules. Originally developed in Sweden, it aims to be fully compatible with MIT Kerberos. Since MIT export restrictions were lifted in 2000, both implementations tends to coexist on a wider scale.
- **Active Directory**. Not a Kerberos implementation by itself, but kind of. Its the Microsoft's directory, that consist of a loose Kerberos implementation with some other services (LDAP). It's not directly compatible with MIT and Heimdal.

- **TrustBroker**. A commercial implementation of Kerberos protocol, developped by CyberSafe. Supports a wide range of Operating Systems (Windows, Unix, Linux, ...), and offers full interoperability with many existing Kerberos implementations, from MIT to Microsoft's AD.
- **Shishi**. A GNU implementation of Kerberos 5.

The author decided to give a try to the MIT implementation. how-tos and commands described below are almost compatible with a Heimdal implementation, the key differences (from an administrator perspective) being that Heimdal stores the server's configuration directly into the database, not in a separated text file like MIT does.

There are two publicly available versions for Kerberos, namely v4 (deprecated) and v5 (often written Kerberos 5). While v4 is still used in some places, it is strongly advised to migrate it to a Kerberos 5 implementation, as v5 offers many more functionalities compared to v4, and an improved security.

Now, we will go into details in Kerberos' functioning. Let's talk about the ticket exchange service.

Ticket Exchange Service

Kerberos' communication is built around the Needham-Shroeder protocol (NS protocol). Described in a paper published in 1978 by Roger Needham and Michael Shroeder, it is designed to provide a distributed secure authentication service, through secret key cryptography.

All those keys are secrets shared by the two ends of a Kerberos connection. It differs from asymmetric systems, like SSL or IPsec, where a public key is known by virtually everybody, while the private key remains secret, and stored on the server.

For a user, the secret key is his "hashed password" (the password is reworked through a one-way hash function and the resulted string is used as a key), usually stored in the Key Distribution Center. For a service, the key is a random generated sequence, acting like a password; it is also stored in Key Distribution Center, and in a file called a keytab on the machine's service side.

For this schema to work, clients and services have to trust a third party service (the Kerberos server), that is capable of issuing the required keys on demand.

The Kerberos communication is based around tickets. Tickets are a kind of encrypted data scheme that is transmitted over the network, and stored on the client's side. The type of storage depends on the client's operating system and configuration. Traditionally, it's stored as a small file in /tmp, for compatibility reasons: each OS has some kind of filesystem, able to save data.

The main central part of a Kerberos network is the the Key Distribution Center (KDC). It consists of three parts:

- an Authentication Server, which answers requests for Authentication issued by clients. Here, we're in the AS_REQUEST and AS_REPLY challenging part (see below for details), where the client gets a Ticket Granting Ticket (TGT).
- a Ticket Granting Server, which issues Ticket Granting Service (TGS) to a client. This is the TGS_REQUEST and TGS_REPLY part, where a client gets a TGS that allows him to authenticate to a service accessible on the network.
- a database, that stores all the secret keys (clients' and services' ones), as well as some information relating to Kerberos accounts (creation date, policies, ...).

Usually, the KDC is a distinguished machine on the network, dedicated only to serve Kerberos services. This is mainly for security purposes: since the KDC stores the database containing all the secret keys, a compromised KDC allows the attacker to impersonate any service and client stored in KDC's database.

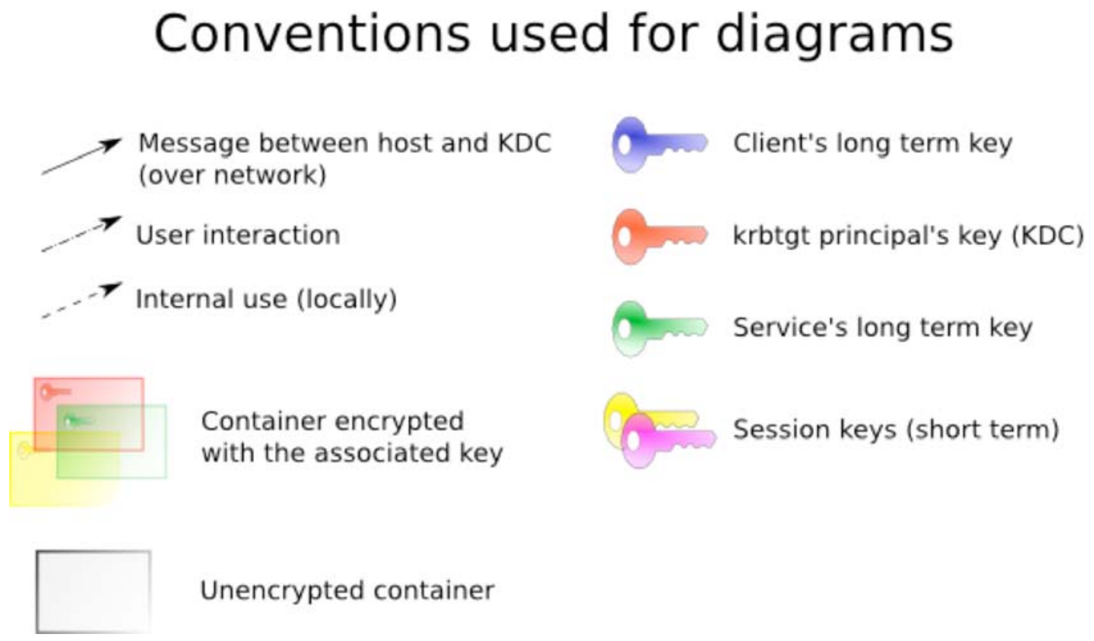
Kerberos accounts are named through principals, the equivalent of the username for a Unix account.

Kerberos' cryptosystem works with DES and his variants, like 3DES. AES' support is ongoing, as described in RFC 3962 "Advanced Encryption Standard (AES) Encryption for Kerberos 5".

Now we will concentrate on Kerberos 5 mechanisms.

Each step is summarized by a picture, showing the different mechanisms taking place during tickets negotiations. Please find them at the end of their corresponding part (Authentication Request, and Service Request).

The convention for the pictures is as follows:


Conventions used for diagrams
Message between host and KDC (over network)
User interaction
Internal use (locally)
Container encrypted with the associated key
Unencrypted container
Client's long term key
krbtgt principal's key (KDC)
Service's long term key
Session keys (short term)

Authentication mechanism—Ticket Granting Tickets

Authentication mechanism is the first step to be done in a Kerberos environment. It provides the user with a Ticket Granting Ticket (TGT), that serves post-authentication for later access to specific services, Single Sign On.

Pre-authentication

Originally, Kerberos 4 protocol implementation was subject to off line attacks and brute force cracking on tickets. This was due to the fact that the KDC issued a TGT with a principal's secret key on each request for an authentication. A malicious guy could recursively ask for TGTs using different principals, get the tickets, and try to brute force the user's long term key off line.

To solve this security flaw, Kerberos 5 introduced pre-authentication methods. The principle lies in the necessity that the client identifies himself to the KDC first before getting a TGT. This way, an attacker must contact the KDC each time he tries a new password.

1st step: Authentication Service Request—AS_REQUEST

This is the first message sent to the KDC, in plain text. It contains:

- the client's principal name,
- the Ticket Granting Server's principal (termed "krbtgt principal", needed to obtain further TGS),
- the client timestamp,
- the requested ticket lifetime (usually, 8 to 10 hours long).

The KDC receives this message, check if the client's principal has a match in the database, and if the timestamp between client's machine and KDC are close enough (typically, 3 to 5 min). This timestamp's check does not prevent replay; it's only used as way to early warn the user from an incorrect time synchronization, before going any further into authentication.

If pre-authentication is mandatory, KDC won't return a TGT. Instead, it will send a NEEDED_PREAUTH message, and ask the client to provide some pre-authentication data before delivering the TGT. traditionally, the method used is PA-ENC-TIMESTAMP, where the current timestamp is encrypted using the user's key, known on the client's side through password.

In this case, the client re-send an AS_REQUEST message, this time with the encrypted timestamp included. Given a successful pre-authentication, KDC will return a TGT; this is the AS_REPLY step.

2nd step: Authentication Service Reply—AS_REPLY

Upon checking, the Authentication Server generates a random session key ("short term" key). The KDC makes two copies of it: one is for the client, and is added to the AS_REPLY message, the second copy remains available for the Ticket Granting Server. This key is mainly used for later negotiations for other tickets concerning kerberized services.

Provided the client succeeded in his authentication, the KDC will return an AS_REPLY message, containing the Ticket Granting Ticket. It will be stored in some kind of credential cache, for future use. The message is encrypted with the user's key, thus preventing any impersonator from decoding it.

The AS_REPLY message is formed of two layers; the first one is encrypted with the user's key, while the second layer is the TGT itself, first encrypted with the Ticket Granting Server's key, then re-encrypted with the user's key. This way, only the trusted user is able to decipher the message and get the TGT.

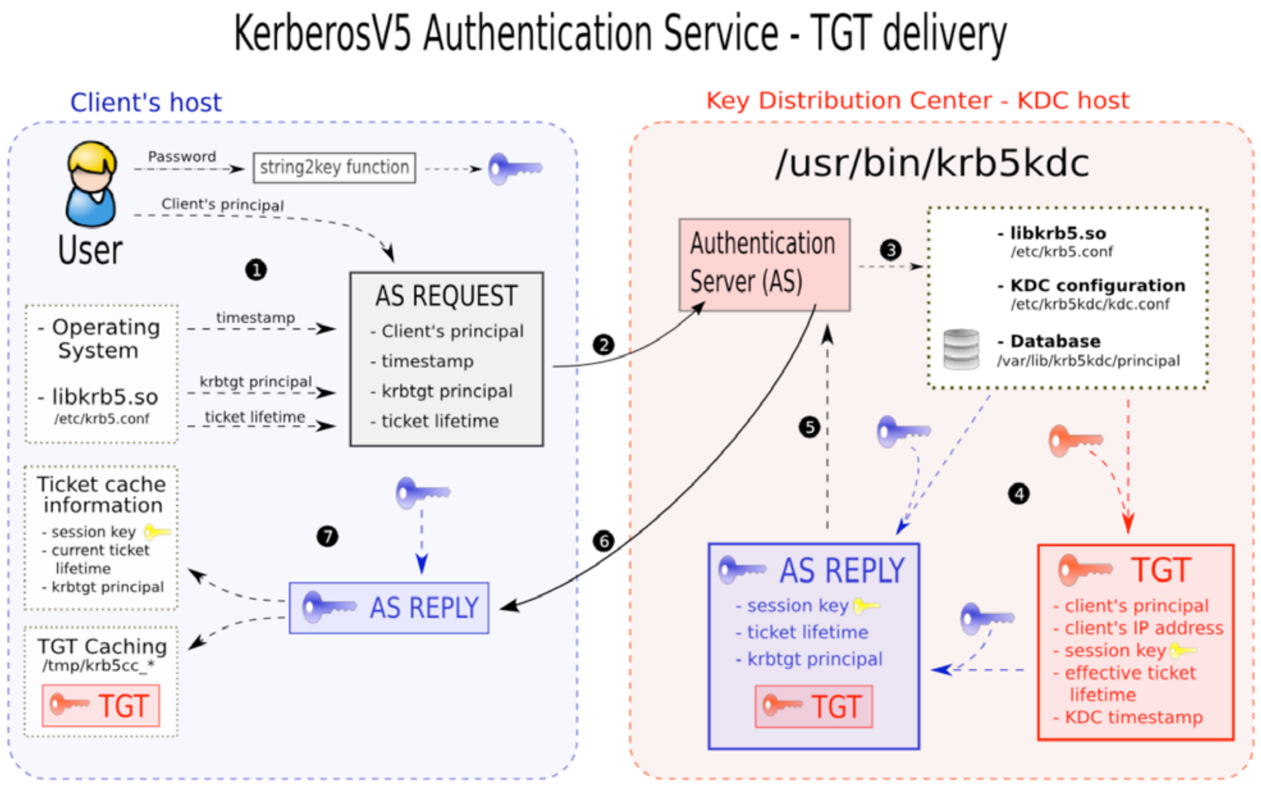
The content of the AS_REPLY message is as follows:

- encrypted with user's key:
  - copy of session key for user
  - Ticket lifetime
  - krbtgt principal name
- first encrypted with Ticket Granting Server's key, then user's key. This is the TGT:
  - copy of session key
  - effective ticket lifetime

- KDC timestamp
- client principal
- client IP address

Note: Although the TGT is decrypted and cached onto the client, its content cannot be read on the client's side. It is effectively encrypted with the Ticket Granting Server's key, which is only known by Ticket Granting Server.

in conclusion, the Authentication mechanism can be represented as follows:



Service's use mechanism—Ticket Granting Service

We suppose that the client has already gone through the authentication mechanism, and has a Ticket Granting Ticket (TGT). Now he's requesting access to a particular service on the network (web server, file sharing...), and for this, he requires a Ticket Granting Service (TGS).

Again, this request is separated into two steps, TGS_REQUEST and TGS_REPLY. Both messages are encrypted, for security reasons.

1st step: Ticket Granting Service Request—TGS_REQUEST

When the user wishes to access to a kerberized service, he must first authenticate himself to it. This pre-requisite needs a separate connection to the Ticket Granting Server: the TGS_REQUEST.

The message sent by the client is composed of several elements:

- the TGS request itself, containing the service principal and the requested lifetime,
- the TGT acquired earlier (on a successful authentication),

- an authenticator.

The authenticator is here to thwart replay. It's a message encrypted with the session key acquired during the AS process, and contains the user's principal and a timestamp. This way, the KDC ensures that this unique message is coming from the right person: first by checking the temporary session key negotiated earlier, and second, through timestamp, which detects fraudulent attempt of replay.

Upon successful request (a valid TGT and correct authenticator), the Ticket Granting Server will return the TGS.

2nd step: Ticket Granting Service Reply—TGS_REPLY

At this stage, the server generates a new set of session keys.

The reply message from the server is encrypted with the session key acquired through AS process. As such, only the client that effectively identified himself some time ago to KDC is able to read its content, and extract the TGS from it.

The message forms the TGS_REPLY part issued by KDC. It contains:

- encrypted with session's key acquired through AS process:
  - copy of new session key, for user
  - effective ticket lifetime
  - service's principal name
- first encrypted with service's long term key, then with the actual session's key. This is the TGS:
  - copy of new session key, for service
  - effective ticket lifetime
  - KDC timestamp
  - client principal
  - client IP address

3rd step: Contacting service

Once the client obtained its TGS, he will use it to authenticate himself to the requested service directly. Since this step depends largely on the service that required Kerberos' help, we won't go into details here.

