Zoho App sec Round - 1 (No Quants)

OWASP Top-10 (concise map you must quote in answers)

- **A01 — Broken Access Control** (IDOR, missing auth checks)
- **A02 — Cryptographic Failures** (weak/absent encryption, bad password storage)
- **A03 — Injection** (SQLi, OS cmd, LDAP, XSS — user input inserted into commands)
- **A04 — Insecure Design** (flaws in architecture/design, missing threat modeling)
- **A05 — Security Misconfiguration** (debug enabled, default creds)
- **A06 — Vulnerable & Outdated Components** (old libs, OSS with CVEs)
- **A07 — Identification & Authentication Failures** (weak session management, broken auth)
- **A08 — Software & Data Integrity Failures** (insecure deserialization, supply chain)
- **A09 — Security Logging & Monitoring Failures** (no detection, logs missing)
- **A10 — Server-Side Request Forgery (SSRF)** (server fetches attacker controlled URL)

(XSS and CSRF are classic — treat XSS as Injection and CSRF under Broken Auth/Design.)

For each vulnerability — examples (Java / Python / C/C++) + how to review + fix

---

1) Injection (SQLi, Command Injection, XSS)

Why it matters: attacker-supplied data gets directly interpreted by a backend system (DB, shell, XML parser, browser).

Java — SQL Injection (vulnerable)

```
// Vulnerable: concatenating user input into SQL
String q = "SELECT * FROM users WHERE username = '" + username + "' AND
password = '" + password + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(q);
```

How to spot in review: string concatenation in SQL queries, user input variables inside quotes.

Fix (secure):

```
String q = "SELECT * FROM users WHERE username = ? AND password = ?";
PreparedStatement ps = conn.prepareStatement(q);
ps.setString(1, username);
ps.setString(2, passwordHash);
ResultSet rs = ps.executeQuery();
```

Notes: use parameterized queries and store hashed passwords (bcrypt/argon2).

Python — SQL Injection (vulnerable)

```
# Vulnerable (psycopg2)
cur.execute("SELECT * FROM users WHERE id = %s" % user_id)
```

Fix:

```
cur.execute("SELECT * FROM users WHERE id = %s", (user_id,))
```

C/C++ — Command injection / format-string / buffer overflow

Command injection (vulnerable):

```
// vulnerable: using system() with untrusted input
std::string cmd = "ping -c 1 " + ip; // ip from user
system(cmd.c_str());
```

How to spot: use of `system()`, `popen()`, or shell invocation with concatenated user input.

Fix: Validate and whitelist input (IP regex), use native libraries (sockets) instead of shell, avoid `system()`.

Format string / buffer overflow example (vulnerable):

```
char buf[64];
strcpy(buf, user_input); // no bounds check -> buffer overflow
printf(buf);             // format string vulnerability if user contains %s
```

Fix: use `snprintf`, `printf("%s", buf)`, and bounds-check copy functions.

XSS (cross-site scripting)

- **Spot:** server prints user input into an HTML page without escaping.
- **Fix:** output encode (HTML-encode), use strict CSP, avoid `innerHTML` with raw values.

Java example (JSP vulnerable):

```
<%= request.getParameter("name") %>
```

Fix: escape: use JSTL `<c:out value="${param.name}" />` or use OWASP Java Encoder.

---

2) Broken Access Control (IDOR, missing server-side checks)

Why: client can access or modify data they shouldn't.

Java (vulnerable)

```
// Vulnerable: trusts client param for role
String role = request.getParameter("role");
if (role.equals("admin")) {
    // give admin page
}
```

Spot: auth/role checks based on client data; absence of server-side authentication/authorization.

Fix: Authenticate via server session (session attributes / JWT with server verification) and enforce RBAC/ABAC on server side. Verify `currentUser.hasRole("admin")`.

Python (Flask example — IDOR)

```
# Vulnerable: allows fetching any user's profile by id param
@app.route('/profile')
def profile():
    user_id = request.args.get('id')
    user = db.get_user(user_id)
    return render_template('profile.html', user=user)
```

Fix: use session to get current user; only fetch `current_user.id` or check `current_user.is_admin`.

