Authentication Protocols

1. Notes from reviews:
   1. Why symmetric key not public key?
   2. Man-in-the-middle attacks?
   3. Single point of failure?
   4. What is Needham & Schroeder?
   5. How does single sign-in work?
   6. How use smart cards?
7. Attacks:
   i. If break into machine with a trusted user, can get access to tickets & session keys and impersonate that user
   ii. Manipulate TGS/AS to create long-lasting ticket that grants access even if account deleted, password changed, etc.
   iii. With realm admin, can read password database, forge long-lasting tickets as anyone
      1. If know TGS password, can create a ticket offline that will be accepted to grant additional tickets (subject to auditing)
      2. Cause: servers trust tickets, don’t do external validation
   iv. Password guessing: can do offline guessing against reply from AS exchange
      1. Solution: pre-authentication where send current time encrypted with client password
8. Use with SPNEGO?
9. Why not IP address; only principal names?
   i. Question: what does a name mean? Can IP addresses for a service change?
10. Proxy problem:
    i. Proxy tickets – allow destination to use with different address. Client request new ticket with client address of service it wants to be the proxy
    ii. Forwardable tickets: send a TGT, allow destination to use it to get new tickets on client behalf
11. Overhead/improvements since this paper?
    i. Not much – more options, public keys, etc.
12. ACLs in KDBM
    i. Idea: let users control their own accounts; if requestor name = target name, allow
ii. If not: look at ACL. Instance name is part of user name: mike/admin vs mike. So, have separate instance of account (separate password) for admin access.

13. Clock synchronization problem?

2. Authentication
   1. Locally: how is it done on a stand-alone machine?
      i. Users enters password
      ii. Hash password
      iii. Compare against password database

2. What can go wrong?
   i. Using strcmp() to compare strings without hashing
      1. Stops at first wrong character; timing can tell you where it stopped
      2. Turns \( O(64^{\ast}8) \) to \( O(64*8) \) if you can guess each character one at a time
   ii. Stealing password database & reversing hash
      1. Dictionary attack: take a set of possible passwords, hash them and compare
      2. Offline: if standard hash, can do once then \( O(1) \) lookup to compare passwords – very fast

3. Strengthening: salt
   i. Store random number with password, incorporate into hash
   ii. Store Hash(pwd,salt)
   iii. On login, read salt, compute hash(pwd,salt) and compare
   iv. Defeats precomputed dictionary attack: with 32 bit salt, would need 4 billion hashes per password

3. Distributed authentication problem statement
   1. Basic problem: you want to know who is talking to you over a network
   2. Assumptions: QUESTION: what are they
      i. Network totally untrustworthy
         1. Can sniff messages
         2. Can inject messages
         3. Can replay messages
         4. Can forge addresses (but harder)
      ii. Trust local hardware to perform correctly
         1. Who? You or sys admins
      iii. Trust the people you trust not to divulge secrets
         1. Not give away passwords

3. Constraints
i. Many servers \(\rightarrow\) had to share passwords with each one
ii. Low overhead \(\rightarrow\) cache information as much as possible, few round trips & messages
iii. Use long-term keys as little as possible

4. Basics of authentication
   1. Want to prove your identity
      i. Possibilities:
         1. Something you know – a secret
            a. Problem: somebody could snoop it off the network
            b. Could be guessed
            c. Can be found – if written down
         2. Something you have – a physical device (e.g. ID card)
            a. Magnetic strip card (credit card) – $50 to duplicate
            b. Smart card – does encryption on card
         3. Something you are – e.g. a fingerprint
            a. Problem: hard to measure precisely
               i. Still need a username to identify the person
               ii. Can have false matches
               iii. Can copy a fingerprint
               iv. Can subvert the sensor (e.g. hack the fingerprint reader)
               v. Hard to work with people following an injury

5. Trust
   1. Making things secure: options
      i. Verify everything yourself
      ii. Trust someone else to do part of the work
   2. Resolves down to: what do you trust?
      i. Essential problem in security: trust
         1. How do you establish trust?
         2. How do you know what to trust?
         3. Who makes the decision?
            a. SW producers?
            b. Individuals?
            c. Administrators?
   4. Inverse side: risk
      a. Different people / different environments have different risk tolerances
ii. Can you trust MS? Probably not to be malicious but not to be safe

iii. Can you trust open source? If you have the source code, is it trustworthy?
   1. git has had security holes
   2. People have injected attack code into open source repositories
   3. Not many people actually read code

