[Th 6] user authentication

User authentication is composed of two phases:

- in the first one, we perform an **identification**, in which the user uses an identification that is public **(like a username)**
- in the second phase, we have the **verification**, where the user needs to prove that is the owner of the identification and this is done **via password, PIN, or other techniques like biometric information.**

The techniques used to perform the authentication can be something that the user:

- knows (passwords, PIN, an answer to a question)
- has (token, smart card, physical device)
- is (static biometrics like the retina, face, etc)
- does (dynamic biometrics like handwriting, typing rhythm)

Authentication systems based on passwords

A password is a secret exchanged between the user and the server.

1. **The server initially stores the username and the password**
2. **The user submits a username and password to the server**
3. **If they match, the user is authenticated**

ATTACKS TO PASSWORDS

The first possible attack is the one targeted to a specific account.
One possible mitigation can be throttling, a rate limit on attempts.

Another attack can be done using well-known passwords, but it can be solved using a policy over the password choice for the users.

Then we have the password guessing on a single username according to the retrieved information about the user.

The last one is computer hijacking where an attacker takes control of the victim's computer.
This can be solved using an auto-login mechanism.

VULNERABILITY

The vulnerability related to passwords can be of different kinds.
It can be related to user mistakes (sharing passwords with friends, writing them on Post-it, etc), user usage of the same passwords for different systems or accounts, or to electronic monitoring (for example the attacker can intercept passwords sent across the network).

PASSWORD STRENGTH

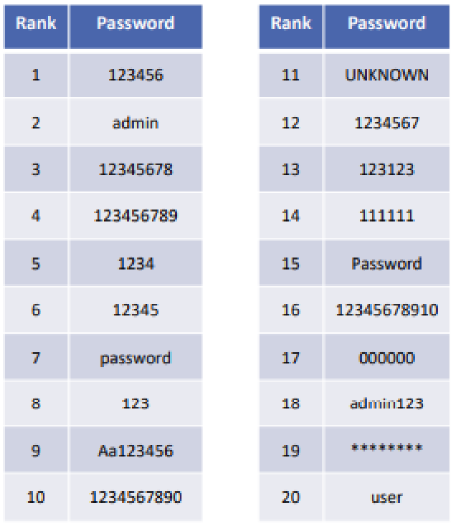
The strength of a password is measured using a metric that is called entropy.
This measure is used to understand which is the chaos of a password and then how much it is difficult to predict it.

For example, the bits needed to represent symbols of a typical set are (calculated with the log of base2):

- 3.32 for representing digits from 0 to 9
- 4.70 for lowercase English letters
- and so on

The very dangerous problem is that human-generated passwords are not very random, in fact, the top 5k passwords are used by 20% of users in the world (2016), or in general most of them are in dictionaries.

Source: NordPass, 2023



Rank	Password
1	123456
2	admin
3	12345678
4	123456789
5	1234
6	12345
7	password
8	123
9	Aa123456
10	1234567890
11	UNKNOWN
12	1234567
13	123123
14	111111
15	Password
16	12345678910
17	000000
18	admin123
19	********
20	user

Mitigation approaches

Mitigation approaches could be:

- user education on the importance of having a strong password
- use computer-generated passwords that are random or easy to pronounce
- proactive password checking when the user is choosing the password

BAD MITIGATION APPROACHES

A bad mitigation approach is to insert constraints on the password content and not on the password length, it is proved that this approach doesn't increase in a suitable way the password entropy but makes them difficult to remember.

The most useful thing is to focus on the verifier. The constraint that makes sense is the one on the password length.

In addition, we need to verify that the password is not present on the list of passwords leaked.

