Distributed Systems Security II

15-440

Take-home points

- What does using public-key for our authentication buy us?
  - Compare kerberos (needham-schroeder) and SSL with a certificate authority
  - Metrics: Scaling, robustness, timeliness
- Motivate & understand perfect forward secrecy and diffie-hellman
- A touch of research: Perspectives SSL auth vs. CA auth

Remember digital signatures

- From last time...
- Shared key crypto with key K_{AB}:
  - Intuition: Hash them together
  - $HMAC(K_{AB}, m) = H((K..) | H(K ... | m))$
- Public key crypto with K_A, K^{-1}_A:
  - Intuition: "signing" is encryption using the private key. But pub key operations are expensive: To make it practical, hash first so that the message is small, fixed-size.
  - $E(K^{-1}_A , H(m))$

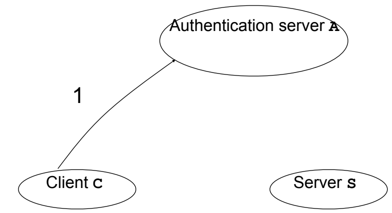
Today: Auth protocols

- Needham-Schroeder - basis of Kerberos authentication
- Goal: Secure, usable authenticaiton system without needing public-key cryptography
- Idea: Everyone shares a key with a trusted third party server
- If A wants to talk to B, on demand, that server generates key K_{AB} and shares it with (and only with) A and B.

Needham-Schroeder and Kerberos

- In following diagrams:
  - Client **C** initiates a connection to server **S**
    - Authentication server **A** generates "session key" K_{SC} for them to use to talk to each other. Only A, S, and C will know this key.
  - Each entity shares a private key with the authentication server:
    - C and A share a secret key K_{AC}
    - **S** and A share secret key K_{AS}
  - Nobody else knows either of those two keys.

Needham-Schroeder and Kerberos

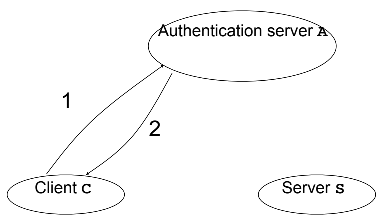


- Messages:

  1: C to A: C,S,n

A nonce: a "number used once." In Kerberos this is usually the time.

Needham-Schroeder and Kerberos



- Messages:

  1: C to A: C,S,n

  2: A to C: $\{K_{cs},S,n\}_{KAC}$ $\{C,S,K_{cs},t_1,t_2\}_{KAS}$

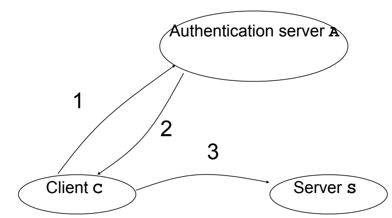
start and end time for K_{CS}

the session key

K_{CS},S,n encrypted with private key K_{AC}

C,S,K_{CS},t_1,t_2 encrypted with secret key K_{AS}

Needham-Schroeder and Kerberos



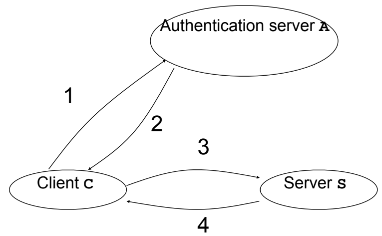
- Messages:

  1: C to A: C,S,n

  2: A to C: $\{K_{cs},S,n\}_{KAC}$ $\{C,S,K_{cs},t_1,t_2\}_{KAS}$

  3: C to S: $\{request,n',\ldots\}_{Ksc}$ $\{C,S,K_{cs},t_1,t_2\}_{KAS}$

Needham-Schroeder and Kerberos



- Messages:
  - 1: C to A: C,S,n
  - 2: A to C: $\{K_{cs},S,n\}_{Kc}$ $\{C,S,K_{cs},t_1,t_2\}_{Ks}$
  - 3: C to S: $\{request,n',\ldots\}_{Ksc}$ $\{C,S,K_{cs},t_1,t_2\}_{Ks}$
  - 4: S to C: $\{n',response,\ldots\}_{Ksc}$

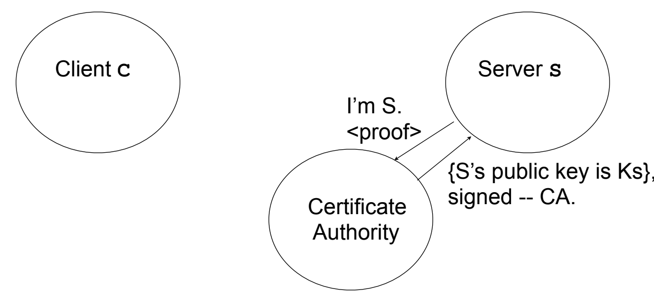
History

- The first version of N-S didn't have the nonce/timestamp.
  - It was vulnerable to a "replay attack"
- *Replay Attack*: An attacker can sniff the traffic and re-play an old value.
  - They don't have to know what it means, necessaril
  - In N-S's case, if an attacker compromised an *old* key, they could use a replay attack to still use that old key.
- Usual warning: Needham and Schroeder are (were - Needham died in 2003) really smart guys. And they goofed this protocol... twice. The vulnerabilities survived in one of the most widely-examined crypto protocols from 1978 until 1995!