3. For authentication, want to trust someone else to vouch for authority
   i. Local machine only has local set of passwords

6. What is reasonable in a network?
   1. What do you trust?
      i. Local machine
      ii. Local OS
      iii. Servers
      iv. Password databases
      v. Users
      vi. Administrators?
      vii. Network
   2. What do you trust it to do?
      i. Execute instructions
      ii. Reliably deliver packets
      iii. Not record everything you do and send it to an attacker
      iv. ...

7. Goals
   1. If B receives a message/connection from A, then
      i. A must be able to create a message that is comprehensible only to B – only B can use the message to determine who it came from
      ii. It must be evident to B that the message came from A

8. Basic NFS/Unix model
   1. Replicate user database to all machines
   2. Log in locally against passwd file
   3. Assume no network spoofing, send UID across network to servers

9. Basic protocol for shared secrets
   1. Challenge response
      i. Client requests challenge
      ii. Server sends challenge
      iii. Client encrypts with pwd, sends to server
      iv. Server verifies
2. Idea: client proves it knows something without exposing what it is.
3. Idea: based on shared knowledge of client and server
4. Problems
   i. Man in the middle
      1. If you can make client connect to you, you can forward messages to server
   ii. Requires server & client to have a shared secret
      1. Either
         a. password database per server
         b. or server forwards challenge/response to auth servers
      2. Used in Windows up to NT 4
   iii. Does not establish a **new** shared secret between client and server; only secret is password
      1. if want to encrypt data, need to do it with the password
   iv. So: not really scalable

10. Scaling protocol: what if have star-formation, an auth server knows all the keys
    1. Needham & Schroeder
       i. Protocol:
          1. \( A \rightarrow AS: A,B, \text{Nonce-a} \)
          2. \( AS \rightarrow A: (\text{Nonce-a}, B, CK, (CK,A)KB)KA \)
          3. \( A \rightarrow B: (CK,A)KB \)
          4. \( B \rightarrow A: (\text{Nonce-B})CK \)
          5. \( A \rightarrow B: (\text{Nonce-B} - 1) CK \)
       ii. points:
          1. Essence: server performs a ticket protocol as compared to a list protocol, AS does a list protocol.
          2. Nonce: preserves freshness – can tie response to request
          3. \( (\text{Nonce-B})CK \) is an authenticator – it demonstrates that B can decrypt CK, and is used to verify that A knows CK as well
          4. Using CK has benefits – can now be used to encrypt additional data between A and B
             a. encrypting a lot of data with a key means key is vulnerable; would like to be able to have new keys all the time
       iii. QUESTION: What do you learn at the end?
          1. Is the user allowed to access the service?
          2. What the name of the user is?
iv. Problems
1. A must enter password each time it talks to a new server
2. No limit on how long message 3 is good for – if can break in and steal CK

v. QUESTION: How do you establish trust with the auth server?
1. Must be out of band – need to have shared secret to start with

11. 4-scenes approach to Kerberos
1. Kerberos goals: same as AFS
   i. 1000s of workstations
   ii. print, email, file, etc. servers
   iii. Make sure user A can’t impersonate user B
      1. Act as B to a service
2. Version 1:
   i. Goals:
      1. Service can identify clients
   ii. Option 1: ask clients for passwords
      1. Every server needs a copy of password database, or users need a different password for each server
   iii. Option 2:
      1. give clients & servers passwords
      2. create “authentication service” that knows all client and server passwords
      3. Protocol:
         a. Client -> AS: password, service name
         b. AS -> Client: ticket = (service name, username)Ksvc
   4. Problem?: sniffing ticket allows reuse from another user/machine
iv. Option 3: add client address
1. Protocol: add client address
   a. Client -> AS: username, password, service name
   b. AS -> Client: ticket = (service name, client_addr, username)Ksvc
   c. Can cache & reuse tickets on client
2. Problems:
   a. IP address spoofing (didn’t’ exist in Kerberos era)
   b. Need client password to get a new ticket
i. Each new service, need to re-enter passwords, or store in clear on client

c. Password goes in clear to AS, can be spoofed

v. Option 4: add TGS

1. Goals:
   a. Users only enter password at beginning of session
   b. Password never sent in clear over net

2. Solution:
   a. Ticket-granting service: allows you to get a ticket to grant you more tickets (layer of indirection) instead of using password each time

