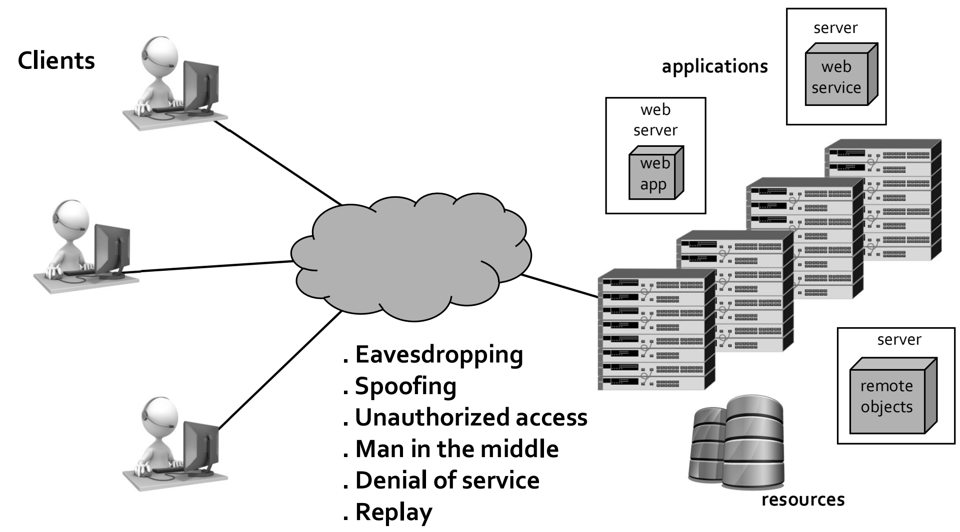
Distributed Systems Security

AUTHENTICATION
KEY DISTRIBUTION
KERBEROS (TICKET BASED AUTHORIZATION)

APM@FEUP

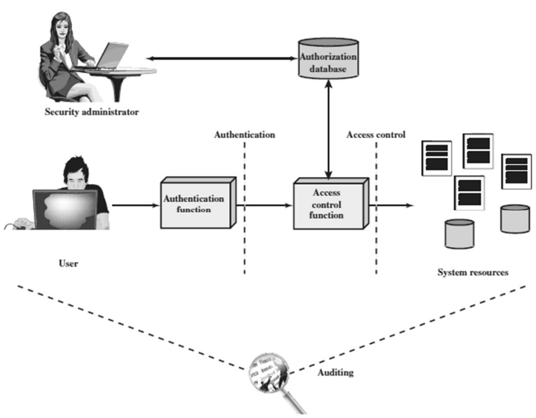
Distributed Applications



Main threats related to the network

Identifying and authorizing

➢ One of main aspects is identifying users and servers and authorizing operations and accesses



➢The other is guarantying confidentiality, integrity and authenticity in information exchanges

Remote authentication

➢ Authenticate a node or user over a network
- We need to assume eavesdropping as a threat, leading to spoofing
- The proof of identity must be protected, and ideally should not be transmitted
- Eavesdropping makes a replay possible and easy
  - Each authentication exchange must be different
  - A protocol using some sort of challenge-response is mandatory

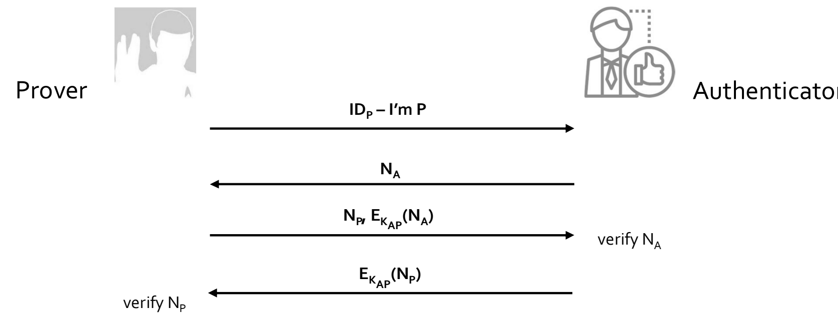
➢ Authentication uses unique data or secret, known by two
- Can be based on passwords, other secrets, biometrics, symmetric or asymmetric keys
- General operation
  - Authenticator sends a challenge (different each time)
  - Prover combines it with secret (using some secure process) and replies it
  - Authenticator verifies both challenge and secret (associated with an identity)

Remote authentication protocols

- It is possible to base authentication only on symmetric or asymmetric cryptography
  - Two basic methods using symmetric or asymmetric keys
- For password-based authentication the CHAP protocol was one of the first published
  - Based on the storage, as a secret on the authenticator side, of H(P(U))
  - The value is combined with a nonce in the client side, and a new hash is transmitted
    - This schema was already presented as the basic challenge-response protocol
- For passwords with salts, the SCRAM protocol is more used
  - SCRAM - Salted Challenge Response Authentication Mechanism
  - Described as a standard in IETF's RFC 5802
- Zero-knowledge password proof
  - Proving that the user knows a secret without saying it
  - SRP is most used, but there are others

