Detailed explanations (Attack → Mitigation → Verification)

Each vulnerability section below has been expanded with additional context, illustrative scenarios, and extended verification steps so you can confidently document and demonstrate the mitigation in your report and demo video. The content intentionally explains concepts and verification methods without giving offensive instructions.

1) SQL Injection — `sql_mitigation.php`

What it is (high level):
SQL Injection (SQLi) happens when an application constructs SQL queries by concatenating strings that include untrusted user input. Because SQL is a programming language for databases, careless concatenation can let untrusted input change the intended query structure.

Real-world scenario (conceptual):
Imagine a login form that runs a query like `SELECT id FROM users WHERE username = 'USER_INPUT' AND password = 'PASS_INPUT'`. If the inputs are not handled as data but concatenated directly, an attacker could manipulate the input to alter the meaning of the query. In a lab, you will observe that malformed input can cause different database responses (e.g., error, more rows returned), which indicates the application is constructing SQL dynamically.

Typical impact:

- Data disclosure (reading tables the application shouldn't expose)
- Authentication bypass (logins validated incorrectly)
- Data manipulation (UPDATE/DELETE queries executed)
- In combination with other server weaknesses, possible remote code execution or escalation

Mitigation approach used in `sql_mitigation.php`:

1. **Prepared Statements / Parameterized Queries:**
    - All SQL statements that accept external input are prepared with parameter placeholders. The DB driver compiles the statement and the values are supplied separately so they cannot alter the SQL structure.
2. **Principle of Least Privilege:**
    - The DB user configured for the application has access only to required tables and operations (SELECT/INSERT/UPDATE limited as necessary).
3. **Fail-safe error handling and logging:**
    - Errors are logged server-side. End-users see generic error messages. Avoiding disclosure of database errors prevents attackers from learning table/column names.
4. **Input validation & normalization (complementary):**

    - Where appropriate, apply strict allowlists (e.g., numeric IDs, enum values) before passing to queries. This reduces the attack surface and improves business logic correctness.

Why this prevents SQLi (conceptual):
Parameterized statements decouple code (SQL syntax) from data (values). No matter what the user types, the database will treat it as a value bound to a parameter—not as SQL—so it cannot alter control flow or inject extra statements.

Extended verification steps (safe):

- **Functional verification:** use valid inputs and confirm the application returns expected records. This ensures the prepared statement logic didn't break legitimate functionality.
- **Behavioral verification:** submit malformed or unexpected input (for testing only in the isolated lab); verify the application does not produce SQL error traces or unexpected query results. Instead, it should return consistent, safe responses (e.g., "no results" or a sanitized error page).
- **Logging verification:** check server-side logs for recorded attempts. Ensure detailed SQL errors are not presented to the user but are properly logged for developers.
- **Privilege verification:** attempt (in-lab) actions that should be prevented by least-privilege (e.g., schema modifications) to confirm the DB account lacks those privileges.

Notes for documentation:

- In the report, include a short code excerpt showing the use of prepared statement placeholders (do not include raw user inputs or offensive payloads). Annotate the snippet explaining where input enters the code and where binding occurs.
- Include a screenshot of a benign query result before/after applying the mitigation and a log snippet showing safe error handling.

---

2) Stored Cross-Site Scripting (Stored XSS) — `xss_stored_mitigation.php`

What it is (high level):
Stored XSS occurs when user-supplied content is saved by the server (in a database or file) and later included in web pages without sufficient encoding or sanitization. Because the payload is persistent, it can affect many users who view the page.

Illustrative scenario (conceptual):
A typical stored XSS case is a blog comment system where user comments are saved and then displayed on post pages. If the saved content isn't sanitized or encoded, a malicious visitor could store markup that executes in later readers' browsers.

