Computer and Network Security
CMSC 414

STANDARDS

Udaya Shankar
shankar@cs.umd.edu

Kerberos 4 (NS chapter 13)

Authentication in network (Realm)
- Realm has KDC and principals (users)
- Users are humans and (distributed) applications (NFS, rsh, etc)
- Human users log in to workstations, use applications (apps)
- Apps can interact with other apps (eg, ftp with NFS)
- KDC authenticates login sessions and apps
- Based on Needham-Schroeder authentication protocol.
- Assumes attacker can eavesdrop and modify messages in transit.
- Assumes DES and IPv4
- Uses timestamps, so nodes need to maintain synchronized clocks.

KDC has
- **master key** for each principal
- Human user's master key obtained from password
- Apps have (high-quality) key
- Secret key K_{KDC} (not shared with any other principal)
  - for encrypting master keys in local database
  - for encrypting TGTs
- Read-only database (except when principal changes master key)

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

When human user wants to access an application
- user's workstation presents KDC with [request, TGT, timestamp] (encrypted with session key)
- KDC returns **credentials** (encrypted with session key) consisting of
  - session key (to talk to application)
  - ticket for application (encrypted with application's master key)
- user's workstation presents application with [request, ticket]

Login handshake

	user A (has pw)	A's workstation	KDC (has A: K_A)
1	start login send [A,passwd]		
2		send [A,KDC, AS_REQ] AS_REQ: "A needs TGT"	
			receive msg retrieve K_A generate session key S_A $tgt_A \leftarrow K_{KDC}\{A, S_A\}$ $crd_A \leftarrow K_A\{S_A, tgt_A\}$
3			send [KDC, A, AS_REP, crd_A]
		receive msg construct K_A from passwd extract S_A, tgt_A from crd_A forget passwd; shell uses S_A henceforth	
4	finish login		

Accessing remote principal

	(LATER IN THE SESSION)		
	A	A's workstation	
1	rlogin B		
2		send [A,KDC,TGS_REQ, "A to talk to B", tgt_A, S_A(ts)] ▪ S_A(ts): authenticator	
			receive msg generate session key K_{AB} get S_A from tgt_A get ts and verify find B's master key K_B $tkt_B \leftarrow K_B\{A, K_{AB}\}$ $crd_B = S_A\{B, K_{AB}, tkt_B\}$ // credential
3			send [TGS_REP, crd_B] to A
		receive msg from KDC	
4		send [A,B, AP_REQ, tkt_B, K_{AB}\{ts\}]	B
5			send [B,A, AP_REP, K_{AB}\{ts+1\}]
6		receive msg	
	end		

Replicated KDCs to improve performance

- One master KDC and several secondary KDCs
- Each secondary KDC has read-only copy of KDC database
- Additions/deletions/changes to master keys always done at master KDC
- Secondary KDCs can generate session keys, TGTs, etc.
- Master disseminates KDC databases to secondary KDCs with integrity protection only (but master keys are encrypted with K_{KDC})

Authentication across multiple realms

- Possible if their KDCs share a key.
- Principal name = [name, instance, realm], each string of 40 chars max

A in realm X	KDC_X	KDC_Y	B in realm Y
send [A, KDC_X, TGS_REQ, A.X, D.Y]			
	receive msg send [KDC_X, A, TGS_REP, cred to KDC_Y]		
receive msg send [A, KDC_Y, TGS_REQ, A.X, B.Y, cred]			
		receive msg send [KDC_Y, A, TGS_REP, cred to B]	
receive msg send [A, B, AP_REQ, cred, …]			
			receive msg

Key version number

If A has a ticket to B and B changes its password, then ticket no longer valid.
To handle this case (without A having to ask KDC for a new ticket):
- Applications remember old master keys (up to expiry time (approx 21 hrs)
- In tickets, the key is sent along with version number
- Human users need not remember old passwords

Network layer address in tickets

- Every ticket has the IPv4 address of the principal given the ticket
- Received ticket is not accepted if ticket sender's IP address does not match
- So if B is to impersonate A, it must also spoof the IP address of A (easy to do)
- Prevents delegation
  - A cannot ask B at another IP address to do work on behalf of A (unless B spoofs IP address of A!)