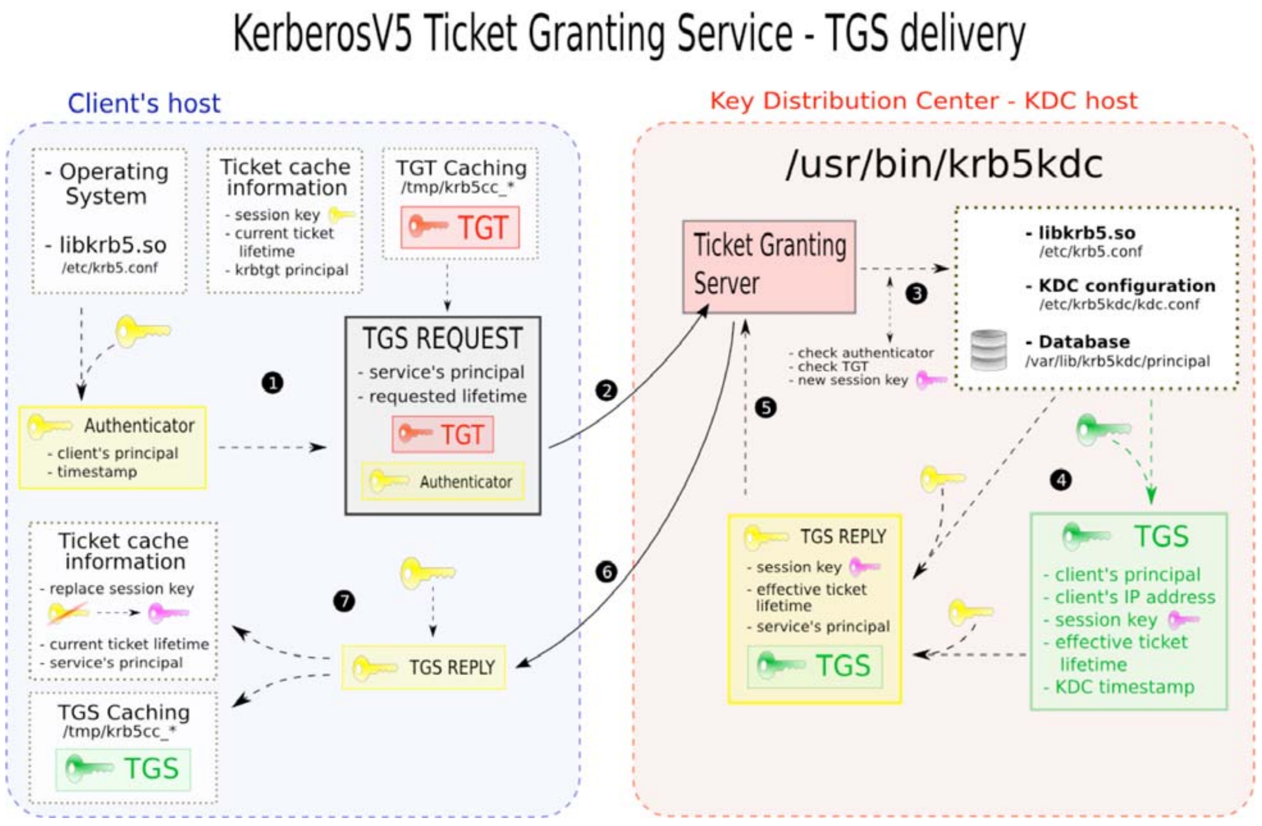
The service has access to its keytab, a file that stores its long term key. This key will allow the service to decrypt the TGS sent by the client, and get all the information needed to identify user and create security context.

Like for the TGT process, the timestamp encoded in the TGS is here to thwart replay.

Traditionally, the session key is used to sign or crypt messages between client and service. It provides both endpoints with a way of checking the integrity of traded messages (if messages are signed), and eventually the creation of a security context, making eavesdropping very difficult.

In this last configuration, Kerberos is complementary to other security protocols, like TLS/SSL or IPsec; the main difference being that they are based on asymmetric cryptography (RSA), whereas Kerberos is built around symmetric cryptography (DES and AES). This situation is often encountered in PKI (Public Key Infrastructure) environment.

Here is a quick visual summary of the TGS steps:



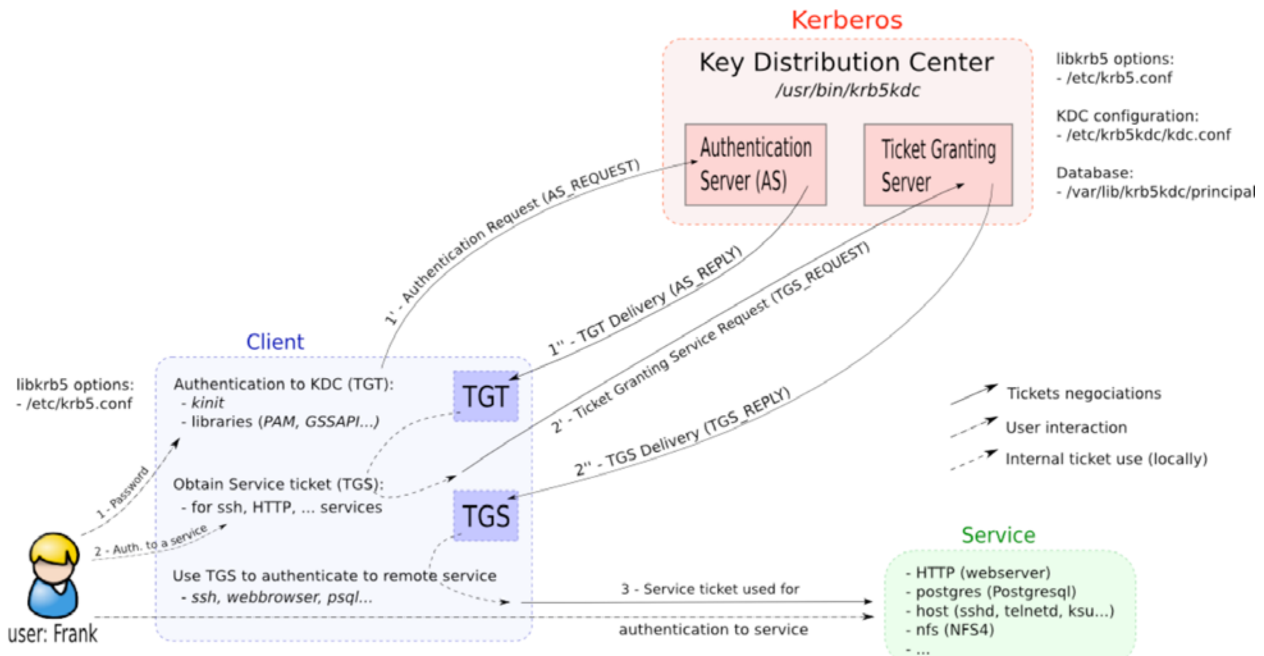
Conclusion

We can divide the Kerberos protocol into three main steps:

1. Authentication process, where the user (and host) obtain a Ticket Granting Ticket (TGT) as authentication token,
2. Service request process, where the user obtain a Ticket Granting Service (TGS) to access a service,
3. Service access, where the user (and host) use TGS to authenticate and access a specific service

The service access step is not really Kerberos related, but merely depends on the service we are authenticating to. This step is mainly covered in the third part of this article.

KerberosV5 Tickets Negociation mechanism


Kerberos
Key Distribution Center
/usr/bin/krb5kdc
Authentication Server (AS)
Ticket Granting Server
libkrb5 options:
- /etc/krb5.conf
KDC configuration:
- /etc/krb5kdc/kdc.conf
Database:
- /var/lib/krb5kdc/principal
1' - Authentication Request (AS_REQUEST)
1'' - TGT Delivery (AS_REPLY)
2' - Ticket Granting Service Request (TGS_REQUEST)
2'' - TGS Delivery (TGS_REPLY)
Client
libkrb5 options:
- /etc/krb5.conf
Authentication to KDC (TGT):
- kinit
- libraries (PAM, GSSAPI...)
TGT
Obtain Service ticket (TGS):
- for ssh, HTTP, ... services
TGS
Use TGS to authenticate to remote service
- ssh, webbrowser, psql...
1 - Password
2 - Auth. to a service
user: Frank
3 - Service ticket used for
authentication to service
Tickets negociations
User interaction
Internal ticket use (locally)
Service
- HTTP (webserver)
- postgres (Postgresql)
- host (sshd, telnetd, ksu...)
- nfs (NFS4)
- ...

Second part - Deploying Kerberos

In our preceding article, we described the theoretical aspects of Kerberos.

We will describe here the basic steps required to get Kerberos working on a single domain, and how to set it up to use it effectively.

Before going any further, you need to be familiar with these terms:

- Key Distribution Center (KDC)
- Principals
- Tickets
- Ticket Granting Ticket (TGT)
- Ticket Granting Service (TGS)
- Key table (keytab)

In case your Kerberos' deployment is part of a migration that deals with authentication, you will have to migrate all the users' password from your old system to Kerberos; and that is not an easy task. Some hints are provided at the end of this chapter, as you require to have a working KDC first.

Installing Kerberos MIT

Many GNU/BSD distributions have tools to automatize all the installation steps described below, but you can also compile and run it directly from source.

For maintenance and ease of administration, it is strongly advised to use your favorite tools to install MIT Kerberos. For quick reference, here are the package names for different systems:

OS	Package name
Debian	krb5-kdc and krb5-admin-server
Redhat/CentOS	krb5-server
FreeBSD ports	security/krb5
NetBSD pkgsrc	security/mit-krb5

In case you do not have such tools at your disposal, or do not want to use them, you can compile and install it from source. The basic steps are described below.

In case you installed it from a package manager, you can jump to the next section, Server configuration.

Source compilation and installation

We will download the sources first. They are available at the MIT Kerberos homepage.

Once you downloaded it, you will have a tar file (an archive). After untarring it, you get the tar.gz source code, and a signature file, to check that the file you have downloaded hasn't been altered by some malicious guy.

We use gpg (GNU Privacy Guard) to ensure the integrity of the package. In the same directory where you have untared the source file in, type (change krb5 version accordingly, if necessary):

```
# gpg --verify krb5-1.5.tar.gz.asc
```

The package is signed with Tom Yu's private key, member of the Kerberos MIT development team. We need his public key to check the files' integrity.

Building the Kerberos 5 does not differ much from any other Unix package. It uses GNU autoconf. A complete list of all the compilation options are accessible through *./configure --help* command. Generally, the defaults are sufficient in almost all cases.

So, we start configuring and compiling the binaries:

```
# ./configure && make
```

Once finished, you should either su root or sudo to make the final step in installation:

```
# make install
```

Note that Kerberos binaries and libraries will be installed under the default prefix directory */usr/local*. If, for some reason, you need a different directory, please do so at ./configure step, through the --prefix=[dir] option.

Last but not least, we need to create the the *krb5kdc* directory which resides in *localstatedir*. It's the directory where all the variable files (KDC's keys databases) will be stored in. By default, *localstatedir* is */usr/local/var/*. So create krb5kdc directory:

```
# mkdir -p /usr/local/var/krb5kdc/
```

Be sure to set the permissions on this directory accordingly. Only root should have access to it.

Server configuration

Now that Kerberos is installed on your system, we should configure it.

We describe here a simple KDC installation (one domain, called a "realm" in Kerberos terminology). More complex infrastructures are described in part three of this article.

To setup MIT Kerberos, we will do it in two steps:

- configure and run the KDC
- check that we can use it to authenticate ourselves correctly

First, edit the configuration file used by Kerberos libraries. By default, it resides in */etc/krb5.conf*. These libraries are used by KDC entities as well as Kerberos client tools.

```
/etc/krb5.conf

[libdefaults]
        default_realm = FOOBAR.COM
        kdc_timesync = 0
        forwardable = true
        proxiable = true

[realms]
FOOBAR.COM = {
        kdc = kdc.foobar.com
        admin_server = kdc.foobar.com
}

[domain_realm]
        .foobar.com = FOOBAR.COM
        foobar.com = FOOBAR.COM

[login]
        krb4_convert = false
        krb4_get_tickets = false
```

Now, we will edit the configuration file of the KDC, namely *kdc.conf*. It resides in the */usr/local/var/krb5kdc* directory (except if your distribution changes the default behaviour).

```
kdc.conf

[kdcdefaults]
        kdc_ports = 750,88

[realms]
FOOBAR.COM = {
              database_name = /var/lib/krb5kdc/principal
              admin_keytab = FILE:/etc/krb5kdc/kadm5.keytab
              acl_file = /etc/krb5kdc/kadm5.acl
              key_stash_file = /etc/krb5kdc/stash
              kdc_ports = 750,88
              max_life = 8h 0m 0s
              max_renewable_life = 1d 0h 0m 0s
              master_key_type = des3-hmac-sha1
              supported_enctypes = des3-hmac-sha1:normal des-cbc-crc:normal des:normal des:v4
des:norealm des:onlyrealm des:afs3
              default_principal_flags = +preauth }
```

We add the +preauth flag for security reason, unless you want compatibility with an existing Kerberos 4 implementation. the Ticket Granting Ticket (TGT) is renewable for a maximum period of 1 day, and expires (unless renewed) after 8 hours (a full work day).

When finished with the conf files, we create the database containing all the principals and their passwords. Luckily, MIT offer a utility to create all the necessary files for you, namely *kdb5_util*. *kdb5_util* is mainly used for low level maintenance (creation, dumping, saving, destruction of KDC database, and so on).

During creation, you will be prompted for the master password. It is the main key that is used by Kerberos to encrypt all the principals' keys in its database. Without it, Kerberos won't be able to parse it. For later convenience, this master password can be stored in a stash file, in order to avoid to retype it each time you restart Kerberos (therefore avoiding unnecessary human interaction).

Execute *kdb5_util* to create database (the -s flag specify the creation of the stash file):

```
# kdb5_util -s create
```

Since this password can decrypt all the Kerberos database, please respect some basic security rules:

- long password, with letters, numbers and special chars
- never share this password.

This password does not provide any kind of authentication to Kerberos. It is used only for encryption of the database, and access to it, for low level maintenance only.

Last part of the configuration: the Access Control Lists (ACLs) to the database. This simple text file will define the rights some principals have on the database: listing principals, changing their policy or their password, or updating their profile.

The path to the ACL file is defined in *kdc.conf*. For our installation, we use very simple (and restrictive!) rules; adapt it to your convenience. The man page of *kadmind* is pretty much straightforward concerning ACLs; first rule that match will be applied, when processed from top to bottom.

```
/usr/local/var/krb5kdc/kadm5.acl

 */admin@FOOBAR.COM    *
```

* acts as a wildcard. This rule grants all rights to any principal authenticated with a /admin instance.

If you change the ACL file later, remember to restart the KDC.

Once finished, start up KDC (*krb5kdc*), and kerberos Administration Server (*kadmind*):

```
# krb5kdc
# kadmind
```

Or, depending on your OS (either SystemV or BSD-like):

```
# /etc/init.d/krb5kdc start
# /etc/init.d/kadmind start
```

```
# /etc/rc.d/kkdc start
# /etc/rc.d/kadmind start
```

Your Kerberos database is ready. The next step will be to connect to it, and check that everything is fine. For that matter, we will now use the Kerberos administration server.

Connection to administration server is done through *kadmin*. Since we did not create any principal yet, connecting to administration server is impossible, as KDC can not authenticate us. So, we use the "local" counterpart of *kadmin*, *kadmin.local*, to connect. It will access directly the Kerberos administration interface without password, but can only be run as **root** on the KDC's host.

```
# kadmin.local
Authenticating as principal root/admin@FOOBAR.COM with password.
kadmin.local:
```

You are now in the Kerberos administration shell. <TAB> key works for autocompletion, and '?' for in line help, in case you need it. If you did not notice, kadmin added a /admin instance to your Unix username upon login, to generate the principal you will be authenticated as for this session. That's the traditionnal behaviour.