Analysis

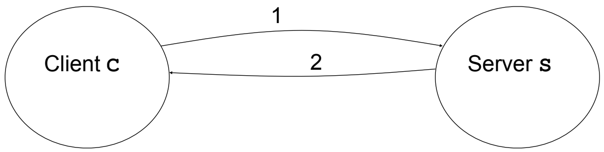
- Everyone trusts the auth server
  - It can read, modify, etc., all traffic. It knows all the keys.
- All connections require a conversation with the auth server.
  - If the auth server goes down, nobody can talk.
- Auth server must store all keys.
  - And must be online and thus exposed to potential compromise.
- Let's fix some of these... with public keys! :)

Simplified SSL/TLS



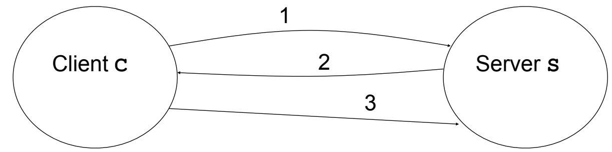
- Step 1: offline, the server gets a "certificate" from the CA that binds its identity to a key it generated.
  - You do this when you configure the server...
- Client C gets the CA's public key (pre-baked in to the software?)

Simplified SSL/TLS



- Online, for C to talk to S...
  1: request
  2: s's X.509v3 certificate, containing its public key signed by a certificate authority

Simplified SSL



- Messages:
  1: request
  2: s's X.509v3 certificate, containing its public key signed by a certificate authority
  3: Client verifies the certificate using the certificate authority's public key, sends session key for subsequent communication (encrypted with s's public key)

Note: Actual TLS protocol is a lot more complicated - it can negotiate different versions, cipher suites, etc...

Analysis

- Public key lets us take the trusted third party *offline*:
  - If it's down, we can still talk!
  - But we trade-off ability for fast *revocation*
    - If server's key is compromised, we can't revoke it immediately...
    - Usual trick:
      - Certificate expires in, e.g., a year.
      - Have an *on-line* revocation authority that distributes a revocation list. Kinda clunky but mostly works, iff revocation is rare. Clients fetch list periodically.
- Better scaling: CA must only sign once... no matter how many connections the server handles.
- If CA is compromised, attacker can trick clients into thinking they're the real server. But...

Forward secrecy

- In N-S, if auth server key K_{AS} is compromised a year later,
  - from the traffic log, attacker can extract session key (encrypted with auth server keys).
  - attacker can decode all traffic retroactively.
- In SSL, if CA key is compromised a year later,
  - Only *new* traffic can be compromised. Cool...
- But in SSL, if *server*'s key is compromised...
  - Old logged traffic can still be compromised...

Diffie-Hellman Key Exchange

- Different model of the world: How to generate keys between two people, securely, no trusted party, *even if someone is listening in*.

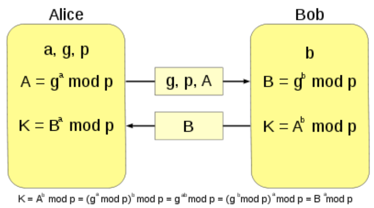


image from wikipedia

- This is cool. But: Vulnerable to man-in-the-middle attack. Attacker pair-wise negotiates keys with each of A and B and decrypts traffic in the middle. No authentication...

Authentication?

- But we already have protocols that give us authentication!
  - They just happen to be vulnerable to disclosure if long-lasting keys are compromised later...
- Hybrid solution:
  - Use diffie-hellman key exchange *with* the protocols we've discussed so far.
  - Auth protocols prevent M-it-M attack if keys aren't yet compromised.
  - D-H means that an attacker can't recover the real session key from a traffic log, even if they can decrypt that log.
  - Client and server discard the D-H parameters and session key after use, so can't be recovered later.
- This is called "perfect forward secrecy". Nice property.

Big picture, usability, etc.

- public key infrastructures (PKI)s are great, but have some challenges...
  - Yesterday, we discussed how your browser trusts many, many different CAs.
  - If any one of those is compromised, an attacker can convince your browser to trust their key for a website... like your bank.
  - Often require payment, etc.
- Alternative: the "ssh" model, which we call "trust on first use" (TOFU). Sometimes called "prayer."

Perspectives approach