Computer and Network Security
CMSC 414

AUTHENTICATION

Udaya Shankar
shankar@cs.umd.edu

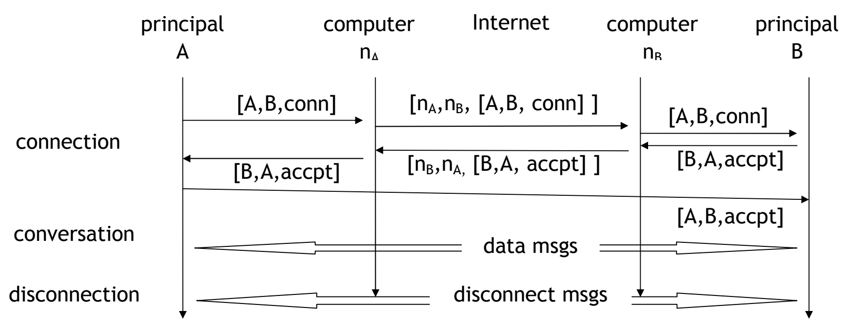
Authentication Overview (NS chapter 9)

Context:
- Large set of principals attached to an open channel (eg, Internet).
- Each principal repeatedly
  - attempts to initiate a connection (i.e., session) with a specified principal
  - upon successful connection establishment, exchanges messages
  - closes the connection
  - waits for an arbitrary (but bounded) time

Authentication is about ensuring
- When a principal A assumes it is connected to a principal B, A is indeed exchanging messages with B, and not some attacker C.
- When principal A assumes confidentiality/integrity of the message exchange, this is indeed the case.

Principal can be a human or an executing computer program
  - Programs can use **high-quality** secrets (eg, from space of 2^{128})
  - Human principals are restricted to **low-quality** secrets (eg, space of 2^{32}) and cannot do cryptographic operations.
  - When we say a program principal A **assumes** it is connected to B, we mean that A's program's variables indicate that A is connected to B.

Typical authentication scenario



Goal: achieves following inspite of attacks
- Connection establishment:
  - A authenticates B: [B,A,accpt] msg sent by B in response to [A,B,conn]
  - B authenticates A: [A,B,accpt] . . . A . . . [B,A,conn]
  - Simultaneously establish a shared secret (session key) for conversation
- Conversation: encryption / MAC
- Disconnection: A and B close their connection and forget any session key

Types of Attacks

An authentication protocol must identify the attacks it is supposed to handle
- network attacker
- end-point attacker
- dictionary attack
- …

An authentication mechanism cannot protect against all attacks, eg,
- overrun (take over) a human principal
- overrun memory while program principal is doing login authentication

Attackers can span multiple classes

Attackes can sequentially mount attacks of different classes
- Eg, record encrypted conversation; much later learn session key

Types of Attacks (contd)

Network-based attacks (roughly in order of increasing difficulty)
- Sending messages with wrong values in fields:
  - spoofing: C at n_c sends messages with sender id as [A]
  - changing "reject" to "accept"
  - spoofing: C at n_c sends messages with sender addr/id as [n_A,A]
  - …
- Eavesdropping: observing messages in the channel.
  - Easy in WLANs and LANs (because of broadcast nature)
  - Not easy in wired point-to-point links (but doable)
    - tap router ports
    - compromise route computation algorithm
- Intercepting messages, changing them, resending them.
  - Relatively easy in WLANs and LANs (because of broadcast nature)
  - Not easy in point-to-point (but doable)

Types of Attacks (contd)

End-host based attacks (roughly in order of increasing difficulty)
- Principal C says it is principal A on a computer n_A (eg, public workstation)
  - online dictionary attack
- Read data on hard disk (or back-up tapes) of n_A or A
  - obtain old keys (encrypted or plaintext) password files, …
  - obtain current keys (encrypted or plaintext) password files, …
  - offline dictionary attack on encrypted passwords
- Overrun computer n_A
  - while A is not at n_A
  - while A is at n_A
- Read data in memory of n_A while A is executing (unlikely)
- Overrun a (human or program) principal
  - mail client, web browser

Types of Attacks (contd)

Dictionary attacks (aka password-guessing attacks)
- Finding a secret by searching through a space of possible secrets
- Doable only if the space is small enough (given reasonable time/resources)
- A secret from a small space is said to be **low-quality**
- A secret from a large space is said to be **high-quality**
- Examples:
  - 128-bit key from a decent random number generator is high-quality
  - 20-bit key from a decent random number generator is low-quality
  - Passwords, and keys obtained from them, are low-quality (typically)
- **Online** dictionary attack: need to interact with authenticator at every guess
- **Offline** dictionary attack: interacts with authenticator just once

Three types of authentication

- **Password-based authentication**
  - Authenticating oneself by showing a secret password to the remote peer (and to the network)
  - Always vulnerable to eavesdropping attack
  - Always vulnerable to online dictionary attack
    - Usually protection: limit frequency of incorrect password entries

- **Address-based authentication**
  - authenticating oneself by using a physically-secured terminal/computer
  - Conceptually similar to password-based authentication ??

- **Cryptography-based authentication**
  - authenticating oneself by showing evidence of a secret key
    to the remote peer (and to the network)
    but without exposing the secret to the peer (or to the network)
  - Note: secret key can be obtained from a password

Password-based authentication

- A authenticates itself by supplying a password.
- Always vulnerable to eavesdropping attack and online dictionary attack

Approach 1:

A (passwd pw_A)	n_A	channel	n_B	B (passwd file with [X, pw_X] for each X)
• enter [A, B, pw_A]				
	• send [n_A,n_B,A,B,pw_A]			
			• check rcvd [A, pw_A] against passwd file • match authenticates A; msgs from n_A until logout assumed to be from A	

- Vulnerable to eavesdropping and to online dictionary attack
  - Defense against latter: limit number of successive failed attempts
- Vulnerable to exposure of password file (overrun of n_B or B)

Password-based authentication (contd)

Approach 2:

- Like approach 1 except B's password file has entries (X, hash(pw_X)) for each X

A (passwd pw_A)	n_A	channel	n_B	B (passwd file with [X, hash(pw_X)] for each X)
• enter [A, B, pw_A]				
	• send [n_A,n_B,A,B,pw_A]			
			• check hash(rcvd pw_A) against passwd file entry for A • match authenticates A	

- Vulnerable to eavesdropping and to on-line dictionary attack (as before)
- Vulnerable to password file exposure but requires **offline** dictionary attack
  - Defense 1: store (X, salt, hash(pw_X, salt))
  - Defense 2: store (X, $encrypt_K(pw_X)$) where K is high-quality key maintained only in B's memory and not hard disk (i.e., manually entered when B is activated).

Password-based authentication (contd)

Handling situation where A may interact with many servers

- Store A's password in every server that A may access.
  - Disadvantage: handling changes to password.
  - Disadvantage: All password files need to be protected well.
- Store A's password in a special **authentication node.**
  - Server authenticates A by checking A's password with authentication node (and presumably forgetting password after authenticating A).
  - Disadvantage: performance bottleneck.
  - Advantage: single node to protect

Address-based authentication

- A uses only a special set of computers
- A is authenticated by the address (network, link level, etc) of its computer.
- Valid if
  - Access to special computers is well-guarded
  - Network is protected against spoofing/interception of messages
- Examples:
  - Unix: os-wide /etc/hosts.equiv file, per-user .rhosts file.
  - VMS: PROXY database
  - Early main-frame machines accessed by dumb terminals.
  - Operator console on many workstations (eg, single-user mode in Linux)
- Conceptually like password-based authentication except that "password" is now associated with a physical device (eg, network interface card).

