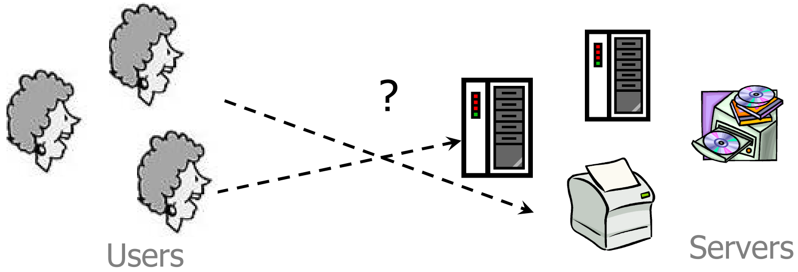
Network Security Standards

Key distribution
Kerberos
SSL/TLS

Many-to-Many Authentication



How do users prove their identities when requesting services from machines on the network?

Naïve solution: every server knows every user’s password

- Insecure: compromise of one server is enough to compromise all users
- Inefficient: to change his password, user must contact every server

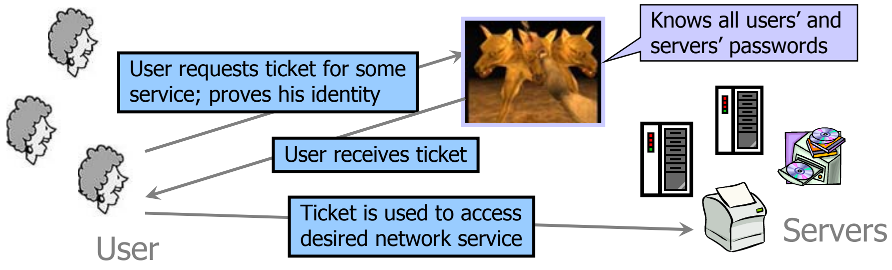
Key Distribution - Secret Keys

- What if there are millions of users and thousands of servers?
- Could configure n^2 keys
- Better is to use a Key Distribution Center
  - Everyone has one key
  - The KDC knows them all
  - The KDC assigns a key to any pair who need to talk

Goals

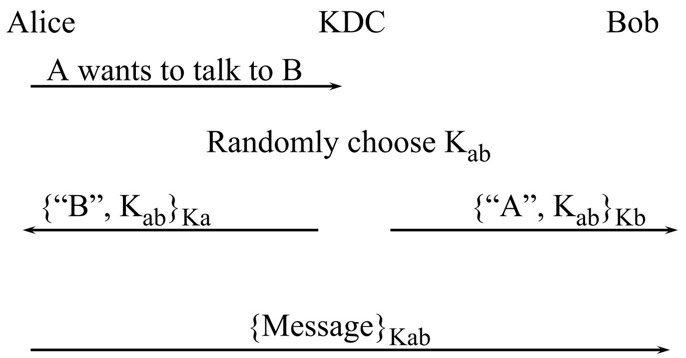
- Requirements
  - Security (sniffers and malicious users)
  - Reliability
  - Transparency
    - Users should not be aware of authentication action
    - Entering password is Ok, if done rarely
  - Scalability
- Threats
  - User impersonation:
    can't trust workstations to verify users' identities
  - Network address impersonation: Spoofing
  - Eavesdropping, tampering and replay to gain unauthorized access

Solution: trusted third party

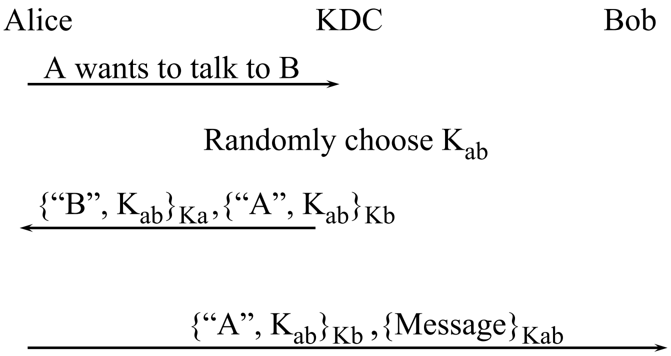


- ❒ Trusted authentication service on the network
  - ❍ Knows all passwords, can grant access to any server
  - ❍ Convenient, but also the single point of failure
  - ❍ Requires high level of physical security

Key Distribution - Secret Keys



A Common Variant



Alice KDC Bob

A wants to talk to B

Randomly choose K_{ab}

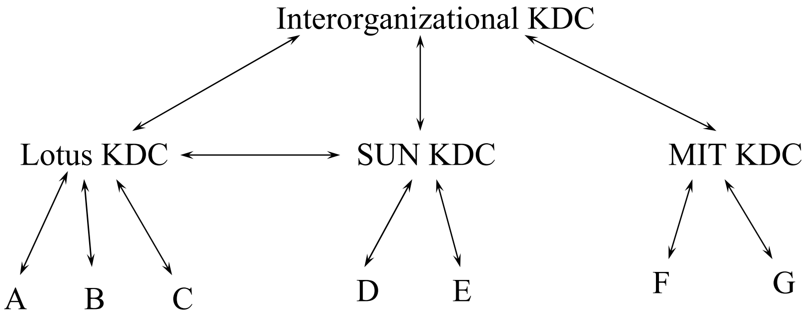
$\{\text{“B”}, K_{ab}\}_{Ka}, \{\text{“A”}, K_{ab}\}_{Kb}$

$\{\text{“A”}, K_{ab}\}_{Kb}, \{\text{Message}\}_{Kab}$

KDC Realms

- KDCs scale up to hundreds of clients, but not millions
- There’s no one who everyone in the world is willing to trust with their secrets
- KDC Realm: a KDC and the users of that KDC

KDC Realms



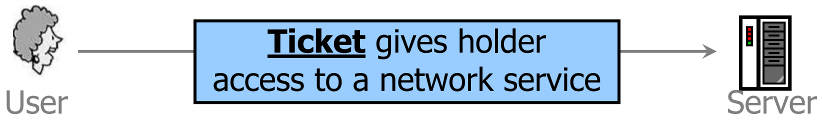
Interrealm KDCs

- How would you talk to someone in another realm?
- How would you know what realm?
- How would you know a path to follow?
- What can bad KDCs do?
- How do you know what path was used? Why do you care?

