Generic Rubric

Scale for all facets:

- **3 (Complete & precise):** Correct, specific, unambiguous; includes all critical elements.
- **2 (Substantially correct):** Core idea is correct; minor gaps, vague terms, or omissions; no material errors.
- **1 (Attempt with misconceptions):** Mentions some relevant aspect but confused, incomplete, or contains errors.
- **0 (Incorrect/irrelevant):** Wrong or irrelevant answer.

Facet 1. Locate (Identify problem code lines)

- **3**: Lists *all essential code locations* that produce the vulnerability (e.g., input point and sink, free + later dereference, validation omission).
- **2**: Identifies the correct function/module and at least one essential site, but not all, or includes non-essential lines.
- **1**: Points only to a broad area (e.g., “input handling,” “database query”) without the critical lines.
- **0**: Wrong location(s).

Note: For this facet, line numbers only are required.

Facet 2. Cause (Static mechanism – Why the vuln exists)

- **3**: Correctly states the root flaw in code/design (e.g., unsanitized input concatenated into SQL, pointer freed but not invalidated, missing boundary check, absent CSRF token).
- **2**: Generally correct but vague or incomplete; identifies the flaw but not the condition that makes it unsafe.
- **1**: Partially relevant but confused; mixes vulnerability types or misuses terminology.
- **0**: Incorrect mechanism.

Facet 3. Behavior (Runtime manifestation – How it plays out)

- **3**: Clear execution story: how input/control flows through the code, how the vuln is triggered, and what happens (e.g., injected query executes, payload renders, memory reused).
- **2**: Mentions plausible runtime outcome (e.g., “crash,” “code execution,” “data leak”) but doesn’t link steps.
- **1**: Vague or incorrect runtime story; some relevant terms but confused.
- **0**: Irrelevant/incorrect description.

Facet 4. Consequence/Impact (Security outcome – So what?)

- **3**: Identifies the primary security impact with specificity (e.g., privilege escalation to admin, exfiltration of PII, execution of arbitrary code, denial of service).
- **2**: Correct impact class but generic or missing scope qualifiers (e.g., just says “security risk” or “data leak” without details).
- **1**: Mostly wrong impact but contains a minor relevant element.
- **0**: Wrong/irrelevant.

Facet 5. Prevention/Mitigation (How to fix/avoid)

- **3**: Proposes effective, specific mitigation for the vuln type (e.g., parameterized queries, output encoding, RAII/smart pointers, bounds checking, CSRF tokens).
- **2**: Generally right but incomplete or generic (e.g., “validate inputs,” “use secure coding practices”); OR suggests only secondary mitigations (firewalls, scanners) without addressing root fix.
- **1**: Largely ineffective or irrelevant suggestion, but security-flavored.
- **0**: Wrong/irrelevant.

Detailed rubric for all vulnerabilities

UAF rubric -1

- **Deallocation (free of a that leads to UAF): L33** `free(a);`
- **Earliest use after free (deref of a): L35** `a = 3;`
- **Additional uses after free:**
  - **L36** → `negate(a);` (deref happens at **L16** `a = -(*a);`)
  - **L37** → `printf("Result: %d\\n", *a);`
- **Not part of the vuln:** the frees inside the OOM branch **L24–L25** (function returns at **L27**).

  If your editor's numbering differs slightly, apply a ±1–2 line tolerance. The "first use after free" is the assignment to *a immediately following free(a).

Facet 1 — Locate (line numbers only)

What to list: the deallocation site that participates in the bug **and** the **earliest** subsequent dereference/use.

- **3 (complete & precise):** Lists **L33** and **L35** (earliest use).
  Accept also: `33, 35–37` or `33, 35, 16` (crediting that L36 calls a deref at L16).
- **2 (substantially correct):** Lists **L33** and **a later** post-free use (e.g., `33, 36` or `33, 37`) **OR** lists the correct function/block and only one essential site (`33` **or** `35`).
- 1 (attempt with notable errors): Points to broadly related but non-essential lines (e.g., `24, 25` (free)() or a function name) without 33/35.
- **0 (incorrect):** Line numbers are unrelated.

  Rater tip: If both correct and incorrect numbers appear, score by the best matching anchor (e.g., 24, 33, 36 → 2, because it includes 33 + a post-free use but not the earliest).

Facet 2 — Cause (static mechanism—why)

Either code-specific or conceptual wording earns full credit.

- **3:** Precisely states that memory for `a` is **freed** and a **dangling pointer** remains that is **dereferenced later** without reallocation or validity reset (e.g., "`a` is freed at L33; `a` is used afterward").
- **2:** Essentially right ("use-after-free / dangling pointer") but vague or missing a key precondition (ownership/alias) — no contradictions.
- **1:** Partly relevant but confused (mixes UAF with null-deref, buffer overflow, or double-free).
- **0:** Wrong/irrelevant mechanism.

Code-grounded exemplar (score 3):

"`a` is freed (L33) but still points to freed memory; it's dereferenced at L35/L36."

Conceptual exemplar (score 3):

"A dangling pointer is dereferenced after its storage is returned to the allocator."

Facet 3 — Behavior (runtime manifestation—how)

- **3:** Coherent execution story: after `free`, the allocator may reuse the chunk; later dereference **reads/writes freed memory**, causing **undefined behavior** (crash, corruption) or enabling exploitation via heap grooming/reallocation.
- **2:** Names plausible outcomes (crash/UB/data corruption) but doesn't connect steps (no mention of reuse/flow).
- **1:** Vague or incorrect flow (e.g., calls it "out-of-bounds index").
- **0:** Irrelevant runtime story.

Code-grounded exemplar (3):

"After L33, `*a = 3` at L35 writes into memory that may already be reused, leading to UB; `negate(a)` at L36 dereferences the same dangling pointer."

Conceptual exemplar (3):

"Freed heap memory can be reallocated to attacker-controlled data; dereferencing the dangling pointer can corrupt or read attacker-supplied content."