Key based generic (symmetric key)

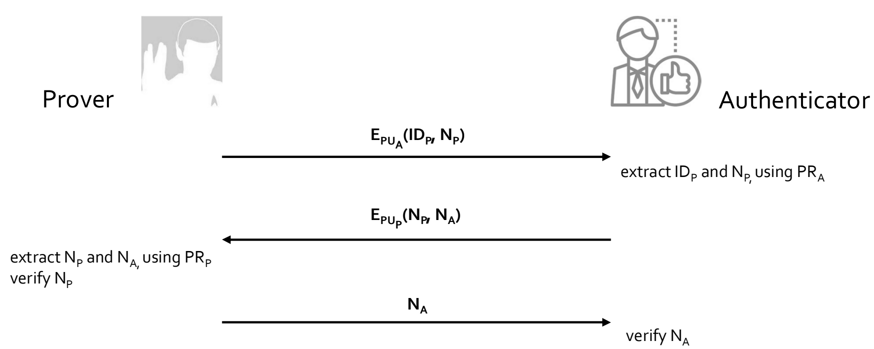
- A symmetric key is known by authenticator(A) and prover(P)
  - Only these two parties know the key (established in installation or registration), e.g., using DH or ECDH
  - Let it be K_{AP}
  - Let N be a nonce (a unique generated random value)
- Mutual authentication or one-way are possible



- For one-way authentication, N_P and the last message are omitted

Key based generic (asymmetric key)

- The public keys of each party are known
  - They are previously established (usually using certificates, CA signed)
  - The correspondent private keys are only known by each party



- What is the purpose of N_P ?

SCRAM protocol

In a database
υ: salt (s), iterations (i), Stored Key: (StK), Server Key: (SrK)

User: υ, password: P

Registration

StP = Salted Password = H_i(s, P) = H_i
U_1 = H(s, P, Int(1)), …, U_k = H(P, U_{k-1})
H_i = $U_1 \oplus \ldots \oplus U_i$
SrK = HMAC(StP, "Server Key") (server Key)
StK = H(HMAC(StP, "Client Key") (stored Key)

User: υ, password: P

Authentication

υ, Nυ (Nυ – random nonce) →

← s, i, Ns (Ns – random nonce)

Computes proof:
ClKey = HMAC(StP, "Client Key")
Prf = ClKey $\oplus$ HMAC(StK, NυNs)

NυNs, Prf →

Verifies if:
H(Prf $\oplus$ HMAC(StK, NυNs)) == StK
Computes verifier (server signature):
SrS = HMAC(SrK, NυNs)

← SrS

Verifies if:
HMAC(HMAC(StP, "Server Key"), NυNs) == SrS

SRP – Secure Remote Password

- **Zero-knowledge password proof**
  - **A method of proving password knowledge without transmitting or storing direct password dependencies**
    - **It claims to prevent eavesdroppers or man-in-the-middle to obtain enough information for a brute-force or dictionary attack**
    - **Specifically designed to avoid existing patents**
      - **Existing for the EKE algorithm (Encrypted Key Exchange)**
  - **Creates a large shared key in a way like Diffie-Hellman, based on the client knowing the password and the server having a cryptographic verifier derived from the password**
    - **The shared key is generated from two random values (one in the client and the other in server)**
    - **Does not need a third party, as Kerberos does**
  - **It needs a finite field Z(N) (N = 2q+1, with q and N primes)**
  - **Also, needs a generator value g of the multiplicative group Z^*_N**
    - **The generator g, is a primitive root of N**
      - **$g^1, g^2, \ldots, g^{N-1}$, generate a permutation of 1..N-1 (elements of group Z^*_N, with N prime)**
      - **All operations performed using mod N**

SRP – Registration and Verification

Both previously agree on N = 2q+1 and g, N and q big primes (arithmetic mod N, and g a generator of Z_N^*)

Registration

Client chooses: Id, P and s
Calculates: $x = H(s, P)$ and $v = g^x$

Server stores: Id, s, v (not knowing x)

Value v is called the verifier

Id – identifier
P – password
s – a random salt
s, P – some combination of s and p

H() is an agreed upon hash, e.g., SHA-256

Login

Both parties can compute k=H(N, g) N, g – some combination of N and g

User supplies Id and P
A random big integer is generated: a
$A = g^a$

→ Id, A

A random big integer is generated: b
$B = kv + g^b$

s, B ←

Both calculate $u = H(A, B)$

$x = H(s, P)$
$S = (B - kg^x)^{a+ux}$
$K = H(S)$

$S = (Av^u)^b$
$K = H(S)$

The value K should be the same

To verify it:

M_1=H(H(N) xor H(g), H(Id), s, A, B, K) → Calculate M_1 and verify. If OK: M_2=H(A, M_1, K)

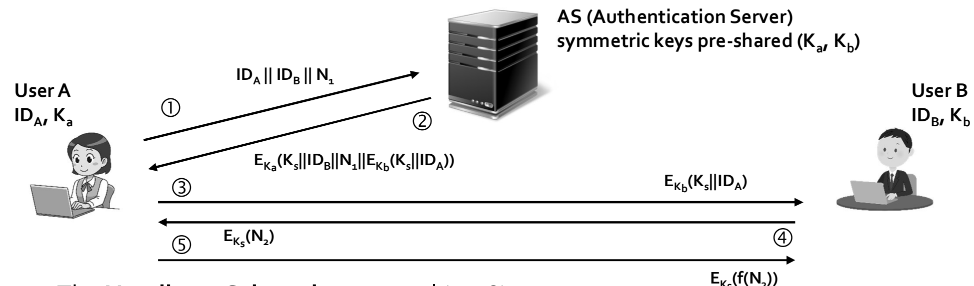
Calculate M_2 and verify. ←

A, B, and u, should be $\neq 0$

Fundamentals
Session (short-term) Key Establishment

- **Needs, usually, a key generation server, trusted by all**
  - **Many times, called the authentication server (AS) (or also known as the KDC – key distribution center)**
- **With symmetric keys, the AS and all other parties pre-share a long-term symmetric key (different for each party)**
  - **It is stored in the AS and each party**
  - **The existence of a centralized AS avoids pre-sharing of N^2 keys**
  - **The authentication is also used to distribute a short-term session key for being used between two parties**
- **With asymmetric keys each node has its own private key**
  - **The AS stores the public keys of every party**
  - **Each party stores the public key of the AS**
- **Protecting against replays needs a nonce or a timestamp**
  - **Timestamps need time synchronization which enlarges the attack surface**

Key establishment: Symmetric key protocols



The **Needham-Schroeder** protocol (1978)

Vulnerability: if someone compromises an old session key K_s, he can replay from message ③

Countermeasure by Denning (1982): include a timestamp at message ② and ③ $E_{K_a}(K_s \| ID_B \| T \| E_{K_b}(K_s \| ID_A \| T))$

But the timestamp T is established in AS and verified in B. A better way was established by Neuman in 1993:

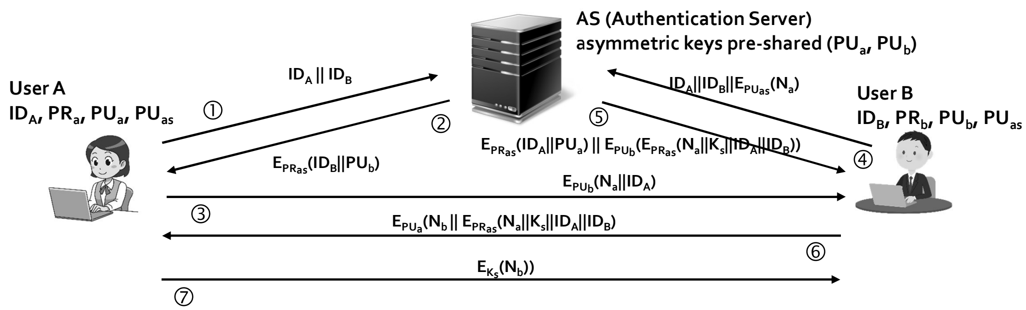
① A → B: $ID_A \| N_a$
② B → AS: $ID_B \| N_b \| E_{K_b}(ID_A \| N_a \| T_b)$
③ AS → A: $E_{K_a}(ID_B \| N_a \| K_s \| T_b) \| E_{K_b}(ID_A \| K_s \| T_b) \| N_b$
④ A → B: $E_{K_b}(ID_A \| K_s \| T_b) \| E_{K_s}(N_b)$

With this proposal by Neuman in 1993 Tb is essentially established and verified in B.
Also, here Tb is a time limit (validity).