KDC Hierarchies

- In hierarchy, what can each compromised KDC do?
- What would happen if root was compromised?
- If it's not a name-based hierarchy, how do you find a path?

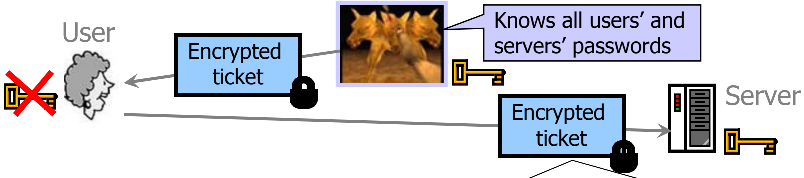
What should a ticket look like?



Ticket gives holder access to a network service

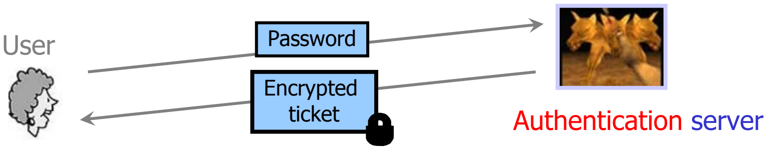
- Ticket cannot include server's plaintext password
  - Otherwise, next time user will access server directly without proving his identity to authentication service
- Solution: encrypt some information with a key derived from the server's password
  - Server can decrypt ticket and verify information
  - User does not learn server's password

What should a ticket include?



- ❒ User name
- ❒ Server name
- ❒ Address of user's workstation
  - ❍ Otherwise, a user on another workstation can steal the ticket and use it to gain access to the server
- ❒ Ticket lifetime
- ❒ A few other things (e.g., session key)

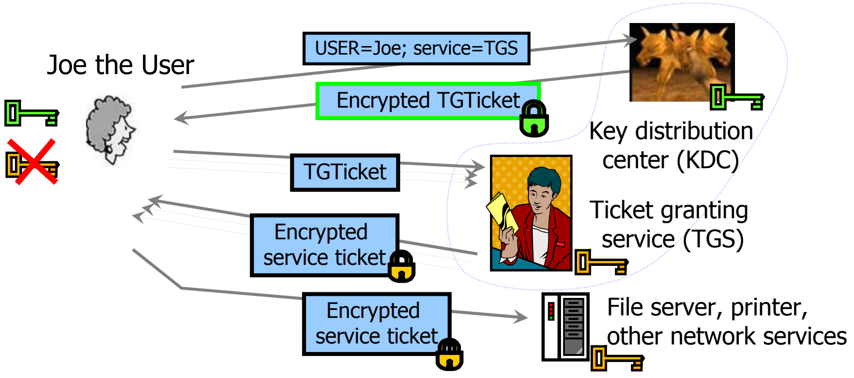
How is authentication done?



- ❒ Insecure: passwords are sent in plaintext
  - ❍ Eavesdropper can steal the password and later impersonate the user to the authentication server
- ❒ Inconvenient: need to send the password each time to obtain the ticket for any network service
  - ❍ Separate authentication for email, printing, etc.

Solution: Two-Step Authentication

- ❐ Prove identity **once** to obtain special TGTicket
  - ❍ Instead of password, use key derived from password
- ❐ Use TGT to get tickets for many network services



Joe the User

USER=Joe; service=TGS

Encrypted TGTicket

TGTicket

Encrypted service ticket

Encrypted service ticket

Key distribution center (KDC)

Ticket granting service (TGS)

File server, printer, other network services

Still Not Good Enough

- ❐ Ticket hijacking
  - ❍ Malicious user may steal the service ticket of another user on the same workstation and use it
    - IP address verification does not help
  - ❍ Servers must be able to verify that the user who is presenting the ticket is the same user to whom the ticket was issued
- ❐ No server authentication
  - ❍ Attacker may misconfigure the network so that he receives messages addressed to a legitimate server
    - Capture private information from users and/or deny service
  - ❍ Servers must prove their identity to users