Facet 4 — Consequence / Impact

- **3:** Specific and correct: memory corruption enabling **RCE/priv-esc/info leak**, or at least **DoS**; explains when severity escalates (attacker controls lifetime/heap layout).
- **2:** Right class but generic ("may crash or leak data") without scope/severity context.
- **1:** Mostly wrong (e.g., claims "SQL injection").
- **0:** Irrelevant.

Exemplar (3):

"Corruption of function pointers/vtables via reuse can lead to arbitrary code execution; at minimum, a reliable crash (DoS)."

Facet 5 — Prevention / Mitigation

(Primary fixes earn full credit; detection alone is secondary.)

- **3:** Effective, specific prevention for UAF: enforce ownership/lifetime discipline (**C++ RAII** with `std::unique_ptr`/`shared_ptr` or C discipline), **don't use after free** (re-order logic so free happens last), **set pointer to NULL immediately after free and guard derefs**, re-allocate before reuse; mention of **ASan/MSan**, static analysis, and fuzzing as detection (not the primary fix).
- **2:** Generally right but incomplete/generic (e.g., "do memory management better," "sanitize inputs"), or proposes only secondary mitigations without the primary fix.
- **1:** Largely ineffective/irrelevant (e.g., "encrypt the data").
- **0:** Wrong/irrelevant.

Code-grounded exemplar (3):

"Move `free(a);` after the last use, or re-allocate `a` before L35; alternatively `free(a); a = NULL;` and guard all derefs; prefer RAII/smart pointers in C++."

Conceptual exemplar (3):

"Prevent dangling pointers via strict ownership discipline and lifetime-safe patterns; detect with ASan in testing."

UAF rubric -2

- **Deallocation (creates dangling pointers):**
  - **L52** `free(password);`
  - **L53** `free(username);`
- **Earliest subsequent use after free:**
  - **L15–16** → `printf(..., username)` dereferences the freed pointer when the loop iterates again.
- **Other post-free uses:**
  - `strcmp(username, "root")` at **L68**
  - `strcmp(temp_uname, username)` and `strcmp(temp_pwd, password)` at **L81**
  - `printf(..., password)` at **L19**
- **Why:** Freed pointers are not set to `NULL`, so they remain non-null and are dereferenced later → classic **use-after-free**.

Facet 1 — Locate (line numbers only)

- **3 (complete & precise):** Lists **52–53** (frees) and **16** (earliest deref). Acceptable: `52–53, 15–16` or `16, 52`.
- **2 (substantially correct):** Lists at least one free + a later deref (e.g., `52, 81`), but not the earliest one at L16; OR lists only the deref.
- **1 (attempt with misconceptions):** Gives a broad area (e.g., “case 3” or `47–54`) but misses free+first use, or lists unrelated lines.
- **0 (incorrect):** Wrong/unrelated lines.

Rater tip: If both correct and incorrect lines are listed, grade based on best anchor match.

Facet 2 — Cause (static mechanism: why)

- **3:** Precisely states that memory for `username`/`password` is freed (L52–53), pointers not reset to NULL, and later dereferenced (L16, L68, L81).
- **2:** Correct but vague (“pointer freed but still used” without details).
- **1:** Partially relevant but confused (mixes with null deref, double free, or buffer overflow).
- **0:** Wrong/irrelevant (e.g., “SQL injection”).

Exemplar (3):

“`username` is freed at L53 but not nullified. On the next loop, L16 prints it as if valid, causing a dangling pointer dereference.”

Facet 3 — Behavior (runtime manifestation: how)

- **3:** Clear story: freed pointers remain non-null, loop dereferences them (`printf`/`strcmp`), so execution touches freed memory → undefined behavior: crash, memory corruption, or exploitable if attacker reallocates heap.
- **2:** Mentions plausible runtime effect (crash, UB, corruption) but doesn’t link steps.
- **1:** Vague or wrong runtime story (e.g., calls it “buffer overflow”).
- **0:** Irrelevant.

Exemplar (3):

“After free, the program still prints or compares the dangling pointer; since the heap chunk may be reused, dereferencing can lead to crashes or controlled corruption.”

Facet 4 — Consequence / Impact

- **3:** Specific impact: memory corruption exploitable for **arbitrary code execution**, privilege escalation, or at least **DoS**. Mentions attacker control over heap layout.
- **2:** Correct but generic (“may crash or leak data”).
- **1:** Mostly wrong, slight relevance.
- **0:** Irrelevant.

Exemplar (3):

“An attacker can exploit this dangling pointer to overwrite sensitive data or hijack control flow, leading to arbitrary code execution; at minimum it causes a crash.”

Facet 5 — Prevention / Mitigation

- **3:** Specific fixes:
  - Reset pointers to NULL after free (`username = NULL; password = NULL;`).
  - Avoid reuse of freed pointers.

    - Enforce safe lifetime management (RAII, smart pointers).
    - Reorder logic so free happens after last use.
    - Use sanitizers (ASan/MSan) for detection.
- **2:** General but incomplete (“manage memory better,” “secure coding”).
- **1:** Ineffective/irrelevant suggestion with security flavor.
- **0:** Wrong/irrelevant.

Exemplar (3):

“After freeing, set pointers to NULL and check before dereference. Better: redesign with RAII/smart pointers to enforce safe lifetimes; use ASan in testing to catch UAFs.”

OOB write-1

- **Vulnerable declaration & use:**
    - `char str2[20];` at **L32** defines a fixed-size stack buffer.
    - **L36** `scanf("%s", str2);` reads arbitrary-length input without a size specifier, allowing data beyond 19 characters (+ `\\0`) to be written.
- **Root flaw:** Unbounded input into a fixed-size array → **stack-based buffer overflow / out-of-bounds write**.
- **Manifestation:** If user enters ≥20 chars, writes overflow `str2` and corrupt adjacent stack memory.
- **Consequences:** Undefined behavior; could crash, corrupt data, or allow attacker to overwrite saved frame pointers/return address for code execution.
- **Fix:** Limit input size (`scanf("%19s", str2)`, `fgets`), validate length, or adopt safer functions.

