Distributed Systems
26. Authentication

Paul Krzyzanowski
Rutgers University
Fall 2016

Security Goals

- **Authentication**
  - Ensure that users, machines, programs, and resources are properly identified
- **Integrity**
  - Verify that data has not been compromised: deleted, modified, added
- **Confidentiality**
  - Prevent unauthorized access to data
- **Availability**
  - Ensure that the system is accessible

Authentication

- For a user (or process):
  - Establish & verify identity
  - Then decide whether to allow access to resources (= authorization)
- For a file or data stream:
  - Validate that the integrity of the data; that it has not been modified by anyone other than the author
  - E.g., digital signature

Identification vs. Authentication

- **Identification**: Who are you?
  - User name, account number, ...

- **Authentication**: Prove it!
  - Password, PIN, encrypt nonce, ...

Local authentication example: login

1. **Identification**
   - Get authentication info
   - Validate
   - exec(login_shell)

2. **Authentication**
   - setuid(user_id)
   - setgid(group_id)
   - Compare given password with stored password

3. **Access Control**
   - Good? Then change user ID and group ID of process
   - Replace the login process with the shell process

Versus Authorization

Authorization defines access control
Once we know a user’s identity:
- Allowed/deny request
- Operating systems
  - Enforce access to resources and data based on user’s credentials
- Network services usually run on another machine
  - Network server may not know of the user
  - Application takes responsibility
  - May contact an authorization server
  - Trusted third party that will grant credentials
    - Kerberos ticket granting service
    - RADIUS (centralized authentication/authorization)
    - OAuth service
Security

The Three A's
Authentication
Authorization
Accounting (+ Auditing)

Authentication

Three factors:
- something you have: key, card
  - Can be stolen
- something you know: passwords
  - Can be guessed, shared, stolen
- something you are: biometrics
  - Usually needs hardware, can be copied (sometimes)
  - Once copied, you're stuck

Multi-Factor Authentication

Factors may be combined
- ATM machine: 2-factor authentication
  - ATM card: something you have
  - PIN: something you know
- Password + code delivered via SMS: 2-factor authentication
  - Password: something you know
  - Code: validates that you possess your phone

Two passwords ≠ Two-factor authentication

PAP: Reusable passwords

Problem #1: Open access to the password file
What if the password file isn’t sufficiently protected and an intruder gets hold of it? All passwords are now compromised!

Even if a trusted admin sees your password, this might also be your password on other systems.

Solution:
- Store a hash of the password in a file
  - Given a file, you don’t get the passwords
  - Have to resort to a dictionary or brute-force attack
  - Example, passwords hashed with SHA-512 hashes (SHA-2)

Common Passwords

Adobe security breach (November 2013)
- 152 million Adobe customer records…with encrypted passwords
- Adobe encrypted passwords with a symmetric key algorithm
- …and used the same key to encrypt every password!

Top 26 Adobe Passwords
**What is a dictionary attack?**

- Suppose you got access to a list of hashed passwords
  - Brute-force, exhaustive search: try every combination
    - Letters (A-Z, a-z), numbers (0-9), symbols (! @#$%...)
    - Assume 30 symbols + 52 letters + 10 digits = 92 characters
    - Test all passwords up to length 8
    - Combinations = $92^8 + 92^7 + 92^6 + 92^5 + 92^4 + 92^3 + 92^2 + 92^1 = 5.189 \times 10^{15}$
    - If we test 1 billion passwords per second: $\approx 60$ days
  - But some passwords are more likely than others
    - 1,991,938 Adobe customers used a password = "123456"
    - 345,834 users used a password = "password"
- Dictionary attack
  - Test lists of common passwords, dictionary words, names
  - Add common substitutions, prefixes, and suffixes

**What is salt?**

- How to speed up a dictionary attack
  - Create a table of precomputed hashes
  - Now we just search a table
- Salt = random string (typically up to 16 characters)
  - Concatenated with the password
  - Stored with the password file (it’s not secret)
  - Even if you know the salt, you cannot use precomputed hashes to search for a password (because the salt is prefixed)
  - You will not have precomputed hash ("am$7b22QLpassword")