First, we will list the content of the database, through the *listprincs* command. You will notice that the database contains already some principals. They are needed for Kerberos to work, during ticket negotiations.

```
# kadmin.local
Authenticating as principal root/admin@FOOBAR.COM with password.
kadmin.local: listprincs
K/M@FOOBAR.COM                      # master key record in KDC database
kadmin/admin@FOOBAR.COM             # admin instance of kadmin
kadmin/changepw@FOOBAR.COM          # instance to change password
kadmin/history@FOOBAR.COM           # used to keep the history of changed passwords (if required)
kadmin/kdc.foobar.com@FOOBAR.COM    # principal for administration from kdc.foobar.com (optional)
krbtgt/FOOBAR.COM@FOOBAR.COM        # Ticket Granting Server's principal
kadmin.local:
```

Do not modify nor delete their properties, or Kerberos will not work anymore.

Now, for testing, we create a test user, **frank** (the command *ank* being an alias of *add_principal* method):

```
kadmin.local: ank frank
WARNING: no policy specified for frank@FOOBAR.COM; defaulting to no policy
Enter password for principal "frank@FOOBAR.COM":
Re-enter password for principal "frank@FOOBAR.COM":
Principal "frank@FOOBAR.COM" created.
kadmin.local:
```

Done! Now, we exit the kerberos shell (*exit* or Ctrl+D) and check that we could authenticate with principal **frank**:

```
# kinit frank
Password for frank@FOOBAR.COM:
```

If you did not receive any error, you should be now authenticated to Kerberos as **frank@FOOBAR.COM** (you now have frank's TGT). To verify, execute *klist*, and check that you have been properly authenticated (see Default principal):

```
# klist
Ticket cache: FILE:/tmp/krb5cc_0
Default principal: frank@FOOBAR.COM

Valid starting     Expires            Service principal
02/21/07 18:27:31  02/22/07 18:27:31  krbtgt/FOOBAR.COM@FOOBAR.COM
```

Now, we will try to connect to administration server as user **frank**. For that matter, we use *kadmin*, with -p flag (remember, by default, *kadmin* adds a /admin instance to username, so we avoid it):

```
# kadmin -p frank
Authenticating as principal frank with password.
Password for frank@FOOBAR.COM:
kadmin.local:
```

Since **frank** has no rights (except changing his own password), try executing some commands:

```
kadmin.local: listprincs
get_principals: Operation requires ``list'' privilege while retrieving list.
kadmin.local: get_policy frank
get_policy: Operation requires ``get'' privilege while retrieving policy "frank".
kadmin.local: ch_password
Enter password for principal "frank":
Re-enter password for principal "frank":
Password for "frank@FOOBAR.COM" changed.
kadmin.local:
```

If everything succeeded, your KDC is up and running! Now, we will go to a client configuration, where we will configure a remote host.

Client configuration

The following part deals with the configuration of a basic computer host on the network, the "client":

IP address	Hostname
192.168.1.1	**kdc-client.foobar.com**
192.168.1.101	**kdc.foobar.com**

The main configuration file is */etc/krb5.conf*. This file is mainly used by the Kerberos library to configure any kerberized client requiring access to KDC.

This *krb5.conf* file does not differ from his "server" counterpart: in fact, even the server hosting Kerberos is, to some extent, a client to the Kerberos service it is hosting. The only difference is that now, ticket's exchange will be done across network.

```
/etc/krb5.conf

[libdefaults]
        default_realm = FOOBAR.COM
        kdc_timesync = 0
        forwardable = true
        proxiable = false

[realms]
FOOBAR.COM = {
        kdc = kdc.foobar.com
        admin_server = kdc.foobar.com
}

[domain_realm]
        .foobar.com = FOOBAR.COM
        foobar.com = FOOBAR.COM

[login]
        krb4_convert = false
        krb4_get_tickets = false
```

Now that your *krb5.conf* file is correct, we will try to initiate an authentication to the KDC. For that matter, we will use *kinit*.

But before trying *kinit*, we will ensure two things, necessary for authentication to work:

- clock synchronization (clock skew between both machines)
- DNS and reverse DNS

Clock sync

Clock sync is usually performed through the use of the NTP protocol, with the help of either *ntpdate* (binary) or *ntpd* (daemon). It is mainly used by Kerberos to avoid replay attack (if an attacker manages to get a ticket, he will not be able to use it indefinately).

Building and installing NTP is not part of this tutorial. Just check that your KDC machine and your client have both a clock skew of less than 5 min (default MIT Kerberos value, specified in *krb5.conf*); the lower, the better.

DNS and reverse DNS

Kerberos relies heavily on DNS, either for contacting KDC, or resolving hostname for authentication (for service use -- see later on).

You do not necesseraly need a DNS server for Kerberos to work. */etc/hosts* file is sufficient, but somewhat limited when network's size grows. Kerberos only needs proper direct- and reverse-resolution of hostname, which should point to their respective FQDN (Fully Qualified Domain Name).

Installing and configuring a DNS does not belong to such an article. For our example, we will use a very simple */etc/hosts* file:

```
/etc/hosts

127.0.0.1       localhost
192.168.1.101   kdc.foobar.com kdc
192.168.1.1     kdc-client.foobar.com kdc-client
...
```

Be sure to specify the FQDN first, then aliases for the host. If not, hostname resolution will fail, and consequently, Kerberos authentication.

Now that DNS and clocks are configured, use *kinit* to initiate a Kerberos authentication:

```
# kinit -p frank    # Initiate a ticket negotiation for principal frank@FOOBAR.COM
Password for frank@FOOBAR.COM
```

If your account's username on this computer is **frank**, simply type *kinit*; it will automatically use your username in order to generate the principal that will be used for authenticating, **frank@FOOBAR.COM**.

Note that you could also use *kadmin* on this host to administer the Kerberos database, or connect to your Kerberos account (to change password for example). The mechanisms are exactly the same as before; if your authentication is successul, Kerberos will connect to administration server through a secured RPC connection.

Changing his own password through *kadmin* is quite tedious. To avoid all the hassle of loging through *kadmin* and typing in commands, a user can use *kpasswd* instead.

To check that the TGT has been correctly received, use *klist*. It will tell you under which principal you are currently authenticated to Kerberos, and, if applicable, which and when you asked for a specific TGS. Since we did not set up any service to use Kerberos yet, you should not see any entry, except the TGT:

```
# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: frank@FOOBAR.COM

Valid starting    Expires           Service principal
02/21/07 23:31:59  02/22/07 23:31:59  krbtgt/FOOBAR.COM@FOOBAR.COM
```

If one of the command failed, or you did not get what you were supposed to, please refer to Troubleshooting section.

If both commands succeeded, congratulations! Your Kerberos realm is now functionnal. Now, let's get to the last chapter of the article, where we will describe how to use Kerberos with other services, and use all the benefits of Single Sign On (SSO). If you need some hints for a migration of your passwords' database, see following section.

Migrating from an existing database

Traditionally, users' passwords are stored in a database (file, or binary), in a hashed version. It could be */etc/shadow* (for SystemV systems) or */etc/master.passwd* (for BSD-like systems), a plain text file (for Apache Basic auth), or even a MySQL/PostgreSQL/Oracle database. It all depends on your installation.

Technically, migrating a password database could be as easy as reversing the hashed version, and get the clear text password from it. Well, in our case, it almost certainly will not work, for obvious reasons. First, Unix system frequently add a salt to the password, so reversing it will not necesseraly give you the proper password back.

Plus, add to the fact that it will cost you more and more time if you are using strong hashing algorithms (sha-1 ou sha256), with complex passwords and thousands of users; this will become a daunting task.

The simplest solution would be to create Kerberos principals for users, and populate it with new random passwords. Things become easier, but you will have to get the clear text passwords, sooner or later, in order to update your Kerberos' database.

When getting the clear text passwords, some few reminders:

- never store them in a file that could be easily eavesdropped by unwelcomed people.
- always implement a secured connection when asking your user's to upgrade their passwords: SSL/TLS, or ssh (if they are familiar with unix or putty).
- check the Kerberos' domain part in your users principals, especially if your system is running in a multi-domain environnement.

Be sure to create the principal with the correct associated domain referrer, otherwise cross realm authentication will not work. For example: if user **foobar** is intended to be in **ex1.example.com**, and you are deploying a Kerberos' realm for **example.com**, be sure to create principal **foobar@EX1.EXAMPLE.COM** and not **foobar@EXAMPLE.COM**.

Anyway, there are still a few tricks you could use to upgrade properly your Kerberos' passwords database. Since there are many prerequesites and conditions (each system having its own set of peculiarities), your mileage may vary.

In this how-to, we suppose that you are at least using a system relying on PAM. Most services today use PAM, or provide a simple way to auth users through it.

To create principals, we need two things:

- the username (obviously)
- the clear text password (not encrypted nor hashed), which is the critical data of the process.

Using PAM

PAM (pluggable authentication modules) is commonly used on Unix for managing Authentication, Authorization and Accounting operations. It is a set of modules that defines how AAA is done.

Updating passwords is done through the pam_script module, which permits the execution of external scripts during PAM mechanisms.

To update the principal's password in KDC, we will use a principal that has rights to change password in database, but can not create nor delete principals. So, first, edit the ACL file and add a given principal (one that will not conflict with your soon-to-be users' principals). Let's call it **ch_passwd**:

```
kadm5.acl

 */admin@FOOBAR.COM                    *
 ch_passwd@FOOBAR.COM      ADMcIL     *@FOOBAR.COM
```

Short flag description: (consult *kadmind*'s man page for more details)

a/A	allow/disallow addition of principals policies on database
d/D	allow/disallow deletion of principals
m/M	allow/disallow modification of principals on database
c/C	allow/disallow password changing of principals
i/I	allow/disallow inquiries to the database
l/L	allow/disallow principals listing of database

These flags will only allow password change for all principals in the form of ***@FOOBAR.COM** (instance being not specified, it defaults to NULL, thus avoiding **ch_passwd@FOOBAR.COM** from changing ***/admin@FOOBAR.COM** passwords).

This ACL suppose that you have created all the principals in KDC database, from an existing list. This is highly recommended, it will avoid the creation of unwanted principals.

Create **ch_passwd** principal:

```
 # kadmin.local
 kadmin.local: ank ch_passwd
 WARNING: no policy specified for ch_passwd@FOOBAR.COM; defaulting to no policy
 Enter password for principal "ch_passwd@FOOBAR.COM":
 Re-enter password for principal "ch_passwd@FOOBAR.COM":
```

Secondly, we store **ch_passwd** credentials in a keytab: this way, we will avoid human interaction for authentication to Kerberos database administration server. Think of this keytab as a private key, so do not forget to set read/write access correctly:

```
 kadmin.local: ktadd -k /root/ch_passwd.keytab ch_passwd
 Entry for principal ch_passwd with kvno 2, encryption type Triple DES cbc mode with HMAC/sha1
added to keytab WRFILE:/root/ch_passwd.keytab.
 Entry for principal ch_passwd with kvno 2, encryption type DES cbc mode with CRC-32 added to keytab
WRFILE:/root/ch_passwd.keytab.
 kadmin.local: exit
```

Now, set the rights:

```
 # chown root:root /root/ch_passwd.keytab
 # chmod 400 /root/ch_passwd.keytab
```

Of course, you can use a simple user, not necessary root.

Now, we will install the pam_script module. It is a PAM module that allows to run shell scripts, while having access to the user's credentials. You can download the sources from http://freshmeat.net/projects/pam_script/.

Compile and install it:

```
# tar -xzf libpam-script-*.tar.gz
# cd libpam-script-*
# make
# cp pam_script.so /lib/security/
```

Then configure the PAM config file you want to use for password updating. Preferably, use the one associated to the service your users are traditionally using to authenticate themselves. Here, we use *login* (note that it may differ from your system):

```
/etc/pam.d/login

auth  sufficient  pam_krb5.so
auth  requisite   pam_unix.so
auth  [default=ignore]  pam_script.so runas=root expose=1 onauth=/root/krb5_passwd.sh
...
```

pam_unix.so is set to requisite: that way, pam_script won't be executed in case the user entered a wrong password (the auth stack will fail on pam_unix module). We avoid PAM from going the auth stack down further, and execute pam_script.so with a wrong password.

pam_script should not contribute to success nor failure of auth stack, so default attitude is ignore.

runas	set it to the user you want the script to run as
expose=1	expose PAM_AUTHTOK (the user password, required for the script)
onauth	execute the script passed as parameter

Please note that this PAM module may not work if it can not access the script during execution. For example, if you turn on privilege separation for OpenSSH and use PAM, chances are that the script will not be executed.

The pam_krb5.so line is not mandatory. Add it if you want to authenticate your user against Kerberos, in case their principal and correct password are correctly entered in KDC database. Anyway, setting it to sufficient avoid any kind of conflict in case Kerberos authentication failed for that user.

The script specified for pam_script is this one. It's a very simple shell script, which updates the password through *kadmin*:

```
/root/krb5_passwd.sh

 #!/bin/sh
 # Some constants
 # Principal with change_password rights
PRINC="ch_passwd@FOOBAR.COM"
 # Keytab of this principal
KEYTAB="/root/ch_passwd.keytab"
LOGFILE="/root/krb5_passwd.log"

if [ $# != "2" ]; then
        exit
else
        LOGIN=$1
        PASSWORD=$PAM_AUTHTOK
fi

DATE=`date`
printf "[$DATE] >> " $LOGFILE

 /usr/sbin/kadmin -p $PRINC -k -t $KEYTAB \
        -q "change_password -pw $2 $1" \
        >> $LOGFILE 2>&1
 # end of script
```

Feel free to test it, and adapt it to your likings (mail notification, password checking...).

Beware: in case the authentication is done via PAM remotely, the machine on which this module (and his keytab) are going to be installed has potentially full write access to users' password database. Something you definetely do not want if this machine is in an insecured place.

Using an HTTP authentication

If your installation uses the traditional HTTP authentication method, you could take advantage from it and create any php/perl/python script that could retrieve the password for you.

Using AuthType Basic and CGI will not work though, as Apache does not give access to the REMOTE_PASSWORD environment variable from a CGI script.

Take care of the security here, as any form of exploit triggered through your script could compromise your database. Also, SSL/TLS is recommended here.

Anyway, if you have no choice and must use a CGI, here's a quick and dirty way to get the traditionnal REMOTE_USER and REMOTE_PASSWORD variables, through Apache mod_rewrite.

First, edit your configuration file for Apache, and turn rewrite engine on. Store the Authentication string in a variable, preferably one that will not conflict with an already existing one:

```
httpd.conf

 <IfModule mod_rewrite.c>
            RewriteEngine on
            RewriteRule .* - [E=AUTH_STRING:%{HTTP:Authorization},L]
 </IfModule>
```

Just allow this rewriting rule only for the script that will update the Kerberos credentials: allowing such a rule to every CGI on your server will have security concerns. For example, you could use a location directive to restrict the RewriteRule to one file specifically:

```
httpd.conf

<Location /get_passwd.cgi>

        ...
        <IfModule mod_rewrite.c>
        RewriteEngine on
        RewriteRule .* - [E=AUTH_STRING:%{HTTP:Authorization},L]
        </IfModule>
        ...
</Location>
```

Next, the CGI script that will handle this variable. Pay attention, we will have to decode the string, as it is base64 encoded. Here is a sample php script to obtain username ($name) and password ($password) from a CGI:

```
get_passwd.cgi

#! /usr/bin/php-cgi
<?php
// headers
echo "Content-type: text/html\n\n";

// If we got the variables USER and PWD
if ( $_SERVER['REMOTE_USER'] && $_SERVER['PHP_AUTH_PW'] ) {
    $name = $_SERVER['REMOTE_USER'];
    $password = $_SERVER['PHP_AUTH_PW'];
} else {
// Well, we assume here that we did not get USER and PWD
// Attempt to read it from AUTH_STRING, thanks to mod_rewrite
    list($auth_type, $auth_string) = split("Basic ", $_SERVER['AUTH_STRING']);
    $auth_string = base64_decode($auth_string);
    list($name,$password) = split(":", $auth_string);
}
// Print name and password
echo "login: " . $name . "<br />password: " . $password;

?>
```

Third part - Using Services with Kerberos

Welcome to the last part of this article, where we deal with Kerberos' interaction with services requiring authentication.