Before the limit Tb expires is still possible to A initiate another session with B, without the AS intervention:

① A → B: $E_{K_b}(ID_A \| K_s \| T_b) \| N'_a$
② B → A: $N'_b \| E_{K_s}(N'_a)$
③ A → B: $E_{K_s}(N'_b)$

Key establishment: Asymmetric key protocol

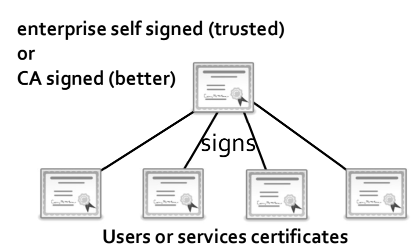


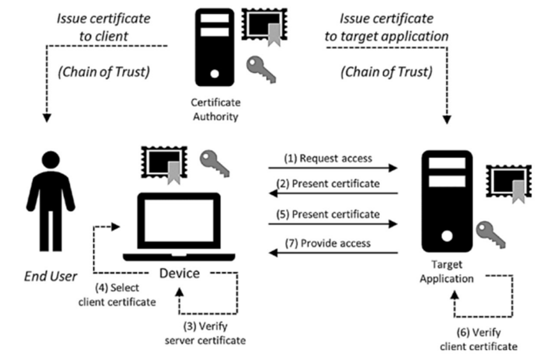
Denning (1982) has proposed also an asymmetric protocol based on timestamps, but it required network synchronization.
Woo and Lam (1992) devised another protocol, without dependencies, and based on nonces, presented above.

In this protocol the AS acts as an authority for certification of the public keys of other users, when it sends those keys with the corresponding ID, encrypted with his private key, like: $\mathbf{E_{PR_{as}}(ID_A \| PU_a)}$ only the AS can produce such a message, which can be decrypted only with his public key (which everyone knows).

Other types of distributed authentication

- **Use of certificates (Mutual authentication e.g., with TLS)**
  - **Can be used by both servers and clients**
    - **The subject of the certificate is the entity to be authenticated**
    - **Should be verified in validity, function (or purpose) and revocation**
    - **From a trusted CA or from the enterprise, and installed as trusted**

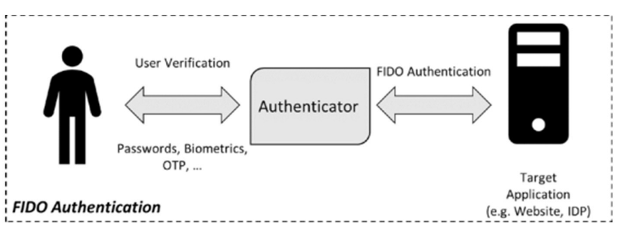




- **Token or ticket based**
  - **Verifiable in specialized services and protocols**
    - **Kerberos, OAuth**

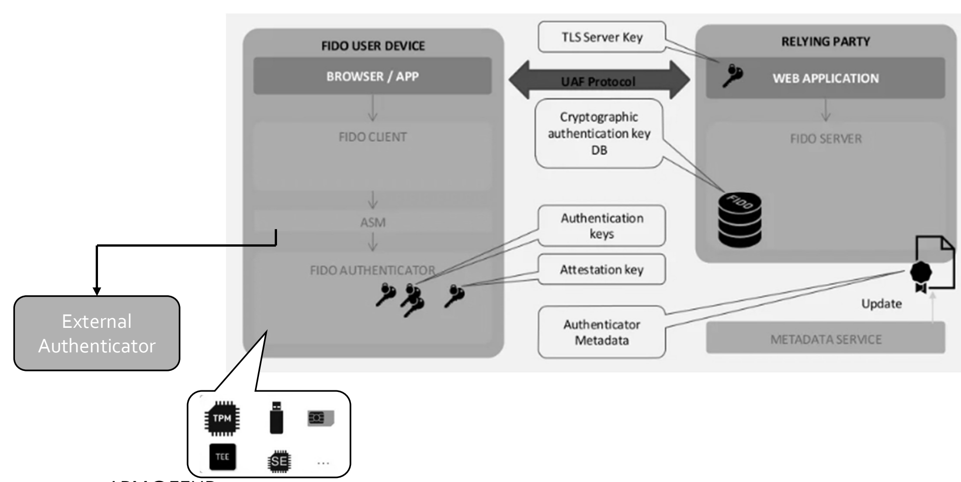
Stronger passwordless authentication

- **Many systems were implemented to allow authentication based on a single action from the user**
  - **FIDO (Fast Identity Online) standardizes for mobile / web Apps/APIs**
  - **Authentication based on possession and/or biometrics supplied on client device and/or external, with asymmetric cryptography**
    - **Better with hardware support on the device (TPM, SE, TEE, …)**
  - **The user presence is verified on an Authenticator application, and then authenticates the user cryptographically with a server**