**PAP: Reusable passwords**

Problem 2: Network sniffing

Passwords can be stolen by observing a user’s session in person or over a network:
- snoop on telnet, ftp, rlogin, rsh sessions
- Trojan horse
- social engineering
- brute-force or dictionary attacks

Solutions:
1. Use one-time passwords
2. Use an encrypted communication channel

**One-time passwords**

Use a different password each time
- If an intruder captures the transaction, it won’t work next time

Three forms
1. Sequence-based: password = f(previous password)
2. Time-based: password = f(time, secret)
3. Challenge-based: f(challenge, secret)

**S/key authentication**

- One-time password scheme
- Produces a limited number of authentication sessions
- Relies on one-way functions

Authenticate Alice for 100 logins
- pick random number, R
- using a one-way function, f(x):

\[
\begin{align*}
x_1 &= f(R) \\
x_2 &= f(x_1) = f(f(R)) \\
x_3 &= f(x_2) = f(f(f(R))) \\
&\vdots \\
x_{100} &= f(x_{99}) = f(\ldots f(f(f(R)))\ldots)
\end{align*}
\]

- then compute:

\[
\begin{align*}
x_{101} &= f(x_{100}) = f(\ldots f(f(f(R)))\ldots)
\end{align*}
\]

Give this list to Alice


**S/key authentication**

Authenticate Alice for 100 logins

store \( x_{101} \) in a password file or database record associated with Alice

\[
\text{alice: } x_{101}
\]

Alice presents the last number on her list:

\[
\text{Alice to host: } \{ \text{"alice"}, x_{100} \}
\]

Host computes \( f(x_{100}) \) and compares it with the value in the database

- if \( (x_{100} \text{ provided by alice}) = \text{passwd(\"alice\")} \)
  - replace \( x_{101} \) in db with \( x_{100} \text{ provided by alice} \)
  - return success
- else
  - fail

next time: Alice presents \( x_{99} \)

if someone sees \( x_{100} \) there is no way to generate \( x_{99} \).

---

**Authentication: CHAP**

Challenge-Handshake Authentication Protocol

The challenge is a nonce (random bits).
We create a hash of the nonce and the secret.
An intruder does not have the secret and cannot do this!

---

**CHAP authentication**

\[
\text{Alice} \quad \text{network} \quad \text{host}
\]

\[
\text{\"alice\"} \quad \text{\"alice\"} \quad \text{look up alice's key, } K
\]

\[
C \quad \text{generate random challenge number } C
\]

\[
R' = f(K, C) \quad R = f(K, C) \quad R = R' ?
\]

an eavesdropper does not see \( K \)

---

**Authentication: MS-CHAP**

Microsoft’s Challenge-Handshake Authentication Protocol

The same as CHAP – we’re just hashing more things in the response

---

**SecurID card**

Username: paul
Password: 1234032848
PIN = passcode from card

Passcode changes every 60 seconds

1. Enter PIN
2. Press 0
3. Card computes password
4. Read password & enter

Password: 354982

Something you know
Something you have
SecurID card

- Proprietary device from RSA
  - SASL mechanism: RFC 2808

- Two-factor authentication based on:
  - Shared secret key (seed) Something you have
  - stored on authentication card
  - Shared personal ID – PIN Something you know
  - known by user

SecurID (SASL) authentication: server side

- Look up user’s PIN and seed associated with the token
- Get the time of day
  - Server stores relative accuracy of clock in that SecurID card
  - historic pattern of drift
  - adds or subtracts offset to determine what the clock chip on the SecurID card believes is its current time

- Passcode is a cryptographic hash of seed, PIN, and time
  - server computes \( f(\text{seed}, \text{PIN}, \text{time}) \)
- Server compares results with data sent by client

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server

Hi Bob, I’m Alice

Hi Bob, I’m Alice

What’s your password?