Mitigation strategy in `xss_stored_mitigation.php`:

1. **Output Encoding:**

    - When rendering stored text into HTML, always encode special characters for the proper context. For instance, use HTML entity encoding for text nodes so characters like < and > render as literal symbols.
2. **Sanitization with a whitelist (when HTML is allowed):**
    - If the application must allow limited HTML (e.g., formatting), use a vetted sanitizer library that enforces a strict whitelist of allowed tags and attributes and removes scripts and event handlers.
3. **Secure cookie flags and session handling:**
    - Ensure cookies use `HttpOnly`, `Secure`, and `SameSite` to make it harder for scripts to exfiltrate session cookies.
4. **Principle of least privilege for stored content display:**
    - Avoid rendering stored content in sensitive contexts (e.g., inside inline `<script>` blocks). If you must, use additional encoding appropriate to that context.

Why this prevents stored XSS (conceptual):
Output encoding changes the representation of data in the HTML so browsers do not treat it as executable markup. Sanitization removes or neutralizes potentially harmful constructs while allowing safe formatting.

Extended verification steps (safe):

- **Rendering verification:** store benign content that contains characters commonly used in markup and confirm the page shows them as text, not interpreted elements.
- **Contextual verification:** verify encoding in different contexts (HTML text, attribute values, and when content is used inside JavaScript). Ensure the output encoding matches the rendering context.
- **Cookie verification:** check in browser devtools that session cookies are set with `HttpOnly` and `Secure` (if using HTTPS in the lab).
- **Sanitizer verification (if applicable):** demonstrate that allowed tags remain but disallowed tags/attributes are removed by the sanitizer.

Notes for documentation:

- Include screenshots of the stored content before and after the fix (sanitized and encoded). Provide a short explanation of where in the rendering pipeline encoding occurs.
- Document the specific HTTP cookie flags applied to sessions and why they matter.

Reflected Cross-Site Scripting (Reflected XSS)

What it is (high level)
Reflected XSS occurs when an application takes user-controlled input from a request (for example, query parameters, form fields, or headers) and includes it in the HTTP response without the right encoding or validation. Because the malicious content is echoed directly in

the response and not stored, an attacker typically convinces a victim to visit a crafted URL or submit a manipulated form so the injected content executes in the victim's browser.

Why it's dangerous

- Attack runs in the context of the victim's browser and origin, so it can perform actions that the user is authorized to perform (subject to same-origin rules).
- It can be used for phishing, UI manipulation, or session abuse (e.g., making the victim's browser execute scripts that contact the site).
- Although usually narrower than stored XSS, reflected XSS is highly practical for targeted attacks where the attacker can lure a user (e.g., via email or social engineering).

Typical indicators in an application

- Search results page or error page reflects the query string or input back into HTML.
- Application echoes form input in page headings, messages, or debug output without apparent escaping.
- Server responses contain user-supplied text in responses or in JavaScript context.

Mitigation (conceptual)

1. **Contextual output encoding:** Always encode user-supplied data according to the context where it appears:
    - HTML text nodes → HTML entity encoding (e.g., replace < with `<`).
    - HTML attributes → attribute encoding.
    - JavaScript strings → JavaScript string encoding or avoid placing untrusted data in inline scripts.
      Encoding must be done as late as possible (right before rendering) and should match the exact context of insertion.
2. **Input validation & normalization:** When possible, validate inputs against strict allowlists (for example, numeric IDs, fixed tokens, or short enumerations). For free text, normalize and remove control characters before rendering.
3. **Content Security Policy (CSP):** Deploy a restrictive CSP that forbids inline scripts (`'unsafe-inline'`) and restricts script loading to trusted origins. CSP is a powerful layer that limits the impact of any script that might make it into a page.
4. **Avoid reflecting data when unnecessary:** Prefer server-side state or lookups rather than reflecting arbitrary user input in pages.
5. **Defense in depth:** Combine encoding + CSP + strict headers (e.g., `X-Content-Type-Options`, `X-Frame-Options`) and secure cookie flags.