Encryption of application data

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

Encryption and Integrity-protection

Recall that standard approach uses two keys and two crypto passes (expensive).
Kerberos uses a modified CBC called Plaintext CBC (PCBC)

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

Encryption for Integrity only

Computes checksum on [session key, msg]
Probably not cryptographically strong

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

Kerberos 5 (NS chapter 14)

More general than V4

- Message formats
  - Defined using ASN.1 and BER (Basic Encoding Rules)
  - Automatically allows for addresses of different formats, etc.
  - Occupies more octets
- Names: [NAME, REALM]
  - Arbitrary strings of arbitrary length (allows ".", "@", "name@org", etc)
  - Allows X.500 names (Country/Org/OrgUnit/LName/PName/…)
  - Kerberos 4 names have size/character limitations
- Cryptographic algorithms
  - Allows choice of crypto algorithms (but DES is the only deployed version)
  - Uses proper integrity protection (rather than pseudo-Juneman checksum)

Kerberos 5

Delegation of rights

- A can ask KDC for a TGT with
  - network addresses different from A's network address (to be used by principals at other IP addresses on behalf of A)
  - no network address (can be used by any principal at any network address)
- Policy decision whether KDC/network issues/accepts such tgts
- Having tgts with explicit addresses:
  - KDC tracks delegation trail
  - A has to interact with KDC for each delegation

A can give a TGT/tickets to B with specific constraints

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

Kerberos 5

TGT/tkt lifetime

- Fields:
  - start-time: when ticket becomes valid
  - end-time: when ticket expires (but can be renewed (see renew-till)
  - authtime: when A first logged in (copied from initial login TGT)
  - renew-till: latest time for ticket to be renewed.
- Allows unlimited duration (upto Dec 31, 9999) subject to renewing (e.g., daily)
  - exchange tgt/tkt at KDC for a new (renewed) tgt/tkt
  - tgt/tkt has to be renewed before expiry (o/w KDC will not renew)
- Allows **postdated** tickets (e.g, for batch jobs).

Kerberos 5

Keys

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

- Needed because tgts/tickets are now renewable and can be postdated.
- For each principal, KDC database stores [key, p_kvno, k_kvno]
  - key: principal's master key encryped with K_{KDC} (current or past version).
  - p_kvno: version number of principal's master key.
  - k_kvno: version number of K_{KDC} used to encrypt
  - ……………………..
  - max_life: max lifetime for tickets issued to this principal
  - max_renewable_life: max total lifetime for tickets issued to this principal
  - expiration: when this entry expires
  - mod_date: when entry last modified
  - mod_name: principal that last modified this entry
  - flags: preauthentication?, forwardable?, proxiable?, etc.
  - password_expiration:
  - last_pwd_change:
  - last_succes: time of last successful login

Human user master key derived from password and realm name.

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

Kerberos 5

Hierachy of realms

Allows KDC chains of authentication (unlike V4)

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

Evading off-line password guessing

- V4 allows off-line password guessing:
  - KDC does not authenticate TGT_REQ before issuing TGT
  - So B can spoof A, get a TGT for A, do off-line dictionary attack on TGT
- In V5
  - Req for TGT for A must contain K_A{timestamp}; so above attack not possible.
  - KDC also does not honor requests for tickets to human users by others.
  - Prevents logged-in B to ask KDC for a ticket (to delegate) for A, on which it can do off-line password guessing.

