Kerberos, SSL, IPsec

Shankar

May 28, 2013

Kerberos 4

Kerberos 4

Authentication in network (Realm)

- Realm has KDC and users (principals)
- Users: human (log in to worsktations)
  apps: NFS, rsh, etc
- Authentication: based on Needham-Schroeder protocol
- Attacker: can read and write messages in transit.
- Assumes DES and IPv4
- Uses timestamps: nodes need synchronized clocks

KDC has

- **Master key** for each user
  - weak key for human; strong key for apps
- Secret-key K_{KDC} not shared with any user
  - to encrypt database (master keys), TGTs
- Database: changes only when user's master key changes
  - mitigates KDC from becoming bottleneck

Kerberos 4

When a human user logs in

- KDC authenticates user based on user's master key
- KDC gives user **credentials** encrypted with user's master key
  - **Session key:** for current login session
    // user's master key not used after login
  - **Ticket Granting Ticket (TGT)** encrypted by K_{KDC}:
    - for user to obtain further tickets from KDC

For human user to access an app

- User sends KDC enc([request,TGT,timestamp], session key)
- KDC returns **credentials** encrypted with session key
  - session key to talk to app
  - ticket encrypted with app's master key (app is not human)
- user sends app [request, ticket]

K4: Login handshake

user A (has pw) at workstation	KDC (has A: K_A)
start login send [“A needs TGT”] receive msg get K_A from pw extract S_A,tgt_A from crd_A forget pw; // use S_A henceforth finish login	 receive msg, retrieve K_A generate session key S_A $tgt_A \leftarrow enc([A, S_A], K_{KDC})$ $crd_A \leftarrow enc([S_A, tgt_A], K_A)$ send [crd_A]

K4: Accessing remote app B

(LATER IN THE SESSION)

<table>
<tr><td>user A at workstation</td><td></td></tr>
<tr><td rowspan="2">rlogin B
send [A,B, tgt_A, enc(ts,S_A)]
// S_A(ts): authenticator

receive msg
extract K, tkt_B
send [A,B,tkt_B, enc(ts,K)]

receive msg
end</td><td>rcv msg, gen sess key K
get S_A from tgt_A
get ts and verify
find B's master key K_B
$tkt_B \leftarrow$ enc([A,K], K_B)
crd_B = enc([B,K,tkt_B] , S_A)
// credential
send [crd_B] to A</td></tr>
<tr><td>B

receive msg from A
extract K, ts
send [B,A, enc(ts+1, K)]</td></tr>
</table>

K4: Replicated KDCs for performance

- One master KDC and several secondary KDCs
- Each secondary KDC has read-only copy of KDC database
- Additions/deletions/changes to master keys always done at master KDC
- Secondary KDCs can generate session keys, TGTs, etc.
- Master disseminates KDC database to secondary KDCs with integrity protection
  (master keys already encrypted with K_{KDC})

K4: Authentication across multiple realms

- Possible only if their KDCs share a key.
- Principal id = [name, instance, realm], each 40 chars max

A in realm X	KDC X	KDC Y	B in realm Y
send [A.X, B.Y] to X			
	receive msg send [cred.Y] to A		
receive msg send [A.X, B.Y, cred.Y] to Y			
		receive msg send [cred.B] to A	
receive msg send [A, B, cred.B]			
			receive msg

K4:Key version number

- If A has tkt to B, and B changes its master key, then ticket no longer valid
- To handle this (w/o A asking KDC for a new ticket):
  - Apps remember old master keys up to expiry time (approx 21 hrs)
  - In tickets, the key is sent along with version number
  - Human users need not remember old passwords

K4: Network layer address in tickets

- Ticket has IPv4 address of the user given the ticket
- Received ticket is not accepted if ticket sender's IP address does not match
- So if B is to impersonate A, it must also spoof the IP address of A (easy to do)
- Prevents delegation
  - A cannot have B at another IP address do work on behalf of A (unless B spoofs IP address of A !)

K4: Encryption/Integrity of data

- After authentication, data exchange can be any of
  - clear
  - encrypted
  - integrity-protected
  - encrypted and integrity-protected
- Choice is up to the application (performance vs security)
- K4 uses adhoc integrity protection (not so safe)

Kerberos 5

Kerberos 5

- More general than Kerberos 4

- Message formats defined using ASN.1 and BER
  - So allows for addresses of different formats, etc.
  - Occupies more octets

- Names: [NAME, REALM]
  - Arbitrary content, length // allows ".", "@",…
  - Allows X.500 names // country/org/name/…