Cryptographic authentication

- A authenticates itself to B by performing a cryptographic operation on a quantity composed of a part supplied by B and a secret shared by A and B.
- Because operation is cryptographic, the secret is not disclosed by eavesdropping.

Limitations if A is human
- A can only remember low-quality secret, ie, password.
- A cannot do cryptographic operations.
- So A inputs password into computer n_A which converts password to key. Hence vulnerable to overrun of n_A.

Transforming password to secret-key-crypto key
- Obtain key by (say) hashing password (and, for AES, taking specified 128 bits).
- Not ok for public key crypto, where keys have constraints. Here is an(wacko?) approach to obtain an RSA key:
  - Use password as seed to specified pseudo-random number generator, and choose first two primes generated.

Using password to get high-quality secret (eg, public-key-crypto key) from directory service

Use password to decrypt a high-quality key kept in a directory service.
- Let K_A be A's high-quality key.
- Let K_{Apw} be the low-quality) key obtained from A's password (eg, by hashing)
- Directory service stores enc(K_A, K_{Apw}) (ie, K_A encrypted by K_{Apw}).
- Computer n_A gets [A, enc(K_A, K_{Apw})] from directory service, K_{Apw} from A's password, and decrypts to get K_A

- Is this vulnerable to offline dictionary attack?
  - Guess candidate password, say cpw.
  - Obtain candidate low-quality key cK_{Apw} (e.g., by hashing cpw).
  - Obtain candidate high-quality key cK_A by decrypting enc(K_A,K_{Apw}) with cK_A. But cannot decide whether cK_A is correct because K_A has no structure. (Note: in RSA, encrypt [d], not [d,n] because latter has structure)
  - But it is vulnerable with a bit more work in some cases, eg,
    - If A uses a session key encrypted with K_A, use cK_A to obtain candidate session key, and check if it can decrypt conversation.
    - If A's signature on a document produced using K_A is available, check if cK_A matches the signature.

Protecting against eavesdropping and server passwd file exposure (spfe)

Easy with public key crypto
- A has private key.
- B stores A's public key (so exposing B's database does no damage).
- Authentication:
  - B sends a random value to A
  - A encrypts using A's private key and sends back
  - B checks received value using A's public key

Handling spfe (but not eavesdropping) with hash/secret-key crypto
- B stores hash of A's password
- Authentication:
  - A sends password to B
  - B compares hash of recieved password with stored hash

Handling eavesdropping (but not spfe) with secret-key crypto
- A and B share a secret K_{AB} (eg, A's password).
- Authentication:
  - A sends [A,login] to B
  - B sends random number R to A
  - A responds with K_{AB}{R} // NOTE: "K_{AB}{R}" short for "enc(R, K_{AB})"

Handling both with secret-key crypto
- Lamport's hash scheme

Lamport's Hash Scheme (NS Chapter 12)

- One-way authentication (B authenticaes A); ie, assumes B is not spoofed.
- A stores password.
- B stores for A:
  - n: positive integer, initially say 1000; number of logins remaining
  - nhpw: n-fold hash of pw; ie, $hash^n$(pw)

A (stores password pw)	**B** (stores (A: n, nhpw))
send [A,B,conn]	
	send [B,A,n]
x ← $hash^{n-1}$(pw) send [A,B,x]	
	if hash(x) = nhpw then A authenticated n ← n–1 nhpw ← x
When n becomes 1, need to reset with new pw and n	

Enhancement with salt:
- Initially: A chooses salt; B stores [A, n, salt, $hash^n$(pw | salt)]
- Login: B responds with [n, salt]; A responds with $hash^{n-1}$(pw | salt)
- To use same pw with many servers: salt = random number | server id.

Lamport's hash scheme (contd)

Reset option 1:

- A chooses new[n, nhpw] and sends it to B unencrypted.
  - Adequate against an attacker that can eavesdrop, intercept, spoof?
  - Adequate given assumption that B-to-A authentication is not needed?

Reset option 2:

- A sends new[n, nhpw] encrypted by a key obtained via Diffie-Helman.
  - Is this any better wrt to the attackers?

Small n attack:

- C impersonates B's network address and waits for A to login
- C responds with m smaller than current n and thus gets $hash^m(pw)$ from A
- C can now impersonate A (for n–m logins)

Lamport's Hash without workstation

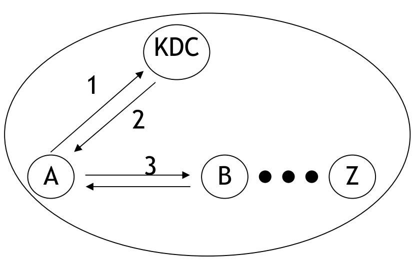
- Instead of just password, A has $hash^i(pw)$ for i = 1, 2, ⋯, n-1 written down
- At each login, A uses last entry and crosses it out.
- Not vulnerable to "small n" attack.
- Is this any different from writing down a high-quality key?

SKEY: Internet deployed version of Lamport's hash

Scaling to network of N principals

- Straightforward approach:
  - Distinct key for every pair of principals.
  - Not scalable:
    - N^2 storage cost at each node
    - N cost for adding new principal
- Use hierarchy of trusted intermediaries
  - **KDC (key distribution center)** in secret-key crypto
  - **CA (certification authority)** in public-key crypto

KDC: single-domain case



- KDC is a host in network; serves shared-secret keys
- Every principal X in domain shares a key K_X with KDC (off-line)
- When A wants to talk to B, it gets a ticket from KDC (online steps 1, 2, 3)

A	KDC	B
send [A, KDC, B, conn]		
	generate session key K_{AB} generate tkt_{AB}= [K_B{A, B, K_{AB}}] (**ticket**) send [KDC, A, K_A{A, B, K_{AB}}, tkt_{AB}]	
send [A, Y, B,conn, tkt_{AB}]		
		decrypt tkt_{AB} and get K_{AB}
< --- authentication betweeen A and B using K_{AB} ---- >		

KDC: single-domain case (cont)

- Advantages of KDC:
  - Adding new principal: one interaction between principal and KDC
  - Revocation of principal: deactivate principal's master key at KDC
- Disadvantages of KDC:
  - KDC can impersonate anyone to anyone.
    KDC compromise makes the whole network vulnerable.
  - KDC failure means no new sessions can be started.
  - KDC can be a performance bottleneck.
  - Last two can be alleviate by having KDC replicas, but
    - need to protect all replicas
    - when a principal's master key is changed, need to sync replicas

KDCs for multi-domain case

Case 1: A in domain (with KDC X) wants to talk to B in domain (with KDC Y), and X and Y share a key, say K_{X-Y}.