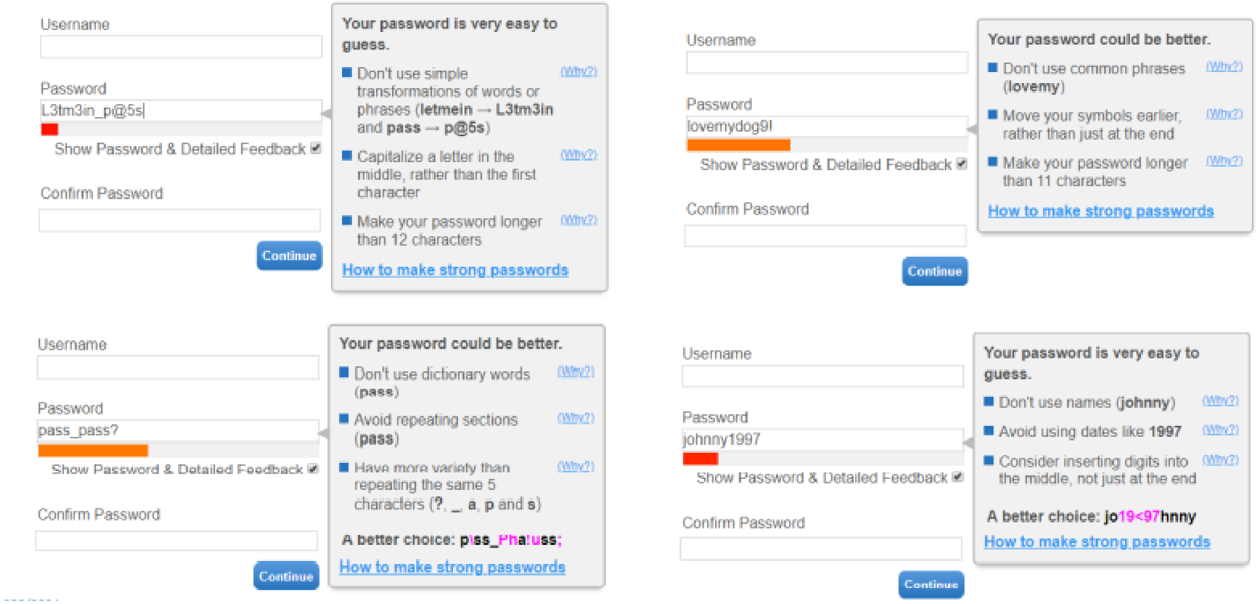
- we can use John the Ripper to do this

A good thing could be to include all UNICODE chars and emoji in order to make passwords easy to remember but difficult to guess.

DATA-DRIVEN PASSWORD METER

Data-driven password meter is a technique in which we use different heuristics (21) that verify how much a password is guessable giving a security score.

When a constraint is not respected then it will give a hint to the user.



Username

Password
L3tm3in_p@5s|

Show Password & Detailed Feedback ☑

Confirm Password

Your password is very easy to guess.

- Don't use simple transformations of words or phrases (**letmein** → **L3tm3in** and **pass** → **p@5s**) (Why?)
- Capitalize a letter in the middle, rather than the first character (Why?)
- Make your password longer than 12 characters (Why?)

How to make strong passwords

Username

Password
lovemydog9!

Show Password & Detailed Feedback ☑

Confirm Password

Your password could be better.

- Don't use common phrases (**lovemy**) (Why?)
- Move your symbols earlier, rather than just at the end (Why?)
- Make your password longer than 11 characters (Why?)

How to make strong passwords

Username

Password
pass_pass?

Show Password & Detailed Feedback ☑

Confirm Password

Your password could be better.

- Don't use dictionary words (**pass**) (Why?)
- Avoid repeating sections (**pass**) (Why?)
- Have more variety than repeating the same 5 characters (**?**, **_**, **a**, **p** and **s**) (Why?)

A better choice: p!ss_Pha!uss;

How to make strong passwords

Username

Password
johnny1997

Show Password & Detailed Feedback ☑

Confirm Password

Your password is very easy to guess.

- Don't use names (**johnny**) (Why?)
- Avoid using dates like **1997** (Why?)
- Consider inserting digits into the middle, not just at the end (Why?)

A better choice: jo19<97hnny

How to make strong passwords

PASSWORD STORING

CLEAR PASSWORD STORING

One approach is to store passwords in clear, but this opens the doors to various attacks:

- insider attacks, a normal user sees the database and now has the passwords of all users
    - mitigation can be the access control on the password database
- insider attacks, the admin sees the database and now has the passwords of all users
    - there is no mitigation
- outsider attacks, an attacker obtains access to the DB and has all user's passwords
    - there is no mitigation

ENCRYPTED PASSWORD STORING

Another approach is to store encrypted passwords with a server private key. The problem is that the server's private key must be stored and if an attacker can steal it there is a serious problem.

HASH VALUES STORING

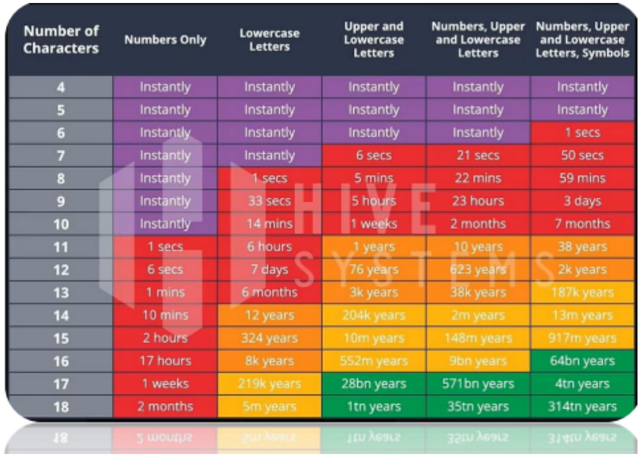
Another approach, the really used one, is to store the hashed value of passwords.