Most services today require some kind of authentication. Traditionally, each of those services manage their own set of AAA (Authorization and Accounting) policies, either by using the one proposed by the local machine (shadow system for example, or nsswitch), or by specially crafted protocols (like RADIUS).

Of the three As, Kerberos was meant to solve all the troubles that goes with authentication. The protocol being standardized, it was designed to be as much application-independent as possible. It should avoid having multiples authentication systems, and multiple password databases.

The Kerberos system we will use in this part is the exact same as the one we configured previously. The differences are that we will not use it for users only, but for users and services.

General thoughts

As we explained before, Kerberos does not make any difference between a user (person) and a service. Both are considered principals. A user has a password, from which we generate a key, while a service has directly a service key (stored in a keytab).

The service's key is only known from the service and the KDC. It means that the keytab should be created with proper rights : only the service which requires it should be allowed to read the keytab. If not, any person (or service) having read access to this keytab could spoof the service's identity.

A keytab can contain more than one key. The file can be used for more than one service, and act as a shared resource.

Since a keytab contains the long term key of the service, if you change the keys regularly, be sure to update the keytabs accordingly. Also, a keytab contains the KVNO (the key version number), that increments each time the given key is changed). KVNOs must match on both ends (in keytab and in KDC database).

Keytab are maintained through the *ktutil* utility. For example, to "read" the content of our *ch_passwd.keytab* used before:

```
# ktutil
ktutil # rkt /root/ch_password.keytab
List its content:
ktutil # list
slot KVNO Principal
---- ---- ---------------------------------------------------------------------
   1    2              ch_passwd@FOOBAR.COM
   2    2              ch_passwd@FOOBAR.COM
```

Depending on your configuration, a keytab may contain more than one entry for a given principal. That is normal; if you use more than one encryption type, a keytab stores all the associated keys you wanted to export in it.

The principal used for each service is completely dependent on the service's configuration. It will slightly differ from a user's one, as a service does not use instances and is usually bound to one main machine (one hostname) in the DNS, whereas a user is not.

For that matter, the service's principal is commonly generated as follows:

```
service_name/fqdn@REALM
```

For example, consider we have an Apache server running on hostname **www.foobar.com**. Usually, the name associated to http service is **HTTP**. Then, Apache's principal will be **HTTP/www.foobar.com@FOOBAR.COM**.

Please note that the required FQDN is the main hostname stored in your DNS tree, and not one of its alias (CNAME). If you do not follow this rule, reverse-DNS mapping will not work, and Kerberos authentication will eventually fail.

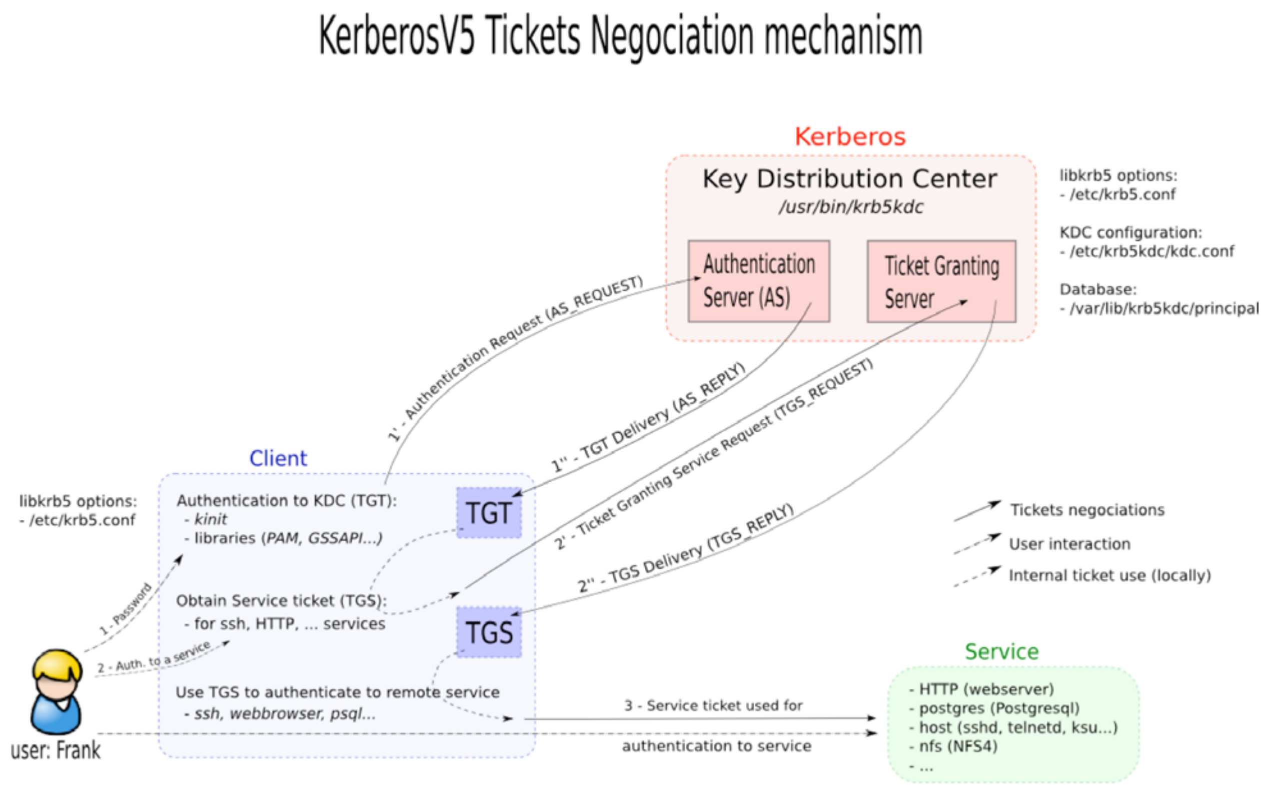
Clients will require Kerberos libraries to use Kerberos mechanisms. So, on each host where you will make use of Kerberos, you will have to edit the */etc/krb5.conf* file, and set the configuration properly.

For this part, we will use the same set of hosts, described in Second part - Client configuration. Their */etc/krb5.conf* files will look like this:

```
/etc/krb5.conf

[libdefaults]
	default_realm = FOOBAR.COM
	kdc_timesync = 0
	forwardable = true
	proxiable = true

[realms]
FOOBAR.COM = {
	kdc = kdc.foobar.com
	admin_server = kdc.foobar.com
}

[domain_realm]
	.foobar.com = FOOBAR.COM
	foobar.com = FOOBAR.COM

[login]
	krb4_convert = false
	krb4_get_tickets = false
```

It simply overrides some default values (like forwardable and proxiable options), and informs the hosts on subnet **foobar.com** to which KDC they should report to; **foobar.com** being mapped to realm **FOOBAR.COM**, hosts will make tickets requests to the appropriate KDC server, **kdc.foobar.com**.

Bear in mind the picture below, from first part of the article. We will use it constantly here.


KerberosV5 Tickets Negociation mechanism
Kerberos
Key Distribution Center
/usr/bin/krb5kdc
Authentication Server (AS)
Ticket Granting Server
libkrb5 options:
- /etc/krb5.conf
KDC configuration:
- /etc/krb5kdc/kdc.conf
Database:
- /var/lib/krb5kdc/principal
1' - Authentication Request (AS_REQUEST)
1'' - TGT Delivery (AS_REPLY)
2' - Ticket Granting Service Request (TGS_REQUEST)
2'' - TGS Delivery (TGS_REPLY)
Client
libkrb5 options:
- /etc/krb5.conf
Authentication to KDC (TGT):
- kinit
- libraries (PAM, GSSAPI...)
TGT
Obtain Service ticket (TGS):
- for ssh, HTTP, ... services
TGS
Use TGS to authenticate to remote service
- ssh, webbrowser, psql...
1 - Password
2 - Auth. to a service
user: Frank
3 - Service ticket used for
authentication to service
Tickets negociations
User interaction
Internal ticket use (locally)
Service
- HTTP (webserver)
- postgres (Postgresql)
- host (sshd, telnetd, ksu...)
- nfs (NFS4)
- ...

Introduction is over. Let's dive in.

Traditional host services

By *host* services, we mean services traditionally tied to Unix machine and accounts: *su*, *ftp*, *rsh* are examples.

For their kerberized counterparts to work (*ksu*, *kftp*, ...), each host should have their entry stored in KDC database, with a principal based on this model:

```
host/fqdn@REALM
```

For example, for host **kdc-client.foobar.com**, the principal will be:

```
host/kdc-client.foobar.com@FOOBAR.COM
```

That way, the host will be able to take part in a Kerberos tickets negotiation.

On the client's host side, the keytab's path is specified in */etc/krb5.conf*, with the default_keytab_name attribute. If not specified, it will default to */etc/krb5.keytab* (see *krb5.conf* manpage for details).

So, firstly, we will create the needed principal, **host/kdc-client.foobar.com**, on KDC:

```
 # kadmin.local
kadmin.local: ank -randkey host/kdc-client.foobar.com@FOOBAR.COM
```

The -randkey option ask to generate a random key for this principal. As we said before, a service has no password, so we use this option to provide a true random key. Now, export its keys by means of a keytab:

```
kadmin.local: ktadd -k /tmp/krb5.keytab host/toulouse.foobar.com
```

The *krb5.keytab* must be copied (with *scp* for example) to the path provided by default_keytab_name on **kdc-client.foobar.com** host. After being done, be sure to remove */tmp/krb5.keytab*, as it contains the key identifying kdc-client over your network.

Once the copied keytab on **kdc-client.foobar.com** is put in the right place, set its rights accordingly:

```
# chown root:root /etc/krb5.keytab
# chmod 400 /etc/krb5.keytab
```

Now, this host is ready to use any kerberized version of its utilities. For example, let's use *ksu*.

Consider that one of your user, named **frank**, with the principal **frank@FOOBAR.COM**, is allowed to *ksu* to root on host **kdc-client**. To allow him to do so, you will simply add his principal to */root/.k5login* on this host:

```
# echo "frank@FOOBAR.COM" >> /root/.k5login
```

Now, each time **frank** is logged on **kdc-client**, considering he is already authenticated to the KDC (and obtained his TGT), he will be able to switch to root account of **kdc-client**, without retyping or reentering its password, through *ksu*.

OpenSSH

For administration, it is quite common to bounce from one host to another, mainly for maintenance tasks. As a consequence, having to retype a full password each time you need to login can be quite cumbersome.

OpenSSH provides a mechanism to avoid typing in a password to authenticate. It is a challenge-based negotiation, built around asymmetric keys.

Basically, the user possesses his own private key (which can be password protected or not); upon a connection request, the ssh server, having the user's public key, encrypts a message with it (we call it a "challenge"), and sends it to the client.

Deciphering the challenge requires the private key. If the client managed to get the correct challenge back, the ssh server assumes that the client is the person he claims to be. If not, it means the client was not able to decipher the challenge, thus not having the private key.

This mechanism is quite common with asymmetric keys. SSL/TLS is based on it. However, it has some drawbacks:

- in this context, SSO is not possible. Providing SSO with asymmetric cryptography is only available through PKI (Public Key Infrastructures), which are heavier to maintain than a simple challenge-based mechanism.
- the credentials are not forwardable. That is, if **frank** connects from host A to host B through ssh and then wants to connect to host C, it will need to expose his private key on host B, or re-enter his password.

- from an administrator point of view, authentication tokens are not necessary stored in a proper manner. The repositery being not centralized, it can become troublesome to manage all the identities.

Properly configured, Kerberos manages to solve these problems.

For the continuation, we will assume that user **frank** (principal: **frank@FOOBAR.COM**) wishes to connect through SSH (version 2) to a host named **ssh-server.foobar.com**, from his workstation, **frank.foobar.com**.

OpenSSH provides a remote shell access, so the service name associated with it is *host* (see previous section for more explanations about host-based services).

As always, before configuring ssh, we will add it to our KDC. So, if you do not have already a host entry for this machine, connect to *kadmin* interface, and add a principal for it:

```
kadmin.local: ank -randkey host/ssh-server.foobar.com@FOOBAR.COM
```

and exports the keys to a keytab file:

```
kadmin.local: ktadd -k /tmp/ssh-server.keytab host/ssh-server.foobar.com
kadmin.local: exit
```

Copy the keytab file to **ssh-server.foobar.com**; for example, through ssh:

```
# scp /tmp/ssh-server.keytab root@ssh-server.foobar.com:/etc/krb5.keytab
```

The keytab should be put into the default path as specified by default_keytab_name in */etc/krb5.conf* on **ssh-server.foobar.com**. Set the rights and owners for it properly:

```
# chown root:root /etc/krb5.conf
# chmod 400 /etc/krb5.conf
```

We are done for the KDC part. Now we will proceed to the server (*sshd*) and client (*ssh*) configuration.

Server configuration, ssh-server.foobar.com

We supposed you want to take full advantage of SSO: ticket forwarding and negotiation. With OpenSSH, this is achieved thanks to the GSSAPI. There are two main options to enable to use Kerberos mechanisms: GSSAPIAuthentication and GSSAPIDelegateCredentials.

Edit */etc/ssh/sshd_config*, and add/uncomment those two options:

```
/etc/ssh/sshd_config

# Ticket negotiation, to allow seemless login through SSO (if possible)
GSSAPIAuthentication yes
# Ask credentials delegation (TGT forwarding)
GSSAPIDelegateCredentials yes
```

Now restart your sshd daemon (OS dependant):

```
# /etc/init.d/sshd restart
```

or

```
# /etc/rc.d/sshd restart
```

Client configuration, frank.foobar.com

The same goes here for the client, except that GSSAPIAuthentication and GSSAPIDelegateCredentials is not enabled by default. So, we will have to enable it in our ssh configuration.

Edit */etc/ssh/ssh_config*, and change the file accordingly. For example, we want to enable Kerberos mechanism for all Hosts:

```
/etc/ssh/ssh_config

Host *
        ....
        GSSAPIAuthentication yes
        GSSAPIDelegateCredentials yes
        ....
```

Save and close file.

These options can also be set directly on the command line, with -o flag:

```
# ssh -o GSSAPIAuthentication=yes -o GSSAPIDelegateCredentials=yes
```

Note that depending on your ssh setup, GSSAPI authentication will not necessarly comes up as first possibility during authentication process. Many default installations use a public key based authentication as first mechanism. See PreferredAuthentication in ssh_config(5) for details.

