Using Cryptography

- Provides foundation for security services
  - touched upon one form of key exchange

- But can it bootstrap itself?
  - must establish shared key
  - straightforward plan
    - one side generates key
    - transmits key to other side
    - but how?
Two Problems

- Peer-to-peer key sharing
- Prob 1: Known peer, insecure channel
- Prob 2: Secure channel, unknown peer
Man in the Middle of DH

DH provides key exchange, but no authentication
- you don’t really know you have a secure channel
- man-in-the-middle
- you exchange a key with eavesdropper (man-in-the-middle), who exchanges key with the person you think are you talking to directly
- eavesdropper relays all messages, but observes or changes them in transit

solutions
- published public values
- authenticated DH (signed or encrypt DH value)
- encrypt the DH exchange
- subsequently send has of DH value, with secret
Security Through Obscurity?

Caesar ciphers
- very simple permutation
- only 25 different cases
- relies strictly on no one knowing the method
- key exchange is really method exchange
Passwords

- Reduces permutation space to key space
  - Caesar cipher: one-letter "key"
  - 10-letter key for MSC reduces $26! \approx 4 \times 10^{20}$ to $26P_{10}$
    - but hard to remember 10-letter key
  - 8-byte key for DES reduces $2^{64!} \approx 10^{10^{200}}$ to $2^{56} \approx 10^{17}$

- But key is more compact and perhaps more readily exchanged out of band
  - in person
  - by telephone (especially for public keys)

- Most security depends on some out of band bootstrap
  - exceptions? are they really exceptions?
    - DH provided key exchange, but not authentication
The German Enigma Machine

- **Rotor**-based
  - rotors are wired codewheels
  - a rotor implements a fixed mono-alphabetic substitution
  - **polyalphabetic substitution** (with a long period) - the encipherment of each plaintext character causes various rotors to move, like an odometer (but not exactly)

- Broken first by Polish, then by English
  - not as easily as widely regarded

- Weaknesses in key distribution
  - day keys plus scramblers (using subkeys)
  - "session keys" encrypted in duplicate
  - Enigma did not use OFB/CFB
Secret Key Distribution

Bill Cheng

http://merlot.usc.edu/cs530-s10
Peer-to-Peer Distribution

- Technically easy
  - by hand!
  - or have a day key

- But it doesn’t scale
  - hundreds of servers...
  - times thousands of users...
  - yields ~ million keys

- Centralized key server
  - building up to the *Needham-Schroeder* approach
Needham and Schroeder - Basic Idea

- User sends request to KDC: \{s\}
- KDC generates a random key: \(K_{c,s}\)
  - encrypted twice, each with a different key
    - \(\{K_{c,s}\}K_c, \{K_{c,s}\}K_s\)
    - \(\{K_{c,s}\}K_c\) is the credentials (contains session key)
    - \(\{K_{c,s}\}K_s\) is the ticket
    - ticket is opaque to the client, it is meant to be forwarded with application request
- No keys ever traverse the net in the clear
KDC (Cont...)
KDC (Cont...)

\[ c, s \{K_{c,s}\}K_c \]

\[ c, s \{K_{c,s}\}K_s \]

\[ \{data\}K_{c,s} \]
Problem #1

How does user know session key is encrypted for the server? And vice versa?

Attacker intercepts initial request, and substitutes own name for server
  - can now read all of user’s messages intended for server
Problem #1 (Cont...)

A \rightarrow_{a} KDC

C \rightarrow_{s} X

S \rightarrow_{K_S} X

K_a

K_c

K_s
Problem #1 (Cont...)

A \rightarrow a \rightarrow KDC

\{K_{c,a}\}K_c

\{K_{c,a}\}K_a

K_a

K_c

K_s
Problem #1 (Cont...)