3. Protocol:
   a. Client->AS: TGS
   b. AS-> Client: (TGS ticket(TGS, client_addr, username)Ktgs)Kuser
      i. If decrypts, user knows password
   c. Client->TGS: TGS ticket, service name
   d. TGS -> Client: service ticket = (service name, client_addr, username)Ksvc

4. Problems:
   a. Tickets can be used if found lying around
   b. Tickets last forever
   c. Can still do IP spoofing

vi. Option 5: add timestamps

1. Protocol:
   a. Client->AS: TGS
   b. AS-> Client: (TGS ticket(TGS, client_addr, username, lifetime)Ktgs)Kuser
      i. If decrypts, user knows password
   c. Client->TGS: TGS ticket, service name
   d. TGS -> Client: service ticket = (service name, client_addr, username, lifetime)Ksvc
   e. Lifetime is 8 hours

2. Problem:
   a. Can still sniff, re-use within 8 hour window

vii. Option 5: add authenticators

1. Problem with old version:
a. Service uses 3 tests:
   i. Can service decrypt ticket
      1. Was generated by TGS/AS
   ii. Has ticket expired
      1. Not replayed too long in future
   iii. Does the address in ticket match sender of ticket?

b. What proved?
   i. No sense that user is who is in ticket – no proof

c. Solution: put session keys inside tickets, use authenticators to prove knowledge of session key

2. Protocol:
   a. Client->AS: username, TGS
   b. AS -> client: {sess_key, {sess_key, tgs,client_addr, username, lifespan}Ktgs}Kuser
      i. If client can decrypt, has ticket & session key
   c. Client -> TGS: (TicketTGS, {username,address}Ksess_key, svc_name
   d. TGS -> client: (ticket_svc, svc_sess_key)Ksess_key
   e. Client -> service: TicketSVC, authenticator = {username, address,timestamp}svc_sess_key
      i. Proves could decrypt sess key that goes with ticket
      ii. Timestamp proves authenticator is fresh
         1. Limit to some # minutes
         2. Cache all authenticators from that period

3. Problem: svc_name sent in TGS request could be replaced by any other SVC, lead to talking to wrong
   a. No mutual authentication – no proof server is real

   viii. Option 6: add service names into tickets & responses
       1. Kerberos

12. Kerberos
i. Goals

1. Want to minimize the number of messages
2. Want to avoid using user passwords for encrypting much data
   a. Things humans can remember make poor keys
3. Allow servers to have stronger keys, so can use for encrypting more data
4. Want to support signing & sealing messages
5. Want to allow multiple administrative domains
6. Want mutual authentication
   a. Server knows who client is
   b. Client knows who server is
7. **QUESTION**: How does the client know what server it wants to talk to?
   a. Replicated servers (I.E google cluster)
   b. Redirects
   c. Servers that contact other servers on user’s behalf
   d. **ANSWER**:
      i. Use the name user types in
      ii. Use DNS to translate & reverse translate
      iii. Same problem as RPC name service

ii. Model

1. Authentication server has keys for users and the TGS
2. Ticket Granting Service has keys for services
3. Users only use their password to get a TGT
4. Users and servers have multipart name
   a. Service name
   b. Instance name
   c. Realm name
   d. i.e. swift.faculty@cs.wisc.edu, printer/copier-7@cs.wisc.edu
5. 3 levels of security
   a. Connection – authenticates connection, relies on lack of IP spoofing for security
      i. **QUESTION**: when reasonable? Cloud?
   b. Integrity – messages not altered
   c. Privacy – messages can’t be altered or understood
   d. **QUESTION**: Why choose one over the other?
      i. Performance – signing cheaper
      ii. Management – easier to debug your network without encryption
iii. Protocol