How to verify the mitigation (safe steps)

- **Rendering test:** Submit benign test input that contains characters used in markup (like < and >), and confirm the application renders them as text rather than HTML.
- **Context checks:** Inspect the rendered HTML in browser devtools; confirm special characters are represented as entities in the exact context (text node, attribute, etc.).
- **Header checks:** In network tools, verify `Content-Security-Policy` header is present and correctly configured. Explain which directives mitigate inline script execution.

- **Behavioral validation:** Show that clicking a crafted link does not result in script execution and that the application still behaves normally for legitimate users.

Documentation notes for report

- Include a short, non-actionable code excerpt showing where output encoding is applied (e.g., mention “HTML entities used when echoing user input” — avoid showing exact encoding functions if not needed).
- Attach a screenshot of the response headers (CSP present) and the rendered HTML showing encoded content.
- Describe remediation steps in plain language: “Use contextual encoding when rendering untrusted input; deploy CSP to block inline script execution.”

Short demo narration lines (for video)

- “This search page previously reflected the query directly. Now we apply contextual encoding and a CSP header. The query appears as literal text and CSP prevents inline scripts — see the response headers here.”

Stored Cross-Site Scripting (Stored XSS)

What it is (high level)
Stored XSS happens when an application accepts user-supplied content (comments, profile fields, forum posts) and persists it on the server, then later renders it into pages without appropriate sanitization or encoding. Because the malicious payload is stored, it can affect many users who view the stored content.

Why it's dangerous

- Persistent effect: once stored, the payload can run in every user’s browser who views the content.
- Potential for wide impact: many users or administrators can be affected, increasing potential harm.
- Can lead to session theft, cross-site request forgery combined attacks, UI manipulation, or planting of phishing content inside the application.

Typical indicators in an application

- User-generated content is saved (database, files) and later rendered in pages unchanged.
- Comments, profile descriptions, or message boards show raw user HTML or script-like text.
- WYSIWYG editors allow HTML but lack server-side sanitization.

Mitigation (conceptual)

1. **Output encoding:** Encode stored content at render time for the context in which it appears (HTML text nodes, attributes, etc.). Encoding ensures stored data is displayed as text, not executed as code.
2. **Sanitization with a whitelist when HTML is required:** If the application must allow formatting, use a well-maintained sanitizer library that enforces a strict allowed list of tags and attributes and removes scripting constructs and event attributes. Avoid fragile regex-based filters.
3. **Cookie & session protections:** Mark session cookies `HttpOnly` and `Secure` (if using HTTPS) to prevent scripts from reading cookies directly. Use `SameSite` to limit cross-site usage.
4. **Avoid dangerous contexts:** Never place user-supplied storage data directly into inline scripts, inline event handlers, or sensitive JSON endpoints unless properly encoded for that specific context.
5. **Least privilege & content policies:** Limit who can post content, moderate posts, or use rate-limits/filters to reduce spam and abuse vectors.

How to verify the mitigation (safe steps)

- **Store benign test content:** Save text containing characters normally interpreted as markup and verify the display shows them as text (encoded).
- **Context verification:** Check rendering in multiple contexts — within body text, inside attributes (e.g., `alt`, `title`), and ensure appropriate encoding is applied per context.
- **Sanitizer verification (if allowing some HTML):** Show stored content that uses permitted tags remains while disallowed tags/attributes are removed.
- **Cookie checks:** Verify `HttpOnly` and `Secure` flags using browser devtools for session cookies. Also show `SameSite` attribute if configured.
- **Logging and moderation:** Show that attempt logs capture submission metadata while the UI shows sanitized content.

Documentation notes for report

- Explain that the fix uses output encoding as the primary control and a whitelist-based sanitizer where limited HTML is permitted.
- Include screenshots showing stored content rendered safely, and highlight the cookie flags in devtools.
- Recommend procedural controls (moderation, rate-limiting) in addition to technical controls.