In this case, we have a security that is related to the hashing algorithm because is not possible to revert a hash value to the original plain text.
In this case, it is always possible to use the brute force approach to try a very large number of passwords, for this reason sometimes the hashing is performed recursively on the plain text.

There are also rainbow tables, which are data structures that contain precalculated hash values for known passwords.

How much time is required to find the hash value of a password?

- RTX 4090 GPU



Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	Instantly	1 secs
7	Instantly	Instantly	6 secs	21 secs	50 secs
8	Instantly	1 secs	5 mins	22 mins	59 mins
9	Instantly	33 secs	5 hours	23 hours	3 days
10	Instantly	14 mins	1 weeks	2 months	7 months
11	1 secs	6 hours	1 years	10 years	38 years
12	6 secs	7 days	76 years	623 years	2k years
13	1 mins	6 months	3k years	38k years	187k years
14	10 mins	12 years	204k years	2m years	13m years
15	2 hours	324 years	10m years	148m years	917m years
16	17 hours	8k years	552m years	9bn years	64bn years
17	1 weeks	219k years	28bn years	571bn years	4tn years
18	2 months	5m years	1tn years	35tn years	314tn years

SALTED HASH VALUES STORING

A better approach is to store the salted hash value of a password.
The idea is to store on the database (also in clear is ok) a random value for each user, which is called salt.

The approach here is to store on the database the hash value of the concatenation between password and salt:

- **hash(password || salt)**

This beats at the possibility of using rainbow tables and precalculated hash values because the same password with different salt has different hash values.

2FA authentication

The 2FA authentication is a technique in which we combine the use of passwords with the use of another authentication mechanism.

Option 1 SMS

- the server stores the user's cellular number on the database

- **when the user performs a login with his password the server sends a code via SMS to the stored phone number**

Option 2 time-based one-time password (TOTP)

- we use a server device that stores internally a secret value stored on the server
- **the user device performs the hashing of the secret and the current time**
    - **hash(secret || current time)**
- This is the approach used by the 2FA authentication application

Option 3 with challenge-response (Universal 2FA)

- the user has a USB that contains a public and a secret key
- the server stores the user's USB's public key
- to log in the server sends the user a challenge string
- the user encrypts it with his private key
- the server verifies that the signature is ok with the user's public key

AUTHENTICATION SCHEMES

Usability features

How easy is it for the user to interact with the authentication schema?

An authentication schema must be:

- **easy to learn**
- **infrequent errors, the schema must be easy to use and must not reject legitimated users**
- **scalable for users**
- **easy-to-recovery-from-loss, a user must be able to recover his credentials if he forgets them**
- **nothing-to-carry, the users don't need to carry additional physical objects to use the scheme**

Deployability features

How easy is it to introduce the schema into a real system?

A schema must be:

- **server-compatible**
- **browser-compatible**
- **accessible, the schema must be used by everyone also if they has disabilities**

Security features

What kind of attacks can the schema prevent?

A schema must be:

- **resilient to physical observation**, an attacker cannot impersonate a user after observing them authenticate one or more times (password schema does not this feature)
- **resilient to targeted impersonation**, an attacker cannot use information about the target to impersonate him
- **resilient to throttled guessing**, an attacker cannot be able to guess the credentials of the users (password schema does not this feature when the entropy is low)
- **resilient to internal observation**, an attacker cannot be able to impersonate him by intercepting the user input (password schema does not this feature)
- **resilient to phishing**, the attacker cannot impersonate the verifier to store information that he can use to impersonate the user (password schema does not this feature)
- **no trusted third party**, the schema must not rely on a third party that could be compromised
- **resilient to leaks from other verifiers**, nothing that a verifier could leak can help an attacker impersonate the user

