EventConnect - Security Features Demo Script

Total Duration: 15-20 minutes **Target Audience:** Instructors, security reviewers, stakeholders **Platform:** Screen recording with narration

---

Pre-Demo Setup Checklist

Required Tools & Access:

- [ ] EventConnect frontend running (https://event-connect.site or localhost:5173)
- [ ] MongoDB Compass connected to database
- [ ] Browser DevTools open (Chrome/Edge recommended)
- [ ] Postman or Thunder Client for API testing
- [ ] Backend server running (https://event-connect-jin2.onrender.com or localhost:5000)
- [ ] Test accounts ready:
  - Organizer: `organizer@test.com` / password
  - Participant: `participant@test.com` / password

Browser Setup:

- [ ] Clear browser cache and cookies
- [ ] Open DevTools (F12)
- [ ] Network tab ready
- [ ] Application tab ready (for cookies/storage)
- [ ] Console tab ready

PART 1: Introduction (1 minute)

Script:

> **[Screen: EventConnect landing page]**
>
> "Hello, I'm presenting the security implementation of EventConnect, an intelligent attendance tracking platform. This demo will showcase **8 security features** we've implemented to protect user data, prevent unauthorized access, and ensure system integrity.
>
> **[Screen: Show security features list]**
>
> We'll demonstrate:
>
> 1. Password Hashing with Bcrypt
> 2. Role-Based Access Control
> 3. Input Validation & Injection Prevention
> 4. Rate Limiting
> 5. Session Management
> 6. Security Headers (Helmet)

> 7. JWT Authentication
> 8. CORS Configuration
>
> Let's begin with user registration and authentication."

Actions:

- Show EventConnect homepage
- Display agenda slide/overlay with 8 security features
- Transition to registration page

PART 2: Password Hashing with Bcrypt (3 minutes)

Script:

> **[Screen: Registration page]**
>
> "First, let's demonstrate how EventConnect securely stores passwords using bcrypt hashing with 12 rounds of salting."

Demo Steps:

Step 1: Weak Password Rejection (30 seconds)

Actions:

1. Navigate to `/register`
2. Enter:
   - Name: `John Doe`
   - Email: `john.demo@test.com`
   - Password: `12345` (only 5 characters)
   - Role: Organizer
3. Click "Register"

Expected Result: Validation error

Script:

> "Notice that when I try to register with a weak password of only 5 characters, the system rejects it with a validation error: 'Password must be at least 6 characters.' This is our first layer of password security - input validation."

[Highlight error message on screen]

Step 2: Successful Registration (30 seconds)

Actions:

1. Change password to: `SecurePass123!`
2. Click "Register"

Expected Result: Success, redirected to dashboard

Script:

> "Now with a strong password, registration succeeds. But here's the critical part - this password is **never stored in plain text**. Let me show you what actually gets saved in the database."

Step 3: Database View - Bcrypt Hash (1 minute)

Actions:

1. Switch to MongoDB Compass
2. Navigate to `eventconnect` database → `users` collection
3. Find the newly created user
4. Highlight the `password` field

Expected View:

```
{
  "_id": "ObjectId('...')",
  "name": "John Doe",
  "email": "john.demo@test.com",
  "password": "$2a$12$KxLZPnR3ZT9hTZ8eVz8YuJ3WxY7nKYOZqM5vZ8YhZ9hZ0hZ1hZ2h",
  "role": "organizer",
  "createdAt": "2025-01-15T10:30:00.000Z"
}
```

Script:

> "As you can see in MongoDB Compass, the password field contains a hash, not the actual password. The format `$2a$12$...` tells us:
>
> - `$2a$` - Bcrypt algorithm
> - `$12$` - 12 rounds of hashing (that's 2^12 = 4,096 iterations)
> - The rest is the salt and hash combined
>
> Even if an attacker gains access to our database, they cannot reverse this hash to get the original password. Let me demonstrate this further."