Kerberos 5

Key inside authenticator

- Suppose A and B share a session key K_{AB} generated by KDC.
- A and B can have another (simultaneous) session using a different key.
- This can be done without involving the KDC:
  - A makes up a key for this second session and gives that to B encryped by K_{AB}

Double TGT authentication

- Allows A to access server B that has session key, say S_B, but not master key K_B
- Needed for X windows: human user runs remote app that can display locally.
  - X server manages display on workstation screen
  - X clients (eg, xterm, browser) run on local or remote workstations
  - X client (A) needs tkt to X server (B) to display on screen.
- No good for A to get from KDC a (regular) tkt encrypted with B's master key
- Instead
  - A gets TGT_B from B, sends ["A to talk to B", TGT_A, TGT_B] to KDC
  - KDC
    - extracts S_B from TGT_B (encrypted with K_{KDC})
    - creates session key $K_{AB,}$
    - generates tkt_B encrypted with S_B [ie, S_B{'A', K_{AB}}] and sends to A

X windows

B (human user)	B's workstation X server	C (may be B's workstation)
• login to X server [B, passwd]		
	• request TGT_B from KDC • obtain [S_B, TGT_B] from KDC • forget B's passwd • start serving B (eg, keybd, mouse)	
• request X client at C (eg, xterm)		
		• X client starts • has info to display at B's screen • get TGT_B from X server • ask KDC for tkt encrypted by S_B • present tkt to X server and info to display
	• X server displays client's info	

PKI: Public-Key Infrastructure (NS Chapter 15)

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

Consists of
- Principal name space
  - usually hierarchical: usr@cs.umd.edu; www.cs.umd.edu/usr;
- Certification authorities (CAs): subset of the principals
- Repository for certificates and CRLs: (e.g., DNS, directory server)
  - searched by principals
  - updated by CAs
- Method for searching repository for a **chain of certificates** given
  - starting CA: **trust anchor** of the chain
  - ending subject: **target** of the chain

Recall certificates, CRLs, certificate chains

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

Updates in PKI

Introduction of public key J of principal X:
- request every CA that can certify X to issue a certificate for [X, J] (online/offline?)
- each such CA checks the request (online/offline?)
  - if the request passes the CA's checks
    then generate a certificate for [X, J] and add to the repository
- if X is also a trust anchor to a set of principals
  - inform every principal in the set of [X, J] (online/offline?)
  - Is this necessary?

Revocation of public key J of principal X:
- request every CA that has certified [X, J] to revoke it in the CA's next CRL
  - if request passes the CA's checks, it includes [X, J] in its next CRL
- if X is also a trust anchor to a set of principals
  - inform every principal in the set that [X, J] is not to be used
  - Is this necessary?

Updates in PKI should preserve the following desired property:
- For every valid certificate chain CC in the repository
  if X is the subject and J the public key of a cft in CC
  then J is X's public key at issue time of earliest CRL in CC prefix upto cft.

Revocation
- Online revocation service (OLRS)
- Delta CRLs
- First valid certificate
- Good-lists vs bad-lists
- Boring…

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

PKI trust model

Defines where a user gets the trust anchors and what chain paths are legal

Monopoly:

- One CA, say R, trusted by **all** organizations and countries.
- Public key of R is the single trust anchor embedded in all software/hardware.
  - every certificate is signed by R
- Advantages:
  - simplicity: verification involves checking one certificate
- Disadvantages:
  - infeasible to change R's public key if it gets compromised
  - R can charge whatever it wants
  - Security of entire world rests on R
  - Bottleneck in obtaining certificates
  - Bottleneck in issuing CRLs

PKI trust model (cont)

Monopoly + Registration Authorities (RAs)

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

Monopoly + Delegated CAs

- Tree of CAs with one **root** CA
- Users can obtain certificates from a delegated CA rather than root CA.
- Verification invovles chain of certificates with root CA as trust anchor

PKI trust model (cont)

Oligarchy

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

Anarchy

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

PKI trust model (cont)

Name constraints

- Each CA is trusted for certifying only a subset of the principal name space.
- Usually hierarchical: i.e., CA x.y is trusted to certify x.y.*, but not x.z.
- Subset can be a function of the user (see below)

Top-down trust model with name constraints

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

Bottom-up trust model with name constraints

- Hierarchical name space
- Down-links (as usual):
  - x.y certifies x.y.z
- Up-link (unusual!):
  - x.y.z certifies x.y
  - Allows x.y.z.a to use x.y.z as trust anchor for users outside x.y.z:
    - e.g., chain [x.y.z , x.y , x , x.p , x.p.q]
- Cross-link: x.y certifies p.q,
    - where x.y and p.q are CAs of two interacting organizations
  - Improves performance. Can also improve security...?
- Allows PKI to be deployed incrementally in (real-world) situation