- Allows choice of crypto algorithms
  - Uses proper integrity protection

K5: Delegation of Rights

- User A can ask KDC for a TGT with
  - network addresses different from A's address (for use by a principal at another address on behalf of A)
  - no address (for use by any principal at any address)
- User A can give a tgt/tkt to B with specific constraints
  - TGT/tkt has "app" field copied by KDC to any derived tkt
- A's TGT can be **forwardable**:
  - A can use it to get a TGT (for B) with different address.
  - A also says whether derived TGT is itself forwardable
- A's TGT can be **proxiable**:
  - A can use it to get tkt (for B) with different address
- Ticket lifetime

K5: TGT/TKT Lifetime

- Fields:
  - start-time: when ticket becomes valid
  - end-time: when ticket expires (but can be renewed)
  - auth-time: when A first logged in (from initial login TGT)
  - renew-till: latest time for ticket to be renewed.
- Allows unlimited duration subject to renewing (e.g., daily)
  - exchange tgt/tkt at KDC for a new (renewed) tgt/tkt
  - tgt/tkt has to be renewed before expiry
- Allows **postdated** tickets (e.g, for batch jobs).

K5: Keys

- KDC remembers old master keys of human users also
  - because tgts/tickets are renewable and can be postdated.
- For each principal B, KDC stores
  - key: B's master key encryped with K_{KDC} (current or past)
  - p_kvno: version number of B's master key
  - k_kvno: version number of K_{KDC} used to encrypt
  - max_life, max_renewable_life: for tickets issued to B
  - expiration: when this entry expires
  - mod_date/mod_name: when entry last modified, by who
  - flags: preauthentication?, forwardable?, proxiable?, …
  - …
- Human user master key derived from pw and realm name.
  - So weak protection from key exposure across realms

K5: Authentication Chains

- Allows KDC chains of authentication (unlike V4)
- Example: KDCs A, B, C, where
  - A-B share key, B-C share key, but A,C do not.
  - K5 allows C to accept tkt sent by A and generated by B
- Each ticket inclues all the intermediate KDCs
  - receiving KDC can reject tkt if it has suspect intermediary

K5: Evading off-line password guessing

- K4 allows off-line password guessing:
  - KDC does not authenticate login requet before issuing TGT
  - So B can spoof A, get a TGT for A,
    do off-line dictionary attack on TGT

- In K5
  - Login req must contain K_A\{timestamp\};
    so above attack not possible
  - KDC also does not honor requests for tickets to human users by others
    - Prevents logged-in B asking KDC for a tkt (to delegate) to A, on which it can do off-line password guessing.

K5: Key inside authenticator

- Suppose A and B share a session key K generated by KDC
- A and B can have another (simultaneous) session using a different key.
- This can be done without involving the KDC:
  - A makes up a key for this second session and gives that to B encryped by K

K5: Double TGT Authentication

- Allows A to access server B that has session key, say S_B, but not master key K_B
- Needed for X windows:
  - X server manages display on workstation screen
  - X clients (eg, xterm) run on local or remote workstations
  - X client (A) needs tkt to X server (B) to display on screen
- No good for A to get from KDC a (regular) tkt encrypted K_B
- Instead
  - A gets TGT_B from B
    sends [“A to B”, TGT_A, TGT_B] to KDC
  - KDC
    - extracts S_B from TGT_B (encrypted with K_{KDC})
    - creates session key $K_{AB,}$
    - generates tkt_B encrypted with S_B (ie, enc([‘A’, K_{AB}], S_B) and sends to A

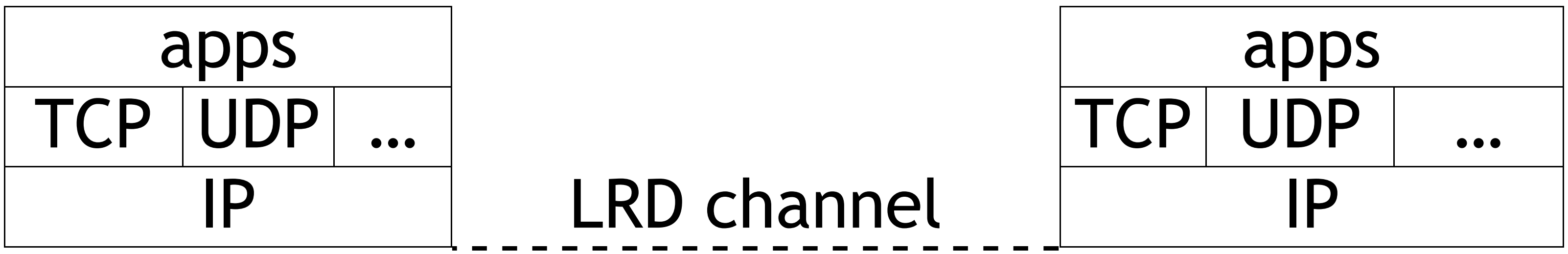
K5: X windows

B at workstation/X server	C (may be B’s workstation)
login to X server (B,pw) • request TGT_B from KDC • obtain [S_B, TGT_B] from KDC • forget B’s passwd • start serving B (eg, keybd, mouse)	
request X client at C (eg, xterm)	
	• X client starts • has info to display at B’s screen • get TGT_B from X server • ask KDC for tkt encrypted by S_B • present tkt to X server and info to display
• X server displays client’s info	