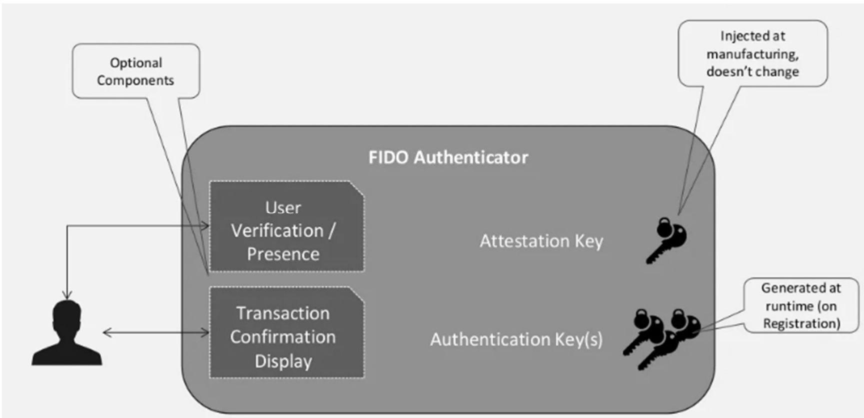
FIDO essential components

- **Several standard software components are needed in a client and a server**
  - **The client runs the user app or a browser, and the server contains a web app or API with the protected remote services**



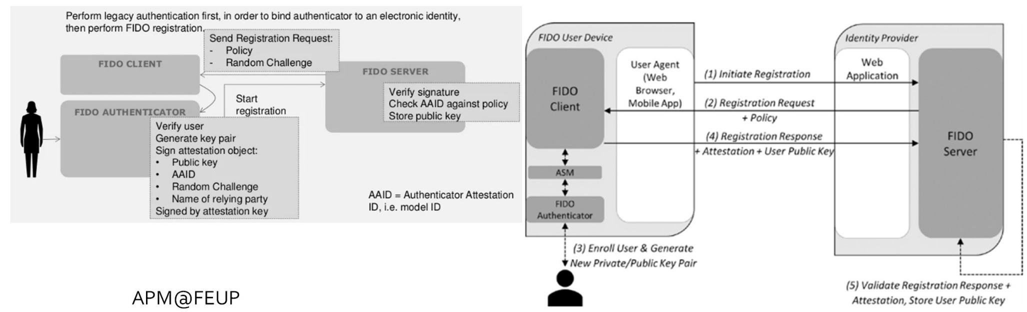
The FIDO Authenticator function

- When an authenticator is built
  - It has an attestation key (private), whose matching key (public) is on the FIDO metadata service (contains also authenticator characteristics)
  - The user authentication keys are generated in registration operations



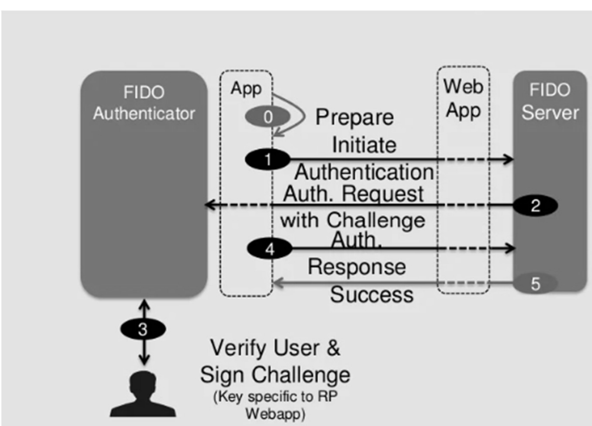
FIDO Registration

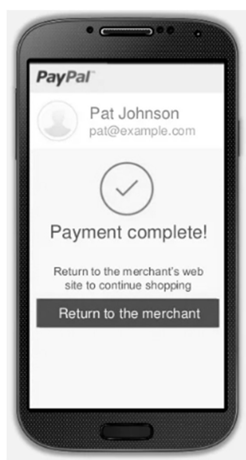
- The registration happens the first time the user executes the client application or accesses the web app
  - The FIDO server creates a binding between the app, the user, and the authenticator
  - Different keys in the authenticator, are generated for different apps
    - Message (2) contains an app identification from message (1) or from the web app in the server
  - Authenticator recognizes always the same user
  - Authenticator is recognized from its attestation signature



FIDO UAF Authentication

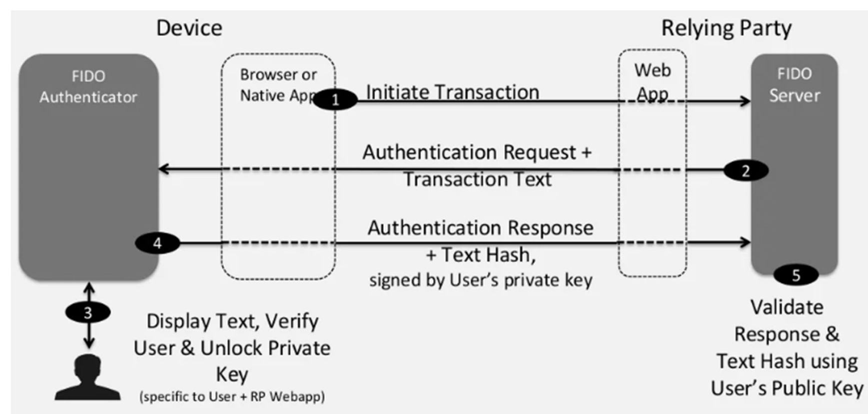
- The UAF (Universal Authentication Framework) is one of the FIDO protocols
- Initiated with a login or a sensitive operation
  - Like a paypal payment, when the service is requested from paypal