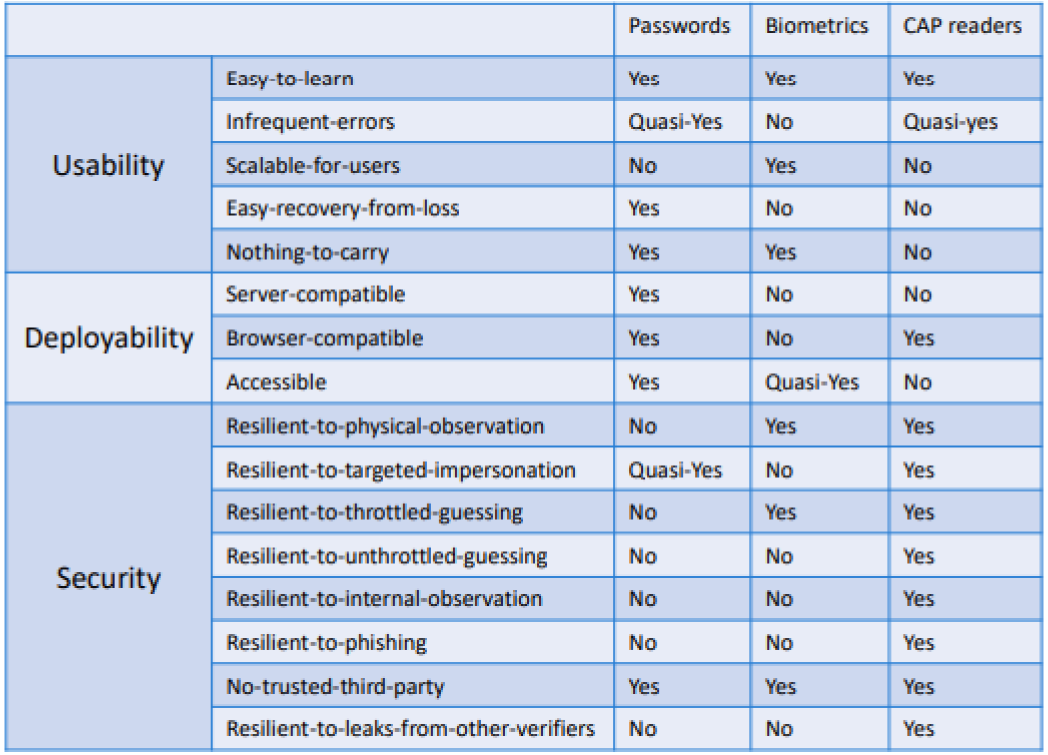
CAP READERS

A cap reader (CAP) is a Chip Authentication Program reader, so it is a device that reads some chips and using a code can output another code to perform a login

For example

1. put your credit card into the reader
2. enter the pin
3. read the 8-digit code for login

Classification of schemas:



		Passwords	Biometrics	CAP readers
Usability	Easy-to-learn	Yes	Yes	Yes
	Infrequent-errors	Quasi-Yes	No	Quasi-yes
	Scalable-for-users	No	Yes	No
	Easy-recovery-from-loss	Yes	No	No
	Nothing-to-carry	Yes	Yes	No
Deployability	Server-compatible	Yes	No	No
	Browser-compatible	Yes	No	Yes
	Accessible	Yes	Quasi-Yes	No
Security	Resilient-to-physical-observation	No	Yes	Yes
	Resilient-to-targeted-impersonation	Quasi-Yes	No	Yes
	Resilient-to-throttled-guessing	No	Yes	Yes
	Resilient-to-unthrottled-guessing	No	No	Yes
	Resilient-to-internal-observation	No	No	Yes
	Resilient-to-phishing	No	No	Yes
	No-trusted-third-party	Yes	Yes	Yes
	Resilient-to-leaks-from-other-verifiers	No	No	Yes

UNIVERSAL SECOND FACTOR (U2f) PROTOCOL

It was produced in order to avoid phishing and man-in-the-middle attacks.

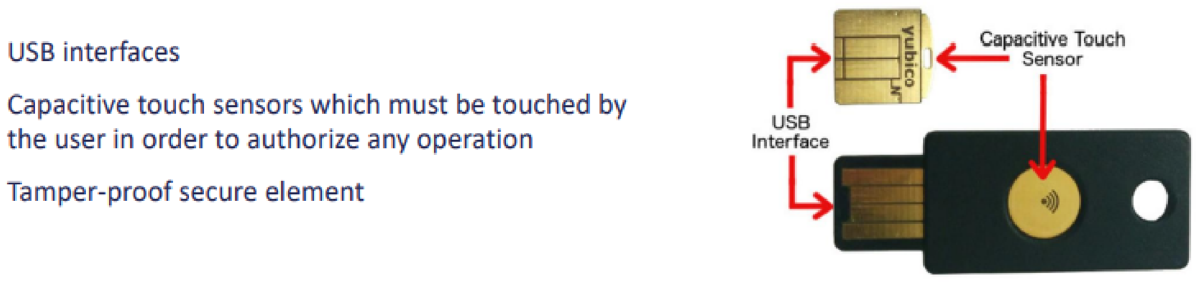
The users must carry a single device that can be used to self-register on any online service that supports the protocol.

