6.858 Lecture 13
Kerberos

Administrivia

Quiz review today (Actual quiz next Wednesday.)
Post your final project idea by tomorrow.

Kerberos setting:

- Distributed architecture, evolved from a single time-sharing system.
- Many servers providing services: remote login, mail, printing, file server.
- Many workstations, some are public, some are private.
- Each user logs into their own workstation, has root access.
- Adversary may have his/her own workstation too.
- Alternatives at the time: rlogin, rsh.
- Goal: allow users to access services, by authenticating to servers.
- Other user information distributed via Hesiod, LDAP, or some other directory.
- Widely used: Microsoft Active Directory uses the Kerberos (v5) protocol

What's the trust model?

- All users, clients, servers trust the Kerberos server.
- No apriori trust between any other pairs of machines.
- Network is not trusted.
- User trusts the local machine.

Kerberos architecture:

- Central Kerberos server, trusted by all parties (or at least all at MIT).
- Users, servers have a private key shared between them and Kerberos.
- Kerberos server keeps track of everyone's private key.
- Kerberos uses keys to achieve mutual *authentication* between client, server.
    - Terminology: user, client, server.
    - Client and server know each other's names.
    - Client is convinced it's talking to server and vice-versa.
- Kerberos does not provide authorization (can user access some resource).
    - It's the application's job to decide this.

Why do we need this trusted Kerberos server?

- Users don't need to set up accounts, passwords, etc on each server.

Overall architecture diagram

```
                                 +-----------------------+
                  c, tgs         |                       |
  [ User: Kc ]  <-------->  [ Kerberos ]                 |
       ^      \                  |           Database:   |
       |       \                 |             c: Kc     |
       V        \    s           |             s: Ks     |
 [ Server: Ks ]  \-------->   [ TGS ]                    |
                                 |         KDC           |
                                 +-----------------------+
```

Basic Kerberos constructs from the paper:

```
Ticket, T_{c,s} = { s, c, addr, timestamp, life, K_{c,s} }
  [ usually encrypted w/ K_s ]
Authenticator, A_c = { c, addr, timestamp }
  [ usually encrypted w/ K_{c,s} ]
```

Kerberos protocol mechanics.

- Two interfaces to the Kerberos database: "Kerberos" and "TGS" protocols.
- Quite similar; few differences:
  - In Kerberos protocol, can specify any c, s; client must know K_c.
  - In TGS protocol, client's name is implicit (from ticket).
  - Client just needs to know K_{c,tgs} to decrypt response (not K_c).
- Where does the client machine get K_c in the first place?
  - For users, derived from a password using, effectively, a hash function.
- Why do we need these two protocols? Why not just use "Kerberos" protocol?
  - Client machine can forget user password after it gets TGS ticket.
  - Can we just store K_c and forget the user password? Password-equivalent.

Naming.

- Critical to Kerberos: mapping between keys and principal names.
- Each principal name consists of (name, instance, realm)
  - Typically written name.instance@realm
- What entities have principals?
  - Users: name is username, instance for special privileges (by convention).
  - Servers: name is service name, instance is server's hostname.
  - TGS: name is 'krbtgt', instance is realm name.
- Where are these names used / where do the names matter?
  - Users remember their user name.
  - Servers perform access control based on principal name.
  - Clients choose a principal they expect to be talking to.
    - Similar to browsers expecting specific certificate name for HTTPS
- When can a name be reused?
  - For user names: ensure no ACL contains that name, difficult.

    - For servers (assuming not on any ACL): ensure users forget server name.
    - Must change the key, to ensure old tickets not valid for new server.

Getting the initial ticket.

- "Kerberos" protocol:
    - Client sends pair of principal names (c, s), where s is typically tgs.
    - Server responds with { K_{c,s}, { T_{c,s} }_{K_s} }_{K_c}
- How does the Kerberos server authenticate the client?
    - Doesn't need to -- willing to respond to any request.
- How does the client authenticate the Kerberos server?
    - Decrypt the response and check if the ticket looks valid.
    - Only the Kerberos server would know K_c.
- In what ways is this better/worse than sending password to server?
    - Password doesn't get sent over network, but easier to brute-force.
- Why is the key included twice in the response from Kerberos/TGS server?
    - K_{c,s} in response gives the client access to this shared key.
    - K_{c,s} in the ticket should convince server the key is legitimate.

General weakness: Kerberos 4 assumed encryption provides message integrity.