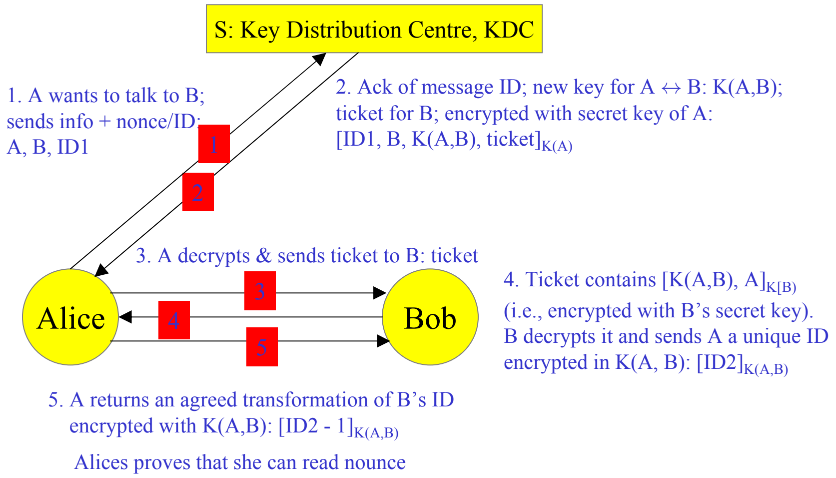
Key management

- Where do keys come from?
  - Symmetric Ciphers: Key Distribution Center (KDC)
  - Why?
    - Shared key for any communication pair does not scale and is cryptographically unwise – uses each key too much!
- Key lifetime / freshness?
  - Long-lived key for authentication and session key negotiation
  - Short-lived key for transfer
  - Why?
    - Long-lived keys are tempting/easy targets (stream ciphers!!!)
    - Compromised old keys

Needham-Schroeder Protocol (1978)

- Basis of Kerberos
- Relies on a key distribution centre (KDC)
- KDC is part of the trusted computing base
  - Knows secret keys of all participants
  - Manages N keys (instead of $N(N-1)/2$)
- Solves two key problems
  - Distribution of shared secret key
  - Mutual authentication

Needham and Schroeder's Protocol



S: Key Distribution Centre, KDC

1. A wants to talk to B; sends info + nonce/ID: A, B, ID1

2. Ack of message ID; new key for A ↔ B: K(A,B); ticket for B; encrypted with secret key of A: $[ID1, B, K(A,B), ticket]_{K(A)}$

3. A decrypts & sends ticket to B: ticket

4. Ticket contains $[K(A,B), A]_{K[B)}$ (i.e., encrypted with B's secret key). B decrypts it and sends A a unique ID encrypted in K(A, B): $[ID2]_{K(A,B)}$

5. A returns an agreed transformation of B's ID encrypted with K(A,B): $[ID2 - 1]_{K(A,B)}$

Alices proves that she can read nounce

Cryptographic protocol design is hard

- Bob never proved his identity to Alice
- If K(A,B) is compromised attacker can impersonate Alice forever
- Denning and Sacco proposed a fix in 1981
- Needham found a flaw in their fix in 1994
- Another flaw found in public key version in 1995 (it is actually only a 3-message protocol)

- Cryptographic protocol design is hard!!!

Kerberos [RFC4120,NeumanTs'94]

- Kerberos: ("der Höllenhund") The watch dog of Hades, whose duty it was to guard the entrance – against whom or what does not clearly appear; … it is known to have had three heads…
  - Ambrose Bierce, The Enlarged Devil's Dictionary
- Designed to authenticate users to servers
- Users use their password to authenticate themselves
- It is possible to protect the Kerberos server
- Assumption: The workstations have not been tampered with!

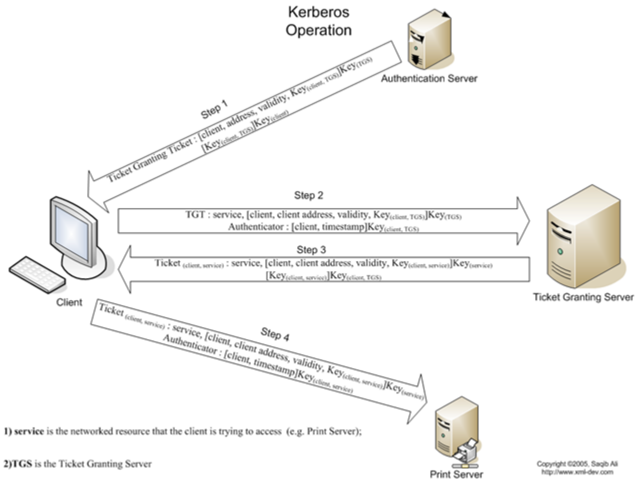
Kerberos lingua

- Principles: Kerberos entity
  - User or system service
  - Triples: (primary name, instance, realm)
    Realm: identifies Kerberos server
  - Examples:
    username@some.domain.name
    somehost/lpr@other.domaim
- Tickets: cryptographically sealed messages with session keys and identifiers
  - Used to obtain a service
- Ticket-Granting ticket (TGT)
  - Ticket to obtain other tickets

How Kerberos works

- Relies on
  - Kerberos key distribution center (KDC)
  - Ticket granting service (TGS)
- Users
  - Have to present a ticket to obtain a service
  - Request TGT from KDC via extended Needham-Schroeder
    (using their shared secret with the KDC)
  - Request tickets from TGS via extended Needham-Schroeder (using the TGT)

Kerberos picture



1) **service** is the networked resource that the client is trying to access (e.g. Print Server);