C/C++ (file access / path traversal)

```
// vulnerable: open file using user-supplied filename
std::ifstream f("/var/www/data/" + filename);
```

Fix: sanitize filename, remove `../`, use a whitelist, or canonicalize path and ensure it is within allowed directory.

---

3) Cryptographic Failures (weak/absent encryption)

Why: passwords or secrets leaked / weak hashing -> account takeover.

Java (vulnerable)

```
MessageDigest md = MessageDigest.getInstance("MD5"); // bad: MD5
md.update(password.getBytes());
byte[] digest = md.digest();
```

Fix: use bcrypt/argon2; never roll your own. Use Java libs like `BCrypt` (Spring Security) or `Argon2` wrappers.

Python (vulnerable)

```
hashed = hashlib.sha1(password).hexdigest()  # weak + no salt
```

Fix: use `bcrypt` or `argon2-cffi` with proper salt & work factor.

C/C++ (vulnerable)

- Using MD5/SHA1 for password hashing. Use libs (libsodium/Argon2) and secure key storage.

Mitigations: TLS everywhere, HSTS, proper certificate validation, never store plaintext secrets, secrets stored in KMS/vault.

---

4) Insecure Design

Why: architectural decisions missing threat modeling, leading to many hidden exposures.

How to show during interview: explain threat model, trust boundaries, threat mitigations (rate limiting, least privilege, validated serialization, secure defaults). Provide design-level measures: threat modeling, secure-by-design, use of frameworks with secure defaults, data flow diagrams.

(No single code snippet — talk about design controls.)

---

5) Security Misconfiguration

Examples: debug enabled in production, default credentials, directory listing enabled, open management endpoints.

Python (Flask)

```
app.run(debug=True)  # debug reveals stacktraces & remote code exec in Werkzeug
```

Fix: Disable debug in production.

Java (Spring Boot)

- Exposed actuator endpoints without auth. Fix: secure actuator, remove sensitive endpoints, use role-based protection.

---

6) Vulnerable & Outdated Components

Spot: dependencies with known CVEs, direct use of old libraries.

Fix: keep dependency inventory (SBOM), run dependency scanners (Snyk/OSS Index/Dependabot), update patches, pin known-safe versions.

---

7) Identification & Authentication Failures

Examples: session fixation, weak password reset, credentials in URL, long-lived tokens.

Java (session fixation)

Bad: reuse old session after login.
Fix: `request.changeSessionId()` (or equivalent) after successful login; set Secure & HttpOnly cookies.

Python (token leakage)

- Placing tokens in query strings (URLs) -> logs and referer leak.
- Fix: send tokens in Authorization header or HttpOnly cookie.

8) Software & Data Integrity Failures (insecure deserialization)

Java (vulnerable):

```
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
Object obj = in.readObject(); // dangerous if input untrusted
```

Python (vulnerable):

```
obj = pickle.loads(user_data)  # executes arbitrary constructors
```

Fix: avoid native serialization of untrusted data; use safe formats (JSON) and strict schema validation. If deserialization is necessary, use allowlist patterns/libraries that are safe.

---

9) Security Logging & Monitoring Failures

Wrong: logging sensitive data (passwords, tokens), insufficient logs, no alerting.

Fix: log important authentications and failures, do not log secrets, centralize logs, alert on repeated failed auths, use immutable retention policy.

---

10) SSRF (Server-Side Request Forgery)

Vulnerable Java:

```
URL url = new URL(request.getParameter("url"));
InputStream in = url.openStream(); // attacker can force server to fetch
internal resources
```

Fix: whitelist allowed hosts / IP ranges, block private IP ranges, use a proxy that blocks internal addresses, validate URL scheme.