[Zoom in on password hash]

Step 4: Same Password, Different Hash (1 minute)

Actions:

1. Create another user with the SAME password
   - Name: `Jane Doe`
   - Email: `jane.demo@test.com`

    - Password: `SecurePass123!` (same as before)
    - Role: Participant
2. Register successfully
3. Switch to MongoDB Compass
4. Show both users side-by-side

Expected View:

```
// John's password hash
"password": "$2a$12$KxLZPnR3ZT9hTZ8eVz8YuJ3WxY7nKYOZqM5vZ8YhZ9hZ0hZ1hZ2h"

// Jane's password hash (DIFFERENT even though password is same)
"password": "$2a$12$Df3kLm9pN4qR6sT9vW2xZ5aC8eG1hJ4kM7oP0qR3sT6uV9wX2yA5"
```

Script:

> "Notice that even though both users have the exact same password, their hashes are completely different. This is because bcrypt generates a unique salt for each password. This prevents rainbow table attacks where attackers use pre-computed hash databases.
>
> Now let's see how password verification works during login."

PART 3: Login & Session Management (3 minutes)

Script:

> "Next, I'll demonstrate our dual authentication system: session-based authentication for web users and JWT for mobile apps."

Demo Steps:

Step 1: Successful Login (1 minute)

Actions:

1. Navigate to `/login`
2. Open Browser DevTools → Application tab → Cookies
3. **BEFORE LOGIN:** Show no session cookie exists
4. Enter credentials:
    - Email: `john.demo@test.com`
    - Password: `SecurePass123!`
    - Role: Organizer
5. Click "Login"
6. **AFTER LOGIN:** Show cookies

Expected Result:

- Redirect to organizer dashboard

- Cookie created

Script:

> "Before logging in, notice there are no session cookies. Now I'll log in with the correct credentials."
>
> **[Enter credentials and login]**
>
> "Login successful! Now let's examine what happened behind the scenes."

[Switch to DevTools → Application → Cookies]

Expected Cookie:

```
Name: connect.sid
Value: s%3Aj8F3kL9mN2pQ5rT8vW1xY4zA7bC0dE6fG9hI2jK5...
Domain: event-connect.site
Path: /
Expires: 2025-01-22 (7 days from now)
HttpOnly: ✓ (checked)
Secure: ✓ (checked)
SameSite: None
```

Script:

> "Here's our session cookie called 'connect.sid'. Notice these critical security properties:
>
> **[Highlight each property]**
>
> 1. **HttpOnly: True** - This prevents JavaScript from accessing the cookie, protecting against XSS attacks
> 2. **Secure: True** - Cookie is only sent over HTTPS, preventing interception
> 3. **SameSite: None** - Allows cross-origin requests (though this has security implications we'll discuss)
> 4. **Expires in 7 days** - Automatic session expiration
>
> Let me prove that JavaScript cannot access this cookie."

Step 2: HttpOnly Cookie Test (30 seconds)

Actions:

1. Open DevTools → Console tab
2. Type: `document.cookie`
3. Press Enter

Expected Result:

```
"" (empty string, session cookie NOT visible)
```

Script:

> "When I try to access cookies via JavaScript using `document.cookie`, the session cookie is not visible. This is the HttpOnly flag in action - even malicious scripts injected via XSS cannot steal our session cookie.
>
> Now let's look at the JWT token."

Step 3: JWT Token Inspection (1 minute 30 seconds)

Actions:

1. DevTools → Application → Local Storage → `https://event-connect.site`
2. Show `token` key with JWT value
3. Copy the JWT token
4. Open new tab → https://jwt.io
5. Paste token in debugger

Expected Token Structure:

```
Header:
{
  "alg": "HS256",
  "typ": "JWT"
}

Payload:
{
  "id": "67a1b2c3d4e5f6g7h8i9j0k1",
  "iat": 1640000000,
  "exp": 1640604800
}

Signature:
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  your-256-bit-secret
)
```

Script:

> "EventConnect also issues a JWT token for mobile app compatibility. Let me decode this token at jwt.io.
>
> **[Paste token]**
>
> The token has three parts:
>
> **[Highlight each section]**

> 1. **Header**: Specifies HS256 algorithm
> 2. **Payload**: Contains user ID and expiration (7 days from now)
> 3. **Signature**: Cryptographically signed with our secret key
>
> This signature ensures the token hasn't been tampered with. Let me demonstrate."

