Notes on Security Protocols
15-440, Fall 2011
Carnegie Mellon University
Randal E. Bryant

References:
Tannenbaum: 9.1, 9.2 (skip 9.2.3)

BASICS

Desired attributes of security:

Condidentiality: Do not reveal information to untrusted parties

Integrity: Ensure that information has not been corrupted

Authentication: Ensure identity of source of information

Availability: Ensure that desired resource can be used.

Attacking strategies (Depict with box diagrams)

Passive
        Code breaking: Determine sensitive data or keys
        Traffice analysis: Learn from overall communication patterns

Active
        Masquerade: Pretend to be someone else
        Replay: Reuse old information
        Alteration: Modify message
        Create spurious messages: (For denial of service)

CRYPTO BUILDING BLOCKS:

1. Private key crypto:

Have key K that must be kept secret between parties

Operations
        K(M): Encrypt message M using key K to create cyphertext C
        K-1(C): Decrypt cyphertext using key K. Should give M.

(Desired) Attributes:
        Can be made fast and safe
        Requires key distribution & preservation
        Given C, hard to determine M or K
        Given C & P, hard to determine K

Attack strategies:
       Ciphertext. Given multiple samples C1, C2, ...
                    Deduce message(s) or K
       Known plaintext. Given multiple pairs (M1, C1), (M2, C2), ...
                    Deduce K
       Chosen plaintext. Given ability to compute K()
                    Generate pairs (M1, C1), ... for chosen messages
                    Deduce K

Examples:

DES: 56-bit keys, encrypt 64-bit blocks

        Security: No longer good enough.
        Can buy hardware DES crackers that do brute-force attack in 1 day

Triple DES: K = (K1, K2, K3) (168 bits total)

        K(M) = K1(K2-1(K3(M)))
        K-1(M) = K3-1(K2(K1-1(C)))

        Why the middle inversion?

```
        If let K1=K2=K3, then get regular DES.
        If let K1=K3, then get double-DES.

AES (a.k.a. Rijndael): 128-bit blocks, key = 128, 192, or 256

2. Public Key Crypto Systems

Two keys:

K+: Public key.  Can be widely disseminated
K-: Private key.  Known only to key owner.

Work both ways as encrpytion/decryption pair:

K+(K-(M)) = M
K-(K+(M)) = M

Stream encryption

Standard encryption schemes are block-based.  Work on fixed size blocks.

What if have message that is longer than single block.

Obvious (Electronic Codebook [ECB]):
        Break M into blocks M1, M2, ..., Mn (Pad final if necessary)
        Encrypt each separately K(M) = [K(M1), ..., K(Mn)]

Better (Cypher block chaining [CBC]):
        (Assume block size and cyphertext size are same)
        Generate random block R
                K(M) = C0, C1, C2, ..., Cn
                where C0 = R, Ci = K(Ci-1 XOR Mi)

        Decoding
                Recover M1, M2, ...
                Mi = K-1(Ci) XOR Ci-1
```

SECURITY PROTOCOLS

Shared Key authentication

Suppose parties Alice (A) and Bob (B) have shared secret key Kab.
A wants to initiate session with B where each party is sure of
who is at the other end. (or at least, that party at each end knows
Kab).

Requires 4 rounds:

1. A->B: A

2. B->A: Rb

   Rb is a "nonce" short for "number used once". Should be randomly
   generated so that no one will be able to guess next nonce to be used
   by B. See go function crypto/rand.Int

3. A->B: Kab(Rb), Ra

   First part is A's way of proving to B that she knows Kab.

4. B->A: Kab(Ra)

   This is B's way of proving that he knows Kab.

Interesting idea: Can we reduce this to 3 rounds:

1. A->B: A, Ra

2. B->A: Rb, Kab(Ra)

   A sure that B knows Kab

3. Kab(Rb)

   B sure that A knows Kab

Vulnerability:

B is providing a free encryption service that can be used by attacker C:

1. C->B: A, Ra

2. B->A (intercepted by C): Rb, Kab(Ra)

1'. C->B: A, Rb

2'. B->A (intercepted by C): Rb', Kab(Rb)

3. C->B: Kab(Rb)

   Bob now thinks that C knows Kab, and so C can masquerade as A.

General principle. Can label each agent by what it knows.

Say Kappa(A, X) means "A knows X"

Examples:

If Kappa(A, K), and B sends A message K(M), then A knows M.

If A knows K, A sends nonce Ra to B, and B sends A K(Ra), then A knows that B knows K. Write Kappa(A, Kappa(B, K))

Revisiting protocol:

Kappa(A, Kab)
Kappa(B, Kab)
Kappa(C, {}) [What does a potential attacker know?]

1. A->B: A

2. B->A: Rb

3. A->B: Kab(Rb), Ra

Kappa(B, Kappa(A, Kab))
Kappa(C, [Rb,Kab(Rb)])

4. B->A: Kab(Ra)

Kappa(A, Kappa(B, Kab))
Kappa(C, [Ra,Kab(Ra)])

All an attacker learns is two plain/cypertext pairs. Can do even better by modifying steps to be:

3′. A->B: Kab(Rb+1), Ra.