It is supported by:

- Design standardized by the FIDO Alliance (see *https://fidoalliance.org*)
- More than 250 companies

amazon
AMERICAN EXPRESS
intel
mastercard
Meta
Google
PayPal
Qualcomm
SAMSUNG
Microsoft
VISA
vmware

Example of devices:



USB interfaces

Capacitive touch sensors which must be touched by the user in order to authorize any operation

Tamper-proof secure element

It is based on the test of user presence (TUP) that is used to understand if a human is present during the command execution.
The TUP implementation is given to the device manufacturer (for example, captive touch sensors, mechanical, power timeouts after USB insertion in a port)

Here we have 2 main operations

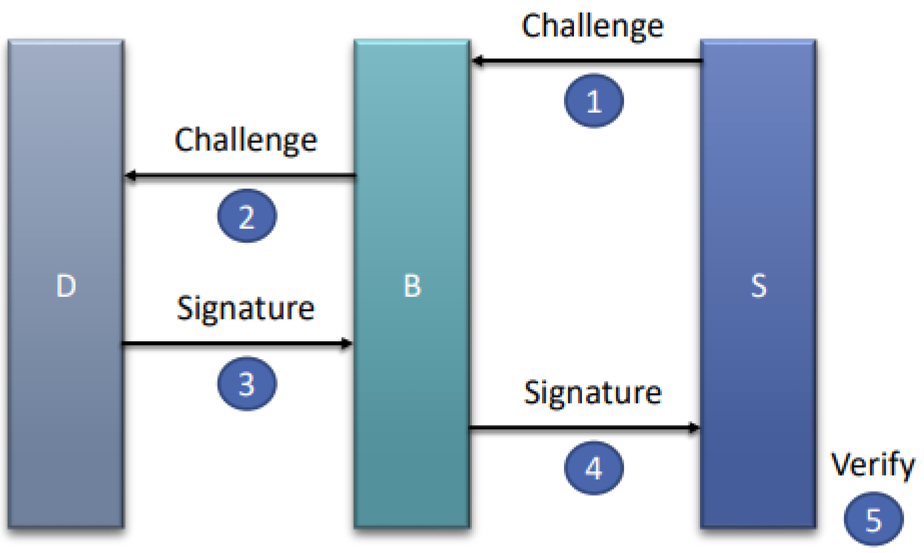
- **sign (K-, message)**
  - **it returns the signature**
- **verify(K+ , message, signature)**
  - **the server verifies that the signature contains the challenge message**

K- is the private key of the device, and k+ is the public key. The server stores k+ for each device.

The schema is composed by:

- server
- browser

- user



Is not possible to actuate a reply attack because each time we connect to the site a new challenge is sent, so also if we can steal the signature it won't work on a later access on the site.

Phishing attack mitigation

It is still possible phishing attack in which the user visits a malicious website and sends a request to the correct site:

- the attacker obtains the challenge from the legit site and forces the victim to sign it with his private key

There is an advanced version of this protocol in which the challenge is not only a string but is the couple:

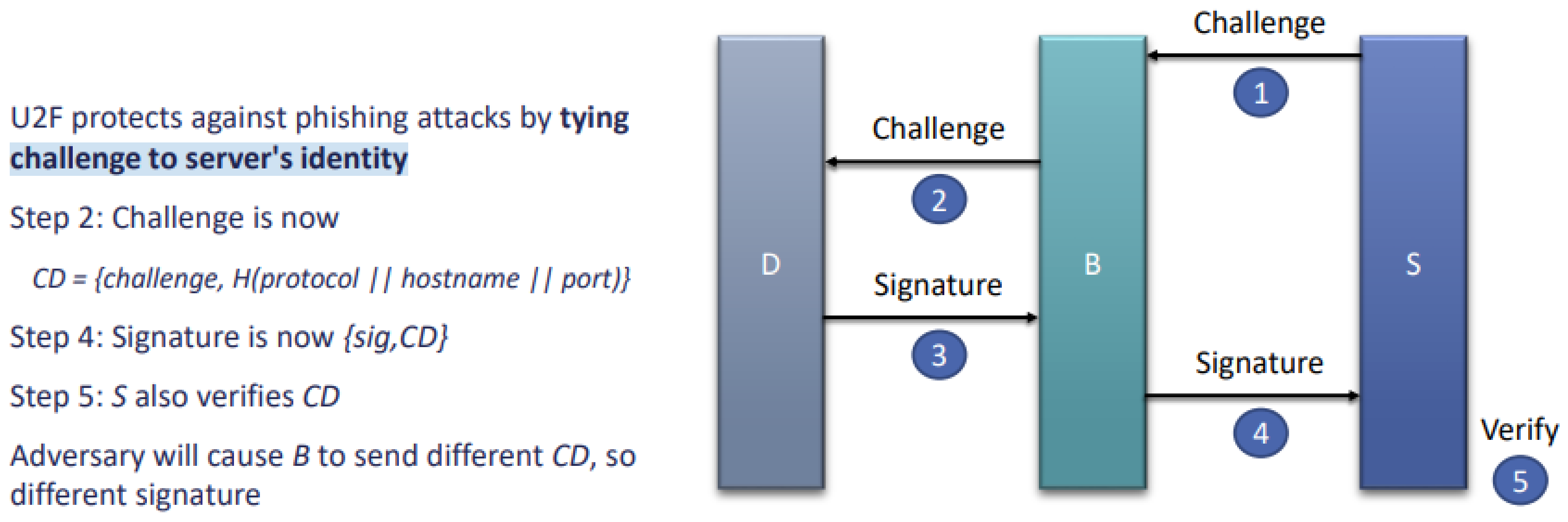
- **CD = {challenge string , has(protocol || hostname || port)**