A	X (KDC of D_1)	Y (KDC of D2)	B
send [A,X, conn to B in D_2]			
	generate session key K_{A-Y} generate tkt_{A-Y} = [K_{X-Y}{A, X, K_{A-Y}}] send [X,A, K_A{A, Y, K_{A-Y}}, tkt_{A-Y}]		
send [A,Y, conn to B in D_2, tkt_{A-Y}]			
		generate session key K_{A-B} generate tkt_{A-B} = [K_{B-Y}{A, B, K_{A-B}}] send [Y,A, K_{A-Y}{A, B, K_{A-B}}, tkt_{A-B}]	
send [A,B, K_{A-B}{A, B, conn}, tkt_{A-B}]			

KDCs for multi-domain case (contd)

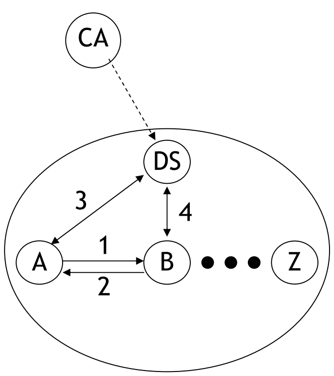
Case 2: KDCs chain from source to destination

- In a large internetwork with many domains, unlikely that every two domains will have a shared key.

- But if there is a sequence of domains D_1, D_2, ⋯, D_N such that for every i, KDC of D_i and KDC of D_{i+1} have a shared key then A of D_1 can securely obtain a session key to talk to B of D_N:
  - Let X_i be the KDC of D_i
  - A talks to X_1 and gets [session-key, $ticket_{A-X2}$] to talk to X_2
  - A talks to X_2 and gets [session-key, $ticket_{A-X3}$] to talk to X_3
  - and so on until
  - A talks to X_N and gets [session-key, $ticket_{A-B}$] to talk to B

- How does A get the sequence X_1, X_2, ⋯, X_N.
  - Static hierarchy with additional links (perhaps cached) for efficiency.
  - Good if A also passes along the sequence of domains to be traversed, so that B can see whether it trusts every KDC on the chain.

CA: single-domain case



- CA is a host but need not be networked; generates certificates (signed public keys) and CRLs (certificate revocations)
- Online directory server (DS) periodically gets certificates and CRLs from CA
- DS serves certificates and CRLs to anyone (online steps 3, 4)
- Every principal X in domain
  - generates a public-key pair
  - gets its public key signed by CA (certificate)
  - gets CA's public key
  (all off-line)
- When A wants to talk to B,
  A shows B its certificate and CRL
  B shows similar documents to A
  (online steps 1, 2)

CA: single-domain case (contd)

- Each principal has a public-key pair.
  Remembers its own private key and CA's public key.
- CA generates **certificate** (signed public key) for each principal X:
  - [(serial no, X, $pubkey_X$, expdate),
    $privkey_{CA}${(serial no, X, $pubkey_X$, expdate)}].
- Certificates are publicly disseminated (e.g., at directory services).

- A authenticates B as follows (ignoring certificate revocation):
  - Obtain certificate for B from anywhere, typically from B.
  - If certificate not expired and signature verifies (using CA's public key), then A has B's public key.
  - A sends challenge and expects challenge encrypted by B's private key, after which A and B settle on a session key.

- Advantages
  - CA does not need to be online or networked, so can be more secure.
  - CA crash does not stop new sessions from starting until expiration date.
  - Certificates need not be secured (except for deletion of certificates).
  - Compromised CA cannot decrypt conversations (unlike KDC). But it can serve false public keys and thus impersonate any principal.

CA: handling revocation

- Certificate revocation is more complex than in KDC.
- CA periodically (eg, hourly) issues CRL (Certificate Revocation List)
  - signed {issue time, list of certificates revoked at issue time}

- A authenticates B (in presence of CRL) by obtaining (typically from B)
  - a certificate for B that has not expired (as above), and
  - a CRL that does not have B and was issued sufficiently recently, eg, at the start of the current period.
  - A sends a challenge and awaits challenge encrypted by B's private key, after which A and B settle on a session key.

- X.509 format for certificate and CRL
  - Certificate =
    [user name, user public key, expiration time, serial number,
    CA's signature on entire contents of certificate]
  - CRL = [issue time, list of serial numers of unexpired revoked certificates]

CAs for multi-domain case

Case 1: A in domain with CA X wants to talk to B in domain with CA Y, and X and Y have certificates for each other.

A	**X directory service**	**Y directory service**	**B**
• Gets from X's directory service a certificate for Y signed by X; A can verify certificate because A has X's public key; so A now has Y's public key. • Gets from Y's directory service a certificate for B signed by Y; A can verify certificate because A now has Y's public key; so A now has B's public key • A can now send messges to B encrypted with B's public key			

CAs for multi-domain case (contd)

Case 2: CA chain from source to destination

- In a large internetwork with many domains, unlikely that the CAs of every two domains will have a certificate for each other.

- But if there is a sequence of domains D_1, D_2, ⋯, D_N such that for every i, directory services of D_i and D_{i+1} have certificates for each other signed by their CA's then A of D_1 can securely obtain the public key of B of D_2 by iterating:
  - Let X_i be the CA of D_i
  - A gets certificate for X_2 signed by X_1
  - A gets certificate for X_3 signed by X_2
  - and so on until
  - A gets certificate for X_N signed by X_{N-1}
  - A gets certificate for B signed by X_N

Session keys

Session keys
- Protect the data exchange after a connection is established
- Should be different from long-term shared key used for authentication
  - so long-term key does not "wear out" (offline crypto attack)
- Should be unique for each session
  - If compromised, only affects data sent in that session.
  - Can be given to relatively untrusted software
- Session key should be forgotten after session ends

Delegation or authentication forwarding
- If A, when logged into B, wants to access C (eg, printer), then B needs to authenticate itself as A to C.
  - A can log into C explicitly (too much trouble)
  - A can give B its password (too risky)
  - A can give B a ticket (called **delegation** or **authentication forwarding**) with
    - types of access allowed by B (eg, A's print queue)
    - expiry time (typically short)

Establishing session key with secret-key authentication (NS Ch 12)

- Consider A and B with shared key K_{AB}.
  During authentication, A and B have exchanged challenges, eg:
  - R_1 (in one-way auth)
  - R_1, R_2 (in two-way auth)

- Session key can be R_1 and/or R_2 encrypted by a specfied function g of K_{AB}, eg,
  - $g(K_{AB})\{R_1\}$ or $(g(K_{AB}))\{R_1 \oplus R_2\}$
  - $g(K_{AB})$ is $K_{AB}+1$, $K_{AB}-1$, $-K_{AB}$, etc

Attack: if C obtains K_{AB} later, C can decrypt (recorded) conversation.

- Session key should not be $g(R_1)$ or $g(R_1,R_2)$ encrypted by K_{AB}, eg, $K_{AB}\{g(R_1)\}$.
  Otherwise, later C can impersonate B, send $g(R_1)$ as a challenge to A, get back $K_{AB}\{g(R_1)\}$, and decrypt earlier conversation between A and B.
  Defense: include sender id in challenges.

- Session key can obtained by Diffie-Hellman after/during authentication (the Diffie-Hellman exchange messages are encrypted by K_{AB}).
  Then even if C obtains K_{AB} later, it still cannot decrypt conversation.