1. Tickets = a basic credential
   a. \((\text{server name, client name, time stamp when issued, lifetime, session key})^{K_{\text{server}}}\)
      i. Safe for client to have, because cannot decrypt
   b. When use, must prove you know the session key with an “authenticator”
      i. \((\text{client name, client addr, timestamp})^{K_{\text{session}}}\)
      ii. Proves knowledge of inside of ticket via session key

2. AS protocol - Used for obtaining ticket-granting tickets
   a. \(A \rightarrow AS: A, TGS == (\text{client name, server name})\)
   b. \(AS \rightarrow A: (K_c,tgs, (T_c,tgs)^{K_{tgs}})^{K_c}\)
      i. \(T_c,tgs = \text{ticket for client to tgs}\)
   c. QUESTION: what does this prove?
      i. Problem: response does not include server’s name in a way that client can read it
      ii. Problem: no nonce tying request to response (could replay response to other server)
      iii. REALITY:
         1. Ticket not encrypted – not needed
         2. Encrypted portion include server name, nonce for freshness
      iv. NOTE: once you get a TGT, client can discard password
         1. Means client is a bit more secure – all authentication material will time out

3. TGS protocol: used for obtaining tickets to services
   a. \(A \rightarrow TGS: s, (T_c,tgs)^{K_{tgs}}, \{\text{Authenticator}\}^{K_c,tgs}\)
   b. \(TGS \rightarrow A: ((T_c,s)^{K_s}, K_c,s)^{K_c,tgs}\)
   c. Notes
      i. Send request including ticket, authenticator proving knowledge of session key
      ii. Server replies with new ticket encrypted with session key
      iii. Same deal as before on replay, nonces, and names

4. AP protocol: used for authenticating to a service
   a. \(A \rightarrow S: (\text{Authenticator})^{K_c,s}, (T_c,s)^{K_s}\)
   b. \(S \rightarrow A: (\text{Timestamp – 1})^{K_c,s}\)
   c. Issues:
i. Client and server now share a session key than can use to further encrypt data

ii. Session key re-used as long as ticket is re-used
   1. Can negotiate new session key by having server return:
      a. (Timestamp -1, Ktmp)Kc,s
      b. Also verifies server identity to client

iii. Replay attacks:
   1. somebody can sniff somebody can sniff network, replay request.
   2. If time in authenticator is still valid (due to clock skew) could be valid
   3. Solution: cache all authenticators (or latest time per client) seen in valid window

2. Show summary:
   i. AS; TGS; AP

ii. Practical considerations
   1. Servers are replicated for availability and load balancing
      a. Single master replication
      b. Password changes propagated slowly – how do you handle this?
         i. client can try backup first (to spread load) then try server
   2. Cross realm authentication
      a. Each password/name database is a “realm”
         i. e.g. cs.wisc.edu
      b. Can cross realms:
         i. Allow TGS for one realm to have a key for another realm
         ii. Issue a cross-domain TGT for another domain
         iii. Other domain accepts this as if it was a TGT from it self
      c. Must include “transited” field – the list of realms the client transited
      d. QUESTION: why?
         i. Server trusts its own AS, but may not trust others. This lets it make a call. But how?
         ii. Can only allow clients from realms it directly trust (no transitivity)

3. Security depends on time synchronization
a. How do you do this?
b. How do you do this secure?

4. Naming
   a. How do you handle DNS aliases?
   b. Canonicalize! But how securely?
   c. QUESTION: What if you don’t want to trust DNS?

5. Key storage: how can you trust clients to securely store keys?
   a. Why use secret keys instead of public keys?

6. Authorization
   a. Kerberos does authentication – how do you figure out what a user is allowed to do? How do you get a Unix user id?

7. Addresses in tickets. How do you handle NATs?

8. Authenticators: have to store all of them
   a. How do you do this in a server that uses TCP/IP and forking a process per client? Hard to share state

9. Password guessing
   a. AS-REP is encrypted in user’s password. Allows off-line guessing attack (similar to obtaining passwd file)
   b. Solution: pre-authenticate by having client encrypt current time

10. Non-networked clients
    a. What if you want to authenticate to DHCP to get an address