Week 5 - Monday

COMP 4290

Last time

- What did we talk about last time?
- Number theory
- RSA

Questions?

Assignment 2

Colm Oneacre Presents

Key Management

Key management

- Once you have great cryptographic primitives, managing keys is still a problem
- How do you distribute new keys?
  - When you have a new user
  - When old keys have been cracked or need to be replaced
- How do you store keys?
- As with the One Time Pad, if you could easily send secret keys confidentially, why not send messages the same way?

Notation for sending

- We will describe several schemes for sending data
- Let ***X*** and ***Y*** be parties and ***Z*** be a message
- { ***Z*** } ***k*** means message ***Z*** encrypted with key ***k***
- Thus, our standard notation will be:
  - $X \rightarrow Y$: { Z } k
  - Which means ***X*** sends message ***Z***, encrypted with key ***k***, to ***Y***
- ***X*** and ***Y*** will be participants like Alice and Bob and ***k*** will be a clearly labeled key
- ***A*** || ***B*** means concatenate message ***A*** with ***B***

Kinds of keys

- Typical to key exchanges is the idea of interchange keys and session keys
- An **interchange key** is a key associated with a particular user over a (long) period of time
- A **session key** is a key used for a particular set of communication events
- Why have both kinds of keys?

Possible attacks using single keys

- If only a single key (instead of interchange and session keys) were used, participants are more vulnerable to:
  - Known plaintext attacks (and potentially chosen plaintext attacks)
  - Attacks requiring many copies of encrypted material for comparison
  - **Replay attacks** in which old encrypted data is sent again from a malicious party
  - **Forward search attacks** in which a user computes many likely messages using a public key and thereby learns the contents of such a message when it is sent

Key exchange criteria

- To be secure, a key exchange whose goal is to allow secret communication from Alice to Bob must meet this criteria:
  1. Alice and Bob cannot transmit their key unencrypted
  2. Alice and Bob may decide to trust a third party (Cathy or Trent)
  3. Cryptosystems and protocols must be public, only the keys are secret

Classical exchange: Attempt 0

- If Bob and Alice have no prior arrangements, classical cryptosystems require a trusted third party Trent
- Trent and Alice share a secret key $\boldsymbol{k_{Alice}}$ and Trent and Bob share a secret key $\boldsymbol{k_{Bob}}$
- Here is the protocol:

1. Alice $\rightarrow$ Trent: {request session key to Bob} $\boldsymbol{k_{Alice}}$
2. Trent $\rightarrow$ Alice: $\{ \boldsymbol{k_{session}} \} \boldsymbol{k_{Alice}} \,||\, \{ \boldsymbol{k_{session}} \} \boldsymbol{k_{Bob}}$
3. Alice $\rightarrow$ Bob: $\{ \boldsymbol{k_{session}} \} \boldsymbol{k_{Bob}}$

What's the problem?

- Unfortunately, this protocol is vulnerable to a replay attack
- (Evil user) Eve records $\{ \boldsymbol{k_{session}} \} \boldsymbol{k_{Bob}}$ sent in step 3 and also some message enciphered with $\boldsymbol{k_{session}}$ (such as "Deposit $500 in Dan's bank account")
- Eve can send the session key to Bob and then send the replayed message
- Maybe Eve is in cahoots with Dan to get him paid twice
- Eve may or may not know the contents of the message she is sending
- The real problem is no **authentication**

Needham-Schroeder: Attempt 1

- We modify the protocol to add random numbers (called **nonces**) and user names for authentication

1. Alice $\rightarrow$ Trent: $\{ \text{Alice} \,||\, \text{Bob} \,||\, \boldsymbol{rand_1} \} \boldsymbol{k_{Alice}}$
2. Trent $\rightarrow$ Alice: $\{ \text{Alice} \,||\, \text{Bob} \,||\, \boldsymbol{rand_1} \,||\, \boldsymbol{k_{session}} \,||\, \{\text{Alice} \,||\, \boldsymbol{k_{session}} \} \boldsymbol{k_{Bob}} \} \boldsymbol{k_{Alice}}$
3. Alice $\rightarrow$ Bob: $\{ \text{Alice} \,||\, \boldsymbol{k_{session}} \} \boldsymbol{k_{Bob}}$
4. Bob $\rightarrow$ Alice: $\{ \boldsymbol{rand_2} \} \boldsymbol{k_{session}}$
5. Alice $\rightarrow$ Bob: $\{ \boldsymbol{rand_2} - 1 \} \boldsymbol{k_{session}}$

Problems with Needham-Schroeder

- Needham-Schroeder assumes that all keys are secure
- Session keys may be less secure since they are generated with some kind of (possibly predictable) pseudorandom generator
- If Eve can recover a session key (maybe after a great deal of computational work), she can trick Bob into thinking she's Alice as follows:

1. Eve $\rightarrow$ Bob: $\{$ Alice $||$ $\boldsymbol{k_{session}}$ $\}\boldsymbol{k_{Bob}}$
2. Bob $\rightarrow$ Alice: $\{$ $\boldsymbol{rand_3}$ $\}$ $\boldsymbol{k_{session}}$ [intercepted by Eve]
3. Eve $\rightarrow$ Bob: $\{$ $\boldsymbol{rand_3} - 1$ $\}$ $\boldsymbol{k_{session}}$