Establishing session key with public-key authentication (NS Ch 12)

- A chooses random R as session key and sends $\{R\}_B$ to B.
  Attack: C spoofs A (after authentication) and choose its own R_1 as session-key.
  So important to have R be part of authentication.

- A chooses R as session key and sends $[\{R\}_B]_A$
  Here C cannot inject spurious R_1 as session-key
  Attack: If C later obtains B's private key, C can extract R and decrypt conversation.

- A picks R_1, B picks R_2, they exchange $\{R_1\}_B$ and $\{R_2\}_A$, set $R_1 \oplus R_2$as session key.
  Attack: Here C has to overrun both A and B to obtain session key.

- Session key can be obtained by Diffie-Hellman after/during authentication (the Diffie-Hellman exchange messages are encrypted or signed).
  Then even if C overruns A and B, it still cannot decrypt conversation.

Authentication of People (KPS 10)

Constraints when authenticating human:
- Can only remember **low-quality** secret (eg, 10 letter "pronounceable" password).
- Cannot perform cryptographic operations.

Human authentication based on one or more of
- What you know: password
- What you have: authentication tokens, eg,
  - physical keys, ATM card
- What you are: biometric features, eg,
  - fingerprint, voice recognition, retina scan

Password limitations
- Eavesdropping
- Online dictionary attack
  - defense: limit number of attempts after which user must talk to admin
    - problem: vandal can easily lock up accounts (denial-of-service)
  - defense: limit speed of attempts
- Exposure of password file on server
  - Doing offline dictionary attack if password file is hashed.
- Exposing passwords in email, script files, etc.

Authentication of People (contd)

Good password, ie, random 128-bit, not feasible

- 20 random digits
- 11 random chars (from 0-9, a–z, A–Z, couple of punctuation marks)
- Computer-generated random pronounceable password
  - Case insensitive: 4.5 bits of randomness per character
  - Every third character a vowel, 6 vowels: 2.5 bits of randomness per vowel
  - Requires 16 characters
- Human-generated passwords
  - About 2 bits of randomness per character
  - So require about 32 character password
- If password is too good, users write it down

Workable approach
- "pass-phrase" with intentional misspelling, punctuation marks, symbols (eg, $ for S), odd capitalization, etc.

Authentication of People (contd)

Login Trojan Horse to capture passwords
- Leave program running on public terminal that imitates login prompt
  - gets password from naive user and attempts to exit inconspicuously
    - eg, exit with “login failed” message
    - better yet: runs virtual OS for duration of user session

- Defenses by OS/hardware:
  - Have special prompt symbol at any input field by non-login program
  - Allow only login screen to fill entire display
  - Non-mappable key to interrupt any running program
    - eg, alt-ctrl-del (but often OS allows remapping of this)
  - Display number of unsuccessful login attempts since last successful login.
  - **Any defense fails given a sufficiently naive user**

Initial distribution of passwords needs to be secure

Passwords can also be used for non-login purposes (protecting individual files)

Authentication of People (contd)

Authentication tokens: physical device that a person carries around:

Magnetic strip cards
- Credit cards, debit cards, id cards, money card, etc
- Can hold high-quality secret and other data (usually read-only)
- If card has picture or signature, then also serves as biometric check by human.

Smart card (embedded CPU and memory)
- can hold high-quality secret
- memory can be password protected
- can do cryptographic operations (challenge/response)

Advantages, disadvantages, features
- Tokens can be lost or stolen (unless it is attached/embedded in user)
  - So usually needs to be augmented with password
  - When token is lost, need an override that is usually not much less convenient than the override for “I forgot my password”
- Requires custom hardware (key slot, card reader, etc) on every access device
  - exception is cryptographic calculator (or readerless smart card)

Authentication of People (contd)

Cryptographic calculator (or readerless smart card)

- Smart card that <u>does not require special hardware</u>.
- Has display and keyboard for human interaction
  - User enters password to unlock device
  - User enters challenge into device and reads cryptographic response
- Time-based alternative
  - User enters password to unlock device
  - Card displays encryption of current time, which user enters as authentication information.
  - Authenticating computer checks that result is valid
    - Needs to check for all possible current times within allowed clock drift.
  - Advantages:
    - Saves half the typing
    - Works with password “form-factor” authentication protocols

Authentication of People (contd)

Biometric authentication devices
- Retinal scanner
  - scans blood vessels in back of your eye
  - expensive and “psychologically threatening” (look into laser device)
- Iris scanner
  - Less intrusive than retinal scanner (can use camera several feet away).
- Fingerprint reader
  - devices available but automation has not been successful for many years
- Face recognition
  - not intrusive but not very accurate; susceptible to false negatives
- Handprint readers
  - More false positives than fingerprint readers, but cheaper/fewer problems
- Voiceprints
  - Cheap and can be as accurate as fingerprinting
  - Can be defeated with tape recording
  - False negatives (voice change due to illness)
- Keystroke timing
  - False negatives (injury)
- Signature
  - Not accurate based only on static signature
  - Accurate if also based on timing info

Security Handshake Pitfalls (NS chapter 11)

Assume A initiates connection to B.

Can classify the authentication protocols along following features:

- One-way authentication:
  - B authenticates A (eg, login) or
  - A authenticates B (server B with public key, client A w/o public key)
- Mutual-authentication:
  - B authenticates A and A authenticates B
- Secret-key crypto vs Public-key crypto

One-Way Authentication

Solution 1.1: one-way auth, secret-key (K_{AB})

A	B
send [A,B, conn]	
	send challenge [B,A,R]
send response [A,B, f{K_{AB}, R}]	

Note

- Response f{K_{AB},R} is a keyed-hash of R or R encrypted with K_{AB}
- Challenge R must be new (a **nonce**) so that f{K_{AB},R} has not been sent before (by A or by B) and hence has not been seen by attacker.
- If challenge R is obtained from a clock or a counter and if B may have received past msgs m to which it sent f{K_{AB},m} responses (eg, another authentication protocol with A using K_{AB}) then
  - B must ensure that challenge R is not among these msgs, or
  - response should also indicate the sender (eg, f{K_{AB},A,R})
- These problems are not there if R is obtained from a random number generator.

Question: Would these attacks, if successful, yield session key?

Some vulnerabilities:

- If K_{AB} is derived from password, an eavesdropper can do offline dictionary attack.
- If attacker gets B's password file, it can impersonate A
  - Protecting password file is harder if B is replicated or A uses same password on different servers.

Solution 1.2: one-way auth, secret-key (K_{AB})

A	B
send [A, B, conn]	
	send challenge [B,A, K_{AB}{R}]
send [A,B, R]	

Note
- Requires challenge to be reversable (ie, encryption, not keyed-hash).
- R should not only be a nonce but unpredictable (ie, randomly generated).
  - Eg, if R is obtained from a counter, an attacker can impersonate A because it would know that the next challenge generated by B is R+1.

Vulnerabilities: as in solution 1.1 plus the following:
- If K_{AB} derived from password and R has structure, then a spoofer (w/o eavesdropping) can get K_{AB}{R} and do offline dictionary attack.
  - Note: R is randomly generated and need not have structure.