PKI trust model (cont)

Certificates with relative names
- Can of worms

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

Internet Security Architecture (NS 16.1)

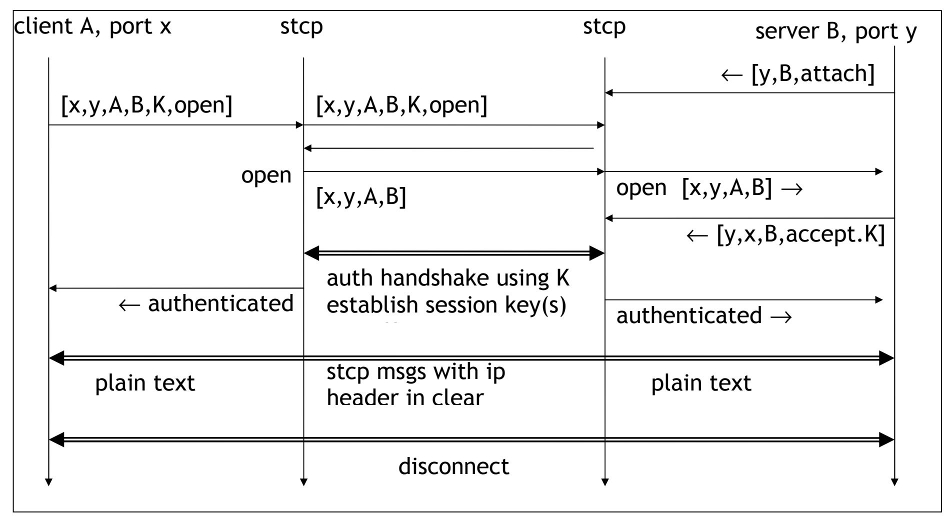
- **TCP/IP stack without security**

- **TCP provides apps with**
  - connection establishment
  - reliable data transfer
- **Want to extend this to handle attackers**
  - network attackers: passive / active
  - endpoint attackers: send messages with arbitrary fields
  - **authentication:** (extends connection establishment)
  - **confidentiality, integrity:** (extends reliable data transfer)

Natural solution to TCP/IP stack with security

- STCP (Secure TCP) like TCP except
  - client app's conn req includes client/server id, authentication secret (K)
  - server app's conn accept includes client/server id, authentication secret (K)
  - stcp conn est does
    - tcp-like 3-way conn est using Internet ids, then
    - auth handshake involving client/server ids, challenges/responses
    - above two can overlap
  - stcp data transfer is tcp-like except
    - ip header is in clear but stcp header and payload encrypted

