MAD 1 WEEK 9

Lecture 9.1: Access Control

What is Access Control?

- Refers to controlling the ability to read, write, or modify information.
- Not all parts of an application are open for public access (e.g., personal, financial, or confidential company data).

Types of Access:

1. **Read-only**: View information without changes.
2. **Read-write**: Create, read, update, delete (CRUD).
3. **Modify-only**: Edit existing content but not create new.

Examples of Access Control in Real Life:

- **Linux Files**:
  - Owner controls access to their files, others cannot modify unless permissions are changed.
  - Admin/root has the ability to override permissions.
- **Email**:
  - You can read your own emails and share them by forwarding, which also involves access.
- **E-commerce Login**:
  - Shopping cart and payment details are private to the logged-in user.

Discretionary vs Mandatory Access Control:

- **Discretionary**:
  - Users can decide who to share information with (e.g., forwarding emails).
- **Mandatory**:
  - Centralized control; users cannot share information without explicit permissions.
  - Common in military or high-security environments.

Role-Based Access Control (RBAC):

- Access is tied to roles, not individuals.

    - Example: A Head of Department (HoD) can access student records, and permissions transfer to the new HoD when roles change.
- Users can have multiple roles (e.g., HoD, Teacher, Club Member).
- Includes role hierarchies (e.g., HoD > Teacher > Student).

Attribute-Based Access Control (ABAC):

- Access depends on attributes like time of day, user age, or citizenship.
- Offers flexibility by combining attributes with roles.

Policies vs Permissions:

- **Permissions**:
    - Static rules (e.g., "Is the user in this group?").
- **Policies**:
    - More dynamic, combining multiple conditions.
    - Example: A bank employee can view ledgers, but only during working hours.

Principle of Least Privilege:

- Users should have the minimum access needed to perform their job.
    - Example: Regular users cannot modify system files.
- Benefits:
    - Enhanced security and stability.
    - Reduced accidental damage.

Privilege Escalation:

- Temporarily gaining elevated access (e.g., using `sudo`).
- Should be logged and used sparingly to avoid security risks.

Access Control in Web Applications:

- Restrict access at multiple levels:
    - **Hardware**: Physical tokens, secure devices.
    - **Operating System**: File access and memory restrictions.
    - **Application**: Database access controls.
    - **Web Applications**:
        - Controllers enforce access rules.

        - Python frameworks (e.g., Flask) use decorators to restrict user permissions.

Example:

- Gradebook:
    - Only admins can add, delete, or modify data.
    - Students can only view their own grades.

Lecture 9.2: Security Mechanisms

Types of Security Checks:

1. **Obscurity (Not Recommended):**
    - Relying on hidden details like using a non-standard port.
    - Easy to bypass if discovered, not a robust solution.
2. **Address-Based:**
    - Access controlled by the origin (e.g., host-based allow/deny rules).
3. **Login Credentials:**
    - Requires username and password for each user.
4. **Tokens:**
    - Unique, hard-to-duplicate tokens for authentication.
    - Common for machine-to-machine communication.

HTTP Authentication:

- **Basic Authentication:**
    - Server enforces authentication; returns status codes like:
        - `401 Unauthorized`: Requires authentication.
        - `403 Forbidden`: No option to authenticate.
        - `404 Not Found`: Resource does not exist.
    - Client sends credentials (access token) in the **header** of the next request.

Problems with Basic Auth:

- Credentials are encoded (not encrypted) and vulnerable without HTTPS.
- Passwords are visible in plain text to the server.
- No standard mechanism for logout.

Digest Authentication:

- Uses cryptographic hash functions (e.g., MD5, SHA256) for secure exchanges.
- Process:
  - Server sends a unique value (nonce) to prevent spoofing.
  - Client computes a response using:
    - `HA1 = MD5(username:realm:password)`
    - `HA2 = MD5(method:URI)`
    - `Response = MD5(HA1:nonce:HA2)`
  - Only the server and client can verify the response, preventing snooping.
- **Nonce** ensures one-time use to prevent replay attacks.

Client Certificates:

- Each client gets a secure certificate for authentication.
- Certificates are cryptographically secure and cannot be reversed to find keys.
- Requires careful handling on the client side.

Form-Based Authentication:

- Credentials (username/password) are sent via form submission.
- Security depends on the connection:
  - **GET Requests**: Data in the URL is insecure and prone to spoofing.
  - **POST Requests**: Data sent in the body is slightly more secure but still requires HTTPS.

Request-Level Security:

- For a single TCP connection: One security check may suffice.
- Without **Connection KeepAlive**:
  - Each request requires a new connection and re-authentication.