Feature

If A and B have clocks that are within D seconds of each other and R has a timestamp (in addition to the random number), then this also authenticates B to A in the following sense:
- A assured that K_{AB}{R} message was originally sent by B within last D seconds
- A not assured that K_{AB}{R} was sent in response to its [A,B,conn] msg
  - Can be fixed by including a nonce in [A,B,conn] and in R.

Solution 1.3: one-way auth, secret-key (K_{AB}), timestamp-based

Assuming A and B have clocks that are within D seconds of each other.

A	B
send [A,B, conn, K_{AB}{ts}]	
	B decrypts, checks that ts within D

Note
- Single transmission suffices, no handshake needed
- B does not need to maintain state per active connection

Vulnerable
- Replay attack within clock skew D
  - defense: B remembers ts sent by A within last D seconds (requires state)
- Replay attack if K_{AB} used with multiple servers
  - defense: include server id along with ts
    May not be doable if servers are replicas of B (with same external id)
- B's clock being set back

If encryption is replaced by keyed-hash, B has much more work
- B has to get keyed-hash of every possible value in D and compare.
- Can overcome by A including unencrypted ts in conn msg. (Is this as secure?)

Solution 1.4: one-way auth, public-key (open challenge, signed response)

A	B
send [A,B, conn]	
	send challenge [B,A, R]
send [A,B, $[R]_A$] // $[R]_A$ is R encrypted with A's private key	

Note
- B's pw file contains A's public key; can be readable (but not modifiable)
- Need to ensure that R has distinct structure that is not used for signing messages

Solution 1.5: one-way auth, public-key (encrypted challenge, open response)

A	B
send [A,B, conn]	
	send challenge [B,A, $\{R\}_A$]
	($\{R\}_A$ is R encrypted with A's public key)
send [A,B, R]	

Note
- B's pw file contains A's public key; can be readable (but not modifiable)
- Need to ensure that R has distinct structure that is not used for sending confidential messages to A
- Why is ok to send response R in the open, instead of say $\{R\}_B$

Mutual (two-way) Authentication (A initiates connection to B)

Solution 2.1: two-way auth, secret key (K_{AB})

	A	B
1	send [A,B, conn]	
2		send challenge [B,A, R_1]
3	send response [A,B, f{K_{AB}, R_1}]	
4	send challenge [A,B, R_2]	
5		send response [B,A, f{K_{AB}, R_2}]

Note
- Consists of two 2-way handshakes
- Messages 3 and 4 can be combined into one message
- Vulnerable to B's passwd file being read
- If K_{AB} obtained from passwd, vulnerable to offline dictionary attack
  - by attacker who can eavesdrop
  - by attacker who can impersonate B
    - Impersonating server B is harder than impersonating client A (assuming server is always connected whereas client is momentary)
- Interchanging order of R_1 and R_2 introduces further vulnerability (below)

Solution 2.2: solution 2.1 with R_1-R_2 order interchanged

	A	B
1	send [A,B, conn, R_2]	
2		send [B,A, R_1, f{K_{AB},R_2}]
3	send [A,B, f{K_{AB}, R_1}]	

Note

- Reduces solution 2.1 to one 3-way handshake
- As usual, vulnerable to B's passwd file being read
- Usual offline dictionary attack if C eavesdrops and K_{AB} obtained from passwd
- If C can spoof A, then C can do offline dictionary attack (without eavesdropping)

Solution 2.2 vulnerable to reflection attack

	C	B
1	send [A,B, conn, R_2]	
2		send [B,A, R_1, f{K_{AB},R_2}]
1'	send [A,B, conn, R_1]	
2'		send [B,A, S_1, f{K_{AB},R_1}]
3	send [A,B, f{K_{AB}, R_1}]	
	C has successfully impersonated A to B	

Possible defenses:

- B remembers R_1 and does not accept it (difficult with replicated servers)
- R has structure indicating sender of challenge (but then offline dictionary attack)
- Use different keys for each direction:
  - K_{AB} (for A → B) and K_{BA} (for A ← B)
  - K_{BA} can be predictably related to K_{AB} [eg, K_{AB}+1, K_{AB}-1, -K_{AB}, or $K_{AB} \oplus (F0F0...F0)_{16}$]

Thumb-rule: Initiator should be first to authenticate itself

Solution 2.3: two-way auth, secret key, timestamps

	A	B
1	send [A,B, conn, f(K_{AB}, ts)]	
2		send [B,A, f(K_{AB}, ts +1)]

Note

- One 2-way handshake suffices
- Msg 1 assures B that msg was generated by A and sent within clock skew
- "ts+1" can be replaced by any predicatable function of ts
  - response should include structure indicating sender (to defend against replay attack), or
  - B must remember timestamp values ts and ts+1 (to defend against replay attack)

Solution 2.4: two-way auth, public keys

	A	B
1	send [A,B, conn, $\{R_2\}_B$]	
2		send [B,A, R_2, $\{R_1\}_A$]
3	send [A,B, R_1]	

Note

- More rugged than secret-key: not vulnerable to overruning B.
- Is it necessary to encrypt response R_1?
- Human A has to obtain its private key and B's public key (already discussed):
  - Directory service supplies A's private key encrypted by A's pwd
  - B supplies B's public key signed by A's private key
  - etc

Solution 2.5: two-way auth, public keys, variant of solution 2.4

	A	B
1	send [A, B, conn, R_2]	
2		send [B,A, $[R_2]_B$, R_1]
3	send [A,B, $[R_1]_A$]	

Extensions for dynamic contex

Dynamic context:
- users join and leave domains
- users do not share pre-assigned keys
- users rely on KDCs / CAs / directory services
- users change passwords
- replicated KDCs
- etc

New attacks become relevant:
- attacker with an old password of a user (trying to impersonate user)
- others?

New situations have to be handled:
- user A presents user B a ticket issued under old password of B
- user A contacts a KDC that still has an old password of A
- etc

Authentication with KDC mediator

A	KDC	B
send [A,KDC, conn to B]		
	generate session key K_{AB} generate tkt_{AB} = [K_B{A, B, K_{AB}}] send [KDC,A, K_A{K_{AB}}, tkt_{AB}]	
send [A,B, conn, tkt_{AB}]		
<------ A and B do mutual authentication using K_{AB} ---------> (example follows)		
		send [B,A, R_1]
send [A,B, R_2, K_{AB}{R_1}]		
		send [B,A, K_{AB}{R_2}]
<--- A and B use K_{AB} (or derivative, eg, (K_{AB}+1){$R_1 \oplus R_2$} as session key data --->		

Note:
- Even if C is spoofing A, C cannot get access to K_{AB}.
- Is authentication between A and KDC needed (or is that already done above)?
- Even if C is spoofing KDC, C cannot give a K_{AB} that B will accept.

Needham-Schroeder Protocol

Below N_1, N_2, N_3 are nonces.