STCP handshake

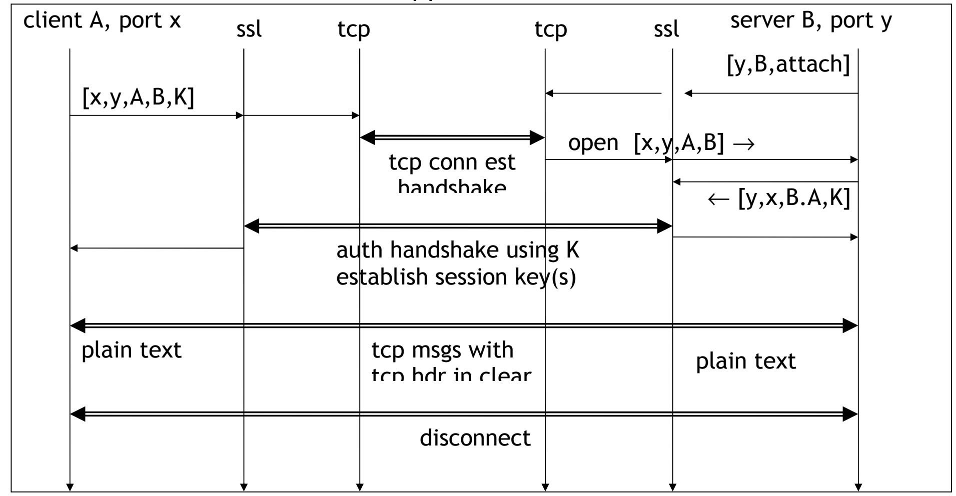


Reality

- Implementors did not want
  - modifications to TCP (which is implemented in OS kernel)
  - another protocol like TCP in OS kernel
  - another protocol like TCP in application space (e.g., above UDP)
- Approach 1: SSL

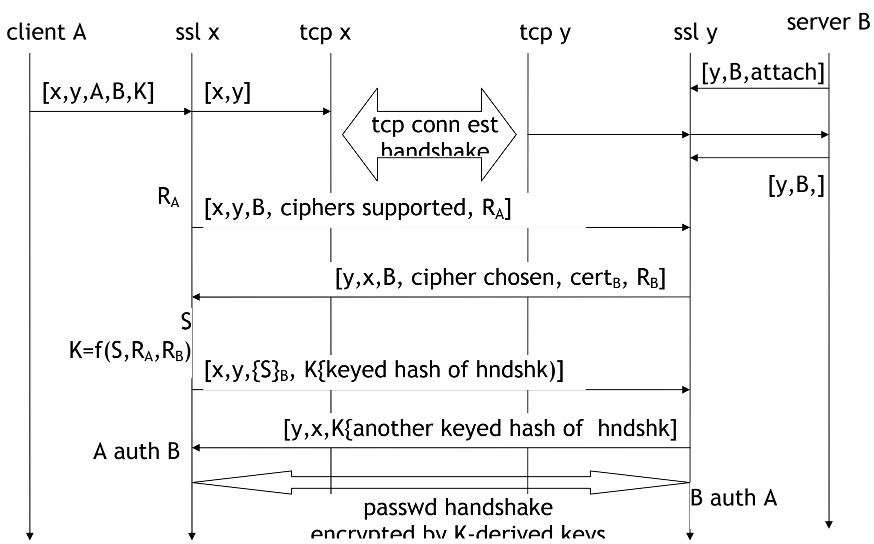
- Approach 2: IPsec

Approach 1: SSL



- tcp hdr in clear => easy denial-of-service attack (rogue packet attack)
  - option 1: restart user or ssl connection
  - option 2: have ssl do retransmissions and acks (i.e. implement tcp)

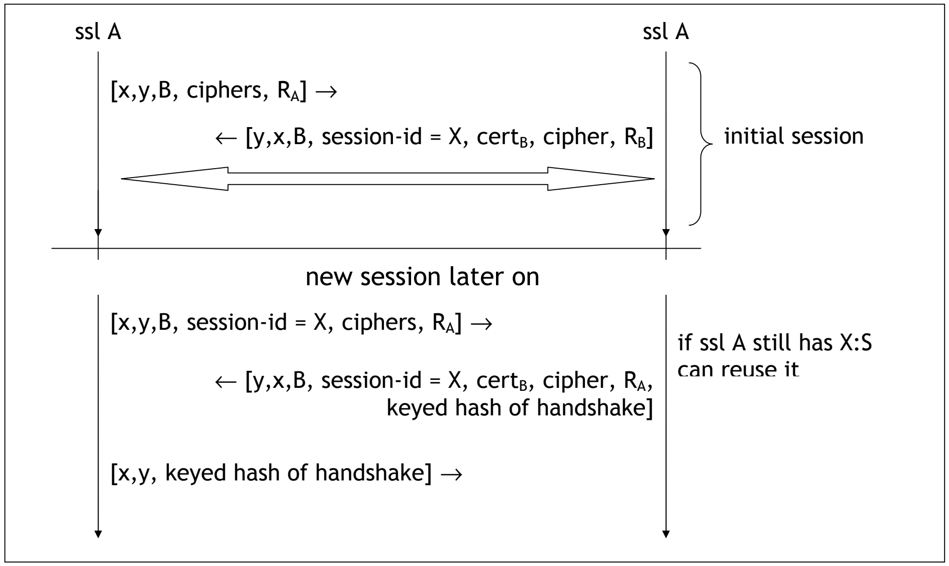
SSL (NS chapter 19)



