

# " KerberosVPN "



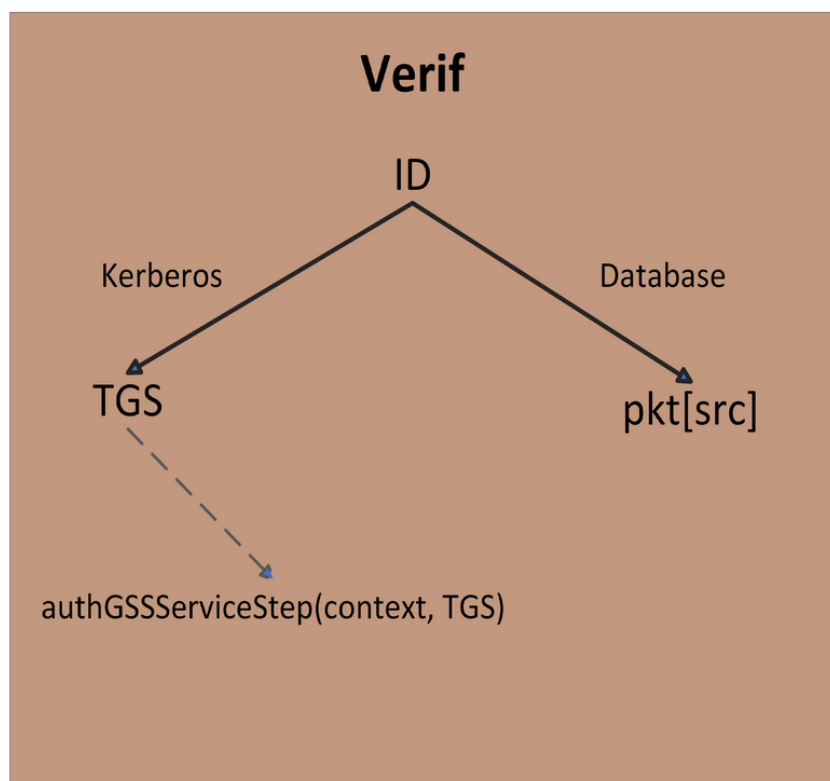
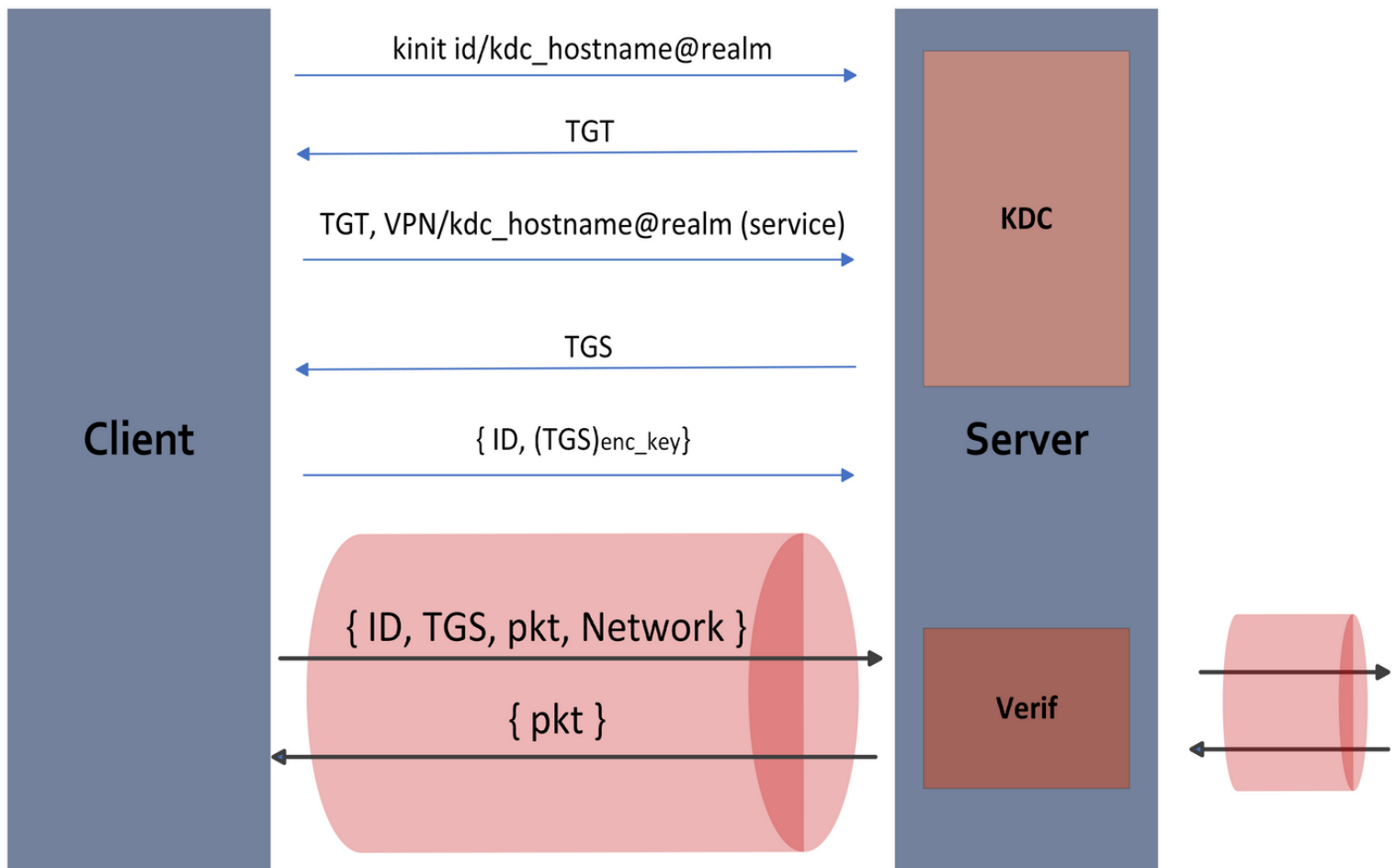
- Implementing a VPN solution using TUN/TAP interfaces with Kerberos authentication.

