Authenticity & Authentication

- I want to provide evidence that an object (e.g. a message) was *endorsed by a particular person*
  - Signatures

- I want to provide evidence that *I am who I say I am*
  - Authentication protocols
Physical Signatures

- Consider a paper check used to transfer money from one person to another
- Signature confirms authenticity
  - Only legitimate signer can produce signature (true?)
- In case of alleged forgery
  - 3rd party can verify authenticity (maybe?)
- Checks are cancelled
  - So they can’t be reused
- Checks are not alterable
  - Or alterations are easily detected
Digital Signatures: Requirements I

- A mark that only one *principal* can make, but others can easily recognize

- **Unforgeable**
  - If P signs a message M with signature $S\{P,M\}$ it is computationally infeasible for any other principal to produce the pair $(M, S\{P,M\})$

- **Authentic**
  - If R receives the pair $(M, S\{P,M\})$ purportedly from P, R can check that the signature really is from P
Digital Signatures: Requirements II

- **Not alterable**
  - After being transmitted, \((M, S\{P, M\})\) cannot be changed by \(P, M\), or an interceptor

- **Not reusable**
  - A duplicate message will be detected by the recipient (why do we care?)
Digital Signatures with Shared Keys

Alice

\[ K_{AT}\{\text{msg}\} \]

Tom

\[ K_{TB}\{\text{Alice, msg, } K_{AT}\{\text{msg}\}\} \]

Bob

\[ K_{TB} \]

Tom is a trusted 3rd party (or arbiter).

**Authenticity:** Tom verifies Alice’s message, Bob trusts Tom.

**No repudiation:** Bob can keep msg, \( K_{AT}\{\text{msg}\} \), which only Alice could produce (or Tom, but he’s trusted not to)
Digital Signatures with Public Keys (RSA)

- RSA is commutative:
  - \( D(E(M, K), k) = E(D(M, k), K) \)

- Opposite from normal use of PK as cipher
  - Let \( K_A \) be Alice’s public key
  - Let \( k_A \) be her private key
  - To sign msg, Alice sends \( D(msg, k_A) \)
  - Bob can verify the message with Alice’s public key

- Works! RSA: \( (m^e)^d = m^{ed} = (m^d)^e \)
Digital Signatures with Public Keys

- No trusted 3rd party
- Simpler algorithm
- More expensive
- No confidentiality
Variations on Public Key Signatures

- Timestamps (to prevent replay)
  - Signature valid for only some time

- Add an extra layer of encryption to guarantee confidentiality
  - Alice sends $K_B{k_A\{msg\}}$ to Bob

- Combined with hashes for performance:
  - Send $K_B \{msg, k_A\{hash(msg)\}\}$
Authentication protocols

- More than just digital signatures…
- How do A and B convince *each other* that they are each A and B?
- Despite the fact that A&B are paranoid
- Cryptographic authentication protocols
Cryptographic Protocols

- Consider communication over a network…
- What is the threat model?
  - What are the vulnerabilities?
General Definition of “Protocol”

- A protocol is a multi-party algorithm
  - A sequence of steps that precisely specify the actions required of the parties in order to achieve a specified objective

- Every participant must know the protocol and the steps in advance
- Every participant must agree to follow the protocol
  - Honest participants
- But may not trust each other…

- Big problem: How to deal with bad participants?
What Can the Interceptor Do?

- Intercept them (confidentiality)
- Modify them (integrity)
- Fabricate other messages (integrity)
- Replay them (integrity)
- Block the messages (availability)
- Delay the messages (availability)
- Cut the wire (availability)
- Flood the network (availability)
Arbitrated Protocols

- Tom is an *arbiter*
  - Disinterested in the outcome (doesn’t play favorites)
  - Trusted by the participants (Trusted 3rd party)
  - Protocol can’t continue without T’s participation
Arbitrated Protocols (Continued)