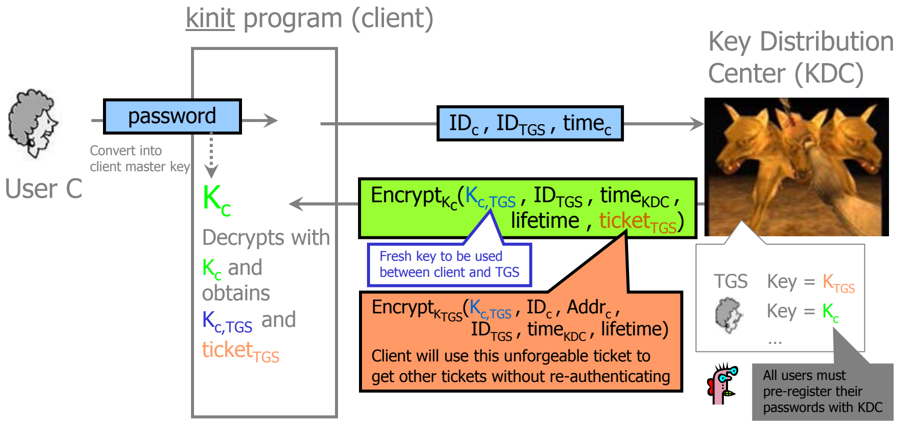
2)**TGS** is the Ticket Granting Server

Copyright ©2005, Saqib Ali
http://www.xml-dev.com

Symmetric keys in Kerberos

- K_c is long-term key for each client C
  - Derived from user's password
  - Known to client and key distribution center (KDC)
- K_{TGS} is long-term key of TGS
  - Known to KDC and ticket granting service (TGS)
- K_v is long-term key of each service V
  - Known to V and TGS; separate key for each service
- $K_{c,TGS}$ is short-term key between C and TGS
  - Created by KDC, known to C and TGS
- $K_{c,v}$ is short-term key between C and V
  - Created by TGS, known to C and V

"Single logon" authentication



kinit program (client)

Key Distribution Center (KDC)

User C

password

Convert into client master key

K_c

Decrypts with K_c and obtains $K_{c,TGS}$ and $ticket_{TGS}$

ID_c , ID_{TGS} , $time_c$

$Encrypt_{K_c}(K_{c,TGS}$, ID_{TGS} , $time_{KDC}$, lifetime , $ticket_{TGS})$

Fresh key to be used between client and TGS

$Encrypt_{K_{TGS}}(K_{c,TGS}$, ID_c , $Addr_c$, ID_{TGS} , $time_{KDC}$, lifetime)

Client will use this unforgeable ticket to get other tickets without re-authenticating

TGS Key = K_{TGS}

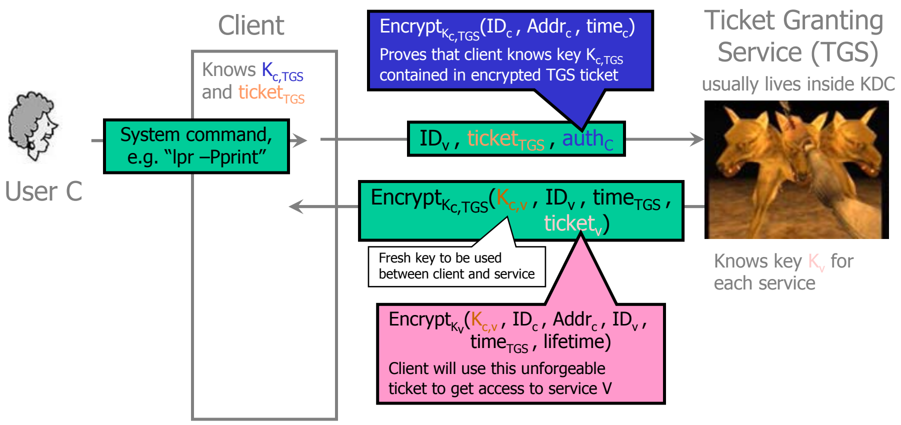
Key = K_c

…

All users must pre-register their passwords with KDC

- Client only needs to obtain TGTicket **once** (say, every morning)
  - Ticket is encrypted; client cannot forge it or tamper with it

Obtaining a service ticket



Client

Knows $K_{c,TGS}$ and ticket_{TGS}

User C

System command, e.g. "lpr –Pprint"

$\text{Encrypt}_{K_{c,TGS}}(\text{ID}_c, \text{Addr}_c, \text{time}_c)$

Proves that client knows key $K_{c,TGS}$ contained in encrypted TGS ticket

$\text{ID}_v, \text{ticket}_{TGS}, \text{auth}_C$

Ticket Granting Service (TGS)

usually lives inside KDC

$\text{Encrypt}_{K_{c,TGS}}(K_{c,v}, \text{ID}_v, \text{time}_{TGS}, \text{ticket}_v)$

Fresh key to be used between client and service

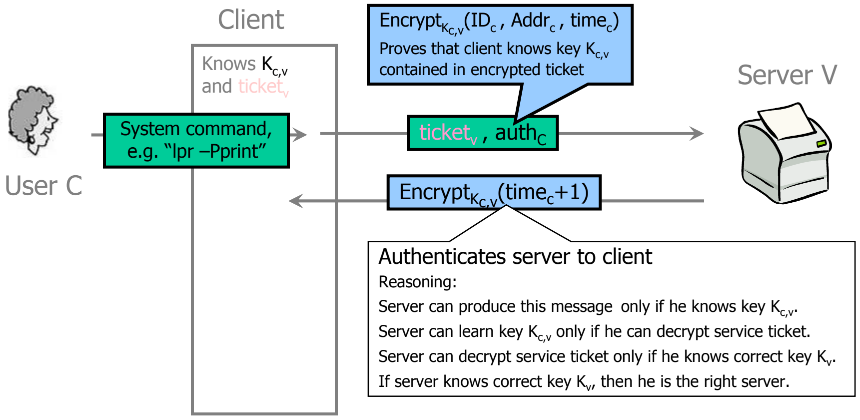
Knows key K_v for each service

$\text{Encrypt}_{K_v}(K_{c,v}, \text{ID}_c, \text{Addr}_c, \text{ID}_v, \text{time}_{TGS}, \text{lifetime})$

Client will use this unforgeable ticket to get access to service V

- Client uses TGTicket to obtain a service ticket and a short-term key for each network service
    - One encrypted, unforgeable ticket per service (printer, email, etc.)