## Project done by :

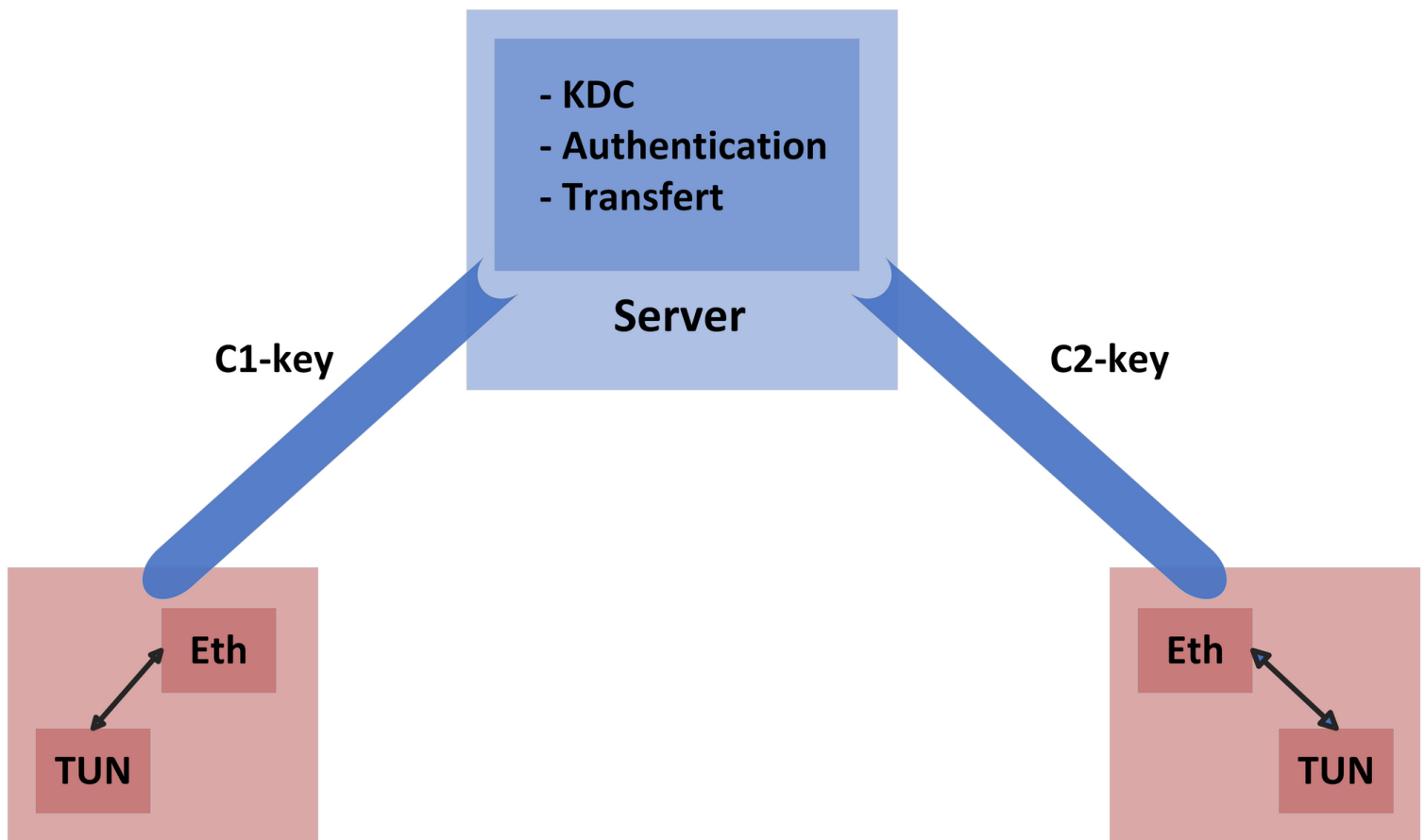
- Bouchhiwa Hassen
- Ben Jemaa Mouhib