- There were some attacks where adversary can tamper with ciphertext.
- No explicit MAC means that no well-defined way to detect tampering.
- One-off solutions: kprop protocol included checksum, hard to match.
- The weakness made it relatively easy for adversary to "mint" tickets.
- Ref: http://web.mit.edu/kerberos/advisories/MITKRB5-SA-2003-004-krb4.txt

General weakness: adversary can mount offline password-guessing attacks.

- Typical passwords don't have a lot of entropy.
- Anyone can ask KDC for a ticket encrypted with user's password.
- Then try to brute-force the user's password offline: easy to parallelize.
- Better design: require client to interact with server for each login attempt.

General weakness: DES hard-coded into the design, packet format.

- Difficult to switch to another cryptosystem when DES became too weak.
- DES key space is too small: keys are only 56 bits, 2^56 is not that big.
- Cheap to break DES these days ($20--$200 via https://www.cloudcracker.com/).
- How could an adversary break Kerberos give this weakness?

Authenticating to a server.

- "TGS" protocol:
    - Client sends (s, {T_{c,tgs}}_{K_tgs}, {A_c}_{K_{c,tgs}})
    - Server replies with { K_{c,s}, { T_{c,s} }_{K_s} }_{K_{c,tgs}}
- How does a server authenticate a client based on the ticket?
    - Decrypt ticket using server's key.
    - Decrypt authenticator using K_{c,s}.
    - Only Kerberos server could have generated ticket (knew K_s).

    - Only client could have generated authenticator (knew K_{c,s}).
- Why does the ticket include c? s? addr? life?
    - Server can extract client's principal name from ticket.
    - Addr tries to prevent stolen ticket from being used on another machine.
    - Lifetime similarly tries to limit damage from stolen ticket.
- How does a network protocol use Kerberos?
    - Encrypt/authenticate all messages with K_{c,s}
    - Mail server commands, documents sent to printer, shell I/O, ..
    - E.g., "DELETE 5" in a mail server protocol.
- Who generates the authenticator?
    - Client, for each new connection.
- Why does a client need to send an authenticator, in addition to the ticket?
    - Prove to the server that an adversary is not replaying an old message.
    - Server must keep last few authenticators in memory, to detect replays.
- How does Kerberos use time? What happens if the clock is wrong?
    - Prevent stolen tickets from being used forever.
    - Bound size of replay cache.
    - If clock is wrong, adversary can use old tickets or replay messages.
- How does client authenticate server? Why would it matter?
    - Connecting to file server: want to know you're getting legitimate files.
    - Solution: send back { timestamp + 1 }_{K_{c,s}}.

General weakness: same key, K_{c,s}, used for many things

- Adversary can substitute any msg encrypted with K_{c,s} for any other.
- Example: messages across multiple sessions.
    - Authenticator does not attest to K_{c,s} being fresh!
    - Adversary can splice fresh authenticator with old message
    - Kerberos v5 uses fresh session key each time, sent in authenticator
- Example: messages in different directions
    - Kerberos v4 included a direction flag in packets (c->s or s->c)
    - Kerberos v5 used separate keys: K_{c->s}, K_{s->c}

What if users connect to wrong server (analogue of MITM / phishing attack)?

- If server is intercepting packets, learns what service user connects to.
- What if user accidentally types ssh malicious.server?
    - Server learns user's principal name.
    - Server does not get user's TGS ticket or K_c.
    - Cannot impersonate user to others.

What happens if the KDC is down?

- Cannot log in.
- Cannot obtain new tickets.
- Can keep using existing tickets.

Authenticating to a Unix system.

- No Kerberos protocol involved when accessing local files, processes.
- If logging in using Kerberos, user must have presented legitimate ticket.
- What if user logs in using username/password (locally or via SSH using pw)?
  - User knows whether the password he/she supplied is legitimate.
  - Server has no idea.
- Potential attack on a server:
  - User connects via SSH, types in username, password.
  - Create legitimate-looking Kerberos response, encrypted with password.
  - Server has no way to tell if this response is really legitimate.
- Solution (if server keeps state): server needs its own principal, key.
  - First obtain user's TGS, using the user's username and password.
  - Then use TGS to obtain a ticket for server's principal.
  - If user faked the Kerberos server, the second ticket will not match.

Using Kerberos in an application.

- Paper suggests using special functions to seal messages, 3 security levels.
- Requires moderate changes to an application.
  - Good for flexibility, performance.
  - Bad for ease of adoption.
  - Hard for developers to understand subtle security guarantees.
- Perhaps a better abstraction: secure channel (SSL/TLS).