SSL (cont)

- A authenticates B using $certificate_B$
- B authenticates A using password (usual case)
  Can also use $certificate_a$ for authenticating A
- S: pre-master secret
- K: master secret
  - $K = f(S, R_A, R_B)$
- keys for data encryption/integrity obtained from K, R_A, R_B
  - A's write (transmit) key = B's read (receive) key
  - B's write (transmit) key = A's read (receive) key
- A does two public-key crypto operations
  - verifying $cert_B$
  - calcluating $\{S\}_B$
- To minimize this, S can be reused across different sessions
  - motivated by http 1.0 (which opens many tcp sessions between same A,B)
  - session id

SSL (cont)



IPsec: AH and ESP (NS chapter 17)

- IPsec sits above IP and below TP (transport protocol: TCP, UDP, IP, …
- IP packet: [IP hdr, IPsec hdr, TP hdr, TP payload]
  ←——- IP payload ——–—→
  ←– IPsec payload →
- TP is IP: "tunnel" mode, because often used to tunnel IP traffic
  TP is not IP: "transport" mode
- IP hdr:
  - sender ip addr, rcvr ip addr
  - hop count // mutable
  - next protocol id: TCP, UDP, IP, IPsec (AH or ESP), …
- IPsec header (**generic**):
  - SPI (security parameter index): identiifies IPsec connection (SA)
  - sequence number: of IPsec packet (for replay attacks)
  - IV (for encryption/integrity)
  - authentication data (integrity check)
  - next protocol id: (TCP, UDP, IP, …)

IPsec: AH and ESP (cont)

- IPsec connection referred to as IPsec SA (**security association**)
  - An SA is one-way, so need two SAs for bi-directional packet flow.
- IPsec entity in a node has
  - **Security policy database** (control path)
    - for <ip addr, port, etc>: crypto or not? type? integrity/encryp, …
  - **SA (security association) database** (data path)
    - outgoing: for remote ip addr: SPI, crypto key/alg, sequence number
    - incoming: for SPI: crypto key/algo, expected seq number, …
- IPsec headers are in two flavors:
  - AH hdr: SPI, sequence number, auth data, next protocol id
    - integrity only but on enclosing IP <payload + "immutable" header>
    - not compatible with NAT, firewalls
  - ESP hdr: SPI, seq number, IV, auth data, next protocol id
    - integrity and/or encryption on enclosing IP payload
    - compatible with NAT, firewalls

IPsec: IKE (NS chapter 18)

- In order for an IPsec SA to operate, its parameters (integrity/encryp, key, …) must be set in the (SA database of the) end-points of the SA
- Can be done manually by end-point administrators or dynamically using IKE
- IKE runs over UDP
- IKE has two phases:
  - Phase 1:
    - end-points do mutual authentication and establish phase-1 session keys
    - 3 ways to prove id:
      - public signature key, public encryption key, or secret key
    - two kinds of handshakes, each involving Diffie-Helman
      - aggressive mode: 3 msgs, less options
      - main mode: 6 msgs, more options
    - so total of 6 types of handshakes (actually 8)
  - Phase 2: establish one or more IPsec SAs
    Each SA:
    - 3 msgs. all encrypted with phase-1 keys
    - session keys generated using phase-1 session key as seed
    - public-key crypto (e.g., Diffie-Hellman) is optional

IPsec IKE: Phase 1

Main mode (generic)

client A (at udp x) server B (at udp y)

[C_A (cookie), CP (crypto supported)] →

← [C_A,C_B,CPA (crypto accepted)]

[C_A,C_B, g^a mod p, nonce_A] →

← [C_A,C_B, g^b mod p, nonce_B]

[C_A,C_B, K{A, proof I'm A}] →

← [C_A,C_B, K{B, proof I'm B}]