Language-specific vulnerability lists (quick reference)

Java — common vulnerabilities

- SQL Injection via string concatenation
- Insecure deserialization (readObject)
- XSS (JSP/servlets printing user input)
- CSRF (no anti-CSRF token)
- Broken Access Control (IDOR, unsecured endpoints)
- Weak crypto (MD5/SHA1, no salt)
- Session fixation & insecure cookies
- Security misconfig (debug, open actuator)
- Use of outdated libs (Log4j-style examples)
- Improper input validation (regex/encoding mistakes)

C / C++ — common vulnerabilities

- Buffer overflow (unsafe `strcpy`, `gets`)
- Use-after-free / double free
- Integer overflow leading to buffer overflow
- Format string vulnerabilities (`printf(user_input)`)
- Command injection via `system()` or `popen()`
- Path traversal (`fopen` with user filename)
- Undefined behavior / memory leaks causing escalation or info leak
- Lack of bounds-checking on network reads

Python — common vulnerabilities

- SQLi when using string formatting in queries
- Insecure deserialization (`pickle.loads`)
- Command injection via `os.system` / `subprocess` with shell=True
- XSS in templates if autoescape disabled or using Markup incorrectly
- Improper crypto usage (hashlib for password storage)
- Directory traversal (unsanitized file paths)
- Using debug mode / exposing stacktraces
- Dependency issues (outdated packages)

Practice Code-Review Snippets (simulate the test)

For each snippet: identify bug quickly (1–2 lines), reason, and fix.

Snippet 1 — Java (SQLi)

```java
String q = "SELECT * FROM users WHERE email = '" + email + "'";
ResultSet rs = stmt.executeQuery(q);
```

Answer: SQL Injection. Fix: `PreparedStatement` with parameter binding.

Snippet 2 — Python (Command injection)

```python
@app.route('/ping')
def ping():
    ip = request.args.get('ip')
    os.system("ping -c 4 " + ip)
    return "done"
```

Answer: OS command injection. Fix: validate/whitelist IPs, avoid `system()`; use native network code or `subprocess` with args list (no `shell=True`), or validate with regex.

Snippet 3 — C (buffer overflow)

```c
char buf[32];
gets(buf); // deprecated and unsafe
```

Answer: Buffer overflow via `gets()`. Fix: `fgets(buf, sizeof buf, stdin)` with bounds checks.

Snippet 4 — JavaScript/HTML (Reflected XSS)

```html
<!-- vulnerable -->
Hello <span id="name"></span>
<script>
document.getElementById('name').innerHTML =
decodeURIComponent(location.search.split('name=')[1]);
</script>
```

Answer: XSS — using `innerHTML` with unsanitized input. Fix: use `textContent` or sanitize.

Snippet 5 — Java (Insecure password storage)

```
// storing plain text
stmt.executeUpdate("INSERT INTO users(email, password) VALUES('" + email +
"', '" + password + "')");
```

Answer: Plaintext passwords and SQLi. Fix: hash with bcrypt/argon2 and use prepared statements.

Snippet 6 — Python (pickle)

```
data = request.data
obj = pickle.loads(data)
```

Answer: Insecure deserialization. Fix: don't use pickle for untrusted input; use json + schema validation.

Snippet 7 — C++ (format string)

```
std::string name = get_user_input();
printf(name.c_str());
```

Answer: Format string vulnerability. Fix: `printf("%s", name.c_str());`

Snippet 8 — Java (Broken access control)

```
// /admin?isAdmin=true
if(request.getParameter("isAdmin").equals("true")) {
  showAdminPage();
}
```

Answer: Trusting client parameters for auth. Fix: check server-side user role.

Snippet 9 — Python (debug enabled)

```
app.run(debug=True)
```

Answer: Debug enabled in prod — reveals detailed error pages & REPL. Fix: disable in production.

Snippet 10 — Java (SSRF)

```
String url = request.getParameter("url");
HttpURLConnection con = (HttpURLConnection) new URL(url).openConnection();
```

Answer: SSRF. Fix: whitelist hosts, parse and validate scheme/host/ip.

(Use these to test yourself: describe bug in <30 sec and then explain fix.)