# # Brief documentation #

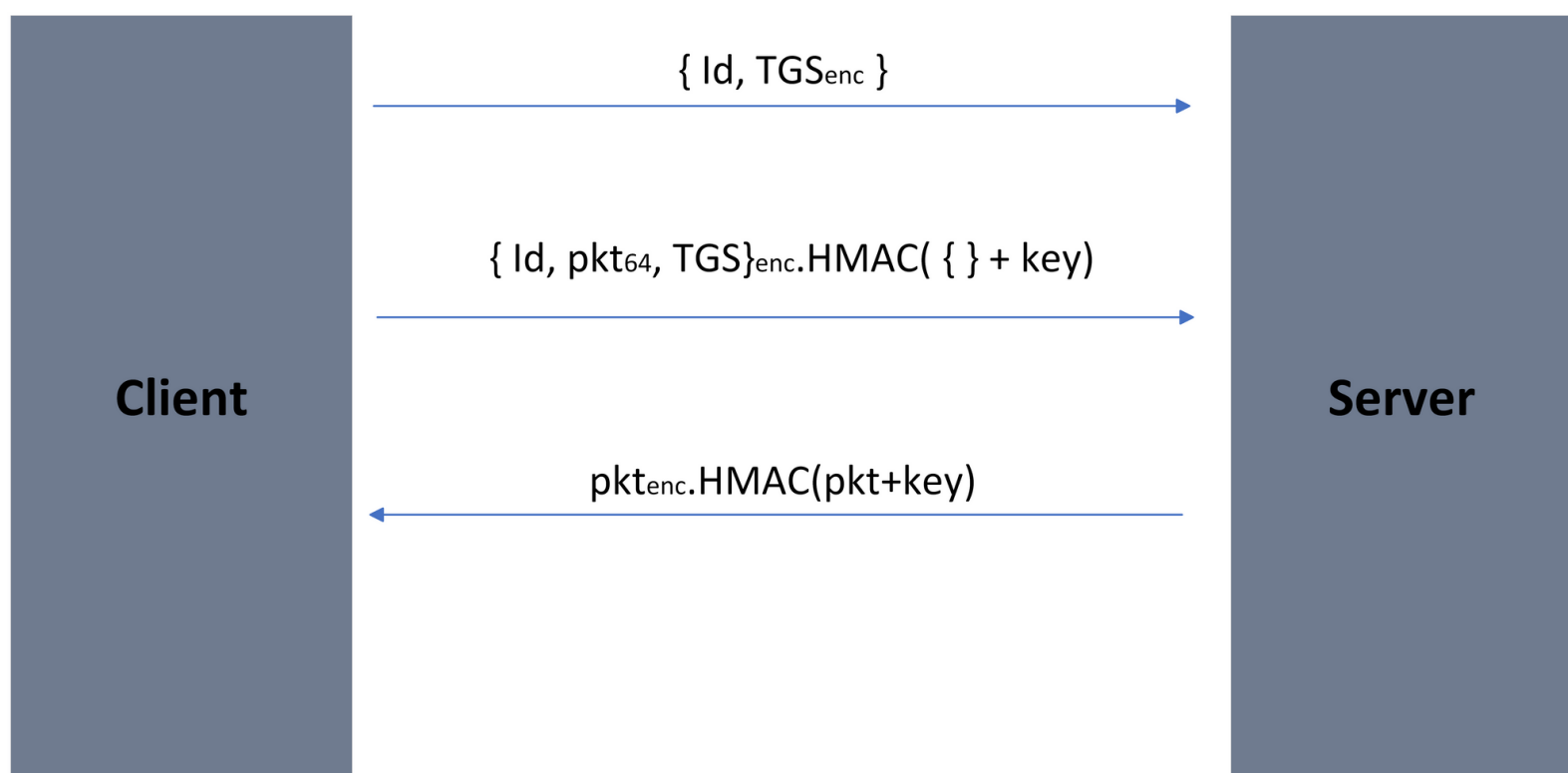
## The first phase of communication between client/server