A \rightarrow_{a} KDC

\{K_{c,a}\}K_{c}

\{K_{c,a}\}K_{a}

\{K_{c,a}\}K_{a}

\{\text{data}\}K_{c,a}
Solution #1

- Add names to ticket, credentials
  - request looks like \{c, s\}
  - \{K_{c,s}, s\}K_c and \{K_{c,s}, c\}K_s, respectively

- Both sides can verify intended target for key sharing

- This is basic **Needham-Schroeder**

```
\begin{center}
\begin{tikzpicture}
  \node[shape=circle,draw] (A) at (0,0) {A};
  \node[shape=circle,draw] (C) at (2,-2) {C};
  \node[shape=circle,draw] (KDC) at (2,0) {KDC};

  \path[->] (A) edge node[above] {a} (KDC);
  \path[->] (C) edge node[above] {c, s} (KDC);
  \path[->] (KDC) edge node[below] {\{K_{c,a}, a\}K_c} (KDC);
  \path[->] (KDC) edge node[below] {\{K_{c,a}, c\}K_a} (KDC);
  \path[->] (A) edge node[below] {K_a} (C);
\end{tikzpicture}
\end{center}
```
Problem #2

- How can user and server know that session key is fresh?
- Attacker intercepts and records old KDC reply, then inserts this in response to future requests
  - can now read all traffic between user and server
Problem #2 (Cont...)
Problem #2 (Cont...)

A

KDC

cracking...

C

K_c

S

K_s
Problem #2 (Cont...)

2 months later...
$K_{c,s}$ cracked!

$K_c$, $K_s$
Problem #2 (Cont...)

\[ \{K_{c,s}\}K_c \{K_{c,s}\}K_s \]
Problem #2 (Cont...)

even if the attacker has not cracked $K_{c,s}$, simply replaying
the credentials can obtained more ciphertext $\{\text{data}\}K_{c,s}$
to help it crack $K_{c,s}$
Solution #2

- Add *nonces* to ticket, credentials
  - request looks like \{c, s, n\}
  - \{K_{c,s}, s, n\}K_c and \{K_{c,s}, c, n\}K_s

- Client can now check that reply made in response to current request
Problem #3

- User now trusts credentials
- But can server trust user?
- How can server tell this isn’t a 3rd-party replay?
- Legitimate user makes electronic payment to attacker; attacker replays message to get paid multiple times
  - attacker can spoof IP address and impersonate the client
  - requires no knowledge of session key
Solution #3

Add *challenge-response*

- server generates second random nonce
- sends to client, encrypted in session key
- client must decrypt, decrement, encrypt
  - if the attacker does not know the session key, it cannot respond

Effective, but adds second round of messages
Problem #4

What happens if attacker does get session key?
- can reuse old sessions key to answer challenge-response, generate new requests, etc.
Solution #4

Replace (or supplement) nonce in request/reply with timestamp [Denning, Sacco]

- \( \{K_{c,s}, s, n, t\}\mathcal{K}_c \) and \( \{K_{c,s}, c, n, t\}\mathcal{K}_s \), respectively
- also send \( \{t'\}\mathcal{K}_{c,s} \) as authenticator, each time the client sends a message to the server with the current time \( t' \)
  - prevents replay without employing second round of messages as in challenge-response
Problem #5

Each client to KDC request yields new known-plaintext pair
- or in this case, verifiable plaintext pair
  - either because the format of data is known or because message conforms to protocol structure

Attacker can sit on the network, harvest client request and KDC replies
Solution #5

- Introduce Ticket Granting Server (TGS)
  - daily ticket plus session keys
  - session keys are random numbers

- \( TGS + AS = KDC \)
  - this is modified Needham-Schroeder
  - basis for \textit{Kerberos}
Problem #6

Active attacker can obtain arbitrary numbers of known-plaintext pairs
  - can then mount dictionary attack at leisure
  - exacerbated by bad password selection

$K_c$ is often weak since it’s usually derived from a passphrase
Solution #6

- Must reduce the exposure of the long-term client key $K_c$

- **Preauthentication**
  - establish weak authentication for user before KDC replies
  - Ex:
    - password-encrypted timestamp
    - hardware authentication
    - single-use key
  - now the attacker must wait for the client to communicate with the KDC in order to obtain known-plaintext pairs
- $K_{c,TGS}$ has a short lifetime (say 8-10 hours)
- $\{K_c,TGS\}K_{TGS}$ is known as the ticket-granting-ticket (TGT)
TGS (Cont...)

→ use the TGT to get a ticket for the server $s$
TGS (Cont...)

