Note on NS chapter 13: Kerberos V4

Authentication in network (Realm)
- Human users log in to workstations, use (distributed) applications (NFS, rsh, etc).
- Realm has KDC that authenticates login sessions and (Kerberozed) applications.
  - Based on Needham-Schroeder authentication protocol.
- Assumes attacker can eavesdrop and modify messages in transit.
- Secret key technology (DES).

KDC has
- **master key** for each principal:
  - Human user: username and password (from which master key is obtained).
  - Application has (high quality) key.
- Secret key $K_{KDC}$ (not shared with any other principal) used for encrypting
  - master keys in local database
  - TGTs

When human user logs in
- KDC authenticates user based on user’s master key.
- KDC provides user credentials (encrypted with master key) consisting of
  - **Session key** for that login session (user master key is not used after login).
  - **Ticket Granting Ticket (TGT)** used to obtain further tickets from KDC.
    - TGT is encrypted by $K_{KDC}$.

When human user wants to access an application
- User workstation presents KDC with [request, TGT] (encrypted by session key).
- KDC returns credentials (encrypted by session key) consisting of
  - Session key (to talk to application)
  - Ticket for application (encrypted with application’s master key).
- User workstation presents application with [request, ticket].
- Note that this is really one application (user shell) accessing another application.
Kerberos login handshake

<table>
<thead>
<tr>
<th>A user</th>
<th>A’s workstation</th>
<th>KDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[A, passwd]</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>start login</td>
<td></td>
</tr>
<tr>
<td></td>
<td>send [A, KDC, AS_REQ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>says “A needs TGT”</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>receive msg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>construct K(_A) from passwd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extract S(_A), tgt(_A) from crd(_A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>forget passwd; shell uses S(_A) henceforth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tell user “login” succeeded</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>login finish</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(LATER IN THE SESSION)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rlogin B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>send [A, KDC,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TGS_REQ, “A to talk to B”, tgt(_A), authenticator (= S(_A)(ts)) ]</td>
<td></td>
</tr>
</tbody>
</table>

 receive msg
 generate session key K\(_{AB}\)
 extract S\(_A\) from tgt\(_A\)
 extract ts from authenticator and verify
 find B’s master key K\(_B\)
 generate tgt\(_B\) = K\(_B\)\{‘A’, K\(_{AB}\)\}
 crd\(_B\) = S\(_A\)\{‘B’, K\(_{AB}\), tgt\(_B\)\} // credential
 send [TGS_REP, crd\(_B\)] to A

 receive msg from KDC
 send [AP_REQ, tgt\(_B\), K\(_{AB}\){ts}] to B
 tell user “rlogin B” succeeded

 send [AP_REP, K\(_{AB}\){ts+1} ] to A
Replicated KDCs to improve performance
- One master KDC and several secondary KDCs, each with read-only copy of KDC.
- Additions/deletions/changes to master keys always done at master KDC
- Secondary KDCs can generate session keys, etc.
- Master disseminates KDC databases to secondary KDCs with integrity protection only
  (but master keys are encrypted with $K_{KDC}$)

Authentication across multiple realms
- Possible if their KDCs share a key.
- Principal name = ["name", "instance", "realm"]

<table>
<thead>
<tr>
<th>A in X</th>
<th>KDC_X</th>
<th>KDC_Y</th>
<th>B in Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>send [A, KDC_X, TGS_REQ, A.X, D.Y]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- receive msg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- send [KDC_X, A, TGS_REP, cred to KDC_Y]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- receive msg</td>
<td>send [KDC_Y, A, TGS_REP, cred to B]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>send [A, B, AP_REQ, cred, …]</td>
<td>receive msg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key version number
Suppose A has a ticket to B and B changes its password. Then ticket no longer valid.
To handle this case (without A having to ask KDC for a new ticket):
- Applications 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.

Network layer address in tickets
- Every ticket has the IPv4 address of the principal 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, i.e., if A wants B to do some work on behalf of A (unless B spoofs
  IP address of A!)
Encryption of application data

- Once a session is authenticated, data can be exchanged in the clear, or encrypted, or encrypted and integrity-protected.
- Choice is up to the application (performance vs security).
- Kerberos V4 uses some adhoc encryption techniques (not so safe).