## The second phase of communication between client/server



## Messages exchanged between server and client after establishing the socket



## - Requirements :

- Installing kerberos :



```
sudo apt install krb5-kdc krb5-admin-server krb5-config
```

- Installing kerberos python module



```
pip install kerberos
```

- Creating the realm



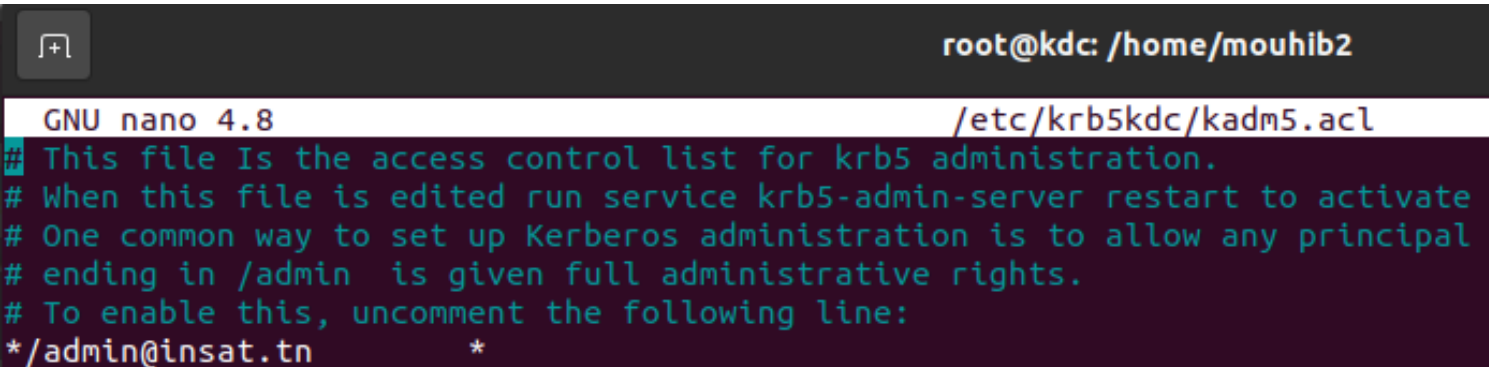
```
# To initiate a new realm :  
krb5_newrealm
```

The command above will create a database for your realm and will initiate a master key principal (K/M@INSAT.TN), you'll be asked to enter a password which will act as a KDC database master key.

```
# Access the ACL file that manages access rights to the Kerberos database
nano kadm5.acl
```