Denning and Sacco: Attempt 2

- Denning and Sacco use timestamps (*T*) to let Bob detect the replay
  1. Alice $\rightarrow$ Trent: $\{ \text{Alice} \,||\, \text{Bob} \,||\, \boldsymbol{rand_1} \}\, \boldsymbol{k_{Alice}}$
  2. Trent $\rightarrow$ Alice: $\{ \text{Alice} \,||\, \text{Bob} \,||\, \boldsymbol{rand_1} \,||\, \boldsymbol{k_{session}} \,||\, \{\text{Alice} \,||\, \boldsymbol{T} \,||\, \boldsymbol{k_{session}}\}\boldsymbol{k_{Bob}} \}\, \boldsymbol{k_{Alice}}$
  3. Alice $\rightarrow$ Bob: $\{ \text{Alice} \,||\, \boldsymbol{T} \,||\, \boldsymbol{k_{session}} \}\boldsymbol{k_{Bob}}$
  4. Bob $\rightarrow$ Alice: $\{ \boldsymbol{rand_2} \}\, \boldsymbol{k_{session}}$
  5. Alice $\rightarrow$ Bob: $\{ \boldsymbol{rand_2} - 1 \}\, \boldsymbol{k_{session}}$
- Unfortunately, this system requires synchronized clocks and a useful definition of when timestamp *T* is "too old"

Otway-Rees: Attempt 3

- The Otway-Rees protocol fixes these problem by using a unique integer ***num*** to label each session
  1. Alice $\rightarrow$ Bob: ***num*** || Alice || Bob || { $\boldsymbol{rand_1}$ || ***num*** || Alice || Bob } $\boldsymbol{k_{Alice}}$
  2. Bob $\rightarrow$ Trent: ***num*** || Alice || Bob || { $\boldsymbol{rand_1}$ || ***num*** || Alice || Bob } $\boldsymbol{k_{Alice}}$ || { $\boldsymbol{rand_2}$ || ***num*** || Alice || Bob } $\boldsymbol{k_{Bob}}$
  3. Trent $\rightarrow$ Bob: ***num*** || { $\boldsymbol{rand_1}$ || $\boldsymbol{k_{session}}$ } $\boldsymbol{k_{Alice}}$ || { $\boldsymbol{rand_2}$ || $\boldsymbol{k_{session}}$ } $\boldsymbol{k_{Bob}}$
  4. Bob $\rightarrow$ Alice: ***num*** || { $\boldsymbol{rand_1}$ || $\boldsymbol{k_{session}}$ } $\boldsymbol{k_{Alice}}$

Kerberos

- Strange as it seems, these key exchange protocols are actually used
- Kerberos was created at MIT as a modified Needham-Schroeder protocol (with timestamps)
  - Originally used to control access to network services for MIT students and staff
  - Current versions of Windows use a modified version of Kerberos for authentication
  - Many Linux and Unix implementations have an implementation of Kerberos
- Kerberos uses a central server that issues tickets to users which give them the authority to access a service on some other server

Public Key Exchange

Public key exchange

- Suddenly, the sun comes out!
- Public key exchanges should be really easy
- The basic outline is:
  1. Alice → Bob: $\{ k_{session} \} e_{Bob}$
- e_{Bob} is Bob's public key
- Only Bob can read it, everything's perfect!
- Except …
- There is still no authentication

Easily fixable

- Alice only needs to encrypt the session key with her private key
- That way, Bob will be able to decrypt it with her public key when it arrives
- New protocol:
  1. Alice $\rightarrow$ Bob: $\{\{ \boldsymbol{k}_{session} \} \boldsymbol{d}_{Alice} \} \boldsymbol{e}_{Bob}$
- Any problems now?

(Wo)man in the middle

- A vulnerability arises if Alice needs to fetch Bob's public key from a public server Peter
- Then, Eve can cause problems
- Attack:
  1. Alice $\rightarrow$ Peter: Send me Bob's key [intercepted by Eve]
  2. Eve $\rightarrow$ Peter: Send me Bob's key
  3. Peter $\rightarrow$ Eve: e_{Bob}
  4. Eve $\rightarrow$ Alice: e_{Eve}
  5. Alice $\rightarrow$ Bob: $\{ k_{session} \} e_{Eve}$ [intercepted by Eve]
  6. Eve $\rightarrow$ Bob $\{ k_{session} \} e_{Bob}$

Key Infrastructure and Storage

Key problems

- The previous man in the middle attack shows a significant problem
- How do we know whose public key is whose?
- We could sign a public key with a private key, but then…
- We would still be dependent on knowing the public key matching the private key used for signing
- It's a massive chicken and egg or bootstrapping problem

Certificate signature chains

- A typical approach is to create a long chain of individuals you trust
- Then, you can get the public key from someone you trust who trusts someone else who … etc.
- This can be arranged in a tree layout, with a central root certificate everyone knows and trusts
  - This system is used by X.509
- Alternatively, it can be arranged haphazardly, with an arbitrary web of trust
  - This system is used by PGP, which incorporates different levels of trust

Hash Function Motivation

Where do passwords go?

- What magic happens when you type your password into…
  - Windows or Unix to log on?
  - Amazon.com to make a purchase?
  - A *Cobra Kai* fan site so that you can post on the forums?
- A genie from the 8th dimension travels back in time and checks to see what password you originally created

In reality…

- The password is checked against a file on a computer
- But, how safe is the whole process?
  - *Cobra Kai* fan site may not be safe at all
  - Amazon.com is complicated, much depends on the implementation of public key cryptography
  - What about your Windows or Unix computer?

Catch-22

- Your computer needs to be able read the password file to check passwords
- But even an administrator shouldn't be able to read everyone's passwords
- Hash functions to the rescue!

Upcoming

Next time…

- Hash functions
- Birthday attacks
- Digital signatures
- Samuel Costa presents

Reminders

- **Office hours end at 3 p.m. today**
- Office hours on Friday from 1:45-4 p.m. are canceled
- Read section 12.5
- Work on Assignment 2
  - **Due Friday**