Obtaining service



Client

Knows $K_{c,v}$ and ticket_v

User C

System command, e.g. "lpr –Pprint"

$\text{Encrypt}_{K_{c,v}}(\text{ID}_c, \text{Addr}_c, \text{time}_c)$

Proves that client knows key $K_{c,v}$ contained in encrypted ticket

$\text{ticket}_v, \text{auth}_C$

Server V

$\text{Encrypt}_{K_{c,v}}(\text{time}_c+1)$

Authenticates server to client

Reasoning:

Server can produce this message only if he knows key $K_{c,v}$.

Server can learn key $K_{c,v}$ only if he can decrypt service ticket.

Server can decrypt service ticket only if he knows correct key K_v.

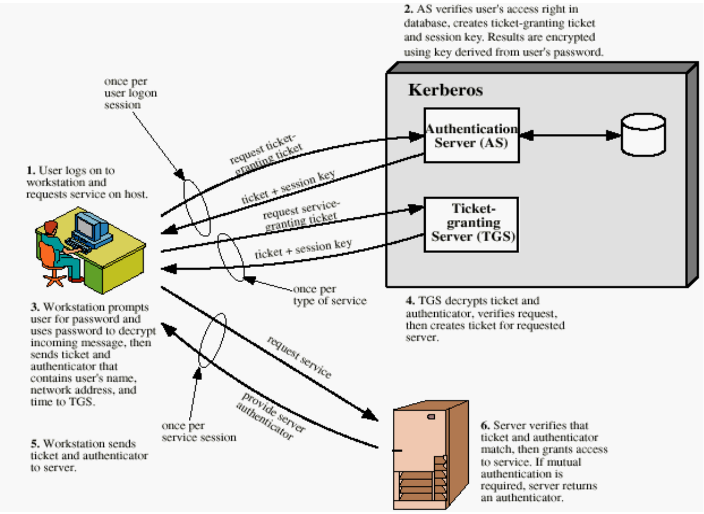
If server knows correct key K_v, then he is the right server.

- For each service request, client uses the short-term key for that service and the ticket he received from TGS

Kerberos in large networks

- One KDC isn't enough for large networks (why?)
- Network is divided into realms
  - KDCs in different realms have different key databases
- To access a service in another realm, users must…
  - Get ticket for home-realm TGS from home-realm KDC
  - Get ticket for remote-realm TGS from home-realm TGS
    - As if remote-realm TGS were just another network service
  - Get ticket for remote service from that realm's TGS
  - Use remote-realm ticket to access service
  - $N(N-1)/2$ key exchanges for full N-realm interoperation

Summary of Kerberos



1. User logs on to workstation and requests service on host.

2. AS verifies user's access right in database, creates ticket-granting ticket and session key. Results are encrypted using key derived from user's password.

3. Workstation prompts user for password and uses password to decrypt incoming message, then sends ticket and authenticator that contains user's name, network address, and time to TGS.

4. TGS decrypts ticket and authenticator, verifies request, then creates ticket for requested server.

5. Workstation sends ticket and authenticator to server.

6. Server verifies that ticket and authenticator match, then grants access to service. If mutual authentication is required, server returns an authenticator.

Important ideas in Kerberos

- Use of short-term session keys
  - Minimize distribution and use of long-term secrets; only used to derive short-term session keys
  - Separate short-term key for each user-server pair
    - But multiple user-server sessions reuse the same key!
- Proofs of identity are based on authenticators
  - Client encrypts his identity, address and current time using short-term session key
    - Also prevents replays (if clocks are globally synchronized)
  - Server learns this key separately (via encrypted ticket that client cannot decrypt) and verifies user's identity
- Symmetric cryptography only

Problematic issues

- Password dictionary attacks on client master keys
- Ticket cache security
- Subverted login command
- Replay of authenticators
  - 5-minute lifetimes long enough for replay
  - Timestamps assume global, secure synchronized clocks
  - Challenge-response would be better
- Same user-server key used for all sessions
- Homebrewed PCBC mode of encryption
  - Tries to combine integrity check with encryption
- Extraneous double encryption of tickets
- No ticket delegation
  - Printer cannot fetch email from server on your behalf

Kerberos Version 5

- Better user-server authentication
  - Separate subkey for each user-server session instead of re-using the session key contained in ticket
  - Authentication via subkeys, not timestamp increments
- Authentication forwarding
  - Servers can access other servers on user’s behalf
- Realm hierarchies for inter-realm authentication
- Richer ticket functionality
- Explicit integrity checking + standard CBC mode
- Multiple encryption schemes, not just DES

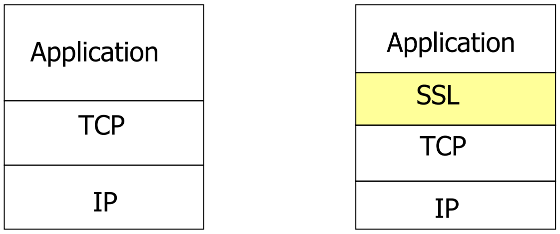
Practical Uses of Kerberos

- Email, FTP, network file systems and many other applications have been kerberized
  - Use of Kerberos is transparent for the end user
  - Transparency is important for usability!
- Standard authentication for Windows (since W2K)
- Local authentication
  - login and su in OpenBSD
- Authentication for network protocols
  - rlogin, rsh, telnet, afs
- Secure windowing systems
  - xdm, kx