To test our new installation, open a new session with user **frank** on **frank.foobar.com**, and check that we have his TGT:

```
# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: frank@FOOBAR.COM

Valid starting     Expires            Service principal
07/14/07 12:41:12  07/14/07 20:01:41  krbtgt/FOOBAR.COM@FOOBAR.COM
        renew until 07/15/07 02:15:28
```

If you do not have a TGT for **frank**, *kinit* for it, and ask for a forwardable ticket (-f switch):

```
# kinit -f frank
```

Now we should be able to connect to **ssh-server.foobar.com** without having to reenter **frank**'s password:

```
# ssh frank@ssh-server.foobar.com
ssh-server ~ #
```

Once our session is opened, retype *klist* to see if the ticket was properly forwarded:

```
ssh-server ~ # klist
Ticket cache: FILE:/tmp/krb5cc_1000_uLicqi7389
Default principal: frank@FOOBAR.COM

Valid starting     Expires            Service principal
07/14/07 12:43:43  07/14/07 20:13:16  krbtgt/FOOBAR.COM@FOOBAR.COM
        renew until 07/15/07 02:13:12
```

Having a TGT on **ssh-server**, **frank** can now successfully use it to authenticate to another ssh server, provided it was configured with GSSAPIAuthentication enabled.

Some notes concerning ticket forwarding: to be forwardable, the TGT must have its Forwardable flag armed. In order to check it, simply use *klist* with the -f switch:

```
# klist –f
07/14/07 12:43:43  07/14/07 20:13:16  krbtgt/FOOBAR.COM@FOOBAR.COM
        renew until 07/15/07 02:13:12, Flags: FRIA
```

If F does not appear in the flags list, then this ticket is not forwardable, and will not be passed to ssh server upon connection. To specifically ask for a forwardable ticket, add the -f switch to *kinit* command, or change the default policy in */etc/krb5.conf* (set the forwardable value to true).

PAM

PAM, the "Pluggable Authentication Module", is a set of modules with very flexible rules that allow to fine tune the authentication (to some extent, authorization and accounting) on most recent Unix and Unix-like systems.

PAM consists of a set of files located in */etc/pam.d/*, typically one by service that will use it (*ssh*, *httpd*, *login*, ...), where the administrator could set the rules for accessing each of those services.

PAM and PAMified services are able to use Kerberos authentication, through the pam_krb5.so module. You will have to modify the service's PAM file in order to use PAM with Kerberos.

For example, for traditionnal *login*, just add pam_krb5.so to your list of modules:

```
/etc/pam.d/login

auth        sufficient    pam_krb5.so
auth        required      pam_unix.so try_first_pass

account     required      pam_krb5.so
account     required      pam_unix.so

password    required      pam_krb5.so
password    required      pam_unix.so

session     optional      pam_krb5.so
session     required      pam_unix.so
```

Now, the service *login* will use Kerberos as authentication possibility. The PAM module will take care of the negotiation and the creation of all tickets and credential caching necessary for SSO.

Note

Using Kerberos through pam_krb5 does not mean that your application is "kerberized". pam_krb5 is a way of checking the user credentials against a Kerberos KDC, and obtain tickets; no more, no less. It does not mean that your application will take full advantages of Kerberos (ticket negotiation, or secure password checking). Using telnet with pam_krb5 is not equivalent to a full kerberized *ktelnet* or *krsh*!

OpenLDAP

OpenLDAP is an open source implementation of the LDAP protocol. Since it supports GSSAPI, you can authenticate to an OpenLDAP using Kerberos (through a TGS).

We won't go into the details of setting up an OpenLDAP directory. That is not the goal of this article.

An LDAP DIT (Directory Information Tree) contains DN (distinguished names), one for each entry. In order to correctly map a DN with a particular Kerberos principal (LDAP and Kerberos are quite different on their naming conventions), we will have to specify the matching rules in *slapd.conf*, through the sasl-regexp option.

Firstly, we will have to check that your LDAP service supports the SASL GSSAPI mechanism, as it is required for Kerberos authentication. So, we request this information from root DSE (DSA Specific Entry):

```
# ldapsearch -x -s base -b "" + | grep GSSAPI
```

(do not forget the + , as we need to display the operational attributes).

If you have a line:

```
supportedSASLMechanisms: GSSAPI
```

Then your *slapd* service supports Kerberos authentication. If not, you have to reinstall your OpenLDAP and enable GSSAPI mechanisms.

For bind operations, the Kerberos principal will be reworked to obtain a certain dn syntax. This dn is only used for binding to the LDAP server, and indicate a special auth mechanism (in our case, GSSAPI). This is not the dn that is found in the LDAP directory.

The convention is as follows:

Kerberos principal	LDAP DN (distinguished name)
foobar@EXAMPLE.COM	uid=foobar,[cn=EXAMPLE.COM],cn=gssapi,cn=auth

Please note that if the principal contains an instance, like **foobar/admin** (**/admin** instance), the dn will be **uid=foobar/admin,cn=gssapi,cn=auth**.

The **cn** for Kerberos realm (**cn=EXAMPLE.COM**) is only needed if your *slapd* service and Kerberos KDC resides on different Kerberos realms.

Here, we have the dn used for bind operation to the OpenLDAP service. Now, we need to use the attributes used in this dn to craft some matching rule, in order to match this dn with a specific entry (object) in our DIT. For that matter, we edit the *slapd.conf* file, and use the sasl-regexp option.

This option is pretty straight-forward. We have a regexp used for attribute matching, and the information extracted from this regexp populates a LDAP request. All matching dn are then automatically bound to this Kerberos principal.

Example: suppose we are authenticated as principal **foobar**, and we wish to bind to the OpenLDAP directory, on the same Kerberos realm **EXAMPLE.COM**. Our Kerberos' principal is therefore: **foobar@EXAMPLE.COM**, and the DN with which we are bound to LDAP is **uid=foobar,cn=gssapi,cn=auth**.

Considering that my corresponding entry in the DIT is **uid=foobar,ou=users,dc=example,dc=com**, the sasl-regexp rule would be:

```
slapd.conf


 sasl-regexp
   uid=(.*),cn=gssapi,cn=auth
   ldap:///ou=users,dc=example,dc=com??sub?(&(uid=$1)(objectClass=inetOrgPerson))
```

The first matching rule will get the **foobar** string, and the LDAP request will look like:
ldap:///ou=users,dc=example,dc=com??sub?(&(uid=foobar)(objectClass=inetOrgPerson))

This request looks for an entry with **uid=foobar** and **objectClass=inetOrgPerson**, in the subtree of **ou=users,dc=example,dc=com**. If it finds one, it will be bound to the designated Kerberos principal.

Of course, you can specify more than one regexp, and make real complicated ones too. See OpenLDAP on-line documentation for details.

Now, after having modified your *slapd.conf* file, restart slapd daemon.

From now on, Kerberos principal **foobar@EXAMPLE.COM** should be authenticated as **dn: uid=foobar,ou=users,dc=example,dc=com**.

One way to check that your regexp are working properly is to used the LDAP "who am I" extended operation. With OpenLDAP, after being correctly authenticated to KDC, use the command line tool:

```
 # ldapwhoami
SASL/GSSAPI authentication started
SASL username: foobar@EXAMPLE.COM
SASL SSF: 56
SASL installing layers
dn:uid=foobar,ou=users,dc=example,dc=com
Result: Success (0)
```

(without any arguments, as we do not want to be authenticated through a simple authentication mechanism).

If the returned dn is the one you're waiting for (from an OpenLDAP point of view), then your sasl-regexp syntax is correct.

Apache

Apache, the web server, can be configured to use many kind of authentication mechanisms, and, to some extent, authorization (for an Intranet, for example).

Traditional identification goes through the use of a login and password, upon a HTTP 401 status response.

However, if the user is already logged in on an Intranet, he has probably entered his username and password upon login already. Reentering them upon accessing the web server is something easily avoidable with Kerberos.

For that matter, you will need to set up Apache properly (server side), and the web browser (client side).

Server side

A specific module has been developed to allow a very good collaboration between Apache's authentication process and Kerberos, namely mod_auth_krb5.

It offers many options. The good part of it is its retro compatibility system: if your client does not support Kerberos authentication (GSSAPI mechanisms), it will propose a classical authentication, where the user will be asked to enter his login and password in a dialog box.

Using *mod_auth_krb5* is preferred for Kerberos authentication, instead of PAM. The main reason being that mod_auth_krb5 sends a Negotiate header, that asks the client's browser to provide the TGS for authentication; whereas PAM will only use a traditional HTTP 401 dialog box.

Through Negotiation, the client's host will present its TGT to KDC and request a TGS for the **HTTP** service. Once the TGS is obtained, it will be cached on the client's host, and used for further authentication to HTTP server (in our case, Apache).

Before installing the module, we must add Apache service as a registered principal in KDC. The service is identified as **HTTP**, but you can change it at will. Be sure to adjust Kerberos module configuration accordingly.

Connect to administration shell (through *kadmin*), and create the **HTTP** service hosted on **webserver.foobar.com**. Since it is a service, make the password key random (with -randkey flag):

```
# kadmin.local
kadmin.local: ank -randkey HTTP/webserver.foobar.com@FOOBAR.COM
```

Please bear in mind that **webserver.foobar.com** is the FQDN of the host where the HTTP service is running. The returned FQDN must match the full hostname of the server, and not the address your users and clients will use to connect to this server (a typical example is when your webserver is found behind a router, or is used through NAT devices).

Now, export the keytab file to *apache_auth.keytab*, as Apache will need it during tickets negotiations:

```
kadmin.local: ktadd -t apache_auth.keytab HTTP/webserver.foobar.com@FOOBAR.COM
```

The keytab file must be readable by the process which uses it; in our case, Apache's user. So set the rights accordingly:

```
# chown root:apache apache_auth.keytab
# chmod 440 apache_auth.keytab
```

From now on, you have a keytab that permits Apache service (**HTTP**) on host **webserver.foobar.com** to authenticate directly to KDC.

We have service **HTTP** and its keytab registered. To use them with Apache, we install the mod_auth_kerb module. It can be found (as well as every documentation you need about it) directly onto the project's web page http://modauthkerb.sourceforge.net/.

To install it, you can either follow their guide, or use your favorite OS package management tools. The module is usually named mod_auth_kerb in the repository.

To use the module, load it in Apache config files. Depending on your Apache installation, you will have to add or uncomment a LoadModule directive:

```
httpd.conf

 LoadModule auth_kerb_module /usr/lib/apache2/modules/mod_auth_kerb.so
```

And then use the new AuthType to take advantage of it. For example, suppose that we want to authenticate users using Kerberos from */testkrb5* Location:

```
httpd.conf

 <Location /testkrb5>
        AuthName "Kerberos authentication on foobar.com domain"
        require valid-user
                # We use Kerberos module method
        AuthType Kerberos
                # The realm it corresponds to (see /etc/krb5.conf)
        KrbAuthRealms FOOBAR.COM
                # The service's name in KDC
        KrbServiceName HTTP
                # We want to use Negotiate method to use TGS from client
        KrbMethodNegotiate on
                # If the client does not support Negotiate, use 401 method instead
        KrbMethodK5Passwd on
                # cache credentials (in case the Negotiate method is not supported)
        KrbSaveCredentials on
                # Where the keytab is located
        Krb5Keytab "/etc/apache2/apache_auth_krb.keytab" </Location>
```

When finished with configuration, restart Apache to take modifications into account.

Now, depending on the Authentication mechanism used, tickets will be either cached on the client's host (if negotiation succeeded), or on webserver's host (thanks to KrbSaveCredentials option).

If the ticket is saved on the webserver, the path to it (in case you need the ticket for further processing) can be obtained from the $KRB5CCNAME environment variable. Note that the ticket is destroyed after the request has been served by Apache, for security reasons.

Note

Due to time resolutions issues (see bug 'Request is a replay' + Basic auth), it is strongly advised to set kdc_timesync value to 0 on the client (in our case, it is the web server).

If you do not do so, in some particular situations with lots of Basic Auth requests, the KDC will return an error, and authentication will eventually fail. Note that this will only happen in case the authentication is done through a Basic Auth mechanism, since in this case, Apache will negotiate a full TGT+TGS demand on each http request anew.

In order to check our setup, we now have to configure a client that will connect to Apache, and deal with Kerberos mechanisms.

Client side

In order to use the SSO mechanisms offered by Kerberos, both ends must support it. We just saw that Apache, through mod_auth_krb5, has full support of Kerberos, through the Negotiate header.

However, things do not look so easy on the client's side. You will have to deal with different systems, mostly Windows, MacOSX, or Linux.

Kerberos support is not that easy to determine on a host. Although Heimdal and MIT libs tends to reduce their discrepancies from day to day, and are commonly used on Unix and Unix compatible systems (*BSD, GNU/Linux, MacOSX), things are a lot messier when using Windows.

Many browsers support GSSAPI mechanisms on Unix. Firefox, Konqueror, and Safari are part of them.

Konqueror and Safari require the least configuration here: they will natively respond to Negotiate method. So, to use them with our Apache web server with Kerberos enabled, simply connect and go to the appropriate address (in our previous example, http://webserver.foobar.com/testkrb5).

Considering you had already obtain a TGT from KDC (through *kinit* and/or upon login), you should be directly authenticated by Apache. Note that it will cache the **HTTP** TGS upon Negotiation:

```
# klist
Valid starting     Expires            Service principal
04/16/07 18:28:56  05/16/07 06:28:56  krbtgt/FOOBAR.COM@FOOBAR.COM
04/16/07 18:29:13  05/16/07 06:29:13  HTTP/webserver.foobar.com@FOOBAR.COM
```

If not (it asks you to enter a login and password), there are many possibilities. The most probable ones are either you do not have a TGT, or your browser does not correctly handles the Negotiate request.

To check you have a TGT, simply use klist. If you get a line with a Service Principal of krbtgt:

```
# klist
...
Valid starting     Expires            Service principal
04/16/07 18:28:56  05/16/07 06:28:56  krbtgt/FOOBAR.COM@FOOBAR.COM
...
```

then you have a TGT. If not, simply use the *kinit* command to issue a TGT request.

Depending on your OS, the credentials are cached in specific locations. Please refer to the man page of *kinit* for more details.

If you still can not use Kerberos and you are sure that your browser support GSSAPI, the server logs will prove helpful, especially for the KDC and Apache. Most common errors include bad FQDN, and key version number (KVNO) mismatch when exporting HTTP keytab. See Troubleshooting section for help.

For Firefox, GSSAPI is also natively supported, but not enabled. To activate it, you should go to the config page. Type in "about:config" in link bar, press enter, and a series of variables will print into your browser's window. They all control Firefox environment.

We are interested in one in particular: network.negotiate-auth.trusted-uris.

This variable control the servers to which Firefox will positively answer to Negotiate method.

Browse (or search) for it, and double-click on its field.

The value is a list of comma-separated server addresses, with the according protocol for access (like http or https).

For our example, the server address is **webserver.foobar.com**, accessed through http. Then, we should enter this URL (without locations, as this is a server-wide parameter):

```
http://webserver.foobar.com
```

Firefox will only respond positively when this address is used. If you access **webserver.foobar.com** through another URL, like **http://webserver**, you should add it to the list in order to activate support for it:

```
http://webserver.foobar.com, http://webserver
```

Same goes for https and/or IPs. For example:

```
https://webserver.foobar.com, https://webserver, http://10.0.240.3
```

Now Firefox should answer properly to a Negotiate request for this server.