- Real-world examples:
  - Lawyers, Bankers, Notary Public

- Issues:
  - Finding a trusted 3rd party
  - Additional resources needed for the arbitrator
  - Delay (introduced by arbitration)
  - Arbitrator might become a bottleneck
  - Single point of vulnerability: attack the arbitrator!
Adjudicated Protocols

- Alice and Bob record an *audit log*
- Only in exceptional circumstances do they contact a trusted 3rd party (3rd party is not always needed)
- Tom as the *adjudicator* can inspect the evidence and determine whether the protocol was carried out fairly
Self-Enforcing Protocols

- No trusted 3\textsuperscript{rd} party involved
- Participants can determine whether other parties cheat
- Protocol is constructed so that there are no possible disputes of the outcome
Examples We’ve Seen

- Arbitrated Protocol
  - Shared key digital signature algorithm
  - Trusted 3rd party provided authenticity

- Adjudicated Protocol
  - Public key digital signature algorithm
  - Bob can keep Alice’s digitally signed message
    - Trusted 3rd party provided non-repudiation
Authentication

- Goal: The honest claimant A is able to authenticate itself to the verifier B.
Threats

- **Impersonation**: The probability is negligible that a party C distinct from A can carry out the protocol in the role of A and cause B to accept it as having A’s identity.

- **Transferability**: B cannot reuse an identification exchange with A to successfully impersonate A to a third party C.
Shared-Key Authentication

Assume Alice & Bob already share a key $K_{AB}$.
- The key might have been decided upon in person or obtained from a trusted 3rd party.

Alice & Bob now want to communicate over a network, but first wish to authenticate to each other.
Solution 1: Weak Authentication

- Alice sends Bob $K_{AB}$
  - $K_{AB}$ acts as a password
- The secret (key) is revealed to passive observers
- Only works one-way
  - Alice doesn’t know she’s talking to Bob
Replay attacks

- **Replay**: the threat in which a transmission is observed by an eavesdropper who subsequently reuses it as part of a protocol, possibly to impersonate the original sender
  - Example: capture secret key $K_{AB}$ from previous example and send it to masquerade as Alice

- Three strategies for defeating replay attacks
  - Nonces
  - Timestamps
  - Sequence numbers
Nonces: Random Numbers

- **Nonce**: A number chosen at random from a range of possible values
  - Each generated nonce is ideally valid only once
- In a challenge-response protocol nonces are used as follows
  - The verifier chooses a (new) random number and provides it to the claimant
  - The claimant performs an operation on it showing knowledge of a secret
  - This information is bound inseparably to the random number and returned to the verifier for examination
Time Stamps

- The claimant sends a message with a timestamp
- The verifier checks that it falls within a some acceptance time window
- The last timestamp received is held, and identification requests with older timestamps are ignored

- Good only if clock synchronization is close enough for acceptance window
  - Attack clock synchronization
Sequence Numbers

- Sequence numbers provide a sequential or monotonic counter on messages.
- If a message is replayed and the original message was received, the replay will have an old or too-small sequence number and be discarded.
  - But, cannot detect forced delay.
- More difficult to maintain when there are system failures.
Solution 2: Strong(er) Authentication w/nonces

- **Challenge/Response**
  - Bob requests proof that Alice knows the secret
  - Alice requires proof from Bob
  - $R_A$ and $R_B$ are randomly generated numbers
- **Protocol doesn’t reveal the secret.**
Why not send more information in each message?
This seems like a simple optimization.
But, it’s broken… how?
**Attack: Marvin can Masquerade as Alice**

- Marvin pretends to take the role of Alice in two runs of the protocol
  - Tricks Bob into doing Alice’s part of the challenge!
  - Interleaves two instances of the same protocol
Lessons

- Protocol design is tricky and subtle
  - “Optimizations” aren’t necessarily good
- Need to worry about:
  - Multiple instances of the same protocol running in parallel
  - Intruders that play by the rules, mostly
General Principles

- In general: be careful – it’s very very easy to get this stuff wrong; resist the urge to create new protocols.

- Don’t do anything more than necessary until confidence is built.
  - Initiator should ideally prove their identity before the responder does any “expensive” action (like encryption).