Step 4: JWT Tampering Test (30 seconds)

Actions:

1. In jwt.io, modify the payload (change user ID)
2. Copy the modified token
3. Back in DevTools → Application → Local Storage
4. Replace the token with modified one
5. Try to access `/organizer-dashboard` or make API call

Expected Result: 401 Unauthorized

Script:

> "If I modify the token - let's say I change the user ID - and try to use it, the server rejects it because the signature no longer matches. The token is invalid."
>
> **[Show 401 error in Network tab]**

Step 5: Session in MongoDB (30 seconds)

Actions:

1. Switch to MongoDB Compass
2. Navigate to `sessions` collection
3. Show the active session document

Expected View:

```
{
  "_id": "GJ3K5L7M9N1P3Q5R7S9T1U3V5W7X9Y1Z",
  "expires": "2025-01-22T08:00:00.000Z",
  "session": {
    "cookie": {
      "originalMaxAge": 604800000,
      "expires": "2025-01-22T08:00:00.000Z",
      "secure": true,
      "httpOnly": true,
      "sameSite": "none"
    },
    "userId": "67a1b2c3d4e5f6g7h8i9j0k1",
    "userRole": "organizer"
    }
  }
```

Script:

> "The session data is stored server-side in MongoDB. Only the session ID is stored in the cookie. This means even if someone steals the session ID, they can't modify the user's role or permissions - that's all on the server."

PART 4: Role-Based Access Control (3 minutes)

Script:

> "Now let's demonstrate role-based access control - how EventConnect restricts functionality based on user roles."

Demo Steps:

Step 1: Participant Cannot Create Events (1 minute)

Actions:

1. Logout from organizer account
2. Login as participant:
   - Email: `participant@test.com`
   - Password: `password`
   - Role: Participant
3. Try to access `/organizer-dashboard` in address bar

Expected Result: Redirect to home page with error toast

Script:

> "I'm now logged in as a participant. Let me try to access the organizer dashboard by typing the URL directly."
>
> **[Type URL and press Enter]**
>
> "Notice I'm immediately redirected and shown an error: 'This page is only accessible to organizers.' This is our frontend route guard in action.
>
> But frontend protection isn't enough - a determined attacker could bypass JavaScript. Let's test the backend protection."

Step 2: API-Level Access Control (1 minute 30 seconds)

Actions:

1. Open Postman/Thunder Client

2. Get the participant's JWT token from localStorage
3. Create POST request:
    - URL: `https://event-connect-jin2.onrender.com/api/events`
    - Headers: `Authorization: Bearer {participant_token}`
    - Body:

```
{
  "title": "Hacker Conference",
  "date": "2025-02-15T09:00:00Z",
  "location": {
    "address": "Test Location",
    "coordinates": {
      "type": "Point",
      "coordinates": [103.8198, 1.3521]
    }
  }
}
```

4. Send request

Expected Response:

```
HTTP/1.1 403 Forbidden
Content-Type: application/json

{
  "success": false,
  "message": "Access denied. Organizer role required."
}
```

Script:

> "Even when I bypass the frontend and send a direct API request with the participant's token, the backend rejects it with 403 Forbidden: 'Organizer role required.'
>
> **[Show response in Postman]**
>
> This is server-side role-based access control. The backend always verifies the user's role before allowing any action. Never trust the client!"

Step 3: Organizer Can Create Events (30 seconds)

Actions:

1. Login as organizer again
2. Copy organizer's JWT token
3. In Postman, replace token with organizer's

4. Send same request

Expected Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "success": true,
  "message": "Event created successfully",
  "data": {
    "event": {
      "_id": "65a1b2c3d4e5f6g7h8i9j0k1",
      "title": "Hacker Conference",
      ...
    }
  }
}
```

Script:

> "Now with the organizer token, the exact same request succeeds. This demonstrates that RBAC is working correctly at the API level."

PART 5: Input Validation & Injection Prevention (3 minutes)

Script:

> "Let's test our input validation and injection prevention mechanisms."

Demo Steps:

Step 1: Empty Field Validation (30 seconds)

Actions:

1. In Postman, modify event creation request:

   ```
   {
     "title": "",
     "date": "2025-02-15T09:00:00Z"
   }
   ```

2. Send request

Expected Response:

```
HTTP/1.1 400 Bad Request

{
  "success": false,
  "message": "Validation failed",
  "errors": [
    {
      "type": "field",
      "msg": "Event title is required",
      "path": "title",
      "location": "body"
    },
    {
      "type": "field",
      "msg": "Event location is required",
      "path": "location.address",
      "location": "body"
    }
  ]
}
```

Script:

> "When I send invalid data - like an empty title - the server responds with detailed validation errors listing exactly what's wrong. This prevents corrupt data from entering our database."

Step 2: Invalid Email Format (30 seconds)

Actions:

1. Try to register with invalid email:

   ```
   {
     "name": "Test User",
     "email": "notanemail",
     "password": "password123",
     "role": "participant"
   }
   ```

Expected Response:

```
HTTP/1.1 400 Bad Request

{
  "success": false,
  "message": "Validation failed",
  "errors": [
```

```
    {
      "msg": "Valid email is required",
      "path": "email"
    }
  ]
}
```

Script:

> "Email format is validated using regex patterns. Invalid emails are rejected before they even reach the database."

Step 3: GPS Coordinates Validation (30 seconds)

Actions:

1. Try to create event with invalid coordinates:

   ```
   {
     "title": "Test Event",
     "location": {
       "address": "Test",
       "coordinates": {
         "type": "Point",
         "coordinates": [200, -100]
       }
     }
   }
   ```

Expected Response:

```
HTTP/1.1 400 Bad Request

{
  "errors": [
    {
      "msg": "Longitude must be valid",
      "path": "location.coordinates.coordinates[0]"
    },
    {
      "msg": "Latitude must be valid",
      "path": "location.coordinates.coordinates[1]"
    }
  ]
}
```

Script:

> "Geographic coordinates are validated to ensure they're within valid ranges: longitude -180 to 180, latitude -90 to 90. Invalid coordinates that could break mapping functionality are rejected."

Step 4: NoSQL Injection Prevention (1 minute)

Actions:

1. Try to login with NoSQL injection payload:

   ```
   {
     "email": { "$ne": null },
     "password": { "$ne": null }
   }
   ```

2. Send to `/api/auth/login`

Expected Result: Either validation error or login failure

Script:

> "A common NoSQL injection attack is to send objects instead of strings, like `{ "$ne": null }` which means 'not equal to null' in MongoDB. This could bypass authentication.
>
> **[Send request]**
>
> However, our system rejects this because:
>
> 1. Express-validator expects strings for email/password
> 2. Mongoose automatically sanitizes and type-checks inputs
>
> The attack fails, and no unauthorized access is granted."

[Show failed login response]

Step 5: XSS Prevention (30 seconds)

Actions:

1. Create event with XSS payload in title:

   ```
   {
     "title": "<script>alert('XSS')</script>",
     "date": "2025-02-15T09:00:00Z",
     "location": { ... }
   }
   ```

2. Event created successfully

3. View event in frontend

Expected Behavior: Script tags displayed as plain text, not executed

Script:

> "If I try to inject a JavaScript payload in the event title, it gets stored but rendered as plain text when displayed. React automatically escapes HTML, preventing XSS attacks."

[Show event list with safe text rendering]

PART 6: Rate Limiting (2 minutes)

Script:

> "To prevent brute-force attacks and API abuse, EventConnect implements rate limiting."

Demo Steps:

Step 1: Normal Request Flow (30 seconds)

Actions:

1. In Postman, send GET request to `/api/events`
2. Show response headers

Expected Headers:

```
HTTP/1.1 200 OK
RateLimit-Limit: 1000
RateLimit-Remaining: 999
RateLimit-Reset: 1640000900
```

Script:

> "Every API response includes rate limit headers:
>
> - `RateLimit-Limit: 1000` - Maximum 1000 requests per 15-minute window
> - `RateLimit-Remaining: 999` - I have 999 requests left
> - `RateLimit-Reset: ...` - When the counter resets
>
> This is generous for normal usage but protects against abuse."

Step 2: Simulate Rapid Requests (1 minute)

Actions:

1. In Postman, create a collection runner or use a tool to send 20 rapid requests
2. Show decreasing `RateLimit-Remaining` value

3. Show all requests still succeeding (under limit)

Script:

> "Let me send multiple requests rapidly. Notice the `RateLimit-Remaining` counter decreases with each request: 999, 998, 997...
>
> All requests are still allowed because we're under the 1000 limit. But if an attacker tried to send 1001 requests..."

Step 3: Rate Limit Exceeded (Optional - if time permits)

Script:

> "If the limit is exceeded, the server would respond with:
>
> ```
> HTTP/1.1 429 Too Many Requests
> Retry-After: 900
>
> {
>   "success": false,
>   "message": "Too many requests from this IP, please try again later.",
>   "retryAfter": 900
> }
> ```
>
> The client is told to wait 15 minutes before trying again."

[Show mock response or code snippet]

PART 7: Security Headers (Helmet) (2 minutes)

Script:

> "EventConnect uses Helmet middleware to set secure HTTP headers that protect against common web vulnerabilities."

Demo Steps:

Step 1: Inspect Security Headers (1 minute 30 seconds)

Actions:

1. In browser, navigate to EventConnect
2. Open DevTools → Network tab
3. Click on any API request
4. Go to Headers tab → Response Headers

Expected Headers:

```
content-security-policy: default-src 'self'; script-src 'self' 'unsafe-inline'
'unsafe-eval' https://cdn.jsdelivr.net; style-src 'self' 'unsafe-inline'
https://fonts.googleapis.com; ...
x-dns-prefetch-control: off
x-frame-options: SAMEORIGIN
x-content-type-options: nosniff
x-xss-protection: 0
strict-transport-security: max-age=63072000; includeSubDomains
access-control-allow-origin: https://event-connect.site
access-control-allow-credentials: true
```

Script:

> "Let me show you the security headers set by our Helmet middleware.
>
> **[Highlight each header]**
>
> 1. **Content-Security-Policy**: Restricts where scripts, styles, and other resources can be loaded from. This prevents injection of malicious external scripts.
> 2. **X-Frame-Options: SAMEORIGIN**: Prevents our site from being embedded in iframes on other domains, protecting against clickjacking attacks.
> 3. **X-Content-Type-Options: nosniff**: Prevents browsers from MIME-sniffing responses, which could lead to security vulnerabilities.
> 4. **Strict-Transport-Security**: Forces browsers to always use HTTPS for our site, preventing downgrade attacks.
> 5. **Access-Control-Allow-Origin**: CORS header specifying which origins can access our API."

Step 2: Test Clickjacking Protection (30 seconds)

Actions:

1. Open browser console
2. Try to embed EventConnect in an iframe:

```
let iframe = document.createElement('iframe');
iframe.src = 'https://event-connect.site';
document.body.appendChild(iframe);
```

Expected Result: Blocked by X-Frame-Options

Script:

> "If I try to embed EventConnect in an iframe from a different domain, the X-Frame-Options header blocks it. This prevents clickjacking attacks where malicious sites overlay invisible frames to trick users

> into clicking."

[Show error in console]

PART 8: CORS Configuration (2 minutes)

Script:

> "CORS (Cross-Origin Resource Sharing) controls which domains can access our API. Let's test this."

Demo Steps:

Step 1: Allowed Origin (1 minute)

Actions:

1. In DevTools → Network tab
2. Make API request from EventConnect frontend
3. Show request headers:

   ```
   Origin: https://event-connect.site
   ```

4. Show response headers:

   ```
   Access-Control-Allow-Origin: https://event-connect.site
   Access-Control-Allow-Credentials: true
   ```

Script:

> "When the frontend (https://event-connect.site) makes a request to the backend, the Origin header shows where the request came from. The backend responds with `Access-Control-Allow-Origin` matching that origin, allowing the request."

Step 2: Blocked Origin (1 minute)

Actions:

1. Open a different website (e.g., example.com)
2. Open DevTools → Console
3. Try to make a request to EventConnect API:

   ```
   fetch('https://event-connect-jin2.onrender.com/api/events', {
     method: 'GET',
     credentials: 'include'
   })
   ```