- Setting up the admin principal for your KDC :

```
# Uncomment the last line to be like this, to grant full access to your admin
# principal
*/admin@insat.tn *
```



```
root@kdc: /home/mouhib2
GNU nano 4.8 /etc/krb5kdc/kadm5.acl
## This file is the access control list for krb5 administration.
# When this file is edited run service krb5-admin-server restart to activate
# One common way to set up Kerberos administration is to allow any principal
# ending in /admin is given full administrative rights.
# To enable this, uncomment the following line:
*/admin@insat.tn *
```

- Creating the principals :
  - vpn/kdc.insat.tn@INSAT.TN (vpnpassword)
  - machine1/kdc.insat.tn@INSAT.TN (password1)
  - machine2/kdc.insat.tn@INSAT.TN (password2)

```

mouhib2@kdc:~/Desktop$ sudo kadmin.local
Authenticating as principal root/admin@INSAT.TN with password.
kadmin.local: addprinc machine1/kdc.insat.tn@INSAT.TN
WARNING: no policy specified for machine1/kdc.insat.tn@INSAT.TN; defaulting to no policy
Enter password for principal "machine1/kdc.insat.tn@INSAT.TN":
Re-enter password for principal "machine1/kdc.insat.tn@INSAT.TN":
Principal "machine1/kdc.insat.tn@INSAT.TN" created.
kadmin.local: addprinc machine2/kdc.insat.tn@INSAT.TN
WARNING: no policy specified for machine2/kdc.insat.tn@INSAT.TN; defaulting to no policy
Enter password for principal "machine2/kdc.insat.tn@INSAT.TN":
Re-enter password for principal "machine2/kdc.insat.tn@INSAT.TN":
Principal "machine2/kdc.insat.tn@INSAT.TN" created.
kadmin.local: addprinc vpn/kdc.insat.tn@INSAT.TN
WARNING: no policy specified for vpn/kdc.insat.tn@INSAT.TN; defaulting to no policy
Enter password for principal "vpn/kdc.insat.tn@INSAT.TN":
Re-enter password for principal "vpn/kdc.insat.tn@INSAT.TN":
Principal "vpn/kdc.insat.tn@INSAT.TN" created.
kadmin.local: get_principals

K/M@INSAT.TN

kadmin/admin@INSAT.TN
kadmin/changepw@INSAT.TN
kadmin/kdc.insat.tn@INSAT.TN
kiprop/kdc.insat.tn@INSAT.TN
krbtgt/INSAT.TN@INSAT.TN

machine1/kdc.insat.tn@INSAT.TN
machine2/kdc.insat.tn@INSAT.TN
root/admin@INSAT.TN
user/kdc.insat.tn@INSAT.TN

vpn/kdc.insat.tn@INSAT.TN

kadmin.local:

```

- Explaining the default principals that appear in the previous image:
  - **"K/M@INSAT.TN"** : Kerberos Master principal that is used by the KDC. Encrypts messages between KDCs and the communications that occur inside the same realm.
  - **"kadmin/kadmin@INSAT.TN"** : Kerberos Administration principal. Used for the management of Kerberos database and performing administrative tasks (adding and deleting principals). It must have all privileges to administer the Kerberos database.
  - **"kadmin/changepw@INSAT.TN"** : A principal capable of changing the password of every other principal that are stored in the Kerberos database. It's called by using the command 'kpasswd' to allow users to change their password.
  - **"kadmin/kdc.insat.tn@INSAT.TN"** : 'kadmin' command uses this principal to communicate with the KDC. It must have administrative privileges to perform administrative tasks (adding and deleting principals).
  - **"kprop/kdc.insat.tn@INSAT.TN"** : Used by 'krb5kdc' (which is the Kerberos database propagation daemon) to propagate any change in the Kerberos database between KDCs in a multi-realm environment.
  - **"krbtgt/INSAT.TN@INSAT.TN"** : This is the TGT - Ticket granting ticket principal that is used to authenticate users and grant them access to services.
  - **"root/admin@INSAT.TN"** : Used by the 'kadmin' command to authenticate users who have been granted administrative privileges. Its use is usually when the root user is performing actions as a root on the KDC machine.
- Checking the keytab file content :

```

root@kdc:/home/mouhib2# ktutil
ktutil: l
slot KVNO Principal
-----
ktutil: rkt /etc/krb5kdc/kadm5.keytab
ktutil: l
slot KVNO Principal
-----
ktutil: █