    - so the challenge is connected to the domain that performs the request

So here the signature becomes:

- **{signature, CD}**
    - **the signature and the entire challenge**

The server verifies the signature and also the CD, so is not possible to perform a phishing attack:



U2F protects against phishing attacks by **tying challenge to server's identity**

Step 2: Challenge is now

CD = {challenge, H(protocol || hostname || port)}

Step 4: Signature is now *{sig,CD}*

Step 5: *S* also verifies *CD*

Adversary will cause *B* to send different *CD*, so different signature

KERBEROS

Kerberos is a protocol born in the second half of the '80.

It is used to perform user authentication in one workstation as root and obtain access to the other servers and services in the network.

So the user logs in to his workstation and has access to the services in the network without any configuration on them.
Microsoft Active Directory uses Kerberos version 5.

It is based on the use of a central server called Kerberos server which must be trusted by the machines in the network.
Each machine trusts only the Kerberos server.

The idea is that:

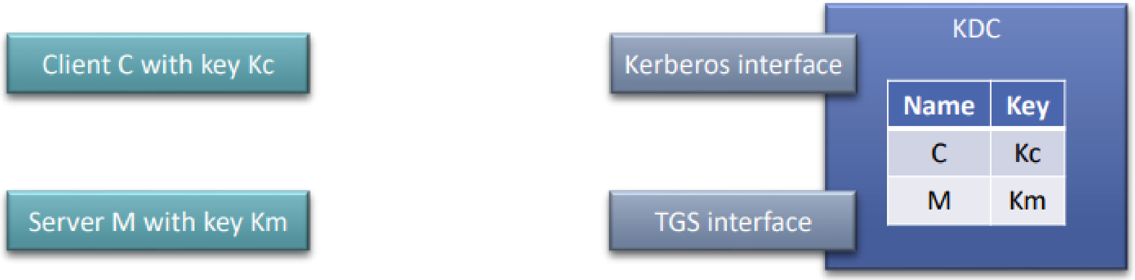
- each machine has a secret key to perform the communication with the Kerberos server
- The Kerberos server stores all machines' secret key

The architecture is based on the KDC (Key distribution center) and on the TGS (Ticket granting service).
Each ticket is used to perform the communication between the machines and has an expiration time.

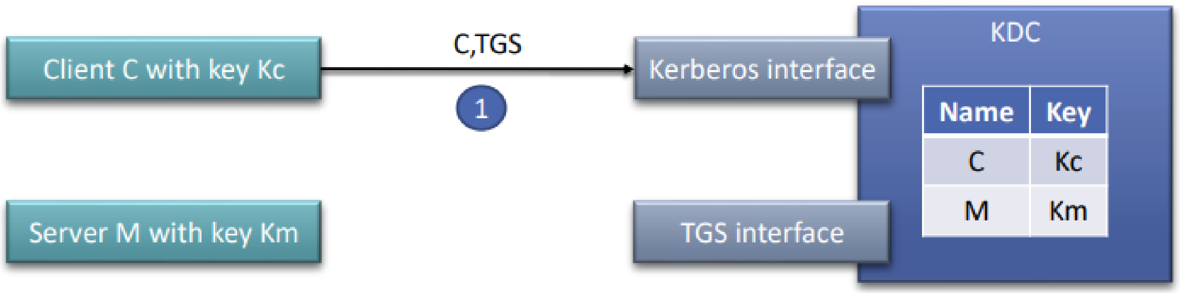
The protocol works in this way:
Imagine having a client C that wants to talk to a server M