- Embed the intended recipient of the message in the message itself (no ambiguity).

- The principal that generates a nonce should be the one that verifies it (i.e., don’t count on others).

- Before encrypting an untrusted message, add “salt” (random string) to prevent chosen plaintext attacks.
Key Establishment

- Key issue: who do you trust?
- Bilateral out-of-band
- Centralized third party
  - Needham-Schroeder
  - Kerberos
- Hierarchical
  - PKI
  - SSL
- Distributed
  - PGP
- Anarchistic
  - SSH
Out-of-band

- Easy… we meet… I validate who you are and give you a unique secret key and you do the same
- Can use these secrets to bootstrap unique point-to-point keys per session
- Should have limited validity duration and uses nonces
Point-to-Point

- Should also use timestamps & nonces.
- Session key should include a validity duration.
Third-party Key Distribution

Give me a key to talk with Bob

Here is the key

Tom gave us this session key
Needham-Schroeder Protocol

1. \( A \rightarrow T : \quad A, B, n_A \)

2. \( T \rightarrow A : \quad K_{AT}\{K_S, n_A, B, K_{BT}\{K_S, A}\} \)
   A decrypts with \( K_{AT} \) and checks \( n_A \) and \( B \). Holds \( K_S \) for future correspondence with \( B \).

3. \( A \rightarrow B : \quad K_{BT}\{K_S, A\} \)
   B decrypts with \( K_{BT} \).

4. \( B \rightarrow A : \quad K_S\{n_B\} \)
   A decrypts with \( K_S \).

5. \( A \rightarrow B : \quad K_S\{n_B - 1\} \)
   B checks \( n_B - 1 \).
Replay attack

- Problem: if an attacker can crack an old value of $K_S$ then can masquerade as Alice

- The attacker records the messages in an old run (in particular, step 3: $(K_{BT}\{K_S, A\})$

- That attacker can then masquerade as Alice by resending the old $K_{BT}\{K_S, A\}$
  - B can’t tell that it isn’t fresh
  - Could be prevented with timestamps

- Subsequent traffic just relies on $K_S$
Attack Scenario 1

1. $A \rightarrow T : \quad A, B, n_A$

2. $T \rightarrow C (A) : \quad K_{AT\{k, n_A, B, K_{BT\{K_S, A}}\}}$

C is unable to decrypt the message to A; passing it along unchanged does no harm. Any change will be detected by A.
Attack Scenario 2

1. \( A \rightarrow C (T) : \) \( A, B, n_A \)
2. \( C (A) \rightarrow T : \) \( A, C, n_A \)
3. \( T \rightarrow A : \) \( K_{AT}\{K_S, n_A, C, K_{CT}\{K_S, A}\}\)

Rejected by A because the message contains C rather than B.
Attack Scenario 3

1. $A \rightarrow C \ (T) : A, B, n_A$

2. $C \rightarrow T : C, B, n_A$

3. $T \rightarrow C : K_{CT}(K_S, n_A, B, K_{BT}(K_S, C))$

4. $C \ (T) \rightarrow A : K_{CT}(K_S, n_A, B, K_{BT}(K_S, C))$

A is unable to decrypt the message.
Attack Scenario 4

1. C → T : C, B, n_A
2. T → C : K_{CT}\{K_S, n_A, B, K_{BT}\{K_S, C}\}
3. C (A) → B : K_{BT}\{K_S, C}\}

B will see that the purported origin (A) does not match the identity indicated by T.
Kerberos

- Key exchange protocol developed at MIT in the late 1980’s
- Central server provides “tickets”
- Tickets – (also known as *capabilities*):
  - Unforgeable
  - Nonreplayable
  - Authenticated
  - Represent authority
- Core part of Windows domain controller
- Also widely used for FS authentication
  - AFS, Windows DFS, an option in NFS
- Also saves on authenticating for each service
  - e.g. with rlogin or rsh.
Kerberos (2)

