Notes on Security Protocols
15-440, Fall 2012
Carnegie Mellon University
Randal E. Bryant

References:
Tannenbaum: 9.1, 9.2 (skip 9.2.3), 9.4.1

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 (from most difficult to easiest):

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

Example encryption schemes:

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

Examples:
- RSA
- Diffie-Hellman (will cover later. Does not have same functionality as RSA)

Besides being useful for encryption, public key systems can be used to "sign" documents.

Given document D:
* Agent A "signs" the document using private key: S = Ka-(D).
* A send message "I have created document D and I affix my signature S on it"
* Given document D, and signature S, others can check authenticity by making sure that Ka+(S) = D.
  - Shows that A indeed generated signature
  - Signature is specific to single document, so others cannot reuse on some other document D'

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 characterize each agent by what it knows.

Say [A:X] means "A knows X"

Inference rules

1. Decryption. If [A:K], and B sends A message K(M), then A knows M.

Write as rule:

```
  [A:K], B -> A: K(M)
  ----------------
     [A:M]
```

2. Challenge/response
   If A knows K, A sends nonce Ra to B, and B sends A K(Ra), then
   A knows that B knows K. Write [A:[B:K]]
   But also, that everyone knows the plain/cypher-text pair Ra, K(Ra)

```
 [A:K], A->B:Ra; B->A:K(Ra)
 --------------------------
  [A:[B:K]], [All: Ra, K(Ra)]
```

Slight variant

1. A->B: Ra
2. B->A: K(Ra+1)

Then [A:[B:K]], but no known plain/cypher-text pair.

```
 [A:K], A->B:Ra; B->A:K(Ra)
 --------------------------
  [A:[B:K]], [All: Ra, K(Ra+1)]
```

Sometimes use K(Ra-1) as well.

Revisiting protocol:

Initially
    [A:Kab]
    [B:Kab]
    [All:{}] (What a potential attacker knows)

1. A->B: A

2. B->A: Rb

3. A->B: Kab(Rb+1), Ra

[B:[A:Kab]]
[All:[Rb,Kab(Rb+1)]]

4. B->A: Kab(Ra+1)

[A:[B:Kab]]
[All:Ra,Kab(Ra+1)]

Key Distribution

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

Impractical to require that each pair X, Y 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 for each individual 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)

[A:Kab])

3. S->B: Kbs(Kab)

[B:Kab]

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

(This protocol is vulnerable to a replay attack. Covered later.)

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)

[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)

[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, using key server S.
* 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))

[A:Kab].
[A:[S:Kas]]

A knows this message was in direct response to #1. So, this is a fresh session key from S.

Fine Points:

* Include B to prevent man-in-middle attack:
1′. A->S (intercepted by C): Ra1, A, B ... C->S: Ra1, A, C

If didn't include identity of second party in #2, would have:
2′. S->A: Kas(Ra1, Kac, Kcs(A,Kac))

& A would proceed to set up session with C.

Including second party in #2, would have:
2′. S->A: Kas(Ra1, C, Kac, Kcs(A,Kac))
and so A could detect incursion.

* 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)

[B:Kab, Ra2]

(Note that having sent ticket in step #2 encrypted was useless, since we've revealed it here, but there's no choice)

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

[A:[B:Kbs, Kab]]

A can be certain that it's communicating with B (knows Kbs)

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

[B:[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 knew Kbs.)

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 A, and current time ~= t,
then A knows that B knows K.

[A:K], B->A:K(t), T[A] ~= t

[A:[B:K]]

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

Public key Needham-Schroeder. A sets up session with B. B generates session key. Does not require separate server.

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

B generates secret session key Kab.

[B:Kab, Ra]

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

A certain that message came (recently) from B.

[A:Kab, [B:Kab]]

3. A->B: Kab(Rb)

[B:[A:Kab]]
B certain that message came from A.

Note: Authentication based on premise that Kb+ is really a public key for B & Ka+ is really a public key for A.

How can we be sure of this? (Will discuss later)

Diffie-Hellman(-Merkle) Key Exchange

Purpose: Given 2 agents A & B

* Each have secret values a & b
* Exchange values
  - A shares something about a without giving away its value
  - B shares something about b without giving away its value
* A & B can then independently generate key K based on their secret value + shared value from other party
* K can be used for private key encryption between them.

Note that this does not provide authentication: A & B cannot be sure of each others identities.