NFSv3 and 4

By default, NFS uses AUTH_SYS as authentication mechanism. Two hosts (the NFS client and NFS server) share the same set of UIDs, synchronized by different means (the most common way being yellow pages systems, like NIS).

When the host is sending out commands or requests (Remote Procedure Calls, or RPC), they are identified by the UID of the user issuing them on the client. They travel over the network until they reach the NFS server, where they are processed.

The RPCs are neither signed, nor mutually verified: the server has no way of knowing that the commands were properly issued by the legitimate user, or someone spoofing his identity. On critical systems, where security and confidentiality are primary objectives, this is not good.

One way to ensure that those RPCs are legitimate is to sign them with a key. This way, the server would be able to check that the command was issued by the right person, and was not tampered. Since Kerberos uses keys for authentication, we can use them to sign the RPC, thus avoiding all the disadvantages mentioned earlier.

Kerberos support for NFS depends heavily on what you want to do, and which tools are at you disposal on your OS. Indeed, they are not all equals in regards to Kerberos support with NFS. For that matter, we will try to remain as general as possible here, and indicate what are the requirements to use Kerberos mechanisms.

To check that Kerberos will work with your NFS system, you will have to rely on the documentation provided with your OS.

From the beginning, adding Kerberos functionality to NFS was not as tricky as it seems: it was already supported for NFS2. We add a more complicated security option to the already existing one, AUTH_SYS.

However, to support a wider set of mechanisms for authentication, Sun defined a new interface, the RPCSEC GSSAPI. Depending on your system, it is not necessarily fully implemented. For example, the latest version of *nfs-utils* for Linux 2.6 do not offer a krb5 security option. To add its support, developers and maintainers generally use patches (provided by the NFS4 project from CITI) to enable such options.

Since NFSv4 does include RPCSEC GSSAPI interface by default, you will be able to use Kerberos with it.

Downside is, NFSv4 is not as widely used as NFSv3 (in 2007, most BSDs and some Linux distributions do not enable NFSv4 by default). So if you use different operating systems on your network, you will probably have to rely on NFSv3.

As development of NFSv4 is an active progress, the documentation may evolve further after the writing of this tutorial. If you plan on using Kerberos with NFSv4, please refer directly to the source.

For a more in depth guide, please consult Kerberized NFSV4 Setup Tutorial, written by Aimé Le Rouzic.

From now on, we will concentrate on using NFS with Kerberos, whether it is NFSv3 or NFSv4.

Please note that installing and configuring NFS shares is beyond the scope of this article. We consider that you already have a working implementation, and just want to add Kerberos to it.

NFS Service configuration

We have three hosts:

- the KDC, **kdc.foobar.com**
- the NFS server (sharing directories), **nfs-server.foobar.com**
- the NFS client (mounting shares), **nfs-client.foobar.com**

Contrary to most services, NFS will require two principals: one for the NFS server, and one for the NFS client. Their respective keytabs will act as a mutual authentication between the two hosts upon mounting NFS shares.

First connect to KDC, and create the required principals:

```
# kadmin.local
Authenticating as principal root/admin@FOOBAR.COM with password.
kadmin.local:  ank -randkey nfs/nfs-server.foobar.com
WARNING: no policy specified for nfs/nfs-server.foobar.com @FOOBAR.COM; defaulting to no policy
Principal "nfs/nfs-server.foobar.com @FOOBAR.COM" created.
kadmin.local:  ank -randkey nfs/nfs-client.foobar.com
WARNING: no policy specified for nfs/nfs-client.foobar.com @FOOBAR.COM; defaulting to no policy
Principal "nfs/nfs-client.foobar.com @FOOBAR.COM" created.
```

Export their keytabs. The one containing the key for the NFS server (**nfs-server.foobar.com**) must be copied onto the NFS server, and the one containing the key for the NFS client (**nfs-client.foobar.com**) must be copied onto the NFS client.

```
 kadmin.local:  ktadd -k nfs-server.keytab nfs/nfs-server.foobar.com
 Entry for principal nfs/nfs-server.foobar.com with kvno 2, encryption type Triple DES cbc mode with HMAC/sha1
added to keytab WRFILE:nfs-server.keytab.
 Entry for principal nfs/nfs-server.foobar.com with kvno 2, encryption type DES cbc mode with CRC-32 added to
keytab WRFILE:nfs-server.keytab.
```

```
 kadmin.local:  ktadd -k nfs-client.keytab nfs/nfs-client.foobar.com
 Entry for principal nfs/nfs-client.foobar.com with kvno 2, encryption type Triple DES cbc mode with HMAC/sha1
added to keytab WRFILE:nfs-client.keytab.
 Entry for principal nfs/nfs-client.foobar.com with kvno 2, encryption type DES cbc mode with CRC-32 added to
keytab WRFILE:nfs-client.keytab.
```

Move each keytab to its respective host.

Note

For NFS server and client, the keytab path will default to */etc/krb5.keytab* (the exact same one as for the "host" service). So, if you already have a keytab containing "host" keys, you cannot simply replace the old keytab with the newly exported one, or you will erase the "host" entries. To merge two different keytabs, use *ktutil*, as described in the general Troubleshooting section.

We will now proceed to NFS server configuration.

NFS Server

After having successfully copied the nfs-server keytab to NFS server (**nfs-server.foobar.com**), connect to it, and set the proper owner and rights to */etc/krb5.keytab* (owner: root, rights: readonly for owner, none for others).

You should have, at least, some nfs service entries in */etc/krb5.keytab*, like those (use *ktutil* to read the keytab content):

```
# ktutil
ktutil: rkt /etc/krb5.keytab
ktutil: list
slot KVNO Principal
---- ---- ---------------------------------------------------------------------
   1    2      nfs/nfs-server.foobar.com@FOOBAR.COM
   2    2      nfs/nfs-server.foobar.com@FOOBAR.COM
...
```

Edit the */etc/exports* file, to add some extra shares, this time with Kerberos support. The entries have exactly the same options as traditional ones, except that for Kerberos, we do not specify machine resolution requirements. The string is replaced by a gss/krb5 entry.

Note that this rule may differ from one operating system from another. Please consult the *exports(5)* man page for details.

> **Note**
>
> Q - Wait! Do you mean that we cannot restrict NFS access based on IPs or subnets for GSS mechanisms?
>
> A - In essence, yes; however, note that a host requires a keytab file and an entry in the KDC for accessing NFS shares. If a host has no keytab (or its principal is revoked from database), he will not be able to mount Kerberos protected exports. Recent *nfs-utils* (starting from version 1.1.0 for Linux) do support machine name format for krb5 entries, but strictly speaking, that is not necessary; you can still use the old syntax.

There are three different ways to export NFS shares, with krb5 (by increasing security level):

- gss/krb5: only user identity is guaranteed.
- gss/krb5i: RPCs are signed. Avoid the RPCs to be altered after sending.
- gss/krb5p: RPCs are encrypted. They cannot be altered nor readable (except by client and server)

Beware: increasing security context will have performance issues, on both sides.

Here, we use the gss/krb5 method.

Our */etc/exports* will look like this:

```
/etc/exports

# /etc/exports: NFS file systems being exported.  See exports(5).
# Traditional exports, on 192.168.1.0 subnets
/exports                192.168.1.0/24(rw,root_squash,sync,subtree_check)
/exports/tmp            192.168.1.0/24(rw,root_squash,sync,subtree_check)

# Same exports, but with Kerberos enabled
/exports                gss/krb5(rw,root_squash,sync,subtree_check)
/exports/tmp            gss/krb5(rw,root_squash,sync,subtree_check)
```

Close file, and reexport directories:

```
# exportfs -r
```

Reinitialize all NFS daemons:

```
# /etc/rc.d/nfsd restart
or
# /etc/init.d/nfsd restart
```

Depending on your OS, you will have to start a daemon that will handle RPCSEC_GSS requests on the server. Usually, it is called *rpc.svcgssd*. So if it was not already started, do so:

```
#  rpc.svcgssd
```

Server configuration is done. Proceed to client configuration.

NFS Client

Like NFS server, move or merge the nfs-client keytab (**nfs-client.foobar.com**) to */etc/krb5.keytab*, and set the proper rights onto it (owner: root, rights: readonly for owner, none for others).

The NFS client configuration resembles the one you traditionally use to mount NFS directories, except that it requires an additional daemon for RPCSEC_GSS support, usually called *rpc.gssd* (the client counterpart of *rpc.svcgssd*). Start it:

```
# rpc.gssd
```

Now, you should be able to mount NFS shares with the krb5 security option. Test it:

```
# mkdir  /tmp/nfs_krb5_test
# mount -t nfs -o sec=krb5 nfs-server.foobar.com:/exports/tmp /tmp/nfs_krb5_test
```

If, for some reason, the mount operation hangs, please consult the NFS troubleshooting section below.

Since the NFS directory is mounted, you should connect to **kdc-client** as a normal user, and ask for a TGT. Upon accessing the NFS share, you will get the **nfs** service ticket:

```
# whoami
frank
# kinit Password for frank@FOOBAR.COM:
# cd /tmp/nfs_krb5_test
# klist
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: frank@FOOBAR.COM

Valid starting     Expires            Service principal
08/15/06 08:01:10  08/15/06 16:01:10  krbtgt/FOOBAR.COM@FOOBAR.COM
        renew until 08/16/06 08:01:08
08/15/06 08:01:30  08/15/06 16:01:10  nfs/nfs-server.foobar.com@FOOBAR.COM
        renew until 08/16/06 08:01:08
```

Troubleshooting kerberized NFS

Using Kerberos with NFS (or any other GSS mechanisms) is quite difficult, part of it being that using NFS with Kerberos is not that common (system administrators tend to rely heavily on samba shares).

Please note that CITI is having a full FAQ concerning NFSv4 and AUTH_GSS. Although it applies to NFSv4, most questions and answers do apply to NFSv3 too: http://www.citi.umich.edu/projects/nfsv4/linux/faq/

1 - Read the logs.

That may be a stupid advice, but in this case, they prove to be helpful. Often, forgetting to just start *portmap* may cause troubles. dmesg is also a good source of investigation. Even if the message you get is cryptic, searching for it or posting it on a mailing list will often get you some good explanations.

For the RPCSEC_GSS part (*rpc.gssd* and *rpc.svcgssd*), both daemon have the possibility to be launched in verbose and debug modes, and stay in foreground.

```
# rpc.gssd -vvv -r -f               # On NFS client
# rpc.svcgssd -vvv -r -f  # On NFS server
```

That way, they will print some valuable information when initializing GSS security contexts. Consult their man pages for more details.

2 - *kdestroy* does not destroy Kerberos NFS session under Linux!

Using kdestroy (as user) does not effectively destroy my Kerberos session for NFS. I can still access my files, even without having the TGS!

Well known issue, due to GSSAPI token caching in kernel.

3 - Using *kinit* as root on nfs-client does not grant me access on the share. Why?

Again, check the CITI FAQ.

4 - Since I use NFS with AUTH_GSS, I can not ssh to hosts with ssh using my private key anymore... What happened?

You are probably using ssh with RSA keys, and put the public key in your ~/.ssh/authorized_keys. However, since the share is not accessible until you get a TGS for it, **nfs-client** can not read this file and therefore, grant you access.

To solve this issue, you should connect to **nfs-client** and use GSSAPI delegation mechanisms, with:

```
ssh -o GSSAPIAuthentication=yes -o GSSAPIDelegateCredentials=yes username@workstation
```

5 - I kept getting "RPC: Couldn't create auth handle (flavor 390003)" on the client

That means that the kernel does not support secure RPC gss_krb5 calls. Either you did not compile in the support (under Linux), or the module associated to it is not loaded. To do so, try to execute:

```
# modprobe rpcsec_gss_krb5
```

Postgresql

PostgreSQL (pg) is one of the two most well-known open source DataBase Management System (DBMS), the other one being MySQL.

Installing and configuring a full Postgresql station is out of the scope of this article. We will concentrate on authentication to pg, and then focus exclusively on its GSSAPI -- Kerberos support.

pg can use many types of authentication (and authorization) mechanisms. These are all configured in the *data/pg_hba.conf* file, which should come with some useful informations into it.

Basically, your initial *pg_hba.conf* file looks similar to this:

```
pg_hba.conf

 # TYPE  DATABASE    USER       CIDR-ADDRESS         METHOD

 # "local" is for Unix domain socket connections only
 local   all        postgres                    ident sameuser
 local   all        all                    md5
 # IPv4 local connections:
 host    all        all        127.0.0.1/32         md5
 # IPv6 local connections:
 host    all        all        ::1/128              md5
```

pg maintains internally its own list of users. *pg_hba.conf* controls who can make what on the system, on a host basis.

From now on, we suppose that PostgreSQL is installed on a host called **pg-server.foobar.com**.

As always, we will first create the keytab file, necessary for pg to function properly with Kerberos. Log in to the KDC, and use *kadmin* to connect to the administration shell. The service's name for PostgreSQL is **postgres**:

```
 # kadmin.local
 # ank -randkey postgres/pg-server.foobar.com
 Principal "postgres/pg-server.foobar.com@FOOBAR.COM" created.
```

Export its keytab:

```
 # ktadd -k /tmp/postgresql.keytab  postgres/pg-server.foobar.com
 Entry for principal postgres/pg-server.foobar.com with kvno 3, encryption type Triple DES cbc mode with
HMAC/sha1 added to keytab WRFILE:/tmp/postgresql.keytab.
 Entry for principal postgres/pg-server.foobar.com with kvno 3, encryption type DES cbc mode with CRC-32 added
to keytab WRFILE:/tmp/postgresql.keytab.
```

Exit *kadmin* shell, and copy the file */tmp/postgresql.keytab* to host **pg-server.foobar.com**. On pg-server, save it in any location you want, provided it is accessible by pg.

```
 # scp /tmp/postgresql.keytab \
 pg-server.foobar.com:/var/lib/postgresql/data/postgresql.keytab
 # rm /tmp/postgresql.keytab
```

Here, we decided to put it in the same directory as postgresql config files. After copy is done, remove */tmp/postgresql.keytab* (for security reasons).

Since the keytab contains the long term key of **postgres** service, change its owner and rights accordingly. The keytab file must be readable by the owner under which postgresql is running, traditionally **pgsql**.

So, on host **pg-server.foobar.com**:

```
# chown postgres:postgres /var/lib/postgresql/data/postgresql.keytab
# chmod 400  /var/lib/postgresql/data/postgresql.keytab
```

We must configure postgreql so that it knows where to look to find the keytab. Edit */var/lib/postgresql/data/postgresql.conf*, and search for the value of krb_server_keyfile. Put in the path of the keytab:

```
postgresql.conf

 krb_server_keyfile = '/var/lib/postgresql/data/postgresql.keytab'
```

Save and close the file.

Now, we must instruct postgresql the conditions under which users must be identified through Kerberos. Here, it merely depends on what you want to do. For this guide, we consider that user **frank** must be identified through Kerberos, and can connect to any database from any host.

So we edit the file *pg_hba.conf*, and add the krb5 line accordingly:

```
pg_hba.conf

 # "local" is for Unix domain socket connections only
local   all     postgres                ident sameuser
local   all     all                     md5
 # IPv4 local connections:
host    all     frank    0.0.0.0/0      krb5
host    all     all     127.0.0.1/32        md5
 # IPv6 local connections:
host    all     all     ::1/128         md5
```

Note

1: *pg_hba.conf* uses a "first match win" ruleset. So, the line concerning **frank** must be put before the line matching all remaining users (the "all" parameter).