- Based on symmetric cryptography

- One Key Distribution Center (KDC) per `Realm`
  - Authentication Service (AS) and
  - Ticket Granting Service (TGS)

- KDC supplies limited-lifetime “tickets” to principals
  - Ticket Granting Ticket, encrypted with hash of password
  - Service Tickets (ST), verified using the TGT

- Every service also shares a secret with the KDC
Kerberos

User Authentication Service

Ticket-granting service

Service Request

Service ticket

Authentication

TGT

Unique keys

File Server

Other Server

Auth. key

Service request
Kerberos Login

- U = User’s machine
- S = KDC Authentication service
  - Has a database of user passwords: userID → pwd
- G = KDC Ticket granting service

- U → S : userID, G, n_U
- S → U : K_{pwd}n_U, K_{UG}, K_{SG}\{T(U,G)\}
- S → G : K_{SG}\{K_{UG}, userID\}

- T(X,Y) = X, Y, L, K_{XY}

Kerberos ticket granting ticket
Ticket lifetime
Session key
Kerberos Service Request

- $U \rightarrow G : K_{UG}\{userID, t\}, K_{SG}\{T\}, \text{req}(F), n'_{A}$

- $G \rightarrow U : K_{UG}\{K_{UF}, n'_{A}\}, K_{F}\{T_{UF}\}$

- $U \rightarrow F : K_{UF}\{userID, t\}, K_{F}\{T_{UF}\}$
Kerberos in quasi-english

- User contacts KDC and gets TGT, encrypted using DES with hash of password as key
- TGT used to encrypt session where ST is requested from KDC TGS
- User gets ticket only when authorized by the KDC AS
- ST encrypted with password of service’s principal
- If service can decrypt ticket, it can be used to exchange new session key
Kerberos Benefits

- Distributed access control
  - No passwords communicated over the network

- Cryptographic protection against spoofing
  - All accesses mediated by G (ticket granting server)

- Limited period of validity
  - Servers check timestamps against ticket validity
  - Limits window of vulnerability

- Timestamps prevent replay attacks
  - Servers check timestamps against their own clocks to ensure “fresh” requests

- Mutual authentication
  - User sends nonce challenges
Kerberos Drawbacks

- Requires available ticket granting server
  - Could become a bottleneck
  - Must be reliable
- All servers must trust G, G must trust servers
  - They share unique keys
- Kerberos requires synchronized clocks
  - Replay can occur during validity period
  - Attack on clock synchronization
- User’s machine could save & replay passwords
  - Password is a weak spot
- Challenges with scaling
  - Must replicate authentication server and ticket granting server
  - Duplicating keys is bad, extra keys = more management
Public Key Infrastructures

- Public Key Infrastructures, or PKI, aims to solve the key distribution problem for public key crypto

- Trusted third party *(Certification Authority)* binds *authentication data* to a *public key*: the Certificate
PKI objects

- The PKI Certificate `X.509`
  - Structured message with:
    - public key
    - identifier(s) (e.g. amazon.com)
    - Lifetime (typically 1 year– fits wall street revenue model)
  - Digitally signed by a trusted third party

- Certification Authority (CA)
  - Binds identifiers to a public key
  - Expected to perform some amount of due diligence before vouching for this binding
  - Popular CA’s: Verisign, Thawte
  - Note that you must trust CA
Example: SSL/TLS

- Handshake protocol
  - Client contacts server
  - Server identifies self, and provides certificate
  - Client validates certificate and, if valid, uses public key to encrypt random session key
  - Client and server move to shared key crypto
  - Lots of other junk (negotiating algorithms, etc)
But I don’t want a single CA!