Cookies:

- Server sets cookies after successful authentication:
  - Example:

        - `Set-Cookie: <name>=<value>; Domain=<domain>; Secure; HttpOnly`.
    - Secure cookies ensure they are transmitted only over HTTPS.
    - **HttpOnly** cookies are inaccessible to JavaScript, reducing XSS risk.
- Cookies are sent with every client request.
- Sessions can have timeouts or allow logout by deleting cookies on the server.

API Security:

- APIs often use tokens or API keys instead of cookies.
- Tokens are better for non-browser clients (e.g., command-line tools, other apps).
- Same rules apply:
    - Use HTTPS.
    - Avoid passing tokens in the URL to prevent leakage.

Lecture 9.3: Session Management

Session Management Overview:

- Clients interact with the server through multiple requests.
- To enhance user experience, **state** information (e.g., login status, preferences like background color) is stored and used to customize responses.

Types of Session Storage:

1. **Client-Side Session:**
    - Data is fully stored in cookies on the client.
    - Security concerns arise as cookies can be modified or accessed.
2. **Server-Side Session:**
    - Data is stored on the server and referenced using an identifier (stored in a cookie).
    - More secure, as sensitive data is not directly exposed to the client.

Cookies:

- Cookies are key-value pairs set by the server using the `Set-Cookie` header.
- The client must send them back with every subsequent request.
- **Uses of Cookies:**
  - Non-sensitive data (e.g., theme, font size) can be directly stored in cookies.
  - Sensitive data (e.g., user permissions, session tokens) must be secure and non-editable.

Session Management with Flask (Examples):

1. **Basic Session Usage in Flask:**

```
from flask import Flask, session, redirect, url_for, request

app = Flask(__name__)
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'  # Secret key for session
encryption

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']  # Store username
in session
        return redirect(url_for('index'))
    return '''
    <form method="post">
        <p><input type=text name=username>
        <p><input type=submit value=Login>
    </form>
    '''
@app.route('/logout')
def logout():
    session.pop('username', None)  # Clear session data
    return redirect(url_for('index'))
```

Security Concerns:

1. **Cookie Modification:**
   - Users could alter their cookies to gain unauthorized access.
2. **Cookie Theft:**
   - If someone steals the cookie, they can impersonate the user.
   - Mitigation:
     - Set timeouts to invalidate sessions.
     - Validate the source IP.
3. **Cross-Site Request Forgery (CSRF):**
   - An attacker crafts a page to send unauthorized requests to another site where the user is logged in.
   - Mitigation: Verify the origin of requests on the server.

Server-Side Information Storage:

- Stores session information at the server, referenced by a minimal cookie value.
- Requires persistent backend storage such as:
  - File systems.
  - Databases.
  - Caching systems like Redis.

Enforcing Authentication:

- Protect sensitive parts of the application.
- Use **tokens** to enforce access.
- In Flask:
  - Protect views by wrapping controllers (Python functions) with **decorators** that check authentication.

Example with Flask-Login:

```
from flask_login import login_required, logout_user, current_user

@app.route('/profile')
@login_required  # Ensures only logged-in users can access
def profile():
    return f'Welcome {current_user.name}!'
```

```python
@app.route('/logout')
@login_required
def logout():
    logout_user()  # Ends session and logs out user
    return redirect(url_for('index'))
```

Transmitted Data Security:

- Assume all network communication can be intercepted (“tapped”).

1. **Risks:**
   - Sensitive data in **HTTP GET URLs** can be logged by firewalls or proxies.
   - Data in **POST requests** or cookies is vulnerable without secure connections.
2. **Mitigation:**
   - Use **HTTPS** to encrypt all transmitted data.
   - Ensure cookies are marked as:
     - `Secure`: Sent only over HTTPS.
     - `HttpOnly`: Inaccessible to client-side scripts, mitigating XSS risks.

Making the Wire Safe (Securing Data in Transit):

- Encrypt all communications using TLS/SSL protocols.
- Avoid transmitting sensitive data (e.g., session tokens) in GET URLs.
- Use tools and frameworks that enforce secure transmission by default.

Lecture 9.4: HTTPS

Normal HTTP Process:

1. Client connects to the server on a fixed network port (default: 80).
2. HTTP request is sent, and the HTTP response is received.

Safety Concerns:

- **Data Tapping:** Information can be intercepted during transmission.

- **Data Alteration:** Data can be modified by attackers.

Secure Sockets (Encryption):

- Creates an **encrypted channel** between the client and server.
- **How it works:**
  1. A shared **secret key** (e.g., a long binary string) is used to encrypt and decrypt data.
  2. **Encryption process:** XOR input data with the key to generate ciphertext.
  3. Without the key, attackers cannot derive the original data.