Application-level bugs — how they work & mitigations

1. **Input validation missing**
   - *How it works:* user input used directly in queries/commands.
   - *Mitigation:* validate/whitelist input, use parameterized interfaces, length checks, type checks, canonicalization.
2. **Poor session management**
   - *How:* session ID in URL, no rotation after login, long expiry.
   - *Mitigation:* HTTPOnly, Secure cookies, SameSite, regenerate session id on auth, reasonable expiry, server-side session store.
3. **Exposing debug or stack traces**
   - *How:* errors show internal paths, DB queries.
   - *Mitigation:* show generic error pages; log detailed errors on server side only.
4. **Missing rate limiting**
   - *How:* brute force or credential stuffing.
   - *Mitigation:* throttle, exponential backoff, CAPTCHA after threshold.
5. **Sensitive data exposure**
   - *How:* logs contain PII, tokens; plaintext passwords.
   - *Mitigation:* mask PII, encrypt in transit & at rest, secure key management, restrict access.
6. **Improper input encoding / output escaping**

- *How:* reflected/stored XSS.
- *Mitigation:* output encoding per context (HTML, attribute, JS), CSP, sanitize inputs.

7. **Use of eval/deserialization with untrusted data**
   - *Mitigation:* avoid `eval`, `pickle`, Java deserialization; use safe formats & validation.

Cookies — what they are & types (interview-ready)

- **What is a cookie:** small key/value stored by browser for a domain. Used for sessions, preferences, tracking.
- **Types:**
  - **Session cookie:** expires when browser closes (no Expires set).
  - **Persistent cookie:** has Expires/Max-Age (survives browser restarts).
  - **Secure cookie:** `Secure` flag — only sent over HTTPS.
  - **HttpOnly cookie:** `HttpOnly` flag — not accessible via `document.cookie` (mitigates XSS theft).
  - **SameSite cookie:** `SameSite=Strict|Lax|None` — control cross-site sending (CSRF mitigation).
  - **Third-party cookie:** set by a domain different from the page domain.
- **Best practices:** HttpOnly + Secure + SameSite=Lax (or Strict where appropriate), minimal lifetime, store session id referencing server data not raw credentials, rotate session ids on privilege changes.

HTTP: what it is & important headers (must-know list with uses)

What is HTTP: stateless request/response protocol. Understand methods: GET (safe), POST (create), PUT/PATCH (update), DELETE (delete). Know status codes: 200, 302, 400, 401, 403, 404, 500, 503.

Important headers & use cases

- **Content-Type:** media type (e.g., `application/json`, `text/html`).
- **Authorization:** `Bearer <token>` or `Basic` — used to pass credentials.
- **WWW-Authenticate:** server challenge for auth (401).
- **Set-Cookie / Cookie:** manage cookies.
- **Content-Security-Policy (CSP):** restrict sources for scripts/styles/media — primary defense for XSS.

- **Strict-Transport-Security (HSTS):** `max-age=...; includeSubDomains` — force HTTPS.
- **X-Frame-Options:** `DENY|SAMEORIGIN` — prevent clickjacking.
- **X-Content-Type-Options:** `nosniff` — stop MIME sniffing.
- **Referrer-Policy:** control referrer header content.
- **Permissions-Policy (Feature-Policy):** control browser features (camera, geolocation).
- **Cache-Control:** `no-store`, `no-cache`, `max-age` — control caching behavior.
- **X-XSS-Protection:** legacy; modern rely on CSP.
- **X-Forwarded-For / Forwarded:** identify client IP behind proxies (use carefully).
- **Expect-CT:** enforce certificate transparency.
- **Public-Key-Pins:** deprecated — don't use.

Interview tip: mention that `Set-Cookie` flags matter: `Set-Cookie: sessionid=abc; HttpOnly; Secure; SameSite=Lax; Path=/; Max-Age=3600`

Additional interview talking points (short answers)

- **How to prevent SQLi?** Parameterized queries (prepared statements), ORM with bound parameters, input validation, least privilege DB user.
- **How to prevent XSS?** Output encoding, contextual escaping, CSP, input validation, HttpOnly cookies.
- **How to prevent CSRF?** Use anti-CSRF tokens, SameSite cookies, require re-auth for sensitive actions.
- **How to handle secrets?** Use environment variables + vault/KMS, rotate keys, avoid hardcoded secrets in code or config repo.
- **How to handle dependencies?** SBOM, SCA scans, automated dependency updates, pinned dependencies.

Common interview code review checklist (copy-paste