Security with TCP/IP

TCP/IP + Security

- TCP/IP stack without security



- TCP provides apps with
  - connection establishment
  - reliable data transfer

- Want to extend this to handle attackers
  - network attackers: passive / active
  - endpoint attackers: send messages with arbitrary fields
  - **authentication:** extends connection establishment
  - **confidentiality, integrity:** extends reliable data transfer

Natural solution: Secure TCP

STCP (Secure TCP) like TCP except

- 3-way connection establishment includes
  - client id, server id, authentication secret
- data transfer is tcp-like except
  - IP header is in clear
  - secure-tcp header encrypted
  - secure-tcp payload encrypted

Reality

Implementors did not want

- modifications to TCP (which is in OS kernel)
- another protocol like TCP in OS kernel or over UDP
- another protocol like TCP in app space (eg, over UDP)

Instead we now have two partial solutions

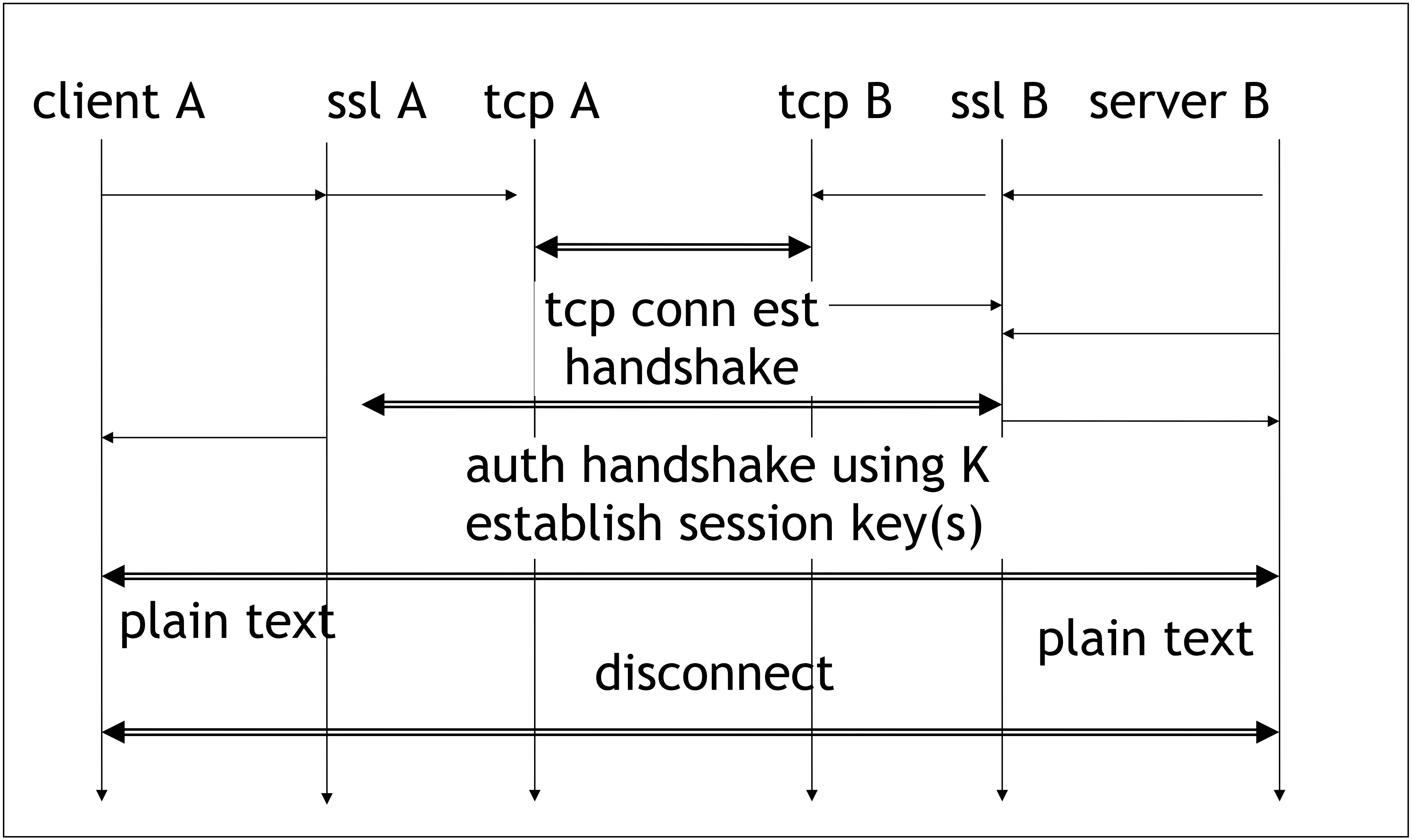
- SSL (Secure Sockets Layer): above TCP
- IPsec: above IP and below transport layer (TCP, UDP)