- TGS issues ticket for talking to the server $s$
- $K_{c,s}$ also has a short lifetime
TGS (Cont...)
K_c is only used *once* for talking to the AS (*single sign-on*)

- may be twice if preauthentication is used
- no need to talk to AS if c needs to talk to *another server*
- for every server c would like to talk to, this would be done only a small number of times per day
Key Distribution Linked to Authentication

- Summary of techniques
  - be explicit about who you wish to talk to (name in request, check name in reply)
  - use nonce (check nonce value in reply)
  - use timestamp
  - use a separate authentication server (minimize use of $K_c$)
  - use preauthentication (to make sure no one else can generate the original request)

- It’s all about knowing who has the keys

- We will revisit Kerberos when we discuss authentication
Public Key Distribution

Bill Cheng

http://merlot.usc.edu/cs530-s10
Public Key Distribution

Public key can be public!
- how does either side know who and what the key is for? private agreement? (not scalable)
  - who are you?
  - how do I know this public key belongs to amazon.com?

Does this solve the key distribution problem?
- no - while confidentiality is not required, integrity is

Must delegate trust
- why?
- how?
- trust VeriSign? trust IE, Netscape? who else are you trusting that you are not aware of? how many levels of delegation?
Certification Infrastructures

- Public keys represented by certificates
- Certificates signed by other certificates
  - user delegates trust to trusted certificates
  - certificate chains transfer trust up several links

Do you trust a certificate signed by Amazon?

Diagram:
- RootCA
- Amazon
- xyz

Do you trust a certificate signed by Amazon?
What Does A Public Key Certificate Look Like?

Example from OpenSSL:

include CA.pl in your path

set path=(~csci551b/openssl/ssl/misc $path) export PATH=~csci551b/openssl/ssl/misc:$PATH

CA.pl -newca

creates:

demoCA/private/cakey.pem: CA private key
demoCA/cacert.pem: CA certificate (self-signed)

CA.pl -newreq-nodes

creates:

newreq.pem: certificate request
newkey.pem: private key

CA.pl -signreq

creates a certificate: newcert.pem copy of this is in demoCA/newcerts
Other Approaches

PGP
- "Web of Trust" (no CA)
- can model as connected digraph of signers
- signature has attributes (e.g., strength)

X.500
- hierarchical model: tree (or DAG?)
- but X.509 certificates use ASN.1
- X.509 uses MD5
  - in 2005, Lenstra and B. de Weger showed one can create a forged X.509 certificate
What Is ASN.1?

Abstract Syntax Notation number One (ASN.1) is a standard that defines a formalism for the specification of abstract data types (standardized first in 1984, way before XML)

- the notation provides a certain number of pre-defined basic types such as:
  - integers (INTEGER)
  - booleans (BOOLEAN)
  - character strings (IA5String, UniversalString...)
  - bit strings (BIT STRING)

- and makes it possible to define constructed types such as:
  - structures (SEQUENCE)
  - lists (SEQUENCE OF)
  - choice between types (CHOICE)

- lots of tools
  - http://asn1.elibel.tm.fr/
What does ANS.1 grammar look like?

Module-order DEFINITIONS AUTOMATIC TAGS ::= BEGIN

Order ::= SEQUENCE {
    header   Order-header,
    items    SEQUENCE OF Order-line
}

Order-line ::= SEQUENCE {
    item-code  Item-code,
    label      Label,
    quantity   Quantity,
    price      Cents
}

Item-code ::= NumericString (SIZE (7))
Label ::= PrintableString (SIZE (1..30))

Quantity ::= CHOICE {
    units INTEGER,
    millimetres INTEGER,
    milligrammes INTEGER
}

Cents ::= INTEGER

END
Other Approaches (Cont...)

- SSH
  - user keys - out of band exchange
    - `ssh-keygen -b 1024 -t rsa`
    - install `~/.ssh/id_rsa.pub` in `~/.ssh/authorized_keys`
    - `ssh -i ~/.ssh/id_rsa ACCONT@HOST`
  - week assurance of server keys
    - was the same host you spoke with last time

- SET (Secured Electronic Transaction) has banks as CA’s and common SET root
  - private key of the SET root CA is split and spread among child CA’s
    - hierarchical
    - multiple roots
    - key splitting