2: by default, postgresql is not configured to listen to TCP connections. To enable it, see the Connection Settings section in */var/lib/postgresql/data/postgresql.conf*.

Once finished, save and close the file and restart postgresql:

```
# pg_ctl -D /var/lib/postgresql/data restart
```

Now, postgresql should be able to use Kerberos mechanisms to authenticate user **frank**.

To check that krb5 authentication functions properly, get the TGT for **frank**, then proceed to connection to database with *psql*:

```
# kinit frank
Password for frank@FOOBAR.COM:
# psql -h pg-server.foobar.com template1 frank
# template1>
```

Normally, *psql* will not ask you for any password. *klist* will return the TGS cached for accessing postgres service:

```
# klist
Default principal: frank@FOOBAR.COM

Valid starting     Expires            Service principal
07/26/07 21:00:42  07/27/07 05:00:42  krbtgt/FOOBAR.COM@FOOBAR.COM
        renew until 07/27/07 21:00:41
07/26/07 22:53:01  07/27/07 05:00:42  postgres/pg-server.foobar.com@FOOBAR.COM
        renew until 07/27/07 21:00:41
```

Note

Kerberos does not provide any form of security mechanism for TCP connections. Kerberos assures only that the authentication process is trustworthy, and that the user is the one he claims to be. Nothing more.

Here, the remaining TCP connection to PostgreSQL might not be encrypted; you will have to use extra security standards (SSL/TLS is a good option there) to prevent any kind of Man In the Middle attack.

Servers' redundancy

Our whole guide is only built around one machine, supporting one KDC. Since Kerberos provides authentication to most services and users, it is a weak link in our network. A single point of failure on our KDC means no one will be able to receive tickets until KDC is back on line.

To avoid this, replication comes to mind. It is mainly used to copy the master's database to a slave KDC, but can serve other purposes, like saving.

Kerberos protocol does not specify replication between multiple hosts. Being not standardized, each implementation has its own ways of doing it.

The propagation is done through a secured encrypted channel, from master to slave ("push"). On the master, a snapshot of the database is done at a given time (the "dump"), and is sent to a service running on the slave, which updates its database.

Clients libraries support secondary servers requests. That is, if one server at any given time failed to respond to ticket delivering demands, clients will automatically try to contact another server (for the same realm), providing you entered their addresses when configuring the client's hosts (in */etc/krb5.conf*).

However, please note that all administrative tasks will remain unavailable until the master is back on line. Slaves KDC provide ticket delivering, but no *kadmin* interface (the Kerberos administration server, *kadmind*, should not run on the slaves' KDCs). Key creation, revocation and policy changes will not be possible when master KDC is down.

There are two ways of adding a Kerberos KDC to the client's list.

The simple way

In part 2 and 3, we made it directly by setting the kdc value to its FQDN, in */etc/krb5.conf*. For more than one server, we just specify multiple values for KDC, one per line:

```
/etc/krb5.conf

[realms]
FOOBAR.COM = {
kdc = kdc.foobar.com
kdc = kdc-slave1.foobar.com
kdc = kdc-slave2.foobar.com
....
admin_server = kdc.foobar.com
}
```

So, if you did not configure your clients earlier with the slaves kdc names, you will have to do it for each of your hosts.

The more technical way

You can register KDC servers right into DNS. Doing so will avoid the hassle of reconfiguring the clients if you add or remove one KDC in a specified realm. However, it supposes that you set up a DNS on your network (like named), and that you have the permission to modify it.

To use DNS records, we will have to use TXT and SRV.

TXT records is the realm associated to a zone of your DNS. It is not used by default; this is why we always specify default_realm value in */etc/krb5.conf*.

SRV records are a bit more complicated. They specify the service name, the service port type, and eventually the priority, weight, contact port and hostname. Please refer to RFC 2052 for more details.

For one KDC, the DNS for our foobar.com zone will look like this (ports are those defined by default for a Kerberos 5 implementation):

```
# Realm registration
_kerberos            IN      TXT     FOOBAR.COM

# port 88, for KDC, Kerberos 5 -- we add tcp and udp
_kerberos._udpIN     SRV   01 00 88 kdc.foobar.com.
_kerberos._tcp  IN   SRV   01 00 88 kdc.foobar.com.

# port 749, for kadmind. tcp only.
_kerberos-adm._tcp IN        SRV   01 00 749 kdc.foobar.com.

# port 464, used for kpasswd (kerberized passwd), on udp. Optional.
_kpasswd._udp        IN      SRV   01 00 464 kdc.foobar.com.
```

To register slave KDCs, add their respective SRV (remember, there should not be any kind of *kadmind* running on slave KDCs):

```
# port 88, for KDC slaves
_kerberos._udpIN        SRV    01 00 88 kdc-slave1.foobar.com.
_kerberos._tcp  IN        SRV    01 00 88 kdc-slave1.foobar.com. ...
```

Save this zone, and restart DNS service.

Now that we have added some slave servers, we should ensure that they receive a copy of the master's database. We will call this step the "Replication step", as we are merely transfering the Kerberos database content from one host to another.

Bear in mind that redundancy only works when you have both components: multiple servers and synchronized databases. We exposed how to have multiple servers; next, we are giving a solution to keep all servers' databases in sync.

Servers' replication

There are multiple ways of doing servers' replication. Strictly speaking, nothing prevents you from mirroring one server to many others. That is one possible solution, but not necessarily the best one.

This part describes how to configure *kprop*, a tool used to replicate a Kerberos database from one host to another. This process is called "Propagation".

Configuring the master

The KDC database must contain two "host" keys: one for the master (**kdc.foobar.com**), and one for the slave (**kdc-slave1.foobar.com**). They are used for mutual authentication when propagating the dump.

Connect to administrative interface on **kdc.foobar.com** and create those two keys:

```
kadmin.local: ank -randkey host/kdc.foobar.com
kadmin.local: ank -randkey host/kdc-slave1.foobar.com
```

Export the host entry of master KDC (default to */etc/krb5.keytab*):

```
# ktadd host/kdc.foobar.com
```

Set the right properly on krb5.keytab:

```
# chown root:root /etc/krb5.keytab
# chmod 400 /etc/krb5.keytab
```

Now, we will do the same, but this time for the slave KDC. Export the key for **host/kdc-slave1.foobar.com** to a keytab, and move the file (via scp) onto the slave KDC **kdc-slave1.foobar.com**:

```
# ktadd -k /tmp/krb5.keytab host/kdc-slave1.foobar.com
# scp /tmp/krb5.keytab root@kdc-slave1.foobar.com:/etc/krb5.conf
# rm /tmp/krb5.keytab
```

Configuring the slave

Note

By default, the propagation is done on the entire content of the master's database. That is, even special principals (like **K/M@FOOBAR.COM**) will be dumped and copied to the slave KDC. Pay attention there: it means that configuration files, as also specific files (like ACLs and stash file) must be copied to the slave host too.

Copying only a part of it will result in a bulky situation. If you forget to copy the stash file for example, the KDC daemon on the slave host will not be able to access the propagated database.

Before connecting to the slave, we will copy all minimum required files from the master for the slave system to work. Initially, it concerns (adapt the paths in accordance to your OS):

- krb5.conf (*/etc/krb5.conf*)
- kdc.conf (*/etc/krb5kdc/kdc.conf*, or found in krb5kdc local directory)
- kadm5.acl (*/etc/krb5kdc/kadm5.acl*, or found in krb5kdc local directory)
- stash file (*/etc/krb5kdc/stash*, or found in krb5kdc local directory)

Connect to the slave, **kdc-slave1.foobar.com**. Move the copied files into their appropriate directories (exactly like on the master KDC).

We will now initialize the slave database. Caution: we will use *kdb5_util*, but without exporting the stash file (-s argument), thus avoiding the obliteration of the one we just copied from master. When asking for the database Master Password, type in anything you want. The whole database will be erased upon the first propagation from master.

```
# kdb5_util create
Loading random data
Initializing database '/var/lib/krb5kdc/principal' for realm 'FOOBAR.COM',
master key name 'K/M@FOOBAR.COM'
You will be prompted for the database Master Password.
It is important that you NOT FORGET this password.
Enter KDC database master key:
Re-enter KDC database master key to verify:
```

The slave configuration is exactly the same as the master's one. Except that:

- it won't run an administrative instance of *kadmind*
- we must add a secondary file, namely *kpropd.acl*. It controls the principals from which the slave machine will allow Kerberos dump updates.

This file typically resides in */var/lib/krb5kdc/kpropd.acl* (the krb5kdc local directory).

Here, updates should only come from **kdc.foobar.com**. So create *kpropd.acl* accordingly:

```
echo "host/kdc.foobar.com@FOOBAR.COM" > /var/lib/krb5kdc/kpropd.acl
```

Do not start slave KDC. Since we still do not have a copy of the master's database, the stash file and the database master password of slave do not match.

Propagation

Connect to master KDC.

Before using *kprop*, we should get a dump of the master, and save the content to some file (in the following example, */root/slave-trans*):

```
# kdb5_util dump /root/slave-trans
```

We should now use *kprop* to propagate this dump to the slave.

```
# kprop -f /root/slave-trans kdc-slave1.foobar.com
Database propagation to kdc-slave1.foobar.com: SUCCEEDED
```

You can use a crontab to make this operation on a hour/daily basis. The dump can also be used as a save file. Once the operation succeeded, connect to slave and start its KDC.

Propagation failed?

If propagation failed with a loud "kprop: Connection refused in call to connect while opening connection", it means that *kprop* did not manage to contact *kpropd* on the remote slave KDC.

This will occur if you set restrictive access rules with a firewall, or if *kpropd* did not start upon connection.

The propagation is done through a tcp stream on port 754. Usually, *kpropd* is not a daemon running on its own: it is started by *inetd* (or its equivalent *xinetd*). However, many systems do not register *kpropd* as a service in their *inetd* database.

You can launch *kpropd* by two different means: either by starting it during boot up with the -S argument (see kpropd(8) for details), or register *kprop* as a potential services to *inetd*.

To register *kpropd*, it depends on whether your are using *inetd* or its more sophisticated equivalent *xinetd*.

First, edit */etc/services*, and look for *kprop* service; the line should look like this:

```
/etc/services

kprop       754/tcp
```

If you did not find it, please add it to the bottom of the file. Save and close.

inetd.conf

Now we should edit *inetd.conf* (see below for *xinetd*), and add this line:

```
/etc/inetd.conf

 kprop    stream    tcp  nowait    root /usr/sbin/kpropd   kpropd
```

Please note that the path to executable may vary from one system to another. Save and close *inetd.conf*, and restart *inetd*.

```
# /etc/rc.d/inetd restart
```

xinetd.conf

All config file for *xinetd* resides in the */etc/xinetd.d* directory. We must add the kprop config file, so that xinetd knows its existence:

```
# vi /etc/xinetd.d/kpropd
```

Create and edit the *kpropd* file:

```
/etc/xinetd.d/kpropd

service kprop
{
socket_type     = stream
wait            = no
user            = root
server          = /usr/sbin/kpropd
only_from  = 0.0.0.0 # Allow anybody to connect to it. Restrictions may apply here.
log_on_success  = PID HOST EXIT DURATION
log_on_failure = PID HOST
}
```

Save and close the file, and restart *xinetd*:

```
# /etc/init.d/xinetd restart
```

You should now be able to propagate the dumps from master **kdc.foobar.com** to slave **kdc-slave.foobar.com**.

Cross Realm Authentication

We managed to configure a KDC on a specific realm. But now, consider that our user **frank** belongs to **FOOBAR.COM**, and he wishes to authenticate to a service found on another realm, like **EXAMPLE.COM**.

Since **EXAMPLE.COM** has no way of deciphering the TGT from **frank** obtained from the KDC of **FOOBAR.COM**, how can the KDC in **EXAMPLE.COM** know that **frank** is the user he claims to be, as it does not have access to **frank**'s key in **FOOBAR.COM**?

Here comes cross realm authentication into play. It comes in three flavors with Kerberos 5: direct, hierarchical and non-hierarchical.

Direct cross realm authentication means that two realms, **REALM1** and **REALM2**, will establish directly their trust relationship, through key shares.

Hierarchical cross realm authentication is more DNS-like. Suppose we have two realms: **FOOBAR.COM** and **SUBNET.FOOBAR.COM**. Provided that KDCs controlling each realm share keys to ensure trust relationship, users from **FOOBAR.COM** will be able to use services offered by **SUBNET.FOOBAR.COM** (and reciprocally, if the relationship is bidirectional).

It resembles direct relationship in some way. The least common multiple being a network with different subnets placed right under it, whereas for a direct relationship, there is not necessarily a least common multiple between realms.

Non hierarchical cross realm authentication is a bit more complex. When the network is growing, and neither a direct nor hierarchical relationship can be established, we have to manually specify a "path" in */etc/krb5.conf* that will instruct the applications how they could obtain a particular service ticket for a distant realm.

A good example is the cohabitation of multiple KDCs, with some of them on realm **FOOBAR.COM** and others on **FOOBAR.NET**. Since there is no direct evidence on how to establish the relationships, it fits into the "non hierarchical" category.

Specifying individual keys in this case would be a painful task. Considering we must provide two keys (for "bidirectional" trust) for each possible combinations of realms, having 6 KDCs will theoretically require 6*5 = 30 keys.

Luckily, Kerberos 5 support transitive relationship. If **REALM1** trusts **REALM2** and **REALM2** trusts **REALM3**, then **REALM1** will "automatically" trust **REALM3**. Still, for complex setups, you will have to manually specify capaths directives (cross authentication paths). Consult *krb5.conf* man page to get explanations on how to write them.

For this how to, we will suppose that we have two realms: **REALM1** and **REALM2**. Having only two different realms in this example, the configuration process tends to be a direct cross realm relationship. But the commands for hierarchical and non-hierarchical are exactly the same (except that non-hierarchical relationship will require capaths directives in */etc/krb5.conf*).

Theory

Cross realm authentication is based solely on keys, shared between two master KDCs. Remember the *krbtgt* service we saw in Part 1 When we initialized the database **FOOBAR.COM**? *kdb5_util* created a principal **krbtgt/FOOBAR.COM@FOOBAR.COM**, based on this model:

```
service/instance@REALM
```

Since *krbtgt* is not a service which is "host" oriented but "realm" oriented, its instance part is a Kerberos realm (that is why instance is written in uppercase).

This principal is used to deliver TGTs for principal's found on realm **FOOBAR.COM**.

Now, we have our user, **frank@REALM1**, who wishes to access a service found in realm **REALM2**, like **host/server1.realm2@REALM2**.

Logically, frank should issue an authentication request to the KDC of **REALM2**, but in fact, he can not: **REALM2** does not store any principal related to **frank**. Since **frank** is not able to identify himself on **REALM2**, the KDC from this realm will not deliver him the precious TGS to access **host/server1.realm2**.

But **frank** is already identified to **REALM1**. So, he can make a request to his KDC, and ask him for a special TGT that could identify him to **REALM2**. Since **frank** will need to authenticate to an instance of **REALM2** from his own realm **REALM1**, **frank** will ask a ticket for a new service, this time for cross realm authentication. The principal for this service is:

```
krbtgt/REALM2@REALM1
```

Using this ticket, frank will now be able to present himself to **REALM2**, and get his TGS for **host/server1.realm2**.