1. **client C sends a request to Kerberos in order to obtain a ticket that will be used to talk to the TGS (the request contains the client identification)**
2. **Kerberos Interface replies with a ticket (Tt) that can be used to talk to the TGS**
3. **C sends a message to TGS using the obtained ticket (Tt) asking for a ticket to talk with the server M**
4. **TGS sends to C a ticket that can be used to talk to the server M (Tm)**
5. **C sends a message to M using the ticket Tm**

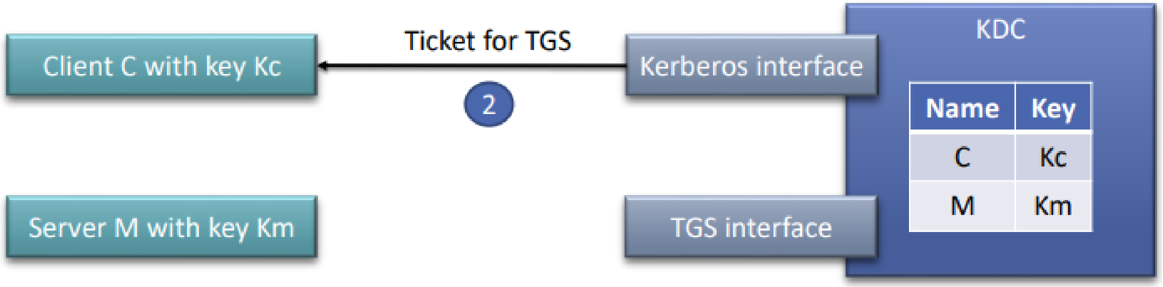
The client doesn't need to authenticate itself when it contacts M because he has the ticket that proves the authenticity.



Name	Key
C	Kc
M	Km

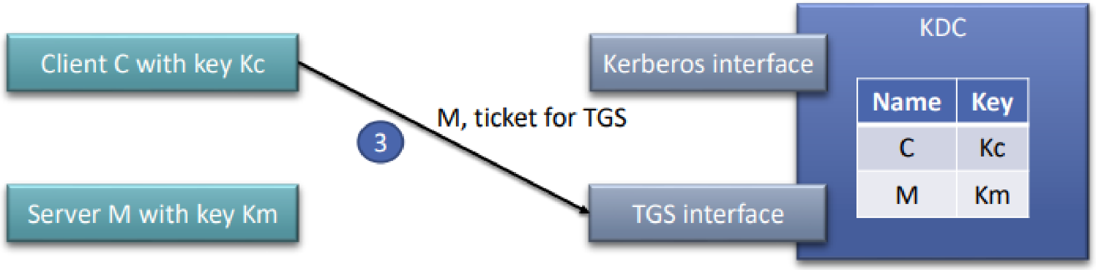


Name	Key
C	Kc
M	Km



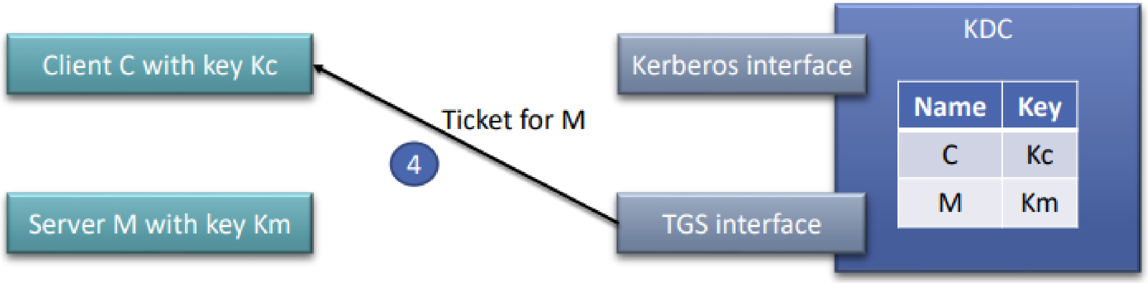
Name	Key
C	Kc
M	Km

Second round: “TGS protocol”