Facet 1 — Locate (line numbers only)

What graders look for: The exact vulnerable code line(s) where the OOB write occurs.

- **3 (complete & precise):** Lists **L36** (`scanf("%s", str2);`) as the overflow site. May also mention the declaration at **L32** in combination with L36 (that’s fine).

- **2 (substantially correct):** Identifies the right block/function and mentions input into `str2` (lines 35–37 or “scanf in main”), but doesn’t isolate L36 precisely OR includes non-essential lines.
- **1 (attempt with misconceptions):** Mentions only a broad area (e.g., “user input” or “reading strings in main”) but doesn’t identify the actual overflow line.
- **0 (incorrect/irrelevant):** Lists unrelated lines (e.g., lines inside `levenshteinDistance`) or no location at all.

Exemplar answer (score 3):

“Line 36, `scanf("%s", str2)`, writes into a fixed 20-byte array with no bounds check.”

Facet 2 — Cause (static mechanism — why the vuln exists)

What graders look for: The root coding/design condition that allows the overflow.

- **3:** Clearly states that `str2[20]` has a fixed size, and `scanf("%s", str2)` does not limit input size, so longer user input will overflow the buffer. Explicit mention of missing bounds check is expected.
- **2:** Essentially correct but vague: mentions “unsafe scanf” or “no input validation” without tying it to the fixed buffer size.
- **1:** Partial relevance but confused: talks about strings or input but calls it “null pointer” or “use-after-free.”
- **0:** Wrong/irrelevant mechanism (e.g., “SQL injection” or “encryption problem”).

Exemplar answer (score 3):

“The buffer `str2` is only 20 bytes, but `scanf("%s")` allows any length input, so input longer than 19 chars + null will write past the array boundary.”

Facet 3 — Behavior (runtime manifestation — how it plays out)

What graders look for: How the vulnerability unfolds when exploited.

- **3:** Provides a coherent runtime story: user enters ≥20 characters → `scanf` keeps writing past `str2` → adjacent stack memory is corrupted → undefined behavior (crash, corruption, or possible control hijack).
- **2:** Mentions a plausible runtime outcome (crash, memory corruption) but doesn't connect steps from input → overflow → effect.
- **1:** Mentions a runtime effect but wrong or highly vague (e.g., "it won't work," "string mismatch").
- **0:** Irrelevant runtime story.

Exemplar answer (score 3):

"If the user types a 30-character string, the extra characters overwrite memory next to `str2` on the stack, potentially clobbering variables or return addresses. This causes crashes or, if controlled, arbitrary execution."

Facet 4 — Consequence / Impact (security outcome)

What graders look for: The real-world risk and what property is violated.

- **3:** Specific, correct impact: stack corruption can be exploited to achieve arbitrary code execution (e.g., overwriting saved return address), or at minimum denial-of-service through a crash.
- **2:** Correct impact class but generic (e.g., "it can cause a crash," "it's a security risk") without explaining stack/exec implications.
- **1:** Mostly wrong consequence but has a minor relevant element (e.g., "data leak" only).
- **0:** Irrelevant impact.

Exemplar answer (score 3):

"An attacker could overwrite the return address on the stack and hijack execution flow, leading to arbitrary code execution; at minimum, the program may crash."

Facet 5 — Prevention / Mitigation (how to fix/avoid)

What graders look for: Effective, specific mitigations that prevent the root cause.

- **3:** Proposes specific, effective fixes:
    - Use bounded format specifiers (`scanf("%19s", str2)`).
    - Or use safe functions (`fgets(str2, sizeof str2, stdin)`).
    - Add explicit length checks before copying/processing.

    - Adopt safer APIs like `snprintf` or use string libraries.
- **2:** General but incomplete: “sanitize inputs,” “be careful with buffer size” without specifics; or mentions secondary mitigations only (stack canaries, ASLR) without addressing the coding flaw.
- **1:** Ineffective or irrelevant suggestion with security flavor (e.g., “encrypt the data”).
- **0:** Wrong/irrelevant.

Exemplar answer (score 3):

“Replace `scanf("%s", str2)` with `scanf("%19s", str2)` or `fgets(str2, sizeof(str2), stdin)` to prevent out-of-bounds writes.”

OOB write-2

- **Core bug (sink): L92** `password[length] = '\\0';` writes one past the end **when `length == MAX_LENGTH` (16)**, because `password` has indices `0..15`.
- **Contributing sites (context):**
    - **L29** `char password[MAX_LENGTH];` allocates exactly 16 bytes.
    - **L34–40** validation **permits** `length == MAX_LENGTH`.
    - The loop at **L56–91** correctly writes `password[0..length-1]`; it’s **not** the bug.
- **Secondary effects:** The overrun can also lead to **subsequent OOB reads** when printing with `%s` (L43) if no in-bounds `'\\0'` exists.

Facet 1 — Locate (identify problem lines; line numbers only)

- **3 (complete & precise):** Lists **L92** (the off-by-one write). It’s also fine (and encouraged) to include the **essential context**: **L29** (buffer size) and **L34–40** (range check allowing `length == MAX_LENGTH`).
    - Acceptable 3-point answers: `92` **or** `29, 34–40, 92`.
- **2 (substantially correct):** Points to the right function/block and a relevant site but misses the exact sink (e.g., `29, 56–91` or “generatePassword loop”), **or** lists only `29` without `92`.
- **1 (attempt with misconceptions):** Broad area only (e.g., “main input” or `31–44`) without the sink; or lists unrelated helper lines.
- **0 (incorrect/irrelevant):** Wrong locations.

Rater tip: Use ±1–2 line tolerance if a student’s editor shifts numbering.

Facet 2 — Cause (static mechanism — why)

- **3:** Explains the **off-by-one**: `password` has size `MAX_LENGTH` (16) → valid indices `0..15`; code writes a NUL at `password[length]`; when `length==16`, this is **one past the end**, i.e., **stack out-of-bounds write**.
- **2:** Essentially right but vague (e.g., “bad bounds check”/“buffer too small”) without stating the off-by-one at `password[length]`.
- **1:** Partially relevant but confused (mixes with UAF/null deref or blames the loop that actually respects bounds).
- **0:** Wrong mechanism.