Encryption for Privacy and Integrity

Recall that Standard technique requires two keys and two crypto passes (expensive).

Kerberos uses a modified CBC called Plaintext CBC (PCBC)

- In CBC: \( c_{n+1} = E_K(m_{n+1} \oplus c_n) \)
  - Modifying any \( c_i \) causes only \( m_i \) and \( m_{i+1} \) to be garbled.

- In PCBC: \( c_{n+1} = E_K(m_{n+1} \oplus c_n \oplus m_n) \)
  - Modifying any \( c_i \) causes all \( m_j \) for \( j \geq i \) to be garbled.
  - Kerberos puts recognizable last block, so tampering detected.
  - However, swapping \( c_i \) and \( c_{i+1} \) makes PCBC get back in synch from \( m_{i+2} \)

Not used in V5

Encryption for Integrity only

Computes checksum on \([\text{session key}, \text{msg}]\)
Probably not cryptographically strong

- May allow attacker to modify msg and pass integrity test
- May allow attacker to obtain session key

Not used in V5

Message formats

Look in text.
Note on NS chapter 14: Kerberos V5

More general than V4

Message formats

- Defined using ASN.1 and BER (Basic Encoding Rules)
- Automatically allows for addresses of different formats, etc.
- Occupies more octets

Names

[NAME, REALM]
- Each is arbitrary string (allows ".", "@", thus "name@org", etc).
- Allows X.500 names (Country/Org/OrgUnit/LName/PName/…).

Delegation of rights

A can ask KDC for TGT with
- One or more network addresses different from A’s network address.
  Principals at other IP addresses can use this on behalf of A.
- No network addresses (can be used by any principal at any network address).

Policy decision whether KDC/network issues/accepts such tgs.
- Advantage: KDC tracks delegation trail
- Disadvantage: A has to interact with KDC for each delegation

A can give a TGT to B with specific constraints.
- TGT/Ticket lists the specific resources that can be accessed.
- TGT/Ticket has a AUTHORIZATION-DATA field that is application specific.
  KDC copies this field from TGT into any derived ticket (used in OSF, Windows).
- A’s TGT can be forwardable:
  - Allows A to use TGT to get a TGT (for B) with different network address.
  - A also says whether derived TGT is itself forwardable.
- A’s TGT can be proxiable:
  - Allows A to use TGT to get tickets (for B) with different network address.
    Referred to as proxy tickets.
Ticket lifetime

TGT/tkt lifetime specified in ANS.1 (17 octets)

- Fields:
  - start-time: when ticket becomes valid
  - end-time: when ticket expires (but can be renewed (see renew-till)
  - auth-time: when A first logged in (copied from initial login TGT)
  - renew-till: latest time for ticket to be renewed.

- Allows unlimited duration (upto Dec 31, 9999) 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 (o/w KDC will not renew)

- Allows postdated tickets (e.g., for batch jobs).

Keys

KDC remembers old master keys of human users (in addition to applications)

- Needed because tgts/tickets are now renewable and can be postdated.
- For each principal, KDC database stores [key, p_kvno, k_kvno]
  - key: principal’s master key encrypted with K
  - p_kvno: version number of principal’s master key.
  - k_kvno: version number of K
  - max_life: max lifetime for tickets issued to this principal
  - max_renewable_life: max total lifetime for tickets issued to this principal
  - expiration: when this entry expires
  - mod_date: when entry last modified
  - mod_name: principal that last modified this entry
  - flags: preauthentication?, forwardable?, proxiable?, etc.
  - password_expiration:
  - last_pwd_change:
  - last_success: time of last successful login

Human user master key derived from password and realm name.

- So even if A uses the same password in several realms, compromising A’s master key (but not password) in one realm does not compromise in another realm.

CryptoGraphic algorithms

Improves upon V4.