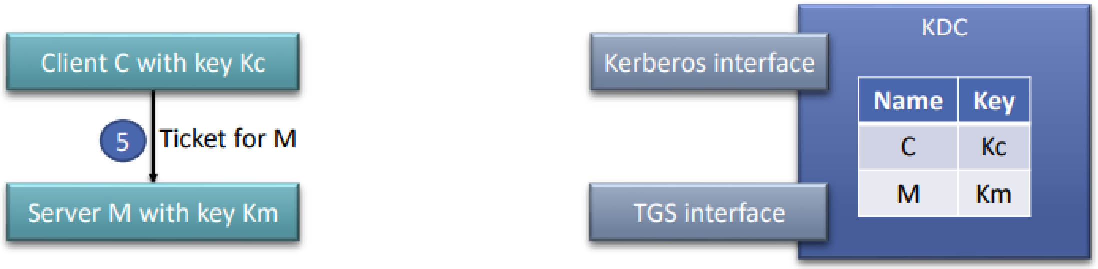


Name	Key
C	Kc
M	Km

Second round: “TGS protocol”



Name	Key
C	Kc
M	Km



Client C with key Kc

5 Ticket for M

Server M with key Km

Kerberos interface

TGS interface

KDC

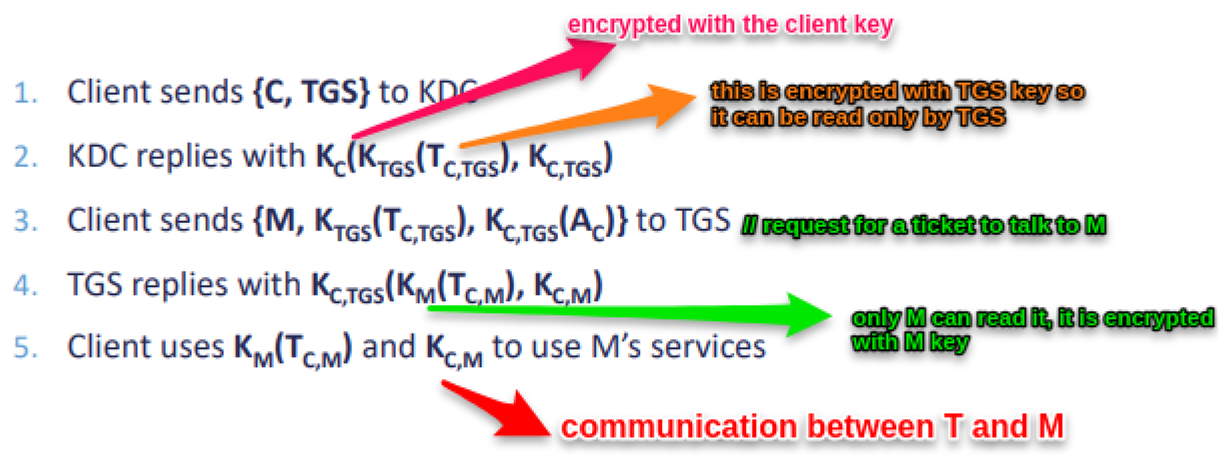
Name	Key
C	Kc
M	Km

Generally, a ticket that is used from C to talk to server S has this form:

$$T_{C,S} = \{S, C, \text{address of } C, \text{timestamp}, \text{TTL}, K_{C,S}\}$$

An authenticator for the client C has the form:

$$A_C = \{C, \text{address of } C, \text{timestamp}\}$$



encrypted with the client key

1. Client sends {C, TGS} to KDC
2. KDC replies with $K_C(K_{TGS}(T_{C,TGS}), K_{C,TGS})$
3. Client sends {M, $K_{TGS}(T_{C,TGS})$, $K_{C,TGS}(A_C)$} to TGS // request for a ticket to talk to M
4. TGS replies with $K_{C,TGS}(K_M(T_{C,M}), K_{C,M})$
5. Client uses $K_M(T_{C,M})$ and $K_{C,M}$ to use M's services

this is encrypted with TGS key so it can be read only by TGS

only M can read it, it is encrypted with M key

communication between T and M

The good things about this protocol are:

- we can avoid replay attacks
- Kerberos v5 offers different encryption schemas, including AES

The bad things are:

- there is no connection between authenticators and the messages exchanged with the client
- ticket requests are not authenticated
  - in fact, Kerberos v5 requires the client to include the encryption with the C key of the timestamp
- there is a single key to perform communication between the server and the client, so it is used by both to encrypt messages

The most important problem here is the reflection attacks:
Let's imagine that server M is a mail server:

1. the attacker sends a mail to the client C that contains "DELETE 5" (this will be for example email 7)
2. when the client requests to the server mail 7 for example, the server will reply with the email "DELETE 5"
3. if the attacker can intercept this message he can send back it to the server forcing it to delete the email 5