	A	KDC	B
1	send [A,KDC, conn B, N_1]		
2		generate session key K_{AB} generate tkt_{AB} = [K_B{A, B, K_{AB}}] send [KDC,A, K_A{N_1, B, K_{AB}, tkt_{AB}}]	
3	send [A,B, tkt_{AB}, K_{AB}{N_2}]		
4			send [B,A, K_{AB}{N_2–1, N_3}]
5	send [A,B, K_{AB}{N_3–1}]		
	<---- use K_{AB} (or derivative, eg, (K_{AB}+1){$N_2 \oplus N_3$} as session key data --->		

Needham-Schroeder (cont)

- Nonce N_1 used to assure A that msg 2 is response by KDC to msg 1
  If N_1 not present, C with an old password of B can impersonate B to A:
  - C records above exchange (refer to them as old msgs 1, 2, 3, 4, 5)
  - C steals K_B; B changes key
  - C decrypts tkt_{AB} and get K_{AB}
  - C waits until A initiates connection to B
  - C intercepts A’s new msg 1, responds with old msg 2 (= K_A{B, K_{AB}, tkt_{AB}})
  - A responds with new msg 2 (= [tkt_{AB} , K_{AB}{new N_2}] to B
  - C intercepts, responds with K_{AB}{new N_2 - 1} (C knows K_{AB})
- Msg 2: id B encrypted by K_A ensures that C cannot replay old KDC reply to C (i.e., KDC reply to request by C to talk to B)
- Msg 2: no need to doubly encrypt tkt_{AB}

Needham-Schroeder (cont)

- If EBC is used (instead of CBC) and each nonce fits in an encryption block, then C can impersonate A to B with reflection attack
  - C eavesdrops and gets msgs 3 and 4
  - Later C replays msg 3
  - B replies with $K_{AB}\{N_2 - 1, N_4\}$ where $N_4 \neq N_3$
  - C needs to get $K_{AB}\{N_4 - 1\}$, which it does as follows:
    - C replays msg 3 with $K_{AB}\{N_4\}$ replacing $K_{AB}\{N_2\}$ and gets $K_{AB}\{N_4 - 1\}$ from B
    - Replacing EBC with CBC makes attack not possible (but then there is no need for N_3–1; can just use N_3)

Needham-Schroeder (cont)

Vulnerability if N_1 sequential

1. Attacker C overhears $N_1 = n$ during normal session between A and B

	A	KDC	B
1	send [A,KDC, conn B, N_1 = n]		
2		generate session key K_{AB} generate ticket $T_{AB} = [K_B\{A, B, K_{AB}\}]$ send [KDC,A, $K_A\{N_1, B, K_{AB}, T_{AB}\}$]	
3	send [A,B, T_{AB}, $K_{AB}\{N_2\}$]		
4			send [B,A, $K_{AB}\{N_2–1, N_3\}$]
5	send [A,B, $K_{AB}\{N_3–1\}$]		
	<------------- A and B exchange data, close ---------->		

Needham-Schroeder vulnerability if N_1 sequential (cont)

2. Attacker C learns K_B, spoofs A to KDC with N_1 = n+1 as follows

	attacker C	KDC	B
6	send [A,KDC, conn B, N_1 = n+1]		
7		generate session key J_{AB} generate ticket $S_{AB} = [K_B\{A, B, J_{AB}\}]$ send [KDC,A, $K_A\{N_1, B, J_{AB}, S_{AB}\}$] (rcvd by C)	

3. C steals K_B. B changes its key.
   C waits for A to connect to B, then impersonates KDC and then B

	A	attacker C	B
8	send [A,KDC, conn B, N_1 = n+1] (intercepted by C)		
9		send [KDC,A, $K_A\{N_1, B, J_{AB}, S_{AB}\}$] (replay msg 7)	
10	send [A,B, S_{AB}, $J_{AB}\{L_2\}$] (intercepted by C)		
		C decrypts S_{AB} (encrypted using (old) K_B) and obtains J_{AB}	
	<---- C can now complete the authentication and impersonate B ----->		

Needham-Schroeder vulnerable to old password exposure

If C gets A's master key (say K_A) and A changes it (to say J_A),
C can still impersonate A to B (because B never talks to KDC).

	A	KDC	B
1	send [A,KDC, B, N_1]		
2		generate session key K_{AB} generate $tkt_{AB} = [K_B\{A, B, K_{AB}\}]$ send [KDC,A, $K_A\{N_1, B, J_{AB}, tkt_{AB}\}$]	
3	send [A,B, tkt_{AB}, $K_{AB}\{N_2\}$]		
4			send [B,A, $K_{AB}\{N_2–1, N_3\}$]
5	send [A,B, $K_{AB}\{N_3–1\}$]		

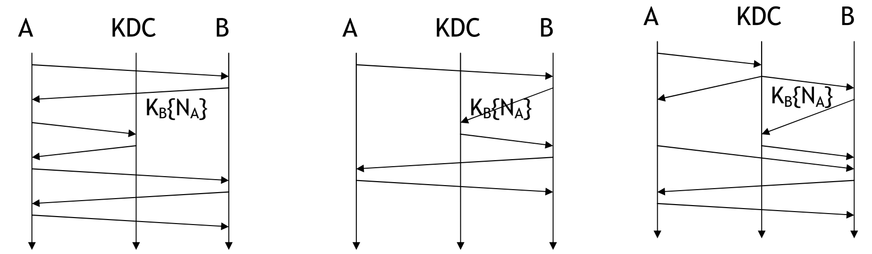
C records above. Then C obtains K_A. Then A changes master key to J_A ($\neq K_A$).

C	B
send [A,B, tkt_{AB}, $K_{AB}\{M_2\}$]	
	send [B,A, $K_{AB}\{M_2–1, M_3\}$]
send [A,B, $K_{AB}\{M_3–1\}$]	

Needham-Schroeder vulnerable to old password exposure (cont)

Fix:
B sends a nonce encrypted by K_B in response to A's connection request, and looks for the nonce in the ticket.

Several ways to include such a B-KDC interaction:



Expanded Needham-Schroeder:	Otway-Rees:	Not good:
▪ 7 msgs	▪ 5 messages	▪ requires KDC to match up messages

Expanded Needham-Schroeder: requires two additional messages

	A	KDC	B
1a	send [A,B, conn]		
1b			send [B,A, $K_B\{N_B\}$]
1	send [A,KDC, conn B, N_1, $K_B\{N_B\}$]		
2		generate session key K_{AB} generate tkt_{AB} = [$K_B\{A, B, K_{AB}, N_B\}$] send [KDC,A, $K_A\{N_1, B, K_{AB}, tkt_{AB}\}$]	
3	send [A,B, tkt_{AB}, $K_{AB}\{N_2\}$]		
4			send [B,A, $K_{AB}\{N_2–1, N_3\}$] (as before)
5	send [A,B, $K_{AB}\{N_3–1\}$] (as before)		
	<---- A and B establish data session key (eg, $(K_{AB}+1)\{N_2 \oplus N_3\}$ --->		

Otway-Rees authentication protocol

Does mutual authentication and handles ticket invalidation in 5 messages

	A	KDC	B
1	generate nonces N_A and N_C send [A,B, N_C, $K_A\{N_A,N_C,A,B\}$]		
2			generate nonce N_B send [B,KDC, $K_A\{N_A, N_C, A, B\}$, $K_B\{N_B, N_C, A, B\}$]
3		if N_C same in $K_A\{\cdots\}$ and $K_B\{\cdots\}$ generate session key K_{AB} send [KDC,B, N_C, $K_A\{N_A,K_{AB}\}$, $K_B\{N_B,K_{AB}\}$]	
4			send [B,A, $K_A\{N_A, K_{AB}\}$]
5	send [A,B, K_{AB}{"hello"}]		
	<--- A and B establish data session key --->		