- Allows choice of crypto algorithms (but DES is still the only deployed version)
- Uses proper integrity protection (rather than pseudo-Juneman checksum).
- Details in text
Hierachy of realms

Allows KDC chains of authentication (unlike V4)

• Suppose KDCs A, B, C, where A, B share key, B,C share key, but A,C do not. Allows C to accept a ticket sent by A and generated by B.
• Each ticket includes all the intermediate KDCs
  o receiving KDC can reject ticket if ticket has a suspect intermediary

Evading off-line password guessing

V4 allows off-line password guessing:

• KDC does not authenticate TGT_REQ
• So B can ask KDC and get a TGT for A, and then do off-line password guessing.

In V5

• Req for TGT for A must contain $K_A\{\text{timestamp}\}$; so above attack not possible.
• KDC also does not honor requests for tickets to human users by others
  o Prevents logged-in B to ask KDC for a ticket to delegate) for A, on which it can do off-line password guessing.

Key inside authenticator

Suppose A and B share a session key $K_{AB}$ 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 encrypted by $K_{AB}$
Double TGT authentication

Suppose a running service B remembers its session key, say $S_B$, but has forgotten its master key (as with a human B after log in (or application program after initialization??)). Suppose principal A wants to access a running service B. No good for A to get from KDC a (regular) tkt encrypted with B’s master key.

Instead
- A asks B for TGT$_B$ and gets it.
- A sends KDC ticket request [“A to talk to B”, TGT$_A$, TGT$_B$]
- KDC
  - extracts session key $S_B$ from TGT$_B$ (encrypted with $K_{KDC}$)
  - creates session key $K_{AB}$
  - generates tkt$_B$ encrypted with $S_B$ [i.e., $S_B$ {‘A’, $K_{AB}$}] and sends to A

Motivated by XWINDOWS

<table>
<thead>
<tr>
<th>B user</th>
<th>B’s workstation</th>
<th>KDC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xwindow server</td>
<td>Xwindow client</td>
</tr>
<tr>
<td>1</td>
<td>▪ [B, passwd] xlogin</td>
<td>▪ request TGT$_B$ from KDC</td>
</tr>
<tr>
<td></td>
<td>▪ to Xwindow server</td>
<td>▪ obtain [S$_B$, TGT$_B$]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ forget passwd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ tell user B login succeeded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>▪ start Xwindow server</td>
</tr>
</tbody>
</table>
Note on NS chapter 15: PKI (Public Key Infrastructure)

PKI: infrastructure for obtaining public keys of principals
- examples: S/MIME, PGP, SSL, Lotus Notes, ...

Recall
- Certificate:
  [issuer C; // name of CA (principal) issuing the certificate
   subject X; // name of principal whose public key is being certified
   subject public key J; // certified public key of X
   expiry time T; // data/time when this certificate expires
   principals that subject can certify; // optional
   serial number; // optional
   signature; // C’s signature of the certificate]
- CRL:
  [issuer C; // name of CA issuing the CRL
   list of revoked certificates; // e.g., listed by serial number
   issue time T; // date/time when this CRL was issued
   signature; // C’s signature of the CRL]
- Certificate chain: // below, ‘cft’ is short for ‘certificate’
  - sequence of < (cft_1, crl_1), …, (cft_n, crl_n) > such that cft_i subject = cft_{i+1} issuer
  - cft_1 issuer is the trust anchor of the chain
  - cft_n subject is the target of the chain
  - chain is valid (my terminology) if for every i in 1, …, n:
    - cft_i is unexpired
    - crl_i is recent enough and does not include cft_i

PKI consists of
- Principal name space
  - usually hierarchical: usr@cs.umd.edu; www.cs.umd.edu/usr;
- Certification authorities (CAs): subset of the principals
- Repository for certificates and CRLs: (e.g., distributed repository like DNS)
  - searched by principals
  - updated by CAs
- Method for searching repository for a chain of certificates given
  - starting CA: trust anchor of the chain
  - ending subject: target of the chain
Updates in PKI should preserve the following desired property:
For every valid certificate chain \( CC \) in the repository
- if \( X \) is the subject and \( J \) the public key of a cft in \( CC \):
  - then \( J \) is \( X \)'s public key currently
    (more precisely, at the issue time of the earliest CRL in the prefix of \( CC \) upto cft)