SSL: Secure Sockets Layer

- Widely deployed
  - Supported by almost all Web browsers and servers
  - **https**
  - Lots $ spent over SSL
- Originally designed by Netscape in 1993
- Proposed standard:
  - TLS: transport layer security (RFC 4346)
- Provides
  - Confidentiality
  - Integrity
  - Authentication
- Original goals:
  - Secure Web e-commerce transactions
  - Encryption (especially credit-card numbers)
  - Web-server authentication
  - Optional client authentication
  - Minimum hassle for business with new merchant
- Available to all TCP applications
  - Secure socket interface

SSL and TCP/IP

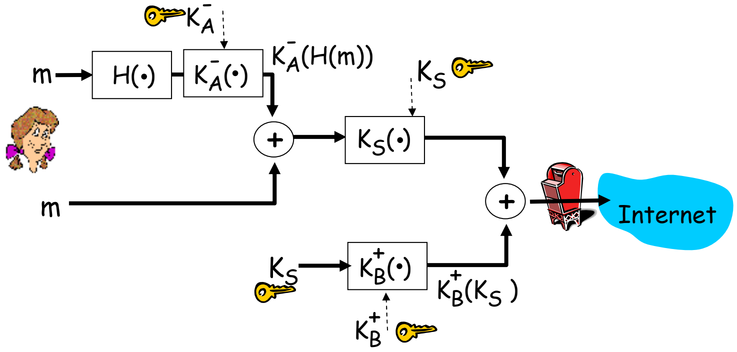


Normal Application

Application with SSL

- SSL provides application programming interface (API) to applications
- Many SSL libraries/classes readily available, including C, C++, Java, Perl, …

Could do something like PGP

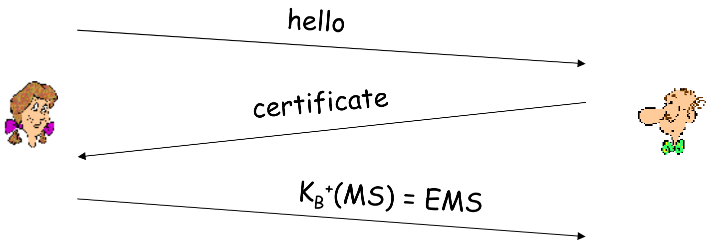


- But want to send byte streams & interactive data
- Want a set of secret keys for entire connection
- Want certificate exchange as part of protocol: handshake phase

Toy SSL: a simple secure channel

- Handshake: Alice and Bob use their certificates and private keys to authenticate each other and exchange shared secret
- Key Derivation: Alice and Bob use shared secret to derive set of keys
- Data Transfer: Data to be transferred is broken up into a series of records
- Connection Closure: Special messages to securely close connection

Toy: simple handshake

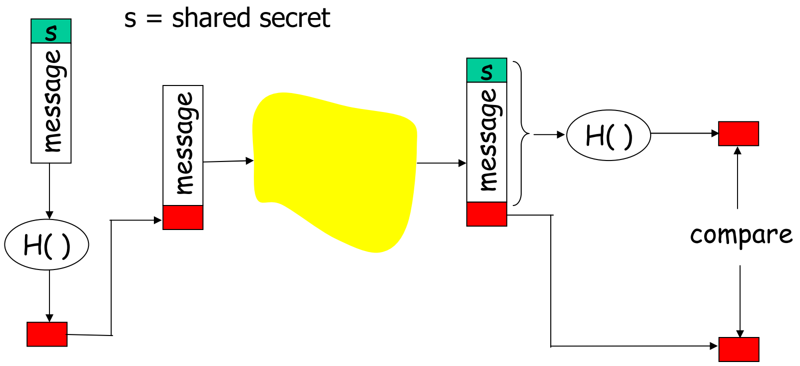


- MS = master secret
- EMS = encrypted master secret

Toy: key derivation

- Bad to use same key for >1 cryptographic op.
  - Different keys for message authentication code (MAC) and encryption
- Four keys:
  - K_c = encryption key for data sent from client to server
  - M_c = MAC key for data sent from client to server
  - E_s = encryption key for data sent from server to client
  - M_s = MAC key for data sent from server to client
- Keys derived via key derivation function (KDF)
  - Takes master secret and (possibly) some additional random data and creates the keys

Recall MAC



- Recall that HMAC is a standardized MAC algorithm
- SSL uses a variation of HMAC
- TLS uses HMAC

Toy: data records

- Why not encrypt data in stream as we write it to TCP?
  - Where to put MAC?
    At end? No message integrity until all data processed.
  - E.g.: instant messaging: how to do integrity check over all bytes before displaying?
- Break stream in series of records
  - Each record carries a MAC
  - Receiver can act on each record as it arrives
- Issue for receiver: how to distinguish MAC from data
  - Want to use variable-length records