Password-changing service (administrative interface).

- How does the Kerberos protocol ensure that client knows password? Why?
  - Special flag in ticket indicates which interface was used to obtain it.
  - Password-changing service only accepts tickets obtained by using K_c.
  - Ensure that client knows old password, doesn't just have the ticket.
- How does the client change the user's password?
  - Connect to password-changing service, send new password to server.

Replication.

- One master server (supports password changes), zero or more slaves.
- All servers can issue tickets, only master can change keys.
- Why this split?
  - Only one master ensures consistency: cannot have conflicting changes.
- Master periodically updates the slaves (when paper was written, ~once/hour).
  - More recent impls have incremental propagation: lower latency (but not 0).
- How scalable is this?
  - Symmetric crypto (DES, AES) is fast -- O(100MB/sec) on current hardware.
  - Tickets are small, O(100 bytes), so can support 1M tickets/second.
  - Easy to scale by adding slaves.
- Potential problem: password changes take a while to propagate.
- Adversary can still use a stolen password for a while after user changes it.

- To learn more about how to do replication right, take 6.824.

Security of the Kerberos database.

- Master and slave servers are highly sensitive in this design.
- Compromised master/slave server means all passwords/keys have to change.
- Must be physically secure, no bugs in Kerberos server software,
    - no bugs in any other network service on server machines, etc.
- Can we do better? SSL CA infrastructure slightly better, but not much.
    - Will look at it in more detail when we talk about browser security / HTTPS.
- Most centralized authentication systems suffer from such problems.
    - globally-unique freeform names require some trusted mapping authority.

Why didn't Kerberos use public key crypto?

- Too slow at the time: VAX systems, 10MHz clocks.
- Government export restrictions.
- Patents.

Network attacks.

- Offline password guessing attacks on Kerberos server.
    - Kerberos v5 prevents clients from requesting ticket for any principal.
    - Must include { timestamp }_{K_c} along with request, proves know K_c.
    - Still vulnerable to password guessing by network sniffer at that time.
    - Better alternatives are available: SRP, PAKE.
- What can adversary do with a stolen ticket?
- What can adversary do with a stolen K_c?
- What can adversary do with a stolen K_s?
    - Remember: two parties share each key (and rely on it) in Kerberos!
- What happens after a password change if K_c is compromised?
    - Can decrypt all subsequent exchanges, starting with initial ticket
    - Can even decrypt password change requests, getting the new password!
- What if adversary figures out your old password sometime later?
    - If the adversary saved old packets, can decrypt everything.
    - Can similarly obtain current password.

Forward secrecy (avoiding the password-change problem).

- Abstract problem: establish a shared secret between two parties.
- Kerberos approach: someone picks the secret, encrypts it, and sends it.
- Weakness: if the encryption key is stolen, can get the secret later.
- Diffie-Hellman key exchange protocol:
    - Two parties pick their own parts of a secret.
    - Send messages to each other.
    - Messages do not have to be secret, just authenticated (no tampering).
    - Two parties use each other's messages to reconstruct shared key.
    - Adversary cannot reconstruct key by watching network messages.

- Diffie-Hellman details:
    - Prime p, generator g mod p.
    - Alice and Bob each pick a random, secret exponent (a and b).
    - Alice and Bob send (g^a mod p) and (g^b mod p) to each other.
    - Each party computes (g^(ab) mod p) = (g^a^b mod p) = (g^b^a mod p).
    - Use (g^(ab) mod p) as secret key.
    - Assume discrete log (recovering a from (g^a mod p)) is hard.

Cross-realm in Kerberos.
- Shared keys between realms.
- Kerberos v4 only supported pairwise cross-realm (no transiting).

What doesn't Kerberos address?
- Client, server, or KDC machine can be compromised.
- Access control or groups (up to service to implement that).
- Microsoft "extended" Kerberos to support groups.
    - Effectively the user's list of groups was included in ticket.
- Proxy problem: still no great solution in Kerberos, but ssh-agent is nice.
- Workstation security (can trojan login, and did happen in practice).
    - Smartcard-based approach hasn't taken off.
    - Two-step authentication (time-based OTP) used by Google Authenticator.
    - Shared desktop systems not so prevalent: everyone has own phone, laptop, ..

Follow-ons.
- Kerberos v5 fixes many problems in v4 (some mentioned), used widely (MS AD).
- OpenID is a similar-looking protocol for authentication in web applications.
    - Similar messages are passed around via HTTP requests.