Exemplar (3):

“The code NUL-terminates at `password[length]`. With a 16-byte buffer, `length==16` writes to index 16 (OOB); that’s an off-by-one stack overflow.”

Facet 3 — Behavior (runtime manifestation — how it plays out)

- **3:** Clear execution story: for `length==16`, loop fills `password[0..15]`; then `password[16]='\\0'` (**L92**) writes past the stack buffer → corrupts adjacent stack memory (e.g., `saved EBP/return address` or nearby locals) → **undefined behavior** (crash, corruption), and may also induce an OOB **read** when `%s` scans for a terminator.
- **2:** Mentions plausible effects (crash/corruption) but doesn’t connect the off-by-one steps.
- **1:** Vague or incorrect runtime story (e.g., claims the random selection or the loop bounds cause it).
- **0:** Irrelevant.

Exemplar (3):

“When `length` equals 16, the terminator write goes to index 16, corrupting adjacent stack memory; printing the string later can also traverse into unintended memory.”

Facet 4 — Consequence / Impact (so what)

- **3:** Specific, correct impact: stack corruption exploitable for **arbitrary code execution** (e.g., overwriting a saved return address/canary) or at least **denial-of-service** via crash. May note risk of **information disclosure** via stray reads.
- **2:** Correct but generic ("may crash," "memory corruption") without scope/severity.
- **1:** Mostly wrong consequence with minor relevance.
- **0:** Irrelevant.

Exemplar (3):

"The one-byte overflow can corrupt control data on the stack, enabling control-flow hijack (arbitrary code execution); minimally, it causes a crash."

Facet 5 — Prevention / Mitigation (how to fix/avoid)

- **3:** Concrete, effective fixes that address the root cause:
    - Adjust termination logic: write `password[length-1] = '\\0'` **only if** there's space, or ensure the loop leaves room for the terminator (iterate `i < length-1`).
    - Or allocate **`MAX_LENGTH+1`** for the buffer (**L29**): `char password[MAX_LENGTH+1];`.
    - Keep the length contract consistent: either restrict input to `MAX_LENGTH-1`, or size the array for `MAX_LENGTH+1` including NUL.
    - Add assertions/tests; enable compiler & runtime hardening (ASan, stack canaries) as **secondary** detection, not the primary fix.
- **2:** Generally right but incomplete (e.g., "check bounds" without specifying the off-by-one, or only mentions sanitizers).
- **1:** Largely ineffective/irrelevant.
- **0:** Wrong/irrelevant.

Exemplar (3):

"Either declare `char password[MAX_LENGTH+1];` **or** ensure you leave one byte for `'\\0'` by filling at most `length-1`chars and then writing `password[length-1]='\\0'`."

SQLi-1

- **Vulnerable sink: L21** — `snprintf` builds an SQL string by interpolating user input (`id`) directly into the query: `"SELECT * FROM existential_crises WHERE id = '%s'"`.
- **Input source: L18** — `fgets(id, sizeof(id), stdin)` reads user-controlled input (note: it may include a trailing newline).
- **Why this is vulnerable:** Concatenating or formatting raw user input into SQL without using parameterized queries or escaping allows an attacker to inject SQL metacharacters (e.g., `'; DROP TABLE ...; --`) and change the query semantics — classic **SQL injection**.
- **Immediate consequences:** An attacker can read, modify, or delete database rows, escalate data exposure, or execute multiple statements depending on DB engine/config (e.g., retrieve confidential records, delete tables, escalate to data destruction).
- **Correct fix:** Use parameterized/prepared statements with bound parameters (e.g., `sqlite3_prepare_v2` on a SQL containing `?` placeholder then `sqlite3_bind_text`), or robust input validation and proper escaping (but prefer parameter binding). Also sanitize/remove newline from `fgets`.

Facet 1 — Locate (line numbers only)

What to list: the input source(s) and where the user data is injected into the SQL statement.

- **3 (complete & precise):** Lists **L18** (input via `fgets`) and **L21** (the `snprintf` that builds the SQL) — i.e., `18, 21`. Accept variants that include the prepare line **L23** for context.
- **2 (substantially correct):** Identifies the SQL-building site (L21) but omits the input source, or lists the input line only.
- **1 (attempt with misconceptions):** Points to related I/O code but not the actual concatenation into SQL (e.g., just `30–34` printing rows) or says “database code” without lines.
- **0 (incorrect):** Wrong or unrelated lines.

Model (3): `18, 21` — `fgets(id,...)` and `snprintf(... "WHERE id = '%s'", id)`.

Facet 2 — Cause (static mechanism — why)

What graders look for: the coding mistake that enables the injection.

- **3:** Precisely states that user input (`id`) is inserted directly into an SQL statement (L21) without parameter binding or proper escaping; constructing SQL via `snprintf` with raw

input allows the user to manipulate SQL syntax → SQL injection. Optionally mention `fgets` newline handling as a nuance.
- **2:** Correctly names SQL injection / unsafe concatenation but is vague about the mechanism (e.g., “bad formatting” without stating the missing parameterization).
- **1:** Partially relevant but confused (e.g., blames `fgets` buffer overflow or says “sqlite3_prepare_v2 is unsafe” without explaining concatenation).
- **0:** Incorrect (e.g., says it’s a buffer overflow or UAF).

Model (3):

`snprintf` inserts `id` directly into the SQL literal; because the code does not use placeholders and bind parameters, an attacker can inject SQL metacharacters to change the query.

Facet 3 — Behavior (runtime manifestation — how)

What graders look for: the execution-level effect when an attacker supplies crafted input.