- Hierarchical trust
- Root CA (can have multiple)
- The sign public keys of second-level CA’s
- Etc
- Client verifies signatures up chain of trust
But I don’t want a single CA

(A) single CA

(B) Two level hierarchical PKI
Issue: Certificate Revocation

- What if a private key is lost or compromised?
  - Limited lifetime of certificates (typically 1yr)
  - Certificate revocation:
    » Certificate Revocation Lists (CRL’s): list of serial numbers of revoked certificates, signed by CA
    » Online Certificate Status Protocol (OCSP): protocol to verify the validity of a certificate online with (a delegate of) the CA
  - Emergency updates of software to remove CAs… (Diginotar)
Secure Shell (SSH)

- “Secure Shell (SSH) is a program to log into another computer over a network, to execute commands in a remote machine, and to move files from one machine to another.
- It provides strong authentication and secure communications over unsecure channels.
- It is intended as a replacement for telnet, rlogin, rsh, and rcp.”
SSH Authentication

- SSH authenticates using one or more of the following:
  - Password
  - Public key (RSA or DSA)
  - Kerberos
SSH with passwords

- Each SSH *daemon* has a RSA* key pair:
  - Public key is sent to the client
  - This is used to encrypt a (symmetric) *session key*
- Password and future data are sent over the encrypted session
Example Use

Logging into hosts:

$ ssh -l username hostname
$ ssh username@hostname
$ ssh hostname

Example:

$ ssh stevez@eniac.seas.upenn.edu uptime
The authenticity of host 'eniac.seas.upenn.edu (158.130.64.177)' can't be established.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'eniac.seas.upenn.edu' (RSA) to the list of known hosts.
stevez@eniac.seas.upenn.edu's password: <PASSWORD>
10:36am  up 31 day(s), 17:47,  72 users,  load average: 0.17, 0.19, 0.20
**SSH Protocol**

- **SSH Client**
- **Request connection**
- **SSH Server**

![Diagram](image)

**KS, KT**

**KS** stored in client vs. **KS** stored in server.

**KS{KT{r}}, 3DES || IDEA**

All traffic encrypted using $r$ and selected algorithm. Can do regular login (or something more complicated).

- $K_S = $ server’s public host key
- $K_T = $ server’s public key, changes every hour
- $r = $ 256-bit random number generated by client
Ssh with passwords (2)

- Problems with SSH password authentication:
  - Key distribution problem
    » how can the client verify that the host public key is correct? Asked first time
    » only trivial alerts against change of host key
  - No single sign-on
    (login to a new host requires typing the password)
  - Leads to using same password everywhere
    (lose password and its all over)
You are connecting to the host "shankly" for the first time. The host has provided you its identification, a host public key.

The fingerprint of the host public key is: "xufed-tacen-toves-recof-rucik-fapyb-caruz-sonih-synon-viryf-foxux"

You can save the host key to the local database by pressing YES. You can continue without saving the host key by pressing NO. You can also cancel the connection by pressing CANCEL.

Do you want to save the new host key to the local database?
ssh Error

HOST IDENTIFICATION HAS CHANGED

WARNING: HOST IDENTIFICATION HAS CHANGED!
1. Either the administrator has changed the host identification, or
2. The host has been upgraded from SSH1 to SSH2, or
3. SOMEONE COULD BE EAVESDROPPING ON YOU RIGHT NOW (man-in-the-middle attack)!

It is NOT RECOMMENDED to connect to the host until you have contacted your system administrator and find out why the host identification has changed.

Do you want to continue with the connection?

[Yes] [No] [Help]
Ssh with client keys

- Have the client generate an RSA key pair locally:
  
  `ssh-keygen → ~/.ssh/id_rsa & ~/.ssh/id_rsa.pub`

- The public part of this key is stored on remote server in user homedir:
  
  `~remoteuser/.ssh/authorized_keys2`

- `ssh remoteuser@remotestar`  
  challenge encrypted with public key sent to user; can he decrypt it?

- same keypair can be used for all hosts
ssh with client keys (2)

- The (local) user keypair is a very valuable target!
- Need to (symmetrically) encrypt the private key to keep it safe (~/.ssh/id_rsa)
- How to keep that symmetric key safe?
- Nothing comes for free…
Next class

- We’re done with crypto!
- Next week we’ll look at the human side of user authentication and then completely change gears and look at side channels
- Read Anderson Chap 2 on Usability and Psychology (on the Web site)