4′. B->A: Kab(Ra+1)

These are good enough for authentication but yield even less information about Kab.

Key Distribution

Suppose have many clients: A, B, C, ...

Impractical to require that each pair has secret key Kxy.

Instead, have centralized "key distribution center" or KDC. Call it S. Assume KDC is trustworthy. Keeps its secrets. Does not attempt malicious behavior.

KDC maintains keys with each client. Kas, Kbs, ...

When A & B want to communicate, get S to create "session key" Kab with which they can conduct a conversation. As name implies, generally used for bounded duration.

Version 1:

1. A->S: A,B

KDC generate Kab

2. S->A: Kas(Kab)

Kappa(A, Kab)

3. S->B: Kbs(Kab)

Kappa(B, Kab)

Observe: A & B aren't really sure of each others identity, but could go through above authentication protocol & work things out.

Version 2:

Same idea, but want to eliminate communication from S to B.

1. A->S: A,B

KDC generate Kab

2. S->A: Kas(Kab), Kbs(Kab)

Kappa(A, Kab)

Second part is a "ticket". A cannot use this information directly. It's like a sealed envelope that it can present to B.

3. A->B: A, Kbs(Kab)

Kappa(B, Kab)

Again, could follow earlier protocol to have A & B authenticate to each other.

VULNERABILLITY:

Both of these protocols are vulnerable to replay attacks. E.g., suppose that C had earlier communicated with A via session key Kac. C could spoof the server:

1'. A->S: A,B. Intercepted by C.

2'. C->A: Kas(Kac), Kcs(Kac) [C saw earlier message going to A]

Now when A & C engage in authentication protocol, A will think it's talking to B.

Needham-Schroeder Protocol

Original protocol proposed in 1978. We show slightly enhanced version here, and describe a known weakness.

Purpose:

Set up secure channel between A & B with session key Kab.
Authenticate A & B to each other.
Make sure all information is fresh.

Use three nonces: Ra1, Ra2, Rb

1. A->S: Ra1, A, B

2. S->A, Kas(Ra1, B, Kab, Kbs(A,Kab))

Kappa(A, Kab).
Kappa(A, Kappa(S, Kas))

A knows this message was in direct response to #1. So, this is a fresh session key from S. Include B to differentiate from other session keys A might be creating.

Ticket Kbs(A,Kab) includes identity of A. Prevents ticket from some other channel from being reused.

Encrypt everything under Kas, so that attacker cannot learn anything about session, or any plain/cyphertext pairs.

3. A->B: Kab(Ra2), Kbs(A,Kab)

Kappa(B, Kab)

(Note that sending ticket in step #2 encrypted was useless, since we've revealed it here.)

4. B->A: Kab(Ra2-1, Rb)

Kappa(A, Kappa(B, Kab))

This is a slight variation on previous nonce technique:

> If A knows K, A sends K(Ra) to B, and B sends A K(Ra-1),
> then A knows that B knows K.

5. A->B: Kab(Rb-1)

Kappa(B, Kappa(A, Kab))

Note that the standard N-S protocol does not include Ra2, and therefore there is no way for A to be sure that it's really talking to B (more specifically, that it is talking to a party that was able to get the value of Kab.)

This protocol is still considered to be a bit weak, in that there is nothing that can assure B that the ticket it receives on step #3 is fresh, not an old ticket that is being reused. This weakness was first described publicly in 1981 by Denning & Sacco, who suggested fixing it by adding a time stamp to the ticket. Later (1987) Needham & Schroeder published a fix that adds two more steps to the beginning, as well as an additional nonce Rb0. Here are the two steps (numbered -1 and 0, to indicate their relationship to the other 5 steps):

-1. A->B: A

A tells B that it wants to set up a session

0. B->A: Kbs(Rb0)

B provides a nonce that A cannot decode.

We incorporate this nonce into the next 3 steps of the protocol:

1′. A->S: Ra1, A, B, Kbs(Rb0)

2′. S->A: Kas(Ra1, B, Kab, Kbs(A, Kab, Rb0))

3′. A->B: Kab(Ra2), Kbs(A, Kab, Rb0)

We see that nonce Rb0 gets incorporated into the ticket that A provides B. B can now be sure that this is a newly generated ticket.

Kerberos

Uses a version of secret key N-S to set up secure channels between users & services. Uses time stamp rather than nonce′s to avoid reuse of stale keys. Based on following:

> If A knows K, B sends K(t) to B, and current time ˜= t,
> then A knows that B knows K.

Requires A & B to maintain synchronized clocks. More precise ==> less vulnerable to replay attacks.

Public key N-S:

1. A->B: Kb+(A, Ra)

B generates secret session key Kab.

Kappa(B, Kab)

2. B->A: Ka+(Ra, Rb, Kab)

A certain that message came from B.

Kappa(A, Kab)
Kappa(A, Kappa(B, Kab))

3. A->B: Kab(Rb)

Kappa(B, Kappa(A, Kab))
B certain that message came from A.