Short demo narration lines (for video)

- “Previously, stored user content was rendered without encoding, risking persistent XSS. After applying output encoding and a sanitizer for allowed markup, stored entries are safe text and cookie protections prevent scripts from reading session cookies.”

Cross-Site Request Forgery (CSRF)

What it is (high level)
CSRF is an attack that tricks a logged-in user's browser into making an unwanted request to a web application where the user is authenticated. Because browsers automatically include cookies and other credentials with same-origin requests, a cross-site forged request can perform state-changing actions (like changing an account password or submitting a transaction) without the user's consent.

Why it's dangerous

- Attack leverages the user's authenticated session to make requests that the user did not intend.
- Can lead to unauthorized account changes, transactions, or privilege escalation when combined with other flaws.
- Hard to detect from the server side unless CSRF protections are in place.

Typical indicators in an application

- Sensitive state-changing operations (password change, email update, profile edit, money transfer) accept POST/GET requests without any per-request validation token.
- Forms lack hidden tokens and server-side checks for request provenance.
- The application relies solely on cookies and referrer checks without cryptographic tokens.

Mitigation (conceptual)

1. **Synchronizer Token Pattern (recommended):** For each user session or per form, generate a cryptographically random token on the server and embed it in the HTML form as a hidden input. On form submission, the server verifies that the token matches the expected value for that user/session. An attacker cannot generate a valid token for a victim's session, so forged cross-site requests fail.
2. **SameSite cookies:** Set session cookies with `SameSite` attribute (e.g., `Lax` or `Strict`) to limit cookie sending on cross-site requests. This reduces exposure but should be used in combination with tokens rather than as a sole defense.
3. **Double-submit cookie (alternative):** Server issues a cookie with a random value and expects the same value in a request parameter. Server-side verifies both match. This is a less robust alternative when synchronizer tokens are not feasible.
4. **Origin/Referer header checks (layered):** Check `Origin` or `Referer` headers for sensitive actions. These headers are useful but may be absent in some legitimate clients; they should be a secondary control, not the only one.
5. **Avoid side-effectful GETs:** Ensure that operations that change state use POST (or other non-GET verbs) and require CSRF tokens.

How to verify the mitigation (safe steps)

- **Form token presence:** Show a sensitive form in the browser that includes a hidden token field. Point out that the token is tied to the session.
- **Server-side verification:** Demonstrate (conceptually) that server rejects submissions lacking a valid token. In the lab, attempt to submit a form without the token and show rejection.

- **Cookie checks:** Confirm `SameSite` attribute on cookies in devtools (explain what `Lax`/`Strict` mean for cross-site requests).
- **Layered checks:** If using origin/referer checks, show response behavior when header is missing or mismatched (rejected). Emphasize fallbacks and user experience implications.

Cross-cutting verification checklist (detailed)

For completeness in your report and demo, include a verification checklist that you tick off for each mitigation. This checklist demonstrates that you tested both functionality and security properties.

1. **Functional tests:**
    - Legitimate user actions (login, form submit, search, profile update) still succeed with valid inputs.
2. **Negative tests:**
    - Application rejects malformed or unexpected inputs gracefully (no stack traces or debug output to users).
3. **Security header checks:**
    - CSP present and configured; `X-Frame-Options`, `X-Content-Type-Options`, `Referrer-Policy`, and `Strict-Transport-Security` configured as appropriate for the lab.
4. **Cookie & session checks:**
    - Session cookies set with `HttpOnly`, `Secure` (if HTTPS), and `SameSite` flags.
5. **Logging & monitoring:**
    - Suspicious inputs are logged server-side. Errors are logged in a way that is useful for incident response.
6. **Access control & privilege verification:**
    - DB accounts limited; file inclusion paths restricted by whitelist; no direct inclusion of user-controlled file paths.

In the report, present this checklist as a table with