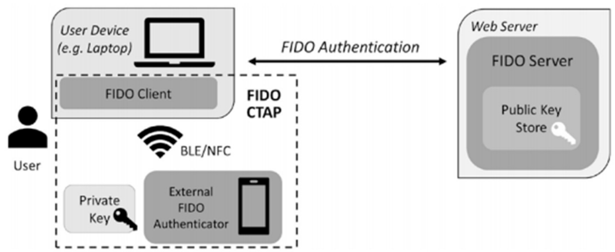
UAF Transaction Confirmation

- Use case where the user has to bound to some statement
  - The working is similar to the previous, but some statement is presented to the user, and signed with the FIDO user private key



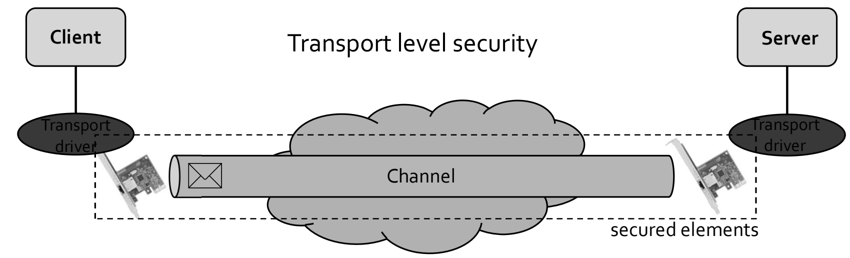
U2F and CTAP

- For applications that need a passwordless second factor
  - U2F – Universal 2nd factor, CTAP – Client to authenticator protocol
  - Can be provided by a FIDO device and authenticator
  - If the authenticator is on another device, it can communicate with the client application using (usually wireless) the CTAP protocol
  - Otherwise, it is like the UAF Authentication or Transaction Confirmation operations, already seen



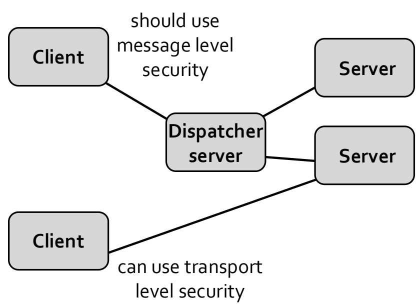
Communications security (1)

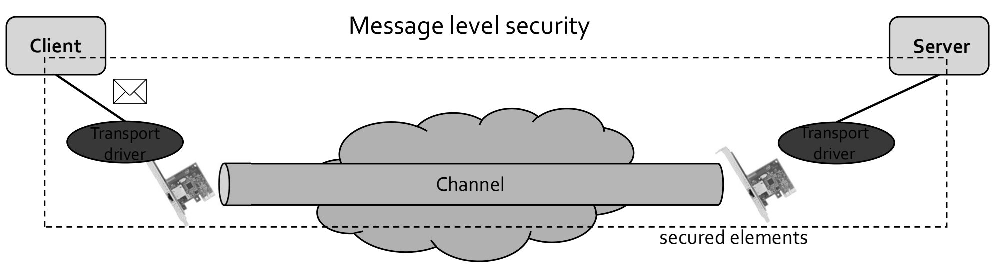
- Should guaranty the security properties (CIA) regarding the transported message
  - Should guaranty confidentiality, integrity, and message authentication
  - Can be at two distinct levels
    - Transport level security
      - Examples: IPSec, SSL/TLS (HTTPS in the HTTP protocol), between nodes
    - Message level security
      - Encryption, MAC, message signature, end-to-end
    - Sometimes both can be active



Communications security (2)

- Sometimes transport security is not enough
  - Transport level is point-to-point
  - Message level is end-to-end (client to final server)