Note:
- Msg 3 assures B that request 1 was by A
- Msg 4 assures A that sender is B

Otway-Rees nonce N_C must be unpredictable, o/w C can impersonate B to A.
Suppose N_C is sequential and equals 007 in one attempt. C does following:

	C	KDC	B
1	send [A,B, N_C=008, grbge]		
2			send [B,KDC, grbge, $K_B\{N_B, N_C=008, A,B\}$] **(C records this)**
		KDC rejects message 2	
	Later A attempts to connect to B		
	A	**KDC**	**C**
3	send [A,B, N_C=008, $K_A\{N_A,N_C=008,A,B\}$]		
4			C intercepts this msg 3 send [B,KDC, msg 3 K_A field, msg 2 K_B field]
5		accepts msg 4 (since its N_C's match) send [KDC,B, N_C, $K_A\{N_A,K_{AB}\}$, $K_B\{N_B,K_{AB}\}$]	
			C intercepts msg 5 send [B,A, $K_A\{N_A, K_{AB}\}$]
6	send [A,B, K_{AB}{"hello" }]		

At this point C has impersonated B to A.
- If A uses a data session key obtained from K_{AB}, C won't succeed (but o/w C can impersonate B to A during the data exchange).

Nonce types:

- Large random number: best nonce
  - crypto operations are the best way to generate them
- Timestamp: not as good
  - clocks must have adequate synchronization and resolution
  - must recover from crashes
- Sequence numbers
  - requires non-volatile storage

Example 1: using seq number nonce when unpredictable nonce is needed

A	B
send [A,B, conn] send [A,B, R_1]	send challenge [B,A, $K_{AB}\{R_1\}$]
If R_1 is sequential, C can impersonate A to B as follows	
C	**B**
send [A, B, conn] send [A,B, R_1+1]	send [B,A, $K_{AB}\{R_2\}$] where $R_2=R_1+1$

Example 2: using sequential nonce when unpredictable nonce is needed

A	B
send [A, B, conn] send [A,B, $K_{AB}\{R_1\}$]	send [B,A, R_1]

C lies in wait for A to initiate to B

- When A initiates to B,
  C intercepts and sends challenge R_1+1 to A and gets $K_{AB}\{R_1+1\}$.
- Then C initiates connection to B impersonating A.
- B sends challenge $R_!$+1, for which C now has the correct response.

Worse than man-in-middle: A does not have to be active for C to do attack.

Example 3: where sequence number nonce is adequate

A sends (A,B, conn);
B sends challenge $K_{AB}\{R\}$
A sends response $(K_{AB}+1)\{R\}$.

Strong Password Protocols (NS chapter12)

- Basic strong password protocols (EKE, SPEKE, PDM)
  - Use Diffie-Hellman
  - Human A with password achieves high-quality authentication with B inspite of eavesdropper
  - No protection against reading of B's db

- Augmented strong password protocols (EKE, SPEKE, PDM)
  - Same as basic protocols except also provide low-quality protection against reading of B's db

- Can be used by human A to obtain a high-quality key (including private key)

EKE basic, SPEKE basic, PDM basic

- Protocols use Diffie-Hellman (DH)
- Mutual authentication
- Strong key protection against eavesdropping
- No protection against attacker reading B's db:
  - attacker gets the key obtained from A's password (no need for offline dictionary attack)

EKE basic

- DH encrypted with password derived key to share high-quality key
- Use shared high-quality key to do two-way authentication
- Strong protection against eavesdropping; none against B db reading

A has password pw	**B** has (A,W) where W = hash(pw)
public DH parameters: g and p	
choose rn a $T_A \leftarrow g^a$ mod p send [A, B, W{T_A}]	
	choose rn b $T_B \leftarrow g^b$ mod p choose challenge C_1 send [B, A, W{T_B,C_1}] $K_B \leftarrow (T_A)^b$ mod-p
$K_A \leftarrow (T_B)^a$ mod-p generate challenge C_2 send [A,B, K_A{C_1,C_2}]	
	send [B,A, K{C_2}]
A and B now share strong key $K_A = K_B = g^{ab}$ mod p	

EKE basic (cont)

To defend against offline dictionary attack, need to ensure that g^a mod p (and g^b mod p) has no structure:

- g^a mod p is less than p
- If encryption block size exceeds $\lceil \log_2 p \rceil$, extra bits must have random pad.
- Require p to be slightly more than a power of 2.
  If p is slightly less than a power of 2, then g^a mod p has structure:
  - Msb = 1 implies most of the bits to the right of msb are zeros
  - Each incorrect candidate pw has 50% chance of violating structure
    - Can quickly narrow down to space of candidate passwords.

  Is this really a EKE issue, rather than a DH issue?

SPEKE basic

Same as EKE except that W takes the place of g.

A	**B**
stores password pw	stores (A,W) where W=hash(pw)
public p (prime)	
choose rn a $T_A \leftarrow W^a$ mod p send [A, B, T_A]	
	choose rn b $T_B \leftarrow W^b$ mod p send [B, A, T_B] $K_B \leftarrow (T_A)^b$ mod-p
$K_A \leftarrow (T_B)^a$ mod-p	
A and B now share strong key $K_A = K_B = W^{ab}$ mod p	
<------ two-way authentication using shared key K ------->	

Note: W must be perfect square mod-p, o/w W^a mod p/W^b mod p have structure

- Otherwise, W^a mod p (or W^b mod p) may not be a perfect square
- Eliminates 50% of candidate passwords.
  But not as bad as EKE because this pruning occurs only once.

PDM basic

- Like EKE but g=2 and prime p is obtained from password (p = f_p(pw))
- To defend against offline dictionary, require
  - p to be a safe prime, i.e., (p–1)/2 is also a prime
  - p mod 24 = 11
  - etc

EKE augmented, SPEKE augmented, PDM augmented, SRP

- Mutual authentication
- Strong-key protection against eavesdropping
- Weak-key protection against attacker reading B's db:
  - attacker can get A's pw by offline dictionary attack

EKE augmented is described next; others are similar.