> **Note**
>
> The cross realm ticket is not delivered by the Authentication Server, as **frank** is already identified. Exceptionally, it is the Ticket Granting Server that will serve the cross realm TGT.

This relationship is one-to-one: **REALM2** trusts principal's from **REALM1**, not the other way around. No user can get a cross realm TGT from **REALM2** to get a TGS from **REALM1**, as we do not have a:

```
krbtgt/REALM1@REALM2
```

at our disposal in **REALM2**.

For all that theory to work, we should now create those keys.

Configuration

> We said that cross realm authentication is based on keys. As you saw, the tickets that are firstly delivered are TGTs, and not TGS. Although the keys must be shared between two different KDCs, we will not export them in keytabs, but rather keep them in the databases; keytabs are mainly used for host based services.
>
> Since Kerberos protects the database with a Master Key Password (contained in the stash file), we can not simply export the keys through a dumpfile (with *kdb5_util*) and load it to the other KDC. As the Master Key Passwords are not identical (we are not in a replication situation!), exporting and importing will result in different keys on both hosts.
>
> To ensure that the keys do correspond correctly from one host to another, we will use a common password on both KDCs, and ensure that we are using the same algorithms and key salts, in order to obtain the same hashes when generating the **krbtgt** keys from the passwords. In the example below, we do not use key salts, to make things easier.

Choose two passwords, preferably different, and complicated ones. They will be used to generate the keys necessary for the bidirectional trust relationship. We will refer to them as password_one and password_two.

We are going to connect to *kadmin* interface for KDC on **REALM1**, and create two principals:

REALM1 trusts REALM2 principals: **krbtgt/REALM1@REALM2**, associated to <*password_one*>.
REALM2 trusts REALM1 principals: **krbtgt/REALM2@REALM1**, associated to <*password_two*>.

Connection:

```
 # kadmin.local
```

The following commands are just examples. You can choose whatever algorithms and key salts you want, just remember to use exactly the same on the other KDC in order to generate the same keys from the same passwords.

First principal:

```
 kadmin.local: ank -kvno 1 -e des3-hmac-sha1:normal -e aes256-cts:normal +requires_preauth
krbtgt/REALM1@REALM2
 WARNING: no policy specified for krbtgt/REALM1@REALM2; defaulting to no policy
 Enter password for principal "krbtgt/REALM1@REALM2": <enter password_one>
 Re-enter password for principal "krbtgt/REALM1@REALM2": <re-enter password_one>
 Principal "krbtgt/REALM1@REALM2" created.
```

Second principal:

```
 kadmin.local: ank -kvno 1 -e des3-hmac-sha1:normal -e aes256-cts:normal +requires_preauth
krbtgt/REALM2@REALM1
 WARNING: no policy specified for krbtgt/REALM2@REALM1; defaulting to no policy
 Enter password for principal "krbtgt/REALM2@REALM1": <enter password_two>
 Re-enter password for principal "krbtgt/REALM2@REALM1": <re-enter password_two>
 Principal "krbtgt/REALM2@REALM1" created.
```

Now, connect to the *kadmin* interface of **REALM2**, and enter the exact same commands.

Both principals are created, cross realm authentication may apply between those two realms. We must also configure */etc/krb5.conf* files, so that kerberized applications know which KDC to contact for each realm:

On **REALM1**:

```
/etc/krb5.conf

[libdefaults]
	default_realm = REALM1
	....

[realms]
REALM1 = {
	kdc = kdc.realm1
	admin_server = kdc.realm1
}
REALM2 = {
	kdc = kdc.realm2
}

[domain_realm]
	.realm1 = REALM1
	realm1 = REALM1
	.realm2 = REALM2
	realm2 = REALM2
...
```

On **REALM2**:

```
/etc/krb5.conf

[libdefaults]
        default_realm = REALM2
        ....

[realms]
REALM2 = {
        kdc = kdc.realm2
        admin_server = kdc.realm2
}
REALM1 = {
        kdc = kdc.realm1
}

[domain_realm]
       .realm2 = REALM2
        realm2 = REALM2
       .realm1 = REALM1
        realm1 = REALM1
...
```

If, for some reasons, cross realm authentication fail, check the logs.

"Deciphering errors" are mostly due to a wrong match between the shared keys (either the hashes do not correspond, or the KVNOs are not equal) between **REALM1** and **REALM2**.

Access refusal to Unix accounts (through *ksu*, or *ssh*) is often due to default_realm value */etc/krb5.conf*. Very often, the standard configuration only allow access to **REALM1** accounts for users from **REALM1** (quite logical, you would say). To grant access to **REALM1** accounts for **REALM2** users, you should edit the *.k5login* file accordingly. See *ksu* man pages for details about it.

Glossary

KDC—The Key Distribution Center(s):

Represents the main part of a Kerberos realm. It consists of three parts:

the Authentication Server (AS), responsible for authenticating users and issuing Ticket Granting Tickets (TGT)

the Ticket Granting Server, responsible for checking a users identity in respect to a service and issuing Ticket Granting Services (TGS)

a database, containing all the long term keys of all services and users of a Kerberos realm, including the administrators of the Kerberos database. This database also contains some more information associated to Kerberos accounts (creation dates, policies, expiration date, ...).

A KDC is usually found in a highly secured environnement, both physically and electronically. Only trusted employees should have physical access to this machine. The KDC is traditionally a machine dedicated solely to delivering Kerberos tickets, in order to minimize compromission through other services it could offer (mail, web server, etc.)

KDCs contains always a master for his own domain, and optionally, slaves, if needed.

Tickets

Some kind of cached information, crypted or not, on a client's machine. It contains all the information required for authentication. Frequently, it's stored as a file in a temporary environnement (/tmp on Unix), or directly in memory.

Ticket Granting Ticket - TGT

The Ticket Granting Ticket. It's the ticket delivered by the Authentication Server on successful identification, traditionally on first login or on request of a ticket renewal, and stored in memory or in a file.

The TGT is used for SSO purposes, and is sent back each time the client needs to authenticate himself to KDC.

Ticket Granting Service - TGS

The Ticket Granting Service. Not to be confused with Ticket Granting Server.

It's the ticket delivered by the Ticket Granting Server, on successful identification through a TGT. This ticket is used when contacting a kerberized service, and provides the service's daemon with all the requirements for authenticating the user.

Key table—keytab

A file containing the long term key associated with a service. Its content is a shared secret between the KDC (which stores service's key in its database) and the service itself. It works mainly in the same way as a password for an individual, albeit it is used by a service (ldap, ssh, apache, nfs...)

For security reasons, this file should only be readable by the service's daemon itself.

KVNO—Key version number

The key version number. It is a counter that increments each time a given key is changed in KDC database, for one principal. It is used mainly as a way to distinguish multiple keys associated with a specific principal, in case they are changed (password change, or key modification for service).

Generally, the KVNO starts at 0 when the principal is first created in the database, and is incremented by one each time the key changes, or is exported (in a keytab, for example).

Principal

This is a Kerberos term which is associated with an entity in the Kerberos realm. Basically, it is the equivalent of the username for a Unix account, but with a wider meaning: it can represent users, but also services or hosts.

A principal is separated into three main parts:

- a username or service name,
- an optional instance name (that allow different contexts for same user or service),
- realm name (a logical part of a network ruled by a set of KDCs).

```
username/instance@REALM
```

Username (or service name) and instance name are separted with a / , and the realm is separated from the first block with an @. By convention, the realm name corresponds to the DNS name in uppercase letters, to avoid confusion.

Although instances define a context and may be bound to a single user, from Kerberos point of view, both instances are treated independently: **frank@FOOBAR.COM** and **frank/admin@FOOBAR.COM** are two different principals.

Kerberized service

This is a program (slapd, sshd, nfsd...) or computer (host) which requires Kerberos as an authentication authority.

GSSAPI

The GSSAPI (Generic Security Services -- Application Program Interface) is an application programming interface used to wrap around security sevices functions, which tends to differ in their implementations.

Being an IETF standard, it is mainly used as a solution to ease the integration of different security systems in programs, providing a core set of functions that are implementation-independant, and hiding the inherent complexities and differences of different protocols to higher level applications.

Today, the dominant mechanism used through GSSAPI is Kerberos, as a way to overcome the technical difficulties by incompatible Kerberos APIs being not standardized. In essence, the GSSAPI is similar to Kerberos protocol: tickets are called tokens, and are used to create a mutual security context.

DES , 3DES, AES, ARCFOUR (RC4), ...

Those are the symmetric cryptography algorithms that are used by Kerberos to cipher and decipher communications. DES is the older of the three, and is based on 56 bits keys. DES is considered insecure today, and it is recommended to use the Triple DES algorithm (3 successive DES operations -- 112 to 168 bits strong), or even better, AES, which is 128, 192, 256 bits strong.

RC4 is the algorithm used by Microsoft own Kerberos implementation (Active Directory), and is commonly used by WEP. It is not recommended to use this algorithm today, as some of his inherent flaws lead to infamous WEP security problems. Microsoft chose it for retro compatibility reasons with NT.

Roughly (and abusively), to compare asymmetric and symmetric keys strength, a 128 bits symmetric key is equivalent to a 3072 bits RSA (asymmetric) key.

Troubleshooting

The troubleshooting guide has no intention of being a "once and for all" solution with Kerberos. Please bear in mind that Kerberos is an extremely complicated protocol, and troubleshooting it isn't easy.

You will find here a good share of the experiences of the author when he got problems with his Kerberos realm, and how he solved them.

Since such a guide could benefit from all the tips and tricks you could have, feel free to contribute to it. You could save quite a lot of time to others if you got some trivial/non trivial solution to someone's problem.

Anyway, if you didn't find anything useful in it, remember: Mails and mailing lists are useful, and google is still your best friend.

1 - I can't contact Kerberos KDC

First, check that krb5kdc is up and listening on ports 88 and 750. If yes, there may be quite a lot of problems:

- firewall
- Improper configuration

For firewall, I can't help you much. Be sure to grant access to:

- ports 88 and 750 (tcp/udp -- for KDC. 88 is for Kerberos 5, and 750 is Kerberos 4),
- port 749 (tcp -- for kadmind),
- (optionally) port 464 (udp, for kpasswd).

To check this, a netstat -anp might help you. Pay attention to lines containing krb5kdc and kadmind.

For improper configuration, double check your configuration files, server's and client's side. This includes all the krb5kdc/* files for server, and krb5.conf for client.

2 - "Can't resolve hostname"

Perhaps the mostly found and the mostly annoying error with Kerberos.

For authentication, Kerberos uses hostname. When you write your conf files, be sure to always indicate the Fully Qualified Domain Name (FQDN), and not a part of it, for server AND client. Bad habbits tend to associate only the last part of the FQDN in the /etc/hosts file, after IP address. That is not good. First, give IP address, then FQDN, then any alias you wish.

On the other hand, if you use a DNS, check that your client and server hostname resolve IP and FQDN correctly. To do this, use these commands:

- *host <IP>*, in order to get the registered name for <IP>. If it doesn't return you the good FQDN of your host, the DNS is misconfigured.
- *host <FQDN>*, which should give you the correct <IP> of your host back.

3 - Clock skew too great!

A very nice one from Kerberos too. Here, that is easy: client and server clocks aren't syncrhonized, and have to much skew between them. Kerberos needs synchronised clocks in order to thwart replay and avoid potential time attacks.

Solution: use NTP for synchronising clocks, or regularly check that the clocks on your machines have not to much skew (3 min max). You shouldn't deactive clock checking, unless you know what you are doing.

4 - It keeps telling me: "PREAUTH_REQUIRED" or "NEEDED_PREAUTH"

It should. Preauthentication is a check flag added with version 5 of Kerberos, which avoids off line brute force attacking on tickets (that could be possible in version 4). Preauthentication should be in default principal flags (see the server configuration part), unless you wan't compatibility with a Kerberos 4 server.

5 - Turning on Logs for Kerberos (libraries, administration and KDC)

For debugging or troubleshooting, turning on logging is quite helpful, especially if you can provide the logs when requesting help on mailing lists.

To turn logging on, add this section to */etc/krb5.conf* (adapt the file paths to your likings):

```
/etc/krb5.conf

 [logging]
        default = FILE:/var/log/krb5.log
        kdc = FILE:/var/log/krb5kdc.log
        admin_server = FILE:/var/log/kadmind.log
```

Be sure to turn it back off when finished, or add some sort of logrotation on those files (especially for the KDC). On busy networks with lots of authentication, the logs can become pretty heavy in just a few hours.

6 - KVNO mismatch (key version number mismatch)

Basically, each time you export a keytab for a service, the KVNO associated to each key is added to the file.

During service's access request, the service must compare the KVNO and key found in its keytab with the one used to generate the TGS. If there is a mismatch between one of the two, it means that the service's keytab is not in sync with the KDC database. Thus, authentication to the service is impossible, due to mismatch.

Solving this issue is easy: you have to update the service's keytab, by re-exporting it and replacing the old one. That way, both key and KVNO will match properly upon access request.

7 - Merging (or editing) a keytab file

Merging or editing keytabs is done through the *ktutil* utility. Suppose we have two keytabs, keytab1 and keytab2, each having their own set of keys, and we would like to merge the two keytabs in one (or create a new keytab containing specific keys). The operation is done through the *ktutil* shell, with *rkt* and *write_kt* commands, and optionally *delent* if you want to delete some entities.

Example:

```
# ktutil
```

Read content of keytab1:

```
ktutil: rkt keytab1
ktutil: list
slot KVNO Principal
---- ---- ---------------------------------------------------------------------
   1    3   <principal and key of keytab1>
   2    3   <principal and key of keytab1>
```

Now, we will read the content of keytab2:

```
ktutil: rkt keytab2
ktutil: list
slot KVNO Principal
---- ---- ---------------------------------------------------------------------
   1    3   <principal and key of keytab1>
   2    3   <principal and key of keytab1>
   3    2   <principal and key of keytab2>
   4    2   <principal and key of keytab2>
```

Save this content in a temporary keytab:

```
ktutil: write_kt /tmp/krb5.keytab
```

This utility is used to duplicate and tweak keytab entries (as its name implies), and remove the need of exporting the keys out of the KDC twice or more (simultaneously avoiding KVNO's increment).

Footnotes

1 - BSD like distributions use a different system. They store passwords and accounting information in the same file. While it does not strictly separate authentication from accounting, principles are exactly the same.

2 - There are other mechanisms, based on asymmetric cryptography, which can provide users' authentication: Public Key Infrastructure (PKI). As of today (in 2008), these systems are still difficult to use, and not as widely supported as Kerberos is. Note that Kerberos can also take advantage of public cryptography (for example, see Distributed Authentication in Kerberos Using Public Key Cryptography), but it is not that common.

References

1. R. M. Needham and M. D. Schroeder, *Using encryption for authentication in large networks of computers*, December 1978.
2. *Kerberos V5 System Administrator's Guide*
3. J. Garman, *Kerberos - The definitive guide*, O'Reilly, August 2003 - ISBN: 0-596-00403-6.
4. F. Ricciardi, *The Kerberos protocol and its implementations*, November 2006
5. K. Hornstein, *Frequently Asked Questions about Kerberos*, August 2000
6. A. Le Rouzic, *Kerberized NFSV4 Setup Tutorial*, June 2007