What’s your password?

It’s 123456

It’s 123456
Man-in-the-Middle Attacks

Password systems are vulnerable to **man-in-the-middle attacks**
- Attacker acts as the server

Guarding against man-in-the-middle

- **Use a covert communication channel**
  - The intruder won’t have the key
  - Can’t see the contents of any messages
  - But you can’t send the key over that channel!

- **Use signed messages**
  - Signed message = \{ message, encrypted hash of message \}
  - Both parties can reject unauthenticated messages
  - The intruder cannot modify the messages
    - Signatures will fail (they will need to know how to encrypt the hash)

Combined authentication and key exchange

Wide-mouth frog

• **Arbitrated protocol** — Trent (3rd party) has all the keys
• Symmetric encryption used for transmitting a session key
**Wide-mouth frog**

Alice ➔ Trent ➔ Bob

- "alice", $E(A, T_A, "bob", K)$
- $E(A, T_A, "alice", K)$

**Trent**
- Creates a new message
- New timestamp
- Identify source of the session key
- Encrypt the message for Bob
- Send to Bob

**Bob**
- Decrypts message
- Validates timestamp
- Extracts sender ("alice")
- Extracts session key K

Since Bob and Alice have the session key, they can communicate securely using the key.

**Kerberos**

- Authentication service developed by MIT
  - project Athena 1983-1988
- Trusted third party
- Symmetric cryptography
- Passwords not sent in clear text
  - assumes only the network can be compromised

Users and services authenticate themselves to each other.

To access a service:
- user presents a ticket issued by the Kerberos authentication server
- service examines the ticket to verify the identity of the user

Kerberos is a trusted third party
- Knows all (users and services) passwords
- Responsible for
  - Authentication: validating an identity
  - Authorization: deciding whether someone can access a service
  - Key exchange: giving both parties an encryption key (securely)
Kerberos

- User Alice wants to communicate with a service Bob
- Both Alice and Bob have keys

Step 1:
- Alice authenticates with Kerberos server
  - Gets session key and sealed envelope

Step 2:
- Alice gives Bob a session key (securely)
- Convinces Bob that she also got the session key from Kerberos

Authenticate, get permission

- Alice 
  - "I want to talk to Bob"
  - If Alice is allowed to talk to Bob, generate session key, S
  - Alice decrypts this:
    - Gets ID of "Bob's server"
    - Gets session key
    - Knows message came from AS
  - Convinces Bob that she also got the session key from Kerberos

Send key

- Alice 
  - Encrypts a timestamp with session key

Bob:
  - Decrypts envelope:
    - Sealed from Kerberos on request from Alice
    - Gets session key
    - Decrypts time stamp
    - Prevents replay attacks

Authenticate recipient of message

- Alice
  - Encrypts Alice's timestamp in return message

Bob:
  - Decrypts message
  - Alice & Bob communicate by encrypting data with S

Kerberos key usage

- Every time a user wants to access a service
  - User's password (key) must be used to decode the message from Kerberos
- We can avoid this by caching the password in a file
  - Not a good idea
- Another way: create a temporary password
  - We can cache this temporary password
  - Similar to a session key for Kerberos — to get access to other services
  - Split Kerberos server into Authentication Server + Ticket Granting Server

Ticket Granting Service (TGS)

- TGS + AS = KDC (Kerberos Key Distribution Center)
  - Authentication Server
    - Authenticates user, gives a session key to access the TGS
  - Before accessing any service, user requests a ticket to contact TGS
  - Ticket Granting Server
    - Anytime a user wants a service, request a ticket from TGS
    - Reply is encrypted with the TGS session key
  - TGS works like a temporary ID
Using Kerberos

$ kinit

Password: enter password

ask AS for permission (session key) to access TGS

Alice gets:

\[ ("TGS", S)_{TGS} \quad \text{TGS Ticket} \]

\[ ("Alice", S)_{TGS} \quad \text{Session key} \]

Compute key (A) from password to decrypt session key S and get TGS ID.