EKE augmented

- Public DH parameters g and p
- A has password pw
  - two keys, W and W', obtained from pw (eg, using different hashes)
- B has [A: W', T_A' (= g^W mod-p)] (so W' is open but not W)
- A and B do DH encrypted by W' to establish session key $g^{a \cdot b}$ mod-p:
  - A: random a; $T_A = g^a$ mod-p; W'{T_A} to B
  - B: random b; $T_B = g^b$ mod-p; W'{T_B} to A
  - $K_A = (T_B)^a$ mod-p = $K_B = (T_A)^b$ mod-p = $g^{a \cdot b}$ mod-p
- A and B also independently generate DH key $g^{W \cdot b}$ mod-p for authentication:
  - A: K_A' ← $(T_B)^W$ mod-p
  - B: K_B' ← $(T_A')^b$ mod-p

EKE augmented (cont)

A has pw, W, W'	**B** has [A, W', T_A' (= g^W mod-p)]
choose rn a; T_A ← g^a mod-p send [A, B, W'{T_A}]	
	extract T_A from W'{T_A} using W' choose rn b; T_B ← g^b mod-p K_B ← $(T_A)^b$ mod-p K_B' ← $(T_A')^b$ mod-p H ← hash(K_B , K_B') send [B, A, W'{T_B}, H]
extract T_B from W'{T_B} using W' K_A ← $(T_B)^a$ mod-p K_A' ← $(T_B)^W$ mod-p verify H = hash(K_A , K_A') to authenticate B H' ← hash'(K_A , K_A'), where hash' is another hash function send [A, B, H']	
	verify H' = hash'(K_B, K_B') to authenticate A
A and B are mutually authenticated and share strong key K = g^{ab} mod p	

Obtaining credential (eg, private key) from network

- Earlier: directory service has $privKey_A$ encrypted by key from A's password
- Can also be solved using strong password protocols

EKE-based protocol for obtaining credential:

- Public DH parameters g and p
- A stores password pw
  - W and W' are two keys obtained from password
- B stores (A, W, Y), where Y = W'{private key of A}

A	**B**
choose rn a compute W = hash(pw) send [A,B, W{ g^a mod p }]	
	choose rn b send [B,A, g^b mod p, (g^{ab} mod p){Y}]
compute g^{ab} mod-p decrypt (g^{ab} mod p){Y} to get private key	

More on authentication (rt comm sec) (NS chapter 16)

Long-term secret of a principal: Master key or private half of a public key pair.

Key escrow:

- Principal's long-term secret held by an escrow agent (eg, law enforcement).
- Principal usually has separate public key pairs for encryption and for signing. Signature key usually not escrowed.
  - (o/w principal can deny a signed message)

Perfect forward security (PFS)

- A session has PFS if an attacker who eavesdrops and later learns long-term secrets of participants still cannot obtain session key.

Escrow-foilage

- A session has escrow-foilage if escrow agent cannot obtain session key by eavesdropping.
  - Of course, escrow agent can always impersonate participant or do man-in-middle attack.

PFS/escrow-foilage usually achieved with authenticated Diffie-Hellman

Example based on public signature keys (below, $[x]_A$ denotes x signed by A):

A (DH params g, p; pub sign key of B)	**B** (DH params g, p; pub sign key of A)
generate a $T_A \leftarrow g^a$ mod p send [A, B, $[A, T_A]_A$]	
	receive msg verify signature on [A, T_A] generate b $T_B \leftarrow g^b$ mod p $K_B \leftarrow (T_A)^b$ mod p // session key send [B, A, $[B, T_B]_B$]
receive message $K_A \leftarrow (T_B)^a$ mod p // session key = K_B send [A, B, H(K_A)] // H: hash	
	receive message if H(K_A) = H(K_B) then A authenticted send [A, B, H(1, K_B)]
receive message if H(1,K_B) = H(1,K_A) then B authenticated	

Protection against denial-of-service attack

- Typically, when a server receives a (potential) connection request, it starts to maintain state for that client (eg, client id, challenge).
- An attacker can overwhelm such a server by flooding it with connection requests.
- Solution:
  - server asks potential client do some work before storing state for the client.
  - The work request is called a **stateless cookie.**
    (Not to be confused with web browser cookies.)

Example: using a stateless cookie

A	**B** (has secret S, not shared with anybody)
send [A, B, conn]	
	receive msg c ← hash(A's ip address, S) // c: stateless cookie send [B, A, c] forget c
receive message send [A, B, conn, c]	
	receive message if c ≠ hash(A's ip addr, S) then abort else continue with authentication handshake

- The above cookie just required A to send it back.
- A more severe cookie c: random string to which the client has to return [x, c], where x is a n-bit number that hashes to c
  - n can be varied to inflict more/less work.

End-point id hiding

Hide the ids of the communicating principals from eavesdroppers, spoofers, etc. Below, A and B are principals, and n_A and n_B are their respective Internet ids.

A (DH params g, p; pub sign key of B)	**B** (DH params g, p; pub sign key of A)
generate a $T_A \leftarrow g^a \bmod p$ send [n_A, n_B, T_A]	
	receive msg generate b $T_B \leftarrow g^b \bmod p$ $K_B \leftarrow (T_A)^b \bmod p$ // session key send [n_B, n_A, T_B]
receive message $K_A \leftarrow (T_B)^a \bmod p$ // session key send [n_A, n_B, K_A{ A, B, $[T_A]_A$ }]	
	receive message send [n_B, n_A, K_B{ B, A, $[T_B]_B$ }]

- Eavesdropper cannot see end-point ids (A and B)
- Spoofer of B (more precisely, of n_B) can learn end-point ids.
- Same can be done with secret key, say L, instead of public key:
  - use L{T_A} and L{T_B} instead of $[T_A]_A$ and $[T_B]_B$ respectively

Reusing DH key across sessions

- Goal: amortize cost of computing DH key
- Approach: define session key as function of DH key and a random nonce.

First session (compute DH key)

A (DH params g, p; pub sign key of B)	**B** (DH params g, p; pub sign key of A)
generate a $T_A \leftarrow g^a \bmod p$ send [A, B, $[T_A]_A$]	
	receive msg generate b, N_1 $T_B \leftarrow g^b \bmod p$ $K_B \leftarrow (T_A)^b \bmod p$ // DH key send [B, A, $[T_B]_B$, N_1] session key $S_{B1} \leftarrow hash(N_1, K_B)$
receive message $K_A \leftarrow (T_B)^a \bmod p$ // DH key session key $S_{A1} \leftarrow hash(N_1, K_B)$	
< -------------- session key $S_{A1} = S_{B1}$ --------------------- > close session **do not forget T_A, K_A and T_B, K_B**	

Reusing DH key across sessions (cont)

Later session (reusing DH key)

A (has T_A, T_B, K_A from before)	**B** (has T_A, T_B, K_B from before)
start new session send [A, B, $[T_A]_A$] // reuse T_A	
	generate N_2 // reuse T_B and K_B session key $S_{B2} \leftarrow hash(N_2, K_B)$ send [B, A, $[T_B]_B$, N_2]
receive message T_B has not changed, so reuse T_A and K_A session key $S_{A2} \leftarrow hash(N_2, K_A)$	
< -------------- session key $S_{A2} = S_{B2}$ --------------------- > close session	

- Above, B authenticates A but not vice versa (ie, attacker can replay B msgs).
- Easy to fix so that A authenticates B also.

What is lost by reusing DH parameters?

Plausible deniability

- Principal A has plausible deniability in a session if nobody can prove that A participated in the session (even though A and B may have authenticated each other in the session).
- Plausible deniability comes for free with secret key (any one participant can cook up the entire session)
- Not possible with public key unless key is escrowed (eg, use encryption public key rather than signature public key).

Negotiating crypto parameters

- In A-B session initiation, A sends crypto options and B responds with crypto accepted.
- Having crypto parameters negotiated allows same protocol to upgrade to better crypto algorithms when they become available.
- Because crypto options are negotiated before authentication, need to reconfirm after authentication (by reiterating the negotiation messages).