```
  .then(res => res.json())
  .then(console.log)
  .catch(console.error);
```

Expected Result: CORS error

Script:

> "If I try to access the EventConnect API from a different website - let's say example.com - the browser blocks it with a CORS error:
>
> **[Show error]**
>
> `Access to fetch at 'https://event-connect-jin2.onrender.com/api/events' from origin 'https://example.com' has been blocked by CORS policy`
>
> This prevents malicious websites from stealing data from our API."

Note: Due to a bug in the current implementation (callback returns true for all origins), you might need to explain: "In our current implementation, there's a bug where unauthorized origins are still allowed. This should be fixed to actually block unauthorized access."

PART 9: Complete Security Workflow (2 minutes)

Script:

> "Let me demonstrate a complete user journey showing all security features working together."

Demo Steps:

Actions (rapid demonstration):

1. **Registration** → Password hashed ✓
2. **Login** → Session created, JWT issued ✓
3. **Session cookie** → HttpOnly, Secure ✓
4. **Create event** (as organizer) → RBAC allows ✓
5. **Input validation** → Coordinates checked ✓
6. **Rate limiting** → Headers show limits ✓
7. **Security headers** → All present ✓
8. **Logout** → Session destroyed ✓

Script:

> "Here's a complete flow:
>
> 1. User registers → password hashed with bcrypt
> 2. User logs in → session created with httpOnly cookie, JWT issued
> 3. User creates event → RBAC verifies organizer role
> 4. Event data validated → coordinates checked, fields required
> 5. Rate limiting active → 1000 requests per 15 minutes

> 6. Security headers sent → CSP, X-Frame-Options, etc.
> 7. User logs out → session destroyed
>
> All 8 security features work seamlessly together to create a defense-in-depth architecture."

[Show quick cuts of each step]

PART 10: Security Gaps & Future Improvements (1 minute)

Script:

> "While we've implemented comprehensive security measures, there are areas for improvement."

[Show slide with gaps]

Current Gaps:

```
🔴 CRITICAL (Fix Immediately):
1. CORS allows all origins (bug in callback logic)
2. localStorage for JWT tokens (XSS vulnerable)
3. sameSite: 'none' allows CSRF attacks
4. .env files in git repository