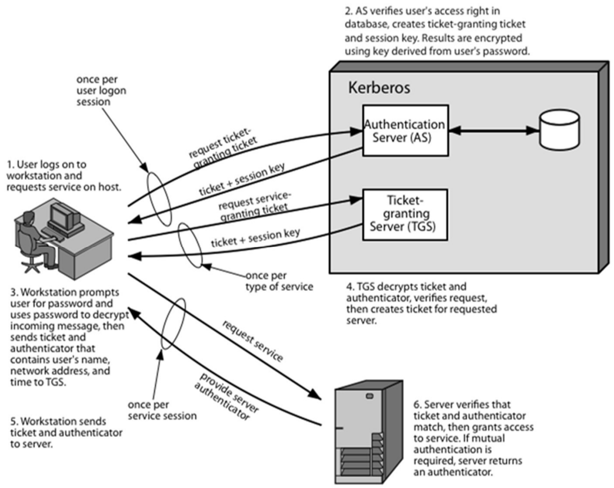


Kerberos

- Designed at MIT to meet distributed systems authentication
  - Two versions are in existence (4 and 5) with version 5 more extensive and secure (Version 5 is an RFC standard (1510, 4120/1)
  - Requires the user to prove his identity to each service invoked. Also, servers should prove their identities to clients
  - Main requirements
    - Secure – prevent eavesdroppers to obtain the information needed to impersonate a user
    - Reliable – distributed architecture with some systems able to replace the functions of others
    - Transparent – users are not aware of its presence beyond the requirement to enter its identifier and password
    - Scalable – To be able to support large numbers of clients and servers and grow
      - Division into realms
  - Servers are not required to trust one another
  - But all systems should trust a third-party authentication server

Kerberos operation

- **Kerberos requires two services with separate functions**
  - **An authentication service (AS) with access to the user and privileges database**
  - **A ticket granting service (TGS) emitting tickets for servers and services**



Earlier design of the protocol

Once per user logon session:

(1) $C \rightarrow AS$: $ID_C \| ID_{tgs}$

(2) $AS \rightarrow C$: $E(K_c, Ticket_{tgs})$

Once per type of service:

(3) $C \rightarrow TGS$: $ID_C \| ID_V \| Ticket_{tgs}$

(4) $TGS \rightarrow C$: $Ticket_v$

Once per service session:

(5) $C \rightarrow V$: $ID_C \| Ticket_v$

$$Ticket_{tgs} = E(K_{tgs}, [ID_C \| AD_C \| ID_{tgs} \| TS_1 \| Lifetime_1])$$
$$Ticket_v = E(K_v, [ID_C \| AD_C \| ID_v \| TS_2 \| Lifetime_2])$$

C – client
AS – authentication service
TGS – ticket granting service

K_c – symmetric key derived from password (stored at AS)
V – server providing the service or resource to the client

AD – network address
TS – time stamp

K_{tgs} – key known by the AS and TGS
K_v – key known by each V and the TGS

I. At logon the client identifies itself and the TGS. The AS responds with a ticket encrypted with a key derived from the client password (stored at the AS). The client asks the user for the password, derives K_c and unencrypts the ticket T_{tgs}. The possession of the correct ticket (T_{tgs}) proves the client identity.

The ticket contains the client Id, its address, a time stamp, and a lifetime, and can only be read by the TGS (the lifetime is typically a few hours)

II. When the client needs some service or resource from V it asks the TGS for a ticket to V, sending the previous ticket (T_{tgs}). The TGS responds with a ticket to V (T_v) if the T_{tgs} is valid. Anytime the client needs services from V it repeats the process.

The ticket to V identifies the client user and the computer network address. It contains also a time stamp and lifetime.

III. Whenever the client needs to contact V, it establishes a session, sending T_v. If V recognizes the ticket, the session is granted.

Shortcomings and improvements

- **The previous protocol**
  - **protects the user password and allows the system to ask the password only once per logon**
  - **But, as the ticket lifetime should be long (hours), there is a window for replay**
    - An attacker can wait for the client to logoff, spoof the network address, and replay message (3) within the original lifetime …
- **New requirements**
  - **There should be a proof that the presenter of a ticket is the same to whom it was emitted**
  - **Servers should also authenticate themselves to clients (preventing server spoofing)**
- **Solving**
  - **AS provides both client and TGS with secret info (e.g., a key) when emitting the T_{tgs}. The client can prove its legitimacy providing that info to the TGS.**
    - Actually, this takes the form of a symmetric key generated by AS ($K_{c,tgs}$), communicated to C and TGS, and used to encrypt messages between the two
  - **For server authentication we can use a handshake (e.g., TS replied by TS+1) and a common key ($K_{c,v}$) generated by the TGS**

The actual Kerberos v4 protocol

(1) $C \rightarrow AS$ $ID_c \| ID_{tgs} \| TS_1$

(2) $AS \rightarrow C$ $E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$

(a) Authentication Service Exchange to obtain ticket-granting ticket

(3) $C \rightarrow TGS$ $ID_v \| Ticket_{tgs} \| Authenticator_c$

(4) $TGS \rightarrow C$ $E(K_{c,tgs}, [K_{c,v} \| ID_v \| TS_4 \| Ticket_v])$

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$
$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$
$$Authenticator_c = E(K_{c,tgs}, [ID_C \| AD_C \| TS_3])$$

