Thanks to the many, many people who have contributed various slides to this deck over the years.
Key Distribution

- Have network with n entities
- Add one more
  - Must generate n new keys
  - Each other entity must securely get its new key
  - Big headache managing $n^2$ keys!
- One solution: use a central keyserver
  - Needs n secret keys between entities and keyserver
  - Generates session keys as needed
- Downsides
  - Only scales to single organization level
  - Single point of failure
Symmetric Key Distribution

- How does Andrew do this?

Andrew Uses Kerberos, which relies on a Key Distribution Center (KDC) to establish shared symmetric keys.
Key Distribution Center (KDC)

- Alice, Bob need shared symmetric key.
- **KDC**: server shares different secret key with *each* registered user (many users)
- Alice, Bob know own symmetric keys, $K_{A-KDC}$, $K_{B-KDC}$, for communicating with KDC.
Q: How does KDC allow Bob, Alice to determine shared symmetric secret key to communicate with each other?

Alice and Bob communicate: using R1 as *session key* for shared symmetric encryption.
How Useful is a KDC?

- Must always be online to support secure communication
- KDC can expose our session keys to others!
- Centralized trust and point of failure.

In practice, the KDC model is mostly used within single organizations (e.g. Kerberos) but not more widely.
Kerberos

- **Trivia**
  - Developed in 80’s by MIT’s Project Athena
  - Used on all Andrew machines
  - Mythic three-headed dog guarding the entrance to Hades
- **Uses DES, 3DES**
- **Key Distribution Center (KDC)**
  - Central keyservier for a Kerberos domain
  - Authentication Service (AS)
    - Database of all master keys for the domain
    - Users’ master keys are derived from their passwords
    - Generates ticket-granting tickets (TGTs)
  - Ticket Granting Service (TGS)
    - Generates tickets for communication between principals
    - “slaves” (read only mirrors) add reliability
    - “cross-realm” keys obtain tickets in others Kerberos domains
Kerberos Authentication Steps
The first step in accessing a service that requires Kerberos authentication is to obtain a *ticket-granting ticket*.

To do this, the client sends a plain-text message to the AS:

- `<client id, KDC id, requested ticket expiration, nonce1>`
Kerberos Authentication Steps

Key Distribution Centre (KDC)

- Authentication Service (AS)
- Ticket Granting Service (TGS)

Client

- (1) AS_REQ
- (2) AS_REPLY
- (3) TGS_REQ
- (4) TGS_REPLY
- (5) APP_REQ
- (6) APP_REPLY

Service
(2) AS_REPLY

- $$\langle K_{c,TGS}, \text{none1} \rangle K_c, \{ticket_{c,tgs}\} K_{TGS}$$

- Notice the reply contains the following:
  - The nonce, to prevent replays
  - The new session key
  - A ticket that the client can’t read or alter

- A ticket:
  - $$ticket_{x,y} = \{x, y, \text{beginning valid time}, \text{expiration time}, K_{x,y}\}$$
Kerberos Authentication Steps

Key Distribution Centre (KDC)

1. AS_REQ
2. AS_REPLY
3. TGS_REQ
4. TGS_REPLY
5. APP_REQ
6. APP_REPLY
(3) TGS_REQUEST

- The TGS request asks the TGS for a ticket to communicate with a particular service.
- $\langle{\text{auth}}_c\rangle_{K_c, TGS}, \{\text{ticket}_c, TGS\}^{K_{TGS}}$, service, nonce2
- $\langle{\text{auth}}_c\rangle$ is known as an *authenticator*; it contains the name of the client and a timestamp for freshness.
Kerberos Authentication Steps
(4) TGS_REPLY

- \({K_{c,\text{service}}, \text{nonce2} \}K_{c, \text{TGS}}, \{\text{ticket}_{c, \text{service}} \}K_{\text{service}}\)>

- Notice again that the client can’t read or alter the ticket
- Notice again the use of the session key and nonce between the client and the TGS
(5) APP_REPLY

- \langle \text{auth}_c \text{K}_{c,\text{service}}, \text{ticket}_{c,\text{service}} \text{K}_{\text{service}}, \text{request}, \text{nonce3} \rangle

- Notice again the use of the session key as well as the protected ticket.
Kerberos Authentication Steps

Key Distribution Centre (KDC)