```

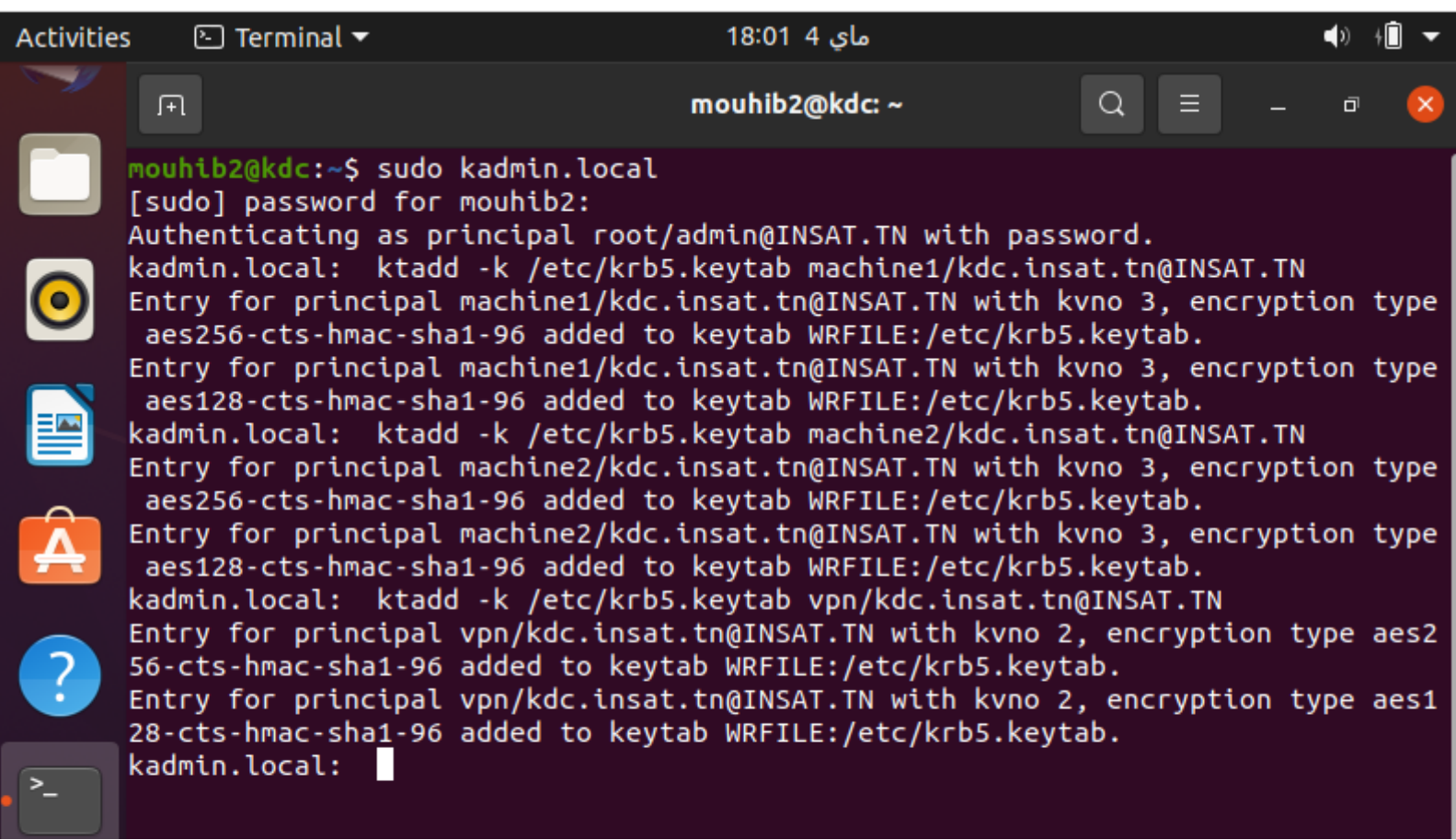
=> We have to add the newly created principles to the keytab, which stores secret keys for each principles that can be used by the root/admin principal to manage the authentication and the policies.