SSL

SSL

- When A connects to B
  - TCP A and TCP B establish a connection
  - SSL A and SSL B authenticate over TCP
    - using A public key and B public key, or
    - using B public key and A password (typical)
- During data transfer:
  - SSL encrypts outgoing / decrypts incoming
- TCP messages have TCP header in clear
  - Easy DOS attack: Rogue packet attack

SSL



client A
ssl A
tcp A
tcp B
ssl B
server B
tcp conn est
handshake
auth handshake using K
establish session key(s)
plain text
plain text
disconnect

SSL

- SSL A: generate R_A
  send [B, ciphers supported, random R_A]
- SSL B: choose cipher, generates R_B
  send [B, cipher chosen, $cert_B$, random R_B]
- SSL A: generate S // pre-master secret
  $K \leftarrow f(S,R_A,R_B)$ // master secret
  send [$enc_P(S, pub_B)$, $hash_1$(handshake,K)]
- SSL B: send [$hash_2$(handshake)]
- SSL A: if $hash_2$ verifies, B authenticated
  send [enc(pw, K-derived-key)]
- SSL B: if pw verifies, A authenticated

A can also use $cert_A$ for authenticating itself to B

SSL

- S: pre-master secret
- K: master secret
  - $K = f(S, R_A, R_B)$
- Keys for data encryption/integrity obtained from K, R_A, R_B
  - A's write (transmit) key = B's read (receive) key
  - B's write (transmit) key = A's read (receive) key

- A does two public-key crypto operations
  - verifying $cert_B$
  - calcluating $\{S\}_B$
- To minimize this, S can be reused across different sessions
  - motivated by http 1.0 (opens many A-B tcp sessions)
  - session id

SSL

ssl A

ssl A

[B, ciphers, R_A] →

← [B, session-id = X, $cert_B$, cipher, R_B]

initial session

new session later on

[x,y,B, session-id = X, ciphers, R_A] →

← [B, session-id = X, $cert_B$, cipher, R_A, keyed hash of handshake]

[x,y, keyed hash of handshake] →

if ssl A still has X:S, can reuse it

IPsec

IPsec

- IPsec sits above IP and below TCP, UDP, …
- IP packet: [IP hdr, IPsec hdr, TP hdr, TP payload]

  ←——–- IP payload ————→

  ←– IPsec payload →
- TP is IP: "tunnel" mode // often used to tunnel IP traffic
- TP is not IP: "transport" mode

IPsec: generic header

- IP hdr
  - sender ip addr, rcvr ip addr
  - hop count   // mutable
  - next protocol id: TCP, UDP, IP, IPsec (AH or ESP), …
- IPsec header (**generic**):
  - SPI (security parameter index): identiifies IPsec connection (SA)
  - sequence number: of IPsec packet (for replay attacks)
  - IV (for encryption/integrity)
  - authentication data (integrity check)
  - next protocol id: (TCP, UDP, IP, …)

IPsec: Security association

- IPsec SA (**security association**): IPsec connection
- An SA is one-way, so need two SAs for bi-directional flow
- IPsec entity in a node has
  - **Security policy database** (control path)
    - for <ip addr, port, etc>:
      - crypto or not? type, integrity/encryp, …
  - **SA (security association) database** (data path)
    - outgoing: for remote ip addr:
      - SPI, crypto key/alg, sequence number
    - incoming: for SPI:
      - crypto key/algo, expected seq number, …

IPsec: AH and ESP

- IPsec headers are in two flavors

- AH hdr:
  - SPI, sequence number, auth data, next protocol id
  - integrity only but on enclosing IP
    <payload + “immutable” header>
  - not compatible with NAT, firewalls

- ESP hdr:
  - SPI, seq number, IV, auth data, next protocol id
  - integrity and/or encryption on enclosing IP payload
  - compatible with NAT, firewalls

IPsec: IKE

- For an IPsec SA to operate,
  its parameters (integrity/encryp, key, …) must be set
  in the (SA database of the) end-points of the SA
- Can be done manually by end-point administrators
  or dynamically using IKE
- IKE runs over UDP, has two phases, and is an UGLY MESS