length	data	MAC

Toy: sequence numbers

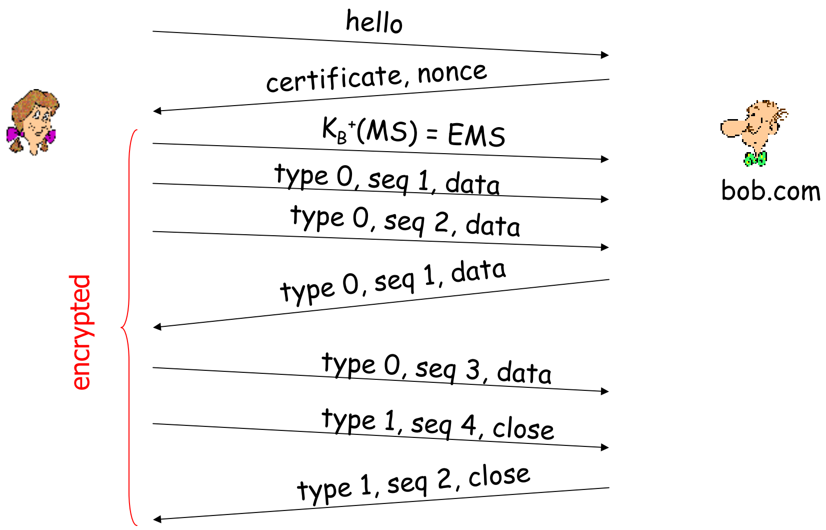
- Attacker can capture and replay record or re-order records
- Solution: put sequence number into MAC:
  - MAC = MAC(M_x, sequence||data)
  - Sequence number serves as nonce for record
  - Note: no sequence number field
- Attacker could still replay all of the records
  - Use session nonce as well

Toy: control information

- Truncation attack:
  - Attacker forges TCP connection close segment
  - One or both sides thinks there is less data than there actually is.
- Solution: record types, with special type for closure
  - Type 0 for data; type 1 for closure
- MAC = MAC(M_x, sequence||type||data)

length	type	data	MAC

Toy SSL: summary



hello

certificate, nonce

$K_B^+(MS)$ = EMS

type 0, seq 1, data

type 0, seq 2, data

type 0, seq 1, data

type 0, seq 3, data

type 1, seq 4, close

type 1, seq 2, close

encrypted

bob.com

Toy SSL is not complete

- How long are the fields?
- What encryption protocols?
- No negotiation
  - Allow support for different encryption algorithms
  - Allow client and server to choose together specific algorithm before data transfer

Most common symmetric ciphers in SSL

- DES – Data Encryption Standard: block
- 3DES – Triple strength: block
- RC2 – Rivest Cipher 2: block
- RC4 – Rivest Cipher 4: stream

Public key encryption

- RSA

SSL cipher suite

- Cipher suite
    - Public-key algorithm
    - Symmetric encryption algorithm
    - MAC algorithm
- SSL supports a variety of cipher suites
- Negotiation:
    - Client offers choice; server picks one

Real SSL: handshake (1)

Purpose

1. Server authentication
2. Negotiation: agree on crypto algorithms
3. Establish keys
4. Client authentication (optional)

Real SSL: handshake (2)

1. Client sends list of algorithms, along with client nonce
2. Server chooses algorithms from list;
   sends back: choice + certificate + server nonce
3. Client verifies certificate,
   extracts server's public key,
   generates pre_master_secret,
   encrypts with server's public key,
   sends to server
4. Client and server independently compute encryption and MAC keys from pre_master_secret and nonces

5. Client sends MAC of all handshake messages
6. Server sends MAC of all handshake messages

Real SSL: handshaking (3)

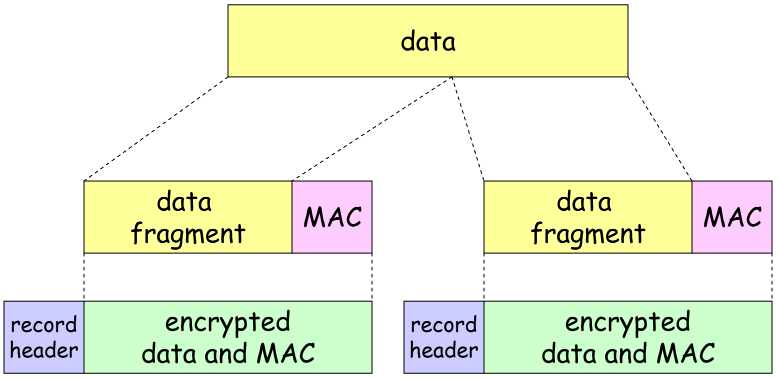
Last 2 steps protect against tampering of handshake

- Client typically offers range of algorithms: some strong, some weak
- Man-in-the middle can delete stronger algorithms
- Last 2 steps prevent this
  - Note: last two messages are encrypted!

Handshake types

- All handshake messages (with SSL header) have 1 byte type field: Types
  - ClientHello
  - ServerHello
  - Certificate
  - ServerKeyExchange
  - CertificateRequest
  - ServerHelloDone
  - CertificateVerify
  - ClientKeyExchange
  - Finished

SSL record protocol

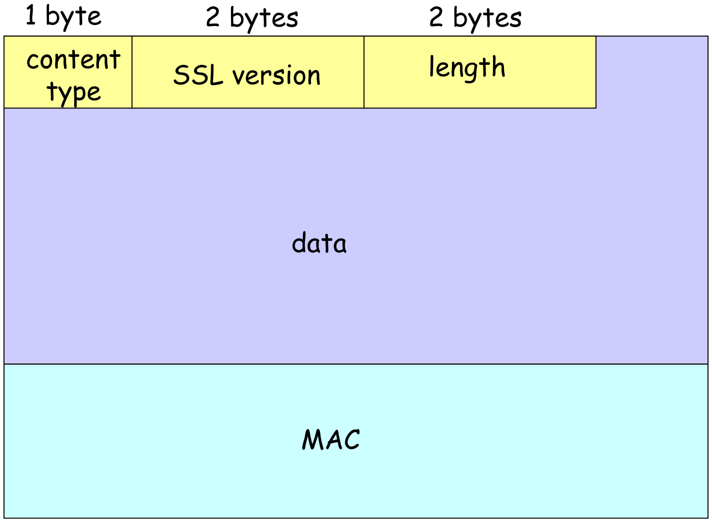


Record header: content type; version; length

MAC: includes sequence number, MAC key M_x

Fragment: each SSL fragment 2^{14} bytes (~16 Kbytes)