What is required:

Public values (Can be used by all parties):
p: A large prime number (typically ~300 bits long)
See Go function crypto/rand.Prime()
g: A value that serves as a "generator" for multiplicative group of integers, mod p (typical values 2, 3, or 5)

Secret values
a, b: large numbers (typically ~100 bits)
See Go function crypto/rand.Int()

Mathematical basis:

Set of values {1, 2, ..., p-1} forms a group with generator g
I.e., Values (g^{i} mod p) for 1 <= i < p generates all {1, ..., p-1}

E.g., p = 5, g = 3

i	g^{i} mod p
1	3
2	9 mod 5 = 4
3	27 mod 5 = 2
4	81 mod 5 = 1

Other important properties:

1. [(x mod p) * (y mod p)] mod p = xy mod p
   Implies (x^u mod p)^v mod p = x^{uv} mod p

2. Euler's Theorem: g^{p-1} mod p = 1

3. Every value x = (g^{i} mod p) has inverse x^{-1}
   * (x * x^{-1} mod p) = 1
   * Computed as (g^{p-i-1} mod p)

D.H. Algorithm:

1. Compute & exchange values:
   - A computes x = g^a mod p
   - B computes y = g^b mod p
   - A & B exchange their values (OK for anyone to see these)
   - The function f(a) = g^a mod p is an example of a one-way function
     * Given x, it is hard to find a such that x = f(a).
     * So A & B can publish values of x & y without disclosing a or b.

2. Compute secret key
   - A computes key K = y^a mod p
   - B computes key K = x^b mod p
   - Note that both values are equal K = g^{ab} mod p
   - Even though A doesn't know b, and B doesn't know a.

3. Using as encryption key
   - Either side encrypts message M as C = (M*K mod p)

4. Decryption
   - Each can compute decryption key k such that (k * K mod p) = 1
   - For A: k = y^{p-1-a} mod p
     = g^{b(p-1-a)} mod p
     = g^{b(p-1)-ab} mod p = g^{-ab} mod p
       [Note above that we compute -ab mod p]
       [Derivation uses property that g^{b(p-1)} = (g^{p-1})^b = 1^b = 1]

   - For B: k = y^{p-1-b} mod p

   - In either case, decrypt C as M = (C*k mod p)

Important properties of Diffie-Hellman:

* Allows any two agents to set up a private session
* Generates both key & encryption method
* It's a symmetric form of public key cryptography.
  - Agent A's private key is a and public key is g^a mod p.
* Unlike RSA, does not lend itself to digital signature scheme
* Much easier to implement than RSA. In particular, it's very easy to generate new secret keys, since they're just big random numbers.

Authentication Protocol based on Diffie-Hellman

The key exchange part of Diffie-Hellman does nothing to authenticate the other party. It just creates a way for two parties to generate a shared key without the resulting key ever being communicated.

The following "Station-to-Station" (STS) protocol uses D-H + public key signatures to both generate a shared secret key & to authenticate the two parties to each other. (Published by Diffie, van Oorschot, & Wiener, 1992).

Components:
* A knows B's public key Kb+
* B knows A's public key Ka+.
* Values of g & p are publicly known.
* A has secret value a & public value x = g^a mod p.
* B has secret value b & public value y = g^b mod p.
* Both a & b were freshly generated to be used only for key exchange.
* A can compute secret key K from values of a & y
* B can compute secret key K from values of b & x

1. A-->B: A, x

* x is used as nonce

2. B-->A: y, K(Kb-(x,y))

* y is also used as nonce
* B signs x to indicate that it is responding to A.
* B signs y to indicate that this is the value B wants A to use in generating K.
* B encrypts this signed information with K
  - So that C cannot interfere by supplying signature Kc-(x,y)
  - Demonstrates to A that B has a working copy of K
  - Prevents B from being spoofed into providing signatures of arbitrary documents

3. A-->B: K(Ka-(x,y))

* Shows that:
  - Responding to step #2
  - Has working copy of K

This protocol has property "Perfect Forward Secrecy." Namely, if someone cracks secret key K generated during one session, that will not give any information about secret key K' generated for a different session. This relies on generating fresh values of a & b for every run of the protocol.

Key Management

We can see the need in several of these schemes for a trusted third party:

* Needham-Schroeder. Need to have trusted server. It manages a key for each authenticated user, and it can be trusted.

* Public key crypto. Need key distribution center that, when requested, can provide public key Ka+ for agent A.

All known schemes require a "root of trust." There must be SOMEONE that you trust.

E.g., public key distribution center. Suppose you trust a key server S, having public key Ks+.

Then A can ask: "What is the public key for B?." Server can respond with signed message [Ka+, Ks-(Ka+)].

In general, maintain tree of certification authorities. Root server S can delegate its authority to lower-level authority T by sending "certificate" [Kt+, Ks-(Kt+)]. Now we can use T as key distribution server just the same as we'd use S.

Usually associate limited time duration to certificate, so that privileges can be will expire if not renewed. That avoids complexities of having to revoke certification explicitly.