You now have a ticket to access the Ticket Granting Service

$ rlogin

rlogin uses the TGS Ticket to request a ticket for the rlogin service on somehost

Alice sends session key S, to TGS

Alice receives session key for rlogin service & ticket to pass to rlogin service

Alice gets:

\[ ("rlogin@somehost", S')_{TGS} \quad \text{TGS Ticket} \]

\[ ("Alice", S')_{TGS} \quad \text{Session key} \]

S' = session key for rlogin ticket for rlogin server on somehost

Public Key Authentication

Demonstrate we can encrypt or decrypt a nonce

This shows we have the right key

• Alice wants to authenticate herself to Bob:
  • Bob: generates nonce, S
    – Sends it to Alice
  • Alice: encrypts S with her private key (signs it)
    – Sends result to Bob

• For mutual authentication, Alice has to present Bob with a nonce that Bob will encrypt with his private key and return

Public key authentication

Bob:
1. Look up ‘alice’ in a database of public keys
2. Decrypt the message from Alice using Alice’s public key
3. If the result is S, then Bob is convinced he’s talking with Alice

For mutual authentication, Alice has to present Bob with a nonce that Bob will encrypt with his private key and return

Public key authentication

• Public key authentication relies on binding identity to a public key
  – How do you know it really is Alice’s public key?
  – One option: get keys from a trusted source
  – Problem: requires always going to the source
    – cannot pass keys around
  – Another option: sign the public key
    – Contents cannot be modified
    – digital certificate
ISO introduced a set of authentication protocols

**X.509: Structure for public key certificates:**
- **Issuer** = Certification Authority (CA)
- **Subject**
- **Public key** (algorithm & key)
- **Signature Algorithm
- **Signature**
- **Certificate data**
- **Version**
- **Serial**
- **Name**
- **Algorithm**
- **Validity Period**
- **Subject Public Key Algorithm & Key**
- **Signature Algorithm & Signature**

*Figure: X.509 v3 Digital Certificate*

### X.509 Certificates

- **Certificate issuance:**
  - **Verifies its signature:**
  - **Public key for certificates:**
  - **X.509: Structure for public key certificates:**
  - **X.509 Certificates:**
  - **Obtain CA's public key (certificate) from trusted source:**
  - **Certificates prevent someone from using a phony public key to masquerade as another person:**
    - *if you trust the CA*

### Reminder: What’s a digital signature?

- **Hash of a message encrypted with the signer’s private key:**
  - **Alice**
  - **Bob**
  - **H(P)**
  - **H(P) = E(D)**

### Built-in trusted root certificates in iOS 9

- **Apple Root CA**
- **CyberTrust**
- **EE Certification Centre Root CA**
- **GeoTrust**
- **Certum**
- **DigiCert**
- **Entrust.net**

*Partial list from support.apple.com/en-us/HT205205*

### Transport Layer Security (TLS)

- **SSLS/TLS:**
  - **aka Secure Socket Layer (SSL), which is an older protocol**
  - **Sits on top of TCP/IP:**
    - **Goal:** provide an encrypted and possibly authenticated communication channel
      - Provides authentication via RSA and X.509 certificates
      - Encryption of communication session via a symmetric cipher
    - **Hybrid cryptosystem:** (usually but also supports Diffie-Hellman)
      - Public key for authentication
      - Symmetric for data communication
    - **Enables TCP services to engage in secure, authenticated transfers:**
      - http, telnet, rtp, smtp, ...
Transport Layer Security (TLS)

1. Establish protocol version, cipher suite
   - Hello (version, cipher suites)
   - Hello (chosen version, chosen cipher suite)
   - Certificate (or public key)
   - Hello done
   - Certificate (only for client authentication)

2. Authenticate: unidirectional or mutual (optional)
   - Client authenticates server (optional)
     - Client nonce
     - Server nonce
   - Server authenticates client (optional)
     - Server nonce
     - Client nonce