SSL record format



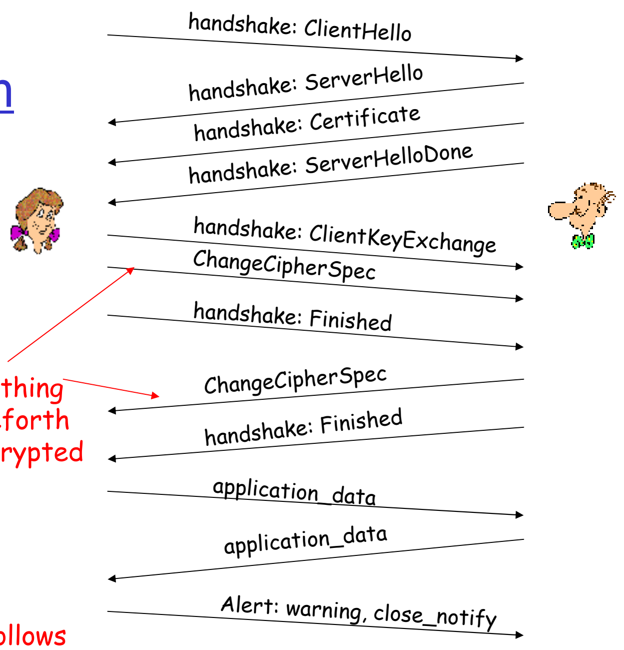
Data and MAC encrypted (symmetric key algorithm)

Content types in record header

- Application_data
- Alert
  - Signaling errors during handshake
- Handshake
  - Initial handshake messages are carried in records of type “handshake”
  - Handshake messages in turn have their own types
- Change_cipher_spec
  - Indicates change in encryption and authentication algorithms

---

SSL: real connection



handshake: ClientHello

handshake: ServerHello

handshake: Certificate

handshake: ServerHelloDone

handshake: ClientKeyExchange

ChangeCipherSpec

handshake: Finished

ChangeCipherSpec

handshake: Finished

application_data

application_data

Alert: warning, close_notify

Everything henceforth is encrypted

TCP Fin follows

Comments about trace messages

ClientHello

- Random: 32-byte nonce

ServerHello

- Cipher suite: RSA key exchange, DES-CBC message encryption, SHA digest
- Random: 32-byte nonce
- Session_id: used for session resumption

Certificate

- X.509 format
- Subject: company info
- Issuer: CA
- Certificate = public key

ClientKeyExchange

- Includes encrypted PreMasterSecret

Finished

- MAC of concatenation of handshake messages

Key derivation

- Client random, server random, and pre-master secret input into pseudo random-number generator.
  - Produces master secret
- Master secret, client and server random numbers into another random-number generator
  - Produces "key block"
- Key block sliced and diced:
  - Client MAC key
  - Server MAC key
  - Client encryption key
  - Server encryption key
  - Client initialization vector (IV)
  - Server initialization vector (IV)

SSL performance

- Recall: big-number operations in public-key crypto are CPU intensive
- Server handshake
  - Typically over half SSL handshake CPU time goes to RSA decryption of the encrypted pre_master_secret
- Client handshake
  - Public key encryption is less expensive
  - Server is handshake bottleneck
- Data transfer
  - Symmetric encryption
  - MAC calculation
  - Neither is as CPU intensive as public-key decryption

Session resumption

- Full handshake is expensive: CPU time
- If the client and server have already communicated once, they can skip handshake and proceed directly to data transfer
  - Session caching
  - For a given session, client and server store session_id, master_secret, negotiated ciphers
- Client sends session_id in ClientHello
- Server then agrees to resume in ServerHello
  - New key_block computed from master_secret and client and server random numbers

Client authentication

- SSL can also authenticate client
- Server sends a CertificateRequest message to client

Who issues Web certificates?

- Browser comes with list of built-in certificate authorities
- Firefox: 138(?) certificate authorities!
- Do you trust them all to be honest and competent?
- Do you even know them?

E.g.: Baltimore Cybertrust

  - Sold its PKI in 2003
  - What about the new owners?

Mountain America Credit Union

- Reputable CA issued certificate for Mountain America
- DNS name: www.mountain-america.net
- Looks OK
- But „real‟ site at www.mtnamerica.org

- Which site is intended by the user?

Technical attack

- Scenario:
  - Usually: shoping via unencrypted pages
  - Click on „Checkout‟ (or „Login‟ on bank Web site)
  - Next page – downloaded without SSL protection – has login link, which uses SSL
- Attack:
  - Tamper with that page
  - Will anyone notice
  - Note some sites outsource payment processing!

- SUGGESTION:
  - Attend our WEB browser security seminar starting this Friday!

- SUGGESTION:
  - Attend our WEB Browser Security seminar starting this Friday!

SSL summary

- Cryptography itself seems correct
  - Indeed is formally verified after many iterations
- Human factors are dubious
- Most users don‵t know what a certificate is, or how to verify one

  Moreover: hard to know what it should say!
- No rational basis for deciding whether or not to trust a CA