- That's why we need to check the encryption algorithm of our principals by getting information for each principal like this :

```
sudo kadmin.local get_principal <principal>
```

- Adding the principals to the keytab :

```
sudo kadmin.local get_principal <principal>  
kadmin.local : ktadd -k <krb5.keytabPATH> <principal>
```



The screenshot shows a terminal window titled "Terminal" with the user "mouhib2@kdc". The terminal output shows the execution of the following commands and their results:

```
mouhib2@kdc:~$ sudo kadmin.local  
[sudo] password for mouhib2:  
Authenticating as principal root/admin@INSAT.TN with password.  
kadmin.local: ktadd -k /etc/krb5.keytab machine1/kdc.insat.tn@INSAT.TN  
Entry for principal machine1/kdc.insat.tn@INSAT.TN with kvno 3, encryption type  
aes256-cts-hmac-sha1-96 added to keytab WRFILE:/etc/krb5.keytab.  
kadmin.local: ktadd -k /etc/krb5.keytab machine2/kdc.insat.tn@INSAT.TN  
Entry for principal machine2/kdc.insat.tn@INSAT.TN with kvno 3, encryption type  
aes256-cts-hmac-sha1-96 added to keytab WRFILE:/etc/krb5.keytab.  
kadmin.local: ktadd -k /etc/krb5.keytab vpn/kdc.insat.tn@INSAT.TN  
Entry for principal vpn/kdc.insat.tn@INSAT.TN with kvno 2, encryption type aes2  
56-cts-hmac-sha1-96 added to keytab WRFILE:/etc/krb5.keytab.  
kadmin.local: 
```



=> Note that this command "kadmin.local ktadd" is used for the purpose of setting up a simple architecture since it adds the principal to the keytab file and sets its keys automatically.

=> However, we could use the command "ktutil add\_entry" to manually add an entry to a keytab file where you can manually specify the encryption type or the key version (it provides more customization for your architecture and your principals).

- Adding the principals to the keytab using "ktutil add\_entry" :

```
sudo su
ktutil add_entry -password -p <principal> -k -1 -e <encr_type>
```

- Checking the keytab contents using ktutil :

```
# The keytab contains a pair of principles and their secret keys
sudo ktutil rkt /etc/krb5kdc/kadm5.keytab
```

- Checking if all configurations were made successfully on the KDC:

```
mouhib2@kdc: ~  
mouhib2@kdc:~$ sudo klist -kte /etc/krb5.keytab  
[sudo] password for mouhib2:  
Keytab name: FILE:/etc/krb5.keytab  
KVNO Timestamp Principal  
-----  
3 2023-05-04T18:00:34 machine1/kdc.insat.tn@INSAT.TN (aes128-cts-hmac-sha1-96)  
3 2023-05-04T18:00:38 machine2/kdc.insat.tn@INSAT.TN (aes256-cts-hmac-sha1-96)  
2 2023-05-04T18:00:49 vpn/kdc.insat.tn@INSAT.TN (aes256-cts-hmac-sha1-96)  
mouhib2@kdc:~$
```

=> Everything is good, we can see our three principals created in our keytab file with their respective encryption key.

- Checking the Kerberos config file :

```
# To navigate into the Kerberos config file  
nano /etc/krb5.conf
```

```
mouhib2@kdc: ~  
GNU nano 4.8 /etc/krb5.conf  
[libdefaults]  
    default_realm = INSAT.TN  
  
# The following krb5.conf variables are only for MIT Kerberos.  
    kdc_timesync = 1  
    ccache_type = 4  
    forwardable = true  
    proxiable = true  
    clockskew = 3600  
  
# The following encryption type specification will be used by MIT Kerberos  
# if uncommented. In general, the defaults in the MIT Kerberos code are  
# correct and overriding these specifications only serves to disable new  
# encryption types as they are added, creating interoperability problems.  
#  
# The only time when you might need to uncomment these lines and change  
# the enctypees is if you have local software that will break on ticket  
# caches containing ticket encryption types it doesn't know about (such as  
# old versions of Sun Java).  
  
#    default_tgs_enctypes = des3-hmac-sha1  
#    default_tkt_enctypes = des3-hmac-sha1  
#    permitted_enctypes = des3-hmac-sha1  
  
# The following libdefaults parameters are only for Heimdal Kerberos.  
    fcc-mit-ticketflags = true
```