Challenge:

- Establishing a shared secret securely over an insecure channel.
- Solutions:
  - **Pre-existing key:** Requires an out-of-band secure method (e.g., SMS or post).
  - **TLS/SSL protocols:** Uses public-key cryptography to establish shared secrets dynamically.

Types of Security in HTTPS:

1. **Channel (Wire) Security:**
   - Ensures data transmitted over the wire cannot be intercepted or altered.
2. **Server Authentication:**
   - Confirms the server's identity (e.g., mail.google.com).
   - Prevents attacks like DNS hijacking by validating the server’s certificate against a trusted authority.
3. **Client Certificates:**
   - Rarely used but required in specific scenarios (e.g., corporate intranets).
   - Allows servers to authenticate clients via client-provided certificates.

Chain of Trust:

- Certificates are issued in a hierarchy:
  - Server’s certificate (e.g., for mail.google.com).
  - Certificate Authority (CA) intermediate certificate (e.g., GTS CA1C3).
  - Root certificate (e.g., GTS Root R1).
- **Root Certificates:**
  - Stored in the operating system or browser.
  - Serve as the ultimate trust anchor for validating all certificates in the chain.

Potential Problems with HTTPS:

1. **Old Browsers:**
    - May lack updated certificate chains, leading to validation failures.
2. **Compromised Root Certificates:**
    - Certificates at the root of trust can be stolen.
    - Mitigation: Certificate revocation and trust store updates.
3. **DNS Hijacking:**
    - Attackers provide false IP addresses and fake certificates.
    - Protection: Validation against trusted root certificates ensures security.

Wildcard Certificates:

- A single certificate that secures all subdomains of a domain (e.g., *.example.com).
- Simplifies certificate management but increases risk if compromised.

Impact of HTTPS:

Advantages:

1. Protects data against wiretapping and eavesdropping.
2. Ensures secure communication, especially on public WiFi networks.

Disadvantages:

1. **Caching Limitations:** Proxies cannot cache encrypted content.
2. **Performance Impact:** Encryption and decryption introduce runtime overhead.

Lecture 9.5: Logging

What is Logging?

- Logging involves recording all accesses and events within an application.

Why is Logging Important?

1. **Debugging:** Identify and resolve bugs.

2. **Usage Analytics:** Track visits and usage patterns.
3. **Optimization:** Discover popular links and optimize performance.
4. **Security Monitoring:** Detect unusual activities or threats.

How to Implement Logging?

- Logging can be built into the app:
    1. **Log File Output:** Store logs in dedicated files.
    2. **Analysis Pipeline:** Direct logs to systems for real-time analysis and insights.

Server-Level Logging:

- **Web Servers:** Logging is often built into servers like Apache and Nginx.
- **Logged Data:**
    - Access details (e.g., URL accessed).
    - Requests per second.
    - Security anomalies (e.g., malformed URLs, unusual request patterns).

Indicators of Security Threats:

- **High Request Volume:** Large bursts of requests in short intervals.
- **Unusual Requests:** Repeated access to obscure or restricted endpoints.

Application-Level Logging:

- **Python Logging Framework:**
    - Outputs logs to files or other "stream" handlers (e.g., console, databases).
- **Logged Data:**
    - Controller and data model interactions.
    - Server errors and application-specific events.
    - Potential security vulnerabilities.

Log Rotation:

- **Challenges:**
    - High-volume logs take significant storage space.
    - Older logs may become irrelevant.
- **Solution:**
    - Rotate logs to manage storage effectively.
    - **Process:**

        1. Keep the last **N log files**.
        2. Delete the oldest file when the limit is reached.
        3. Rename files sequentially (e.g., `log.1` → `log.2`).
    - Ensures a **fixed storage footprint** for logs.

Logs on Custom App Engines:

- **Custom Logging:** Platforms like Google App Engine provide tailored logging capabilities.
- **Features:**
    - Generate reports for app performance and usage.
    - Perform automated security analyses on log data.

Time-Series Analysis in Logs:

- **Logs with Timestamps:** Essential for tracking when events occur.
- **Applications:**
    - **Event Density:** Monitor the number of events per time unit.
    - **Incident Analysis:** Pinpoint the exact time of specific incidents.
    - **Pattern Detection:** Identify trends like periodic spikes or sudden traffic increases.
- **Time-Series Databases:**
    - Tools like **RRDTool**, **InfluxDB**, and **Prometheus** are used for:
        - Storing logs as time-series data.
        - Analyzing and visualizing trends effectively.