- **3:** Explains: attacker supplies input containing quotes/SQL (e.g., `1' OR '1'='1` or `1'; DROP TABLE existential_crises; --`) → the constructed SQL’s syntax is altered; the DB will execute the modified query leading to unauthorized data retrieval, modification, or destruction. Also mention newline/trailing `\\n` from `fgets`and that proper trimming is missing.
- **2:** Names plausible outcomes (data leak, delete rows) but doesn’t fully connect attacker input → altered SQL string → DB executes it.
- **1:** Vague or wrong runtime story (e.g., “it will crash” only).
- **0:** Irrelevant.

Model (3):

“If user enters `1' OR '1'='1`, the SQL becomes `... WHERE id = '1' OR '1'='1'` which matches all rows — attacker can bypass intended filtering. If input includes `'; DROP TABLE ...; --`, the attacker can cause destructive operations (depending on DB settings).”

Facet 4 — Consequence / Impact (security outcome)

What graders look for: the real-world security risk and its severity.

- **3:** Specific impacts: unauthorized data disclosure (read arbitrary rows), authentication bypasses if used in login flows, data manipulation/deletion (DROP/UPDATE), and potential pivot to more serious compromises depending on DB privileges. States severity and context (e.g., data destruction, PII exposure).

- **2:** General correct impact ("may leak or modify data") without scope or examples.
- **1:** Mostly wrong but minor relevance.
- **0:** Irrelevant (e.g., "network issue").

Model (3):

"SQL injection can let an attacker read or modify sensitive data (PII), drop tables, or bypass access controls — consequences range from data leakage to full data loss and application compromise."

Facet 5 — Prevention / Mitigation (how to fix/avoid)

What graders look for: concrete, secure mitigations; ranked preference: parameterized queries > proper escaping > validation; mention trimming newline & limiting input size.

- **3:** Strong, specific mitigations: use prepared statements with placeholders and bind parameters (e.g., change L21 to `SELECT * FROM existential_crises WHERE id = ?` then `sqlite3_prepare_v2` + `sqlite3_bind_text` with `id` after trimming newline). Also recommend input validation (ensure `id` matches expected pattern), remove trailing newline from `fgets` (`id[strcspn(id, "\\n")] = 0;`), limit input length, least-privilege DB user, and logging/monitoring. Note that escaping is error-prone vs. parameter binding.
- **2:** General suggestions like "sanitize inputs" or "escape quotes" without giving the best-practice code-level remedy.
- **1:** Weak or irrelevant suggestions (e.g., "use encryption" or only mention client-side checks).
- **0:** Wrong/irrelevant.

Model (3):

"Use parameterized queries and binding: prepare `SELECT * FROM existential_crises WHERE id = ?`; call `sqlite3_bind_text(res, 1, id_trimmed, -1, SQLITE_TRANSIENT)` before `sqlite3_step`. Also trim newline after `fgets`, validate `id` format, and run DB with least privileges."

SQLi-2

1. **Primary sink**: **L50** — `snprintf(sql, sizeof(sql), "SELECT * FROM tokens WHERE id = %s", id);` — user-controlled `id` is inserted directly into SQL.
2. **Input source**: **L42** `fgets(id, sizeof(id), stdin)` reads user input (may include a trailing `\\n`).
3. **Intended validation**: **L5–11** `validate_id` attempts to ensure `id` contains only digits.
4. **Practical problems that create risk**:
    - `fgets` keeps the newline (`\\n`), so typical numeric input like `123\\n` contains a non-digit → `validate_id`will reject it (logic bug), causing the happy path to fail. That means the program may be brittle; attackers may craft input that bypasses or breaks validation flow.
    - If validation is incorrectly bypassed (or the input is pre-trimmed elsewhere), placing `id` directly into SQL **without quotes** (L50) is dangerous if `id` contains unexpected characters (e.g., `1 OR 1=1`) or whitespace; building SQL by concatenation or formatting is inherently prone to SQL injection unless the input is strictly sanitized and controlled.
    - The call builds SQL without quoting the id; if the database expects a string, missing quotes may produce syntactically different SQL. Either way, this pattern is fragile and not safe.
5. **Best fix**: Use parameterized queries / prepared statements (e.g., `mysql_stmt_prepare` + `mysql_stmt_bind_param`) or at minimum trim input and perform strict validation on a trimmed string; never build SQL by concatenation.

-

Facet 1 — Locate (line numbers only)

What to expect: input read site(s) and the SQL construction/execute site(s).

- **3:** Lists **L42** (input via `fgets`) and **L50** (the `snprintf` that places `id` into the SQL) and optionally **L52**(mysql_query) or **L5–11** (validate_id) for context. e.g., `42, 50` (best) or `42, 50, 52`.
- **2:** Identifies the SQL construction (L50) but omits input source (L42), or lists input only.
- **1:** Points to an I/O area or DB code in general (e.g., "mysql code in main") without lines.
- **0:** Wrong/unrelated lines.

Model (3): `42, 50` — `fgets` reads id and `snprintf` injects it into SQL.

Facet 2 — Cause (static mechanism — why)

What to expect: clear identification of the concatenation/formatting mistake and any validation logic problems.

- **3:** Explains that user input from L42 is directly inserted into the SQL string at L50 using `snprintf`, without parameter binding or proper sanitization; additionally notes the program's validation/trimming issues (fgets newline) that make the validation brittle. Mentions that constructing SQL using formatted strings allows malicious input to alter the query (SQLi).
- **2:** States "SQL injection due to building SQL from user input" but omits the nuance about newline/validation or the lack of quotes.
- **1:** Partial or confused — e.g., blames only `snprintf` overflow or calls it a buffer overflow rather than injection.
- **0:** Incorrect.

Model (3):

`snprintf` places `id` directly into the SQL without parameter binding or escaping. Because the program formats SQL with raw user data, an attacker can inject SQL operators (e.g., `OR`, `;`) to change query semantics. Also note `fgets` leaves a newline which breaks validate_id logic.

Facet 3 — Behavior (runtime manifestation — how)

What to expect: demonstration of the concrete runtime effect of crafted inputs.