(b) Ticket-Granting Service Exchange to obtain service-granting ticket

(5) $C \rightarrow V$ $Ticket_v \| Authenticator_c$

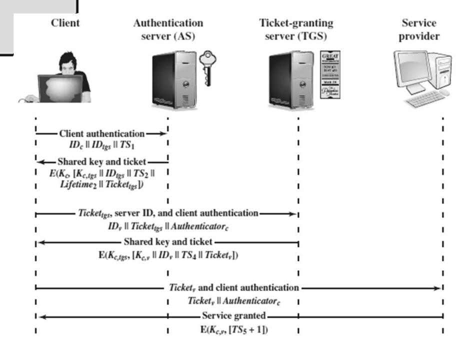
(6) $V \rightarrow C$ $E(K_{c,v}, [TS_5 + 1])$ (for mutual authentication)

$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$
$$Authenticator_c = E(K_{c,v}, [ID_C \| AD_C \| TS_5])$$

(c) Client/Server Authentication Exchange to obtain service

$K_{c,tgs}$ and $K_{c,v}$
keys communicated to client and TGS, and client and V, respectively (by AS (the first) and TGS (the second)

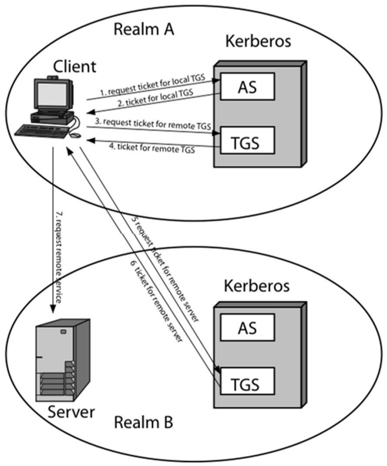
Authenticator$_c$
Proof that the origin is C
The lifetime of these messages is very short



Message flow

Kerberos scalability

- Users, servers, services and resources can be divided in realms
  - Each realm has its own AS and TGS
  - Each realm TGS can emit tickets to the other realms TGSs



Kerberos servers share symmetric keys between them

Kerberos v5 improvements

- Initially standardized in the mid 90's (RFC 1510 (1993))
  - Was revised in 2005 (RFC 4120/1) with updates in algorithms since then
  - addresses encryption, network protocol, byte order, lifetimes, forwarding, and inter-realm authentication
    - allows other encryption algorithms, including asymmetric
    - v4 supports only IP network protocol, v5 allows others
    - network byte order can be specified in messages
    - lifetimes are now specified as start time and end time
    - allows a server V to use other services and resources in other servers in behalf on the same client (requesting tickets)
    - uses inter-realm authentication without using N^2 Kerberos-to-Kerberos relationships (in a system with N realms)
    - It allows the derivation of sub-session keys
    - Allows user pre-authentication
  - All these new features and extensions have conducted to a more complex protocol

Kerberos version 5 summary

(1) C → AS *Options* || *IDc* || *Realmc* || *IDtgs* || *Times* || *Nonce1*

(2) AS → C *Realmc* || *IDC* || *Tickettgs* || E(*Kc*, [*Kc,tgs* || *Times* || *Nonce1* || *Realmtgs* || *IDtgs*])

Tickettgs = E(*Ktgs*, [*Flags* || *Kc,tgs* || *Realmc* || *IDC* || *ADC* || *Times*])

(a) Authentication Service Exchange to obtain ticket-granting ticket

(3) C → TGS *Options* || *IDv* || *Times* || || *Nonce2* || *Tickettgs* || *Authenticatorc*

(4) TGS → C *Realmc* || *IDC* || *Ticketv* || E(*Kc,tgs*, [*Kc,v* || *Times* || *Nonce2* || *Realmv* || *IDv*])

Tickettgs = E(*Ktgs*, [*Flags* || *Kc,tgs* || *Realmc* || *IDC* || *ADC* || *Times*])

Ticketv = E(*Kv*, [*Flags* || *Kc,v* || *Realmc* || *IDC* || ADC || *Times*])

Authenticatorc = E(*Kc,tgs*, [*IDC* || *Realmc* || *TS1*])

(b) Ticket-Granting Service Exchange to obtain service-granting ticket

(5) C → V *Options* || $Ticket_V$ || $Authenticator_C$

(6) V → C $E_{K_{C,V}}$ [TS_2 || *Subkey* || *Seq#*]

Ticketv = E(*Kv*, [Flags || *Kc,v* || *Realmc* || *IDC* || *ADC* || *Times*])

Authenticatorc = E(*Kc,v*, [*IDC* || *Realmc* || *TS2* || *Subkey* || *Seq#*])

(c) Client/Server Authentication Exchange to obtain service