Updates in PKI

Introduction of public key \( J \) of principal \( X \):
- request every CA that can certify \( X \) to issue a certificate for \([X, J]\);
  (request can be in-band only if \( X \) has “currently-valid public key”,
  i.e., a key that is currently certified and has not been compromised)
  each such CA does the following:
    check out the request (in/out-band??);
  if the request passes the CA’s checks
    then generate a certificate for \([X, J]\) and add to the repository
- if \( X \) is also a trust anchor to a set of principals
  inform every principal in the set of \([X, J]\)
  (can be done in-band only if \( X \) has currently-valid public key)

Revocation of public key \( J \) of principal \( X \):
- request every CA that has certified \([X, J]\) to revoke it in the CA’s next CRL;
  if request passes the CA’s checks, it includes \([X, J]\) in its next CRL
- if \( X \) is also a trust anchor to a set of principals
  inform every principal in the set that \([X, J]\) is not to be used
  (can be done in-band only if \( X \) has currently-valid public key)
PKI trust model
• defines where a user gets the trust anchors and what chain paths are legal

Monopoly:
• One CA, say R, trusted by all organizations and countries.
• Public key of R is the single trust anchor embedded in all software and hardware.
  o so every certificate has form [R, X, J, ...] signed by R
• Advantages:
  o simplicity: verification involves checking one certificate
• Disadvantages:
  o infeasible to change R’s public key if it gets compromised.
  o R can charge whatever it wants.
  o Security of entire world rests on R.
  o Bottleneck in obtaining certificates.

Monopoly + Registration Authorities (RAs)
• Like monopoly except
  o CA chooses other organizations (RAs) to interact with world
  o CA interacts only with RAs
• Has all the disadvantages of monopoly except CA is not a bottleneck.
• May be less secure because RAs may not be as careful as CA.

Monopoly + Delegated CAs
• Tree of CAs with one root CA
• Users can obtain certificates from a delegated CA rather than root CA.
• Verification involves chain of certificates with root CA as trust anchor

Oligarchy
• Multiple root CAs (trust anchors)
• Advantage: monopoly pricing is not possible
• Disadvantage:
  o More CAs to go wrong.
  o Choice/control over the CAs pre-installed in your program/hardware.
  o Adding new trust anchors possible, hence vulnerable to
    • adding malicious CA
    • modifying an existing trust anchor’s public key

Anarchy
• Each user independently chooses some trust anchors.
• Advantage: not dependent on other organizations.
• Disadvantage:
  o unorganized certificate space
  o not easy to find certification chains that are acceptable to user.
Name constraints

- Each CA is trusted for certifying only a subset of the principal name space.
- Usually hierarchical: i.e., CA x.y is trusted to certify x.y.*, but not x.z.

Top-down with name constraints

- Monopoly with delegated CAs except
  - each CA can only certify principals in its subtree (excluding itself).

Bottom-up with name constraints

- Hierarchical name space
- Down-links (as usual):
  - x.y certifies x.y.z
- Up-link (unusual!):
  - x.y.z certifies x.y
  - Allows user to use itself as trust anchor:
    e.g., chain [ x.y.z , x.y , x , x.p , x.p.q ]
- Cross-link: x.y certifies p.q,
  where x.y and p.q are CAs of two interacting organizations
  - Avoids having to go through root CA, hence smaller chains.
    - Can enhance performance.
    - Can improve security (if x.y and p.q more trustworthy than root)
  - Allows PKI to be deployed incrementally in (real-world) situation where
    root CA may not be present or may be needlessly expensive
- Cost/ease of obtaining certificates and revoking certificates??
  - There are now many more CAs…
  - Any principal can be its own trust anchor…

Certificates with relative names

Can of worms

Policies in certificates

- Which CAs are acceptable as trust anchors
- Which CAs are not acceptable in chains
- etc

END OF PKI trust models
Revocation
- Online revocation service (OLRS)
- Delta CRLs
- First valid certificate
- Good-lists vs bad-lists
- Boring…

Directories and PKI
- Directory (lookup service) is helpful but not essential
  - X can include its certificate when it sends a message to Y
  - Or Y can ask X for a certificate
- Store certificates in repository by subject or issuer

PKIX and X.509
X.509 certificates used in Internet PKIs

THAT’S IT…