[ Read 103 lines ]

^G Get Help	^O Write Out	^W Where Is	^K Cut Text	^J Justify	^C Cur Pos
^X Exit	^R Read File	^_ Replace	^U Paste Text	^T To Spell	^_ Go To Lin

- Explaining the variables that appear in the previous image:
  - "**kdc\_timesync = 1**" : This parameter is used to tell the Kerberos client that it needs to synchronize its clock with the Kerberos KDC before requesting a ticket (KDC uses timestamps to prevent replay attacks).
  - "**ccache\_type = 4**" : To define the type of credential cache to use, in our case we'll store the credential cache on memory (it could be stored on disk).
  - "**forwardable = true**" : We specify that the ticket should be forwardable which means the ticket can be forwarded to another machine (it is useful if a user wants to access resources on a remote server).
  - "**proxiable = true**" : We define that the ticket should be proxiable which means that the ticket can be passed to a proxy server (It can be useful if the user wants to access resources on a remote server that is behind a firewall)
  - "**clockstew = 3600**" : We specify that the maximum allowed clockstew between the client and KDC is 3600 seconds (1 hour, if the client's clock is more than this value then the authentication will fail).
  - "**fcc-mit-ticketflags = true**" : We specify that the Kerberos Ticket Flags should be stored in the MIT Kerberos File-Based Credential Cache (FCC). These flags are a set of attributes that describe the properties of a Kerberos Ticket.

- Another way of checking the Kerberos config file :



```
# Checking if you defined the correct realm  
cat /etc/krb5.conf | egrep 'INSAT' --color
```



```
# Checking if you defined the correct DNS (for the admin server and KDC)  
cat /etc/krb5.conf | egrep 'insat' --color
```

```
root@kdc:/home/mouhib2# cat /etc/krb5.conf | egrep 'INSAT' --color  
    default_realm = INSAT.TN  
    INSAT.TN = {  
root@kdc:/home/mouhib2# cat /etc/krb5.conf | egrep 'insat' --color  
        kdc = kdc.insat.tn  
        admin_server = kdc.insat.tn  
root@kdc:/home/mouhib2#
```

- Brief explanation on the functions we used from the Kerberospython module :
  - **!- Note** : All these functions are using the GSSAPI interface - Generic Security Service Application Programming Interface which is a security protocol ensuring the encrypted secure exchange of messages between a client and a server application over the network. It can be used with any authentication protocol, in our case the Kerberos python module is using it in its predefined functions.
  - **"authGSSClientInit(service,principal)"** : Initiates a client context for the GSSAPI client-side authentication.
  - **"authGSSServerInit(service)"** : Initiates a server context for the GSSAPI server-side authentication.
  - **"authGSSClientStep(client\_context, " ")"** : Send the initiated client\_context with an empty string to indicate that it's the first step in the authentication process (behaves like a handshake). This function is generally used to do another stage in the authentication process.
  - **"authGSSClientResponse(client\_context)"** : If the client-side GSSAPI step was successful, it will return the TGS corresponding to the client that wants to access a specific service.
  - **"authGSSServerStep(server\_context, response)"** : Send the initiated server\_context with the response generated by the client which represents the TGS to verify if it's correct or not (verify if it matches the principal, the timestamp, the lifetime, the service name). This function is also generally used to do another stage in the authentication process.
  - **!- Note** : The client context represent the security state of the client (for example, the credentials used to authenticate the client to the server) while the server context is a variable that represents the security state of the server (it can also include the service principal for which the client is requesting access). Both contexts are used to store the state of the authentication between both parties. We're using the authGSSServerStep and authGSSClientStep to perform another stage of the authentication process and with each stage we need the previous context so that we can update the contexts with information provided in the exchanged messages. The context can be destroyed after the authentication process is finished so that the client and the server can start communicating securely.
  - **"authGSSServerUsername(server\_context)"** : This function will return the username of the principal trying to authenticate to the server. We call this only after the step function returns a complete or continue response code.
  - **"AUTH\_GSS\_COMPLETE"** : A flag used in the GSSAPI interface and can be returned from the authGSSServerStep to indicate that the authentication was successful.