3. Establish a session key for symmetric cryptography
   - Pick a session key
     - Encrypt with server's public key
   - Decrypt nonce with server's private key

4. Exchange data (symmetric encryption)
   - E(session key)
   - Encrypt & decrypt with session key and symmetric algorithm (e.g., RC4 or AES)

Optimizing reconnections: abbreviated handshake
- Goal: cache symmetric keys for clients
- Server sends a session ID during initial hello message
- Client & server save negotiated parameters and master secret (key)
- Client can use the session ID when reconnecting
- Clients and servers

SSL Keys ... more details
- SSL really uses four session keys
  - Ec – encryption key for messages from Client to Server
  - Mc – MAC encryption key for messages from Client to Server
  - Es – encryption key for messages from Server to Client
  - Ms – MAC encryption key for messages from Server to Client
- They are all derived from the random key selected by the client

You don't need to remember this!
OAuth 2.0

Service Authorization

- You want an app to access your data at some service
  - E.g., access your Google calendar data
- But you want to:
  - Not reveal your password to the app
  - Restrict the data and operations available to the app
  - Be able to revoke the app’s access to the data

OAuth 2.0: Open Authorization

- OAuth: framework for service authorization
  - Allows you to authorize one website (consumer) to access data from another website (provider) — in a restricted manner
  - Designed initially for web services
  - Examples:
    - Allow the Moo photo printing service to get photos from your Flickr account
    - Allow the NY Times to tweet a message from your Twitter account
- OpenID Connect
  - Remote identification: use one login for multiple sites
  - Encapsulated within OAuth 2.0 protocol

OAuth setup

- OAuth is based on
  - Getting a token from the service provider & presenting it each time an application accesses an API at the service
  - URL redirection
  - JSON data encapsulation
- Register a service
  - Service provider (e.g., Flickr):
    - Gets data about your application (name, creator, URL)
    - Assigns the application (consumer) an ID & a secret
    - Presents list of authorization URLs and scopes (access types)

OAuth Entities

You want an app to access your data at some service

- Application needs a Request Token from the Service Provider
- Request contains: client ID, client secret, scope (list of requested APIs)
- User must authenticate at the provider
- User authorizes the requested access
- Service Provider redirects back to consumer with a one-time-use authorization code
- Application exchanges the authorization code with the Service Provider
- Service Provider returns Access Token
- Application makes API requests to Service Provider using the Access Token
Key Points

- You still may need to log into the Provider’s OAuth service when redirected
- You approve the specific access that you are granting
- The Service Provider validates the requested access when it gets a token from the Consumer

Identity Federation: OpenID Connect

- Designed to solve the problem of
  - Having to get an ID per service (website)
  - Managing passwords per site
- Decentralized mechanism for single sign-on
  - Access different services (sites) using the same identity
  - Simply account creation at new site
  - User chooses which OpenID provider to use
  - OpenID does not specify authentication protocol — up to provider
  - Website never sees your password
- OpenID Connect is a standard but not the only solution
  - Used by Google, Microsoft, Amazon Web Services, PayPal, Salesforce, …
  - Facebook Connect — popular alternative solution (similar in operation but websites can share info with Facebook, offer friend access, or make suggestions to users based on Facebook data)

OpenID Connect Authentication

- OAuth requests that you specify a “scope”
  - List of access methods that the app needs permission to use
- To enable user identification
  - Specify “openid” as a requested scope
- Send request to server (identity provider)
  - Server requests user ID and handles authentication
- Get back an access token
  - If authentication is successful, the token contains:
    - User ID
    - Approved scopes
    - Expiration
e tc.

Cryptographic toolbox

- Symmetric encryption
- Public key encryption
- One-way hash functions
- Random number generators
  - Used for nonces and session keys

Examples

- Key exchange
  - Public key cryptography
- Key exchange + secure communication
  - Random # + Public key + symmetric cryptography
- Authentication
  - Nonce (random #) + encryption
- Message authentication codes
  - Hashes
- Digital signature
  - Hash + encryption with private key
The End