- **3:** Connects attacker input → constructed SQL string → DB executes modified query. Gives examples (e.g., input like `1 OR 1=1`, or `0; DROP TABLE tokens; --`) and explains how that changes semantics (bypasses filter, deletes rows). Mentions the newline/trimming quirk and how missing trimming may cause valid IDs to be rejected. Notes also that id is inserted unquoted, so whitespace or operators may directly change SQL.
- **2:** States likely outcomes (data retrieval, modification, or deletion) but doesn't show concrete example or the transformation of the SQL string.
- **1:** Vague or wrong (e.g., "it will crash") without linkage.
- **0:** Irrelevant.

Model (3):

"If attacker supplies `123 OR 1=1`, the SQL becomes `SELECT * FROM tokens WHERE id = 123 OR 1=1` which returns all rows; if they supply `0; DROP TABLE tokens; --` and the server permits multiple statements, the DB will execute destructive commands. Because `fgets` adds `\\n` and `validate_id` expects digits only, the program may reject typical input — attackers may instead inject via pre-trimmed channels or by bypassing validation."

Facet 4 — Consequence / Impact

What to expect: specific security consequences and severity.

- **3:** Explicitly enumerates risks: unauthorized data disclosure (all tokens), authentication/authorization bypass (if used in auth flows), data modification/deletion, and potential full DB compromise or pivot. Notes severity depends on DB privileges and server configuration (e.g., whether multiple statements are allowed).
- **2:** Correct general risk ("may leak or modify data") without context.
- **1:** Slightly relevant but imprecise.
- **0:** Irrelevant.

Model (3):

"SQLi can lead to disclosure/modification/deletion of sensitive token data, permit attackers to bypass access controls, or perform destructive operations — consequences range from data leak to total data loss and application compromise depending on DB privileges."

Facet 5 — Prevention / Mitigation

What to expect: concrete, prioritized, code-level fixes.

- **3:** Recommends parameterized/prepared statements (the correct primary fix) over ad-hoc escaping. Specifically: use `mysql_stmt_prepare` with `?` placeholder and `mysql_stmt_bind_param` to bind an integer parameter if `id` is numeric. Trim newline from `fgets` before validation (`id[strcspn(id, "\\n")] = 0;`) and then validate the trimmed string. Also recommend limiting input length, using least-privilege DB accounts, applying logging/monitoring, and running DB with safe settings (disable multiple statements if not needed). If `id` must be numeric, parse with `strtol` and validate the numeric range rather than concatenating.
- **2:** Says "sanitize/validate input" or "escape quotes" but lacks the preferred binding approach.
- **1:** Weak or irrelevant fixes (e.g., "use encryption" or only client-side checks).
- **0:** Wrong/irrelevant.

Model (3):

“Trim the newline after `fgets`, then parse and validate the ID as an integer (e.g., `long idnum = strtol(id_trimmed, NULL, 10);` and check range), **and** use a prepared statement with `?` placeholder and bind `idnum` as an integer before executing. Don’t build SQL by concatenation. Also use least privilege for DB credentials and enable audit/logging.”

-

XSS-1

Primary untrusted source: `$_GET['email']` (user-controlled input) — accessed in PHP and echoed into the page at **L25**.

Primary sink(s):

- **L25 → L28:** The PHP-echoed value is placed inside a JavaScript single-quoted string literal (`email = '...';`) and later concatenated into `out.innerHTML` (L28). This is a **JS-context → DOM sink**.
- **L28:** `out.innerHTML = 'Welcome: <b>'+email+'</b>';` inserts `email` into the DOM as HTML (innerHTML) — any attacker-controlled content that reaches this assignment as HTML/JS can produce XSS.

Why this is vulnerable (short):

- `htmlspecialchars()` is used at L25, but by default it **does not escape single quotes** unless called with `ENT_QUOTES`. Embedding user input inside a *single-quoted JavaScript string literal* is dangerous if the input can contain an unescaped single quote (`'`) or a sequence that breaks out of the string and injects JS.
- Even if the PHP encoding prevents literal `<script>` tags, breaking out of the JavaScript string or using event/DOM injection can still lead to XSS because the value is later inserted into the DOM with `innerHTML`(which parses HTML).
- In short: **wrong escaping for the JS embedding context + unsafe DOM insertion** (innerHTML).

Additional notes:

- Using `innerHTML` with any untrusted content is risky — prefer `textContent`/`textNode` or safe serialization.
- `htmlspecialchars()` protects HTML contexts but embedding into JS requires a JS-safe encoding (e.g., `json_encode()`), or `htmlspecialchars(...,`

`ENT_QUOTES)` plus careful quoting. Best is to avoid string concatenation into `innerHTML`.

- **3 — Complete & precise:** Correct, specific, and explains the exact mechanism and lines.
- **2 — Substantially correct:** Core idea present but missing context/nuance (e.g., quoting/encoding detail).
- **1 — Attempt with misconceptions:** Related but confuses contexts (HTML vs JS vs DOM) or gives wrong remediation.
- **0 — Incorrect/irrelevant.**

Facet 1 — Locate (line numbers only)

What to list: the input source line(s) and the point(s) where it is placed into JS/DOM.

- **3 (complete & precise):** Lists **L25** (PHP echo into JS string) and **L28** (the `out.innerHTML = ... + email + ...` DOM insertion). Optionally include **L20** (the target element `out`) and **L24** (fetching the element). Example accepted: `25, 28` or `24–28`.
- **2:** Identifies only one of source/sink (e.g., only L28 or only L25).
- **1:** Mentions the general region (script block or form) without the exact lines.
- **0:** Wrong/unrelated lines.

Model (3): `25, 28` — `htmlspecialchars($_GET['email'])` echoed into JS and then inserted into DOM via `innerHTML`.

Facet 2 — Cause (static mechanism — why)

What to expect: explanation of why the input flow leads to XSS, with attention to contexts (PHP → JS string → DOM).