- C_A, C_B (cookies): distinguish different phase 1 connections between A,B. Must be different for each connection attempt.
- K = f(g^{ab} mod p, nonce_A, nonce_B)

IPsec IKE: Phase 1 (cont)

Aggressive mode (generic)

client A (at udp x) server B (at udp y)

[C_A, g^a mod p, A, nonce_A, CP] →

← [C_A,C_B, g^b mod p, nonce_B, CPA, proof I'm B)]

[C_A,C_B, A, proof I'm A}] →

- If aggressive mode is rejected (perhaps because CP not acceptable to B), A should use main mode (rather than aggressive with different CP).

IPsec IKE: Phase 1 (cont)

Negotiating crypto parameters

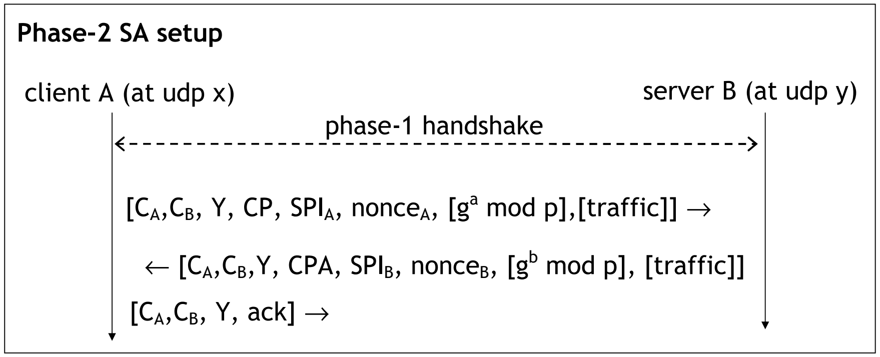
- Algorithms
  - encryption: DES, 3DES, ...
  - hash: MD5, SHA-1, ...
  - authentication method:
    - pre-shared keys
    - RSA signature
    - DSS
    - RSA encryption (original)
    - RSA encryption (improved)
    - ...
  - Diffie-Hellman group
    - modular exponentiation, choice of g and p
    - ellicptic curve, choice of parameters
    - ...
    - **Not negotiable** in aggressive mode
- Lifetime of SA
  - duration and/or quantity of data transferred
- **Must-implement defaults**

IPsec IKE: Phase 1 (cont)

Session keys

- Integrity and encryption keys
  - used on last of phase-1 msgs and all phase-2 handshake msgs
- Seed for phase-2 SA keys
- Keys obtained from hashing (**prf)** quantities of handshake
  - e.g., DES CBC residue, HMAC, …
- **SKEYID** (key seed)
  = prf(nonces, g^{ab} mod p) if public signature key used for auth
  = prf(hash(nonces), cookies) if public encryption key used for auth
  = prf(pre-shared secret key, nonces) if pre-shared secret used for auth
- **SKEYID_d** (seed) = prf(SKEYID, g^{ab} mod p, cookies, 0)
- **SKEYID_a** (integrity key) = prf(SKEYID, SKEYID_d, g^{ab} mod p, cookies, 1)
- **SKEYID_e** (encryp key) = prf(SKEYID, SKEYID_a, g^{ab} mod p, cookies, 2)
- Proof of id for A = prf(SKEYID, g^a, g^b, cookies, A's CP, A)
  Accompanied by certificate (if used)
- Proof of id for B = prf(SKEYID, g^b, g^a, cookies, A's CP, B)
  Accompanied by certificate (if used)

IPsec IKE: Phase 2



- Phase-2 initiator need not be same as phase-1 initiator
- C_A, $C_{B:}$ from phase 1
- Y: 32-bit id of this phase-2 SA
- msgs after "C_A,C_B,Y" under phase-1 keys (SKEYID_e, SKEYID_a)
- IV for msg 1 is final ciphertext block of last phase-1 msg hashed with Y
  IV for later msgs is final ciphertext block of previous msg hased with Y
- traffic descriptor [optional]
- DH [optional]