⚠ HIGH PRIORITY:
5. No account lockout after failed logins
6. Password policy only 6 chars (should be 12+)
7. JWT expires in 7 days (should be 15 minutes)
8. No CSRF token protection
```

Script:

> "We've identified critical security gaps:
>
> **Critical**: The CORS implementation has a bug that allows all origins instead of blocking unauthorized ones. JWT tokens in localStorage are vulnerable to XSS attacks. The sameSite cookie setting allows CSRF attacks.
>
> **High Priority**: There's no account lockout mechanism, password requirements are too weak, and JWT tokens have a 7-day expiration which is too long.
>
> These issues are documented in our security report and are planned for the next sprint."

PART 11: Conclusion (1 minute)

Script:

> **[Screen: Summary slide]**
>
> "To summarize, EventConnect has implemented **8 core security features**:

> ✅ Password Hashing with Bcrypt (12 rounds) ✅ Role-Based Access Control (Organizer/Participant) ✅ Input Validation (Express-validator) ✅ Rate Limiting (1000 requests per 15 minutes) ✅ Session Management (MongoDB, httpOnly cookies) ✅ Security Headers (Helmet middleware) ✅ JWT Authentication (7-day tokens) ✅ CORS Configuration (Origin allowlist)
>
> These features provide defense-in-depth protection against:
>
> - Brute-force attacks
> - SQL/NoSQL injection
> - Cross-Site Scripting (XSS)
> - Clickjacking
> - Session hijacking
> - Unauthorized access
> - API abuse
>
> While there are areas for improvement, the current implementation provides a solid security foundation for EventConnect.
>
> Thank you for watching this security demonstration. For detailed documentation, please refer to the SECURITY_IMPLEMENTATION_REPORT.md file."

[End screen with contact info or next steps]

Post-Production Checklist

Video Editing:

- [ ] Add intro slide (5 seconds)
- [ ] Add section title overlays
- [ ] Highlight important UI elements with circles/arrows
- [ ] Zoom in on critical text (headers, errors, database fields)
- [ ] Add captions/subtitles
- [ ] Add background music (low volume)
- [ ] Add outro slide with summary (10 seconds)

Visual Enhancements:

- [ ] Blur sensitive data (real emails, API keys)
- [ ] Use screen annotations to highlight features
- [ ] Speed up slow parts (MongoDB loading, etc.)
- [ ] Add "Security Feature" badges when demonstrating each feature

Quality Check:

- [ ] Audio is clear
- [ ] Screen is readable (1080p minimum)
- [ ] Transitions are smooth
- [ ] Total time under 20 minutes
- [ ] All 8 features demonstrated

- [ ] No sensitive information exposed

Alternative: Live Presentation Notes

If doing a **live demonstration** instead of recorded video:

Preparation:

1. Have all accounts pre-created
2. Keep MongoDB Compass open in background
3. Have Postman collections ready
4. Test all demos beforehand
5. Prepare backup slides in case of technical issues

Contingency Plans:

- **Internet fails**: Have screenshots/screen recording as backup
- **Database slow**: Have pre-loaded data
- **Live demo breaks**: Have video clips ready
- **Questions**: Prepare FAQ document

Time Management:

- Allocate 5 minutes Q&A at end
- Have "skip" markers for if running over time
- Core demos: Parts 2, 3, 4, 5 (must show)
- Optional demos: Parts 6, 7, 8 (if time permits)

Appendix: Common Questions & Answers

Q1: "Why is CORS allowing all origins?"

A: "Good catch! This is a known bug in our implementation. Line 95 in `server.js` has `callback(null, true)` even for blocked origins. This should be `callback(new Error('Not allowed by CORS'))` to actually block unauthorized access. We've documented this as a critical issue to fix in the next deployment."

Q2: "Is 7-day JWT expiration too long?"

A: "Yes, absolutely. Industry best practice is 15-minute access tokens with a separate refresh token. We've identified this as a high-priority security gap. The long expiration was initially chosen for development convenience but needs to be shortened for production."

Q3: "What about Multi-Factor Authentication?"

A: "MFA is not currently implemented but is planned for Phase 2. Our security plan originally proposed email-based OTP, but we prioritized the 8 core features for this implementation. MFA would significantly improve authentication security and is our top priority for future enhancements."

Q4: "How do you prevent brute-force login attacks?"

A: "Currently, we rely on the global rate limiter (1000 requests per 15 minutes). However, this is too generous for login attempts. We should implement account lockout after 5 failed attempts within a short period. This is documented as a high-priority gap in our security report."

Q5: "What happens if someone steals the JWT token?"

A: "If a JWT token is stolen, it remains valid until expiration (currently 7 days). We don't have a token blacklist mechanism, so even logging out doesn't invalidate the token. This is a known limitation. For production, we should:

1. Reduce token expiration to 15 minutes
2. Implement a token blacklist using Redis
3. Rotate tokens regularly
4. Consider using httpOnly cookies instead of localStorage"

Recording Equipment Recommendations

Software:

- **Screen Recording**: OBS Studio (free) or Camtasia (paid)
- **Video Editing**: DaVinci Resolve (free) or Adobe Premiere Pro (paid)
- **Annotations**: Camtasia or ScreenFlow

Settings:

- **Resolution**: 1920x1080 (1080p minimum)
- **Frame Rate**: 30 fps
- **Audio**: 44.1 kHz, mono
- **Format**: MP4 (H.264 codec)

Best Practices:

- Record in a quiet environment
- Use a good microphone (USB condenser mic recommended)
- Speak clearly and at moderate pace
- Pause between sections for easier editing
- Record extra takes of important parts

End of Demo Script

Total Estimated Time: 15-20 minutes (depending on pace and Q&A) **Difficulty:** Intermediate **Preparation Time:** 30-60 minutes **Required Knowledge:** EventConnect architecture, security concepts, API testing