- **3:** Precisely explains two failures: (1) the PHP escaping used (`htmlspecialchars`) is not appropriate for **embedding into a single-quoted JS string** (it does not escape single quotes by default), and (2) the value is later inserted via `innerHTML`, which interprets HTML — so any content that breaks out of the JS string or contains HTML, once interpreted, leads to XSS. Mentions the correct contextual encoding is missing (`json_encode()`/JS-escaping or ENT_QUOTES + safe usage).

- **2:** Correctly identifies that user input is inserted into the page and that `innerHTML` is unsafe, but omits the subtlety about `htmlspecialchars` lacking single-quote escaping for JS single-quoted literal.
- **1:** Partly relevant but confuses contexts (e.g., says “htmlspecialchars prevents XSS” without recognizing the JS embedding issue) or mislabels it as only an HTML-only issue.
- **0:** Incorrect (e.g., claims there's no issue because htmlspecialchars is used).

Model (3):

`$_GET['email']` is echoed into a JS single-quoted string at L25 using `htmlspecialchars()` (which by default does not escape single quotes). If the attacker supplies a single quote or a payload that breaks out of the string, they can inject JS; because `email` is later placed into the DOM via `innerHTML` (L28), this results in XSS.

Facet 3 — Behavior (runtime manifestation — how it plays out)

What to expect: concrete runtime flow: what an attacker can achieve and how the browser executes it.

- **3:** Describes how attacker-controlled input can break the JS string or produce HTML that `innerHTML` parses, leading to execution of injected scripts or insertion of attacker-controlled markup. Gives a conceptual example (e.g., input containing an unescaped single quote to terminate `'...'` and append `;/*malicious JS*/`), and explains that once `innerHTML` receives malicious content it will interpret it as HTML/JS in the DOM context. Mentions that some vector payloads depend on what client-side encoding is applied.
- **2:** States that an attacker could inject script or HTML and cause XSS, but does not explain the JS-string vs HTML parsing steps.
- **1:** Vague or incorrect runtime description (e.g., says “it will crash” or only references server-side issues).
- **0:** Irrelevant.

Model (3):

“At runtime, if the echoed `email` contains characters that close the single-quoted JS string or valid HTML markup, the resulting JavaScript/DOM assignment can execute attacker-supplied code. Because the code sets `innerHTML` with `'Welcome: <b>' + email + '</b>'`, any HTML or script the attacker injects into `email` will be parsed and can run in the page’s origin.”

Facet 4 — Consequence / Impact

What to expect: what properties are violated and severity.

- **3:** States specific impacts: full **reflected/DOM XSS** leading to cookie/session theft, CSRF token exfiltration, account takeover, or arbitrary actions in victim's context. Mentions that severity depends on page privileges (sensitive UI, authenticated session).
- **2:** Generic ("XSS allows running scripts") without connecting to real consequences.
- **1:** Slightly off or trivial (e.g., "layout changes only").
- **0:** Irrelevant.

Model (3):

"This is a DOM/Reflected XSS which can execute script in victims' browsers under the site's origin; attackers could steal session cookies, perform actions on behalf of users, or display phishing UI — high severity for authenticated pages."

Facet 5 — Prevention / Mitigation

What to expect: concrete, context-aware fixes (server-side and client-side), prioritized.

- **3 (best):** Provides specific, safe fixes such as:

Do not embed raw user data into JS string literals using HTML escapers. Instead, serialize server-side to a JS-safe value with `json_encode()` in PHP:

```
<script>

  const email = <?= json_encode(isset($_GET['email'])? $_GET['email'] : '') ?>;

  // use email safely; then set textContent or create elements

</script>
```

1. `json_encode()` properly escapes quotes and control characters for JS.

Avoid innerHTML for inserting untrusted content. Use `textContent` or `createTextNode` to insert text safely:

```
out.textContent = 'Welcome: ' + email;
```

2. or build elements and set their `textContent`.
3. If HTML must be built, sanitize on the server or use a well-audited client-side sanitizer library and whitelist tags/attributes.
4. If you must embed into HTML contexts, use the correct encoder for that exact context (HTML-attribute, HTML-text, JS-string, URL, CSS) — e.g., `ENT_QUOTES` for HTML quotes and `json_encode()` for JS.
5. Trim/validate the `email` server-side (validate format) and consider server-side canonicalization.

- **2:** Suggests general “escape/validate” or “use htmlspecialchars” without specifying the correct contextual method for JS embedding or recommending `textContent` instead of `innerHTML`.
- **1:** Weak/inapplicable fixes (e.g., “use client-side validation only”).
- **0:** Wrong/irrelevant.

Model (3) — safe replacement snippets:

Server → JS safe embed:

```
<script>

 const email = <?= json_encode(isset($_GET['email']) ? $_GET['email'] : '') ?>;

 const out = document.getElementById('out');

 if (email.split('@').length === 2) {

  out.textContent = 'Welcome: ';

  const b = document.createElement('b');

  b.textContent = email;

  out.appendChild(b);

 }

</script>
```

- `json_encode()` produces a JS string literal safely; `textContent`/`createElement` avoids parsing HTML.
- **If embedding in HTML** (not JS): use `htmlspecialchars($value, ENT_QUOTES, 'UTF-8')` and place it into an HTML text node or attribute with proper quoting.

XSS-2

- **User-controlled input (source):** `$_REQUEST['todo-username']` — provided by the GET form (L45 → submitted to server).
- **Server-side transformation:** `preg_replace('/[<>]/', '', $_REQUEST['todo-username'])` at **L70** removes only `<` and `>` characters. This does **not** escape quotes (`"` or `'`), backslashes, newlines, or other characters that matter inside a JavaScript string literal.
- **Sink / injection context:** the result is embedded directly into a **double-quoted JavaScript string literal** passed to `SessionManager(...)` — see **L70**: `new SessionManager("...here...");`. That string is then assigned to `this.username` and later used inside template literals (`console.log(...)`) — but the primary risk is that the unescaped insertion in L70 can **break out of the JS string** and inject arbitrary JS. This is a reflected XSS (server echoes user input into page → immediate execution).
- **Why it's vulnerable:** removing only `</>` does not neutralize characters that can break a JS string (for example a `"`character, a `\\` backslash, or a sequence that closes the string and appends `; /*malicious*/`). Because the input is placed inside JS quotes and not JSON-encoded or JS-escaped, an attacker can craft input that terminates the string and executes script in the page's origin.
- **Attack surface:** reflected JS injection → arbitrary script execution in victims' browsers, with all consequences (cookie theft, CSRF, UI spoofing).
- **Note:** `testTodoClass = new todoClass();` is an unrelated JS error (possible noise) and not the root of XSS.