- Authentication Service (AS)
- Ticket Granting Service (TGS)

Client

- TGS_REQ
- TGS_REPLY
- APP_REQ
- APP_REPLY

Service

Server

- TGT
- Service TKT

Diagram:

1. AS_REQ
2. AS_REPLY
3. TGS_REQ
4. TGS_REPLY
5. APP_REQ
6. APP_REPLY
(6) APP_REPLY

• $\langle\text{nonce3}\rangle K_{c,\text{service}}, \text{response}\rangle$

• Because of the use of the encrypted nonce, the client is assured the reply came from the application, not an imposter.
Using Kerberos

- **kinit**
  - Get your TGT
  - Creates file, usually stored in /tmp
- **klist**
  - View your current Kerberos tickets
  ```shell
  unix41:~ebardsle> klist
  Credentials cache: FILE:/ticket/krb5cc_61189_9FT1N6
  Principal: ebardsle@ANDREW.CMU.EDU
<table>
<thead>
<tr>
<th>Issued</th>
<th>Expires</th>
<th>Principal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr 18 19:40:50</td>
<td>Apr 19 20:40:49</td>
<td>krbtgt/ANDREW.CMU.EDU@ANDREW.CMU.EDU</td>
</tr>
<tr>
<td>Apr 18 19:40:50</td>
<td>Apr 19 20:40:49</td>
<td><a href="mailto:afs@ANDREW.CMU.EDU">afs@ANDREW.CMU.EDU</a></td>
</tr>
<tr>
<td>Apr 18 19:40:51</td>
<td>Apr 19 20:40:49</td>
<td>imap/cyrus.andrew.cmu.edu@ANDREW.CMU.EDU</td>
</tr>
</tbody>
</table>
  ```
- **kdestory**
  - End session, destroy all tickets
- **kpasswd**
  - Changes your master key stored by the AS
- “Kerberized” applications
  - kftp, ktelnet, ssh, zephyr, etc
  - afslog uses Kerberos tickets to get AFS token
Asymmetric Key Crypto:

- Instead of shared keys, each person has a “key pair”
  - $K_B$ Bob’s public key
  - $K_B^{-1}$ Bob’s private key

- The keys are inverses, so: $K_B^{-1} (K_B (m)) = m$
Asymmetric Key Crypto:

- It is believed to be computationally unfeasible to derive $K_B^{-1}$ from $K_B$ or to find any way to get $M$ from $K_B(M)$ other than using $K_B^{-1}$.

$\Rightarrow$ $K_B$ can safely be made public.

Note: We will not detail the computation that $K_B(m)$ entails, but rather treat these functions as black boxes with the desired properties.
Asymmetric Key: Confidentiality

Bob’s public key

Bob’s private key

Encryption algorithm

Ciphertext

$K_B(m)$

Decryption algorithm

Plaintext message

$m = K_B^{-1} (K_B(m))$
Asymmetric Key: Sign & Verify

- If we are given a message $M$, and a value $S$ such that $K_B(S) = M$, what can we conclude?

- The message must be from Bob, because it must be the case that $S = K_B^{-1}(M)$, and only Bob has $K_B^{-1}$!

- This gives us two primitives:
  - **Sign** ($M$) = $K_B^{-1}(M)$ = Signature $S$
  - **Verify** ($S$, $M$) = test($K_B(S)$ == $M$)
Asymmetric Key: Integrity & Authentication

- We can use Sign() and Verify() in a similar manner as our HMAC in symmetric schemes.

**Integrity:**

$S = \text{Sign}(M)$

Message $M$

Receiver must only check Verify($M$, $S$)

**Authentication:**

Nonce

$S = \text{Sign}(\text{Nonce})$

Verify(Nonce, $S$)
Asymmetric Key Review:

- **Confidentiality**: Encrypt with Public Key of Receiver
- **Integrity**: Sign message with private key of the sender
- **Authentication**: Entity being authenticated signs a nonce with private key, signature is then verified with the public key

But, these operations are computationally expensive*
Cryptographic Hash Functions

- Given arbitrary length message $m$, compute constant length digest $h(m)$
- Desirable properties
  - $h(m)$ easy to compute given $m$
  - Preimage resistant
  - $2^{nd}$ preimage resistant
  - Collision resistant
- Crucial point: These are not inverted, they are recomputed
- Example use: file distribution (you're well aware of that!)
- Common algorithms: MD5, SHA
Digital Signatures