Facet 1 — Locate (line numbers only)

Expect: the input read site and the server-to-client injection site.

- **3 (complete & precise):** Lists **L45** (input field `name="todo-username"`) and **L70** (PHP echo inserted into JS: `new SessionManager("<?= preg_replace... ?>");`). Accept variants `45, 70` or `45, 70, 53–58` (context).
- **2:** Identifies the injection site (L70) but omits the form/input line (L45), or vice versa.
- **1:** Mentions script block or form area without exact lines.
- **0:** Wrong lines.

Model (3): `45, 70`.

Facet 2 — Cause (static mechanism — why)

Expect: explanation of how the flow/context enables XSS.

- **3:** Precisely says: server removes only `<` and `>` via `preg_replace` (L70) but **does not perform JavaScript escaping or JSON encoding**; user data is embedded directly inside a **double-quoted JS string literal**, so characters like `"` or `\\` or `;` can terminate or alter the JS and allow arbitrary script execution. The contextual encoder is wrong/incomplete.
- **2:** Correctly identifies unescaped user input embedded in JS leads to injection, but omits the exact mismatch (i.e., that only `</>` are removed).
- **1:** Mentions user input is echoed but confuses with HTML-context escaping only.
- **0:** Incorrect.

Model (3):

`preg_replace('/[<>]/', '', ...)` strips `</>` but does not escape quotes/backslashes/newlines; because the result is placed inside `new SessionManager("...")` (a JS string), an attacker can craft input to break out and inject JS → reflected XSS.

Facet 3 — Behavior (runtime manifestation — how)

Expect: concrete runtime story and exploitation vector.

- **3:** Explains attacker flow: attacker submits a specially crafted `todo-username` that contains characters that close the JS string (e.g., `"; alert(1); //`) or a backslash sequence; the browser executes the resulting injected JavaScript when parsing the script block. Because `this.username` is later used in console logs and could be used elsewhere, the attacker can run arbitrary script in the page origin. Example transformation: server outputs `new SessionManager(""; alert(1); //");` which runs `alert(1)`. Mentions reflected nature (GET parameter → immediate page response).
- **2:** Says XSS possible and attacker can run scripts, but doesn't connect exact JS-string-breakout mechanics.

- **1:** Vague or wrong runtime description (e.g., only says “it prints username” without exploit mechanics).
- **0:** Irrelevant.

Model (3):

“At runtime the server echo is inserted into a double-quoted JS literal; if input contains `"` and code to follow, it will terminate the literal and inject new statements — the browser will parse and execute them, leading to script execution under the site’s origin.”

Facet 4 — Consequence / Impact

Expect: concrete impact scenarios and severity.

- **3:** Specific: reflected XSS allows script execution in users’ browsers under the site origin → session cookie theft, token exfiltration, forced actions, UI spoofing, or persistent attacks if logged somewhere. Severity = high for authenticated contexts. Mention that consequences depend on what the page exposes (console logs are low-value, but attackers can escalate).
- **2:** Generic (“allows script execution”) without examples.
- **1:** Slightly off.
- **0:** Irrelevant.

Model (3):

“Reflected JS injection gives an attacker the ability to run arbitrary script in victims’ browsers (cookie/session theft, CSRF, phishing UI); severity is high if users are authenticated or sensitive page functions exist.”

Facet 5 — Prevention / Mitigation

Expect: concrete, context-aware fixes; recommend best practices and show secure code.

- **3 (best):** Multiple precise fixes, prioritized:

Proper JS encoding: embed user data into JS using `json_encode()` (or a dedicated JS encoder) on the server so the value becomes a safe JS string literal:

```
<script>

  const username = <?= json_encode($_REQUEST['todo-username'] ?? '') ?>;
  const test = new SessionManager(username);

</script>
```

1. `json_encode()` handles quotes, backslashes, and control chars correctly.

Avoid direct execution contexts: do not place untrusted data inside script code; pass data via data attributes and read via `dataset`, or set server-rendered text in elements and read `textContent`. Example:

```
<div id="user" data-username="<?= htmlspecialchars($u, ENT_QUOTES|ENT_SUBSTITUTE, 'UTF-8') ?>"></div>

<script>

  const username = document.getElementById('user').dataset.username;

</script>
```

2.
3. **Validate & canonicalize** server-side: enforce allowed character set and length for usernames (e.g., alnum + limited punctuation). Validation is not a substitute for encoding, but it reduces risk.
4. **Avoid ad-hoc filters:** `preg_replace('/[<>]/','',...)` is insufficient — never rely on removing a couple of characters to secure a different context.
5. **Use CSP** as an additional defense-in-depth to limit script execution.

- **2:** General advice like "escape input" but without specifying the right encoder for JS or alternatives (json_encode vs htmlspecialchars).
- **1:** Weak advice (client-side validation only).
- **0:** Wrong.

Model (3) code snippet:

```
<?php

  $safe = isset($_REQUEST['todo-username']) ? $_REQUEST['todo-username'] : '';

?>

<script>

  // json_encode returns a safe JS literal (with quotes)
  const username = <?= json_encode($safe) ?>;

  const test = new SessionManager(username);

  test.startSession();

</script>
```

Or using data-attribute:

```
<div id="user" data-username="<?= htmlspecialchars($safe,
ENT_QUOTES|ENT_SUBSTITUTE, 'UTF-8') ?>"></div>

<script>

  const username = document.getElementById('user').dataset.username;

</script>
```