- Alice wants to convince others that she wrote message m
  - Computes digest \( d = h(m) \) with secure hash
  - Send \( <m,d> \)

- Digital Signature Standard (DSS)
The Dreaded PKI

• Definition:
  Public Key Infrastructure (PKI)

1) A system in which “roots of trust” authoritatively bind public keys to real-world identities

2) A significant stumbling block in deploying many “next generation” secure Internet protocol or applications.
Certification Authorities

- **Certification authority (CA):** binds public key to particular entity, E.
- An entity E registers its public key with CA.
  - E provides “proof of identity” to CA.
  - CA creates certificate binding E to its public key.
  - Certificate contains E’s public key AND the CA’s signature of E’s public key.

```
CA generates S = \text{Sign}(K_B)
```

- Certificate = Bob’s public key and signature by CA
Certification Authorities

- When Alice wants Bob’s public key:
  - Gets Bob’s certificate (Bob or elsewhere).
  - Use CA’s public key to verify the signature within Bob’s certificate, then accepts public key

\[ \text{Verify}(S, K_B) \]

If signature is valid, use \( K_B \)

\( K_B \)

\( K_{CA} \)
Certificate Contents

- Info algorithm and key value itself (not shown)
- Cert owner
- Cert issuer
- Valid dates
- Fingerprint of signature
Pretty Good Privacy (PGP)

- **History**
  - Written in early 1990s by Phil Zimmermann
  - Primary motivation is email security
  - Controversial for a while because it was too strong
    - Distributed from Europe
  - Now the OpenPGP protocol is an IETF standard (RFC 2440)
  - Many implementations, including the GNU Privacy Guard (GPG)

- **Uses**
  - Message integrity and source authentication
    - Makes message digest, signs with public key cryptosystem
    - Webs of trust
  - Message body encryption
    - Private key encryption for speed
    - Public key to encrypt the message’s private key
Secure Shell (SSH)

- Negotiates use of many different algorithms
- Encryption
- Server-to-client authentication
  - Protects against man-in-the-middle
  - Uses public key cryptosystems
  - Keys distributed informally
    - kept in ~/.ssh/known_hosts
  - Signatures not used for trust relations
- Client-to-server authentication
  - Can use many different methods
  - Password hash
  - Public key
  - Kerberos tickets
SSL/TLS

- History
  - Standard libraries and protocols for encryption and authentication
  - SSL originally developed by Netscape
    - SSL v3 draft released in 1996
    - TLS formalized in RFC2246 (1999)
- Uses public key encryption
- Uses
  - HTTPS, IMAP, SMTP, etc
Transport Layer Security (TLS) aka Secure Socket Layer (SSL)

- Used for protocols like HTTPS
- Special TLS socket layer between application and TCP (small changes to application).
- Handles confidentiality, integrity, and authentication.
- Uses “hybrid” cryptography.
Setup Channel with TLS “Handshake”

Handshake Steps:

1) Clients and servers negotiate exact cryptographic protocols

2) Client’s validate public key certificate with CA public key.

3) Client encrypt secret random value with servers key, and send it as a challenge.

4) Server decrypts, proving it has the corresponding private key.

5) This value is used to derive symmetric session keys for encryption & MACs.
How TLS Handles Data

1) Data arrives as a stream from the application via the TLS Socket.

2) The data is segmented by TLS into chunks.

3) A session key is used to encrypt and MAC each chunk to form a TLS “record”, which includes a short header and data that is encrypted, as well as a MAC.

4) Records form a byte stream that is fed to a TCP socket for transmission.
Works Cited/Resources

- [http://www.psc.edu/~jheffner/talks/sec_lecture.pdf](http://www.psc.edu/~jheffner/talks/sec_lecture.pdf)

Textbook: 8.1 – 8.3

- Wikipedia for overview of Symmetric/Asymmetric primitives and Hash functions.
- OpenSSL ([www.openssl.org](http://www.openssl.org)): top-rate open source code for SSL and primitive functions.
- “Handbook of Applied Cryptography” available free online: [www.cacr.math.uwaterloo.ca/hac/](http://www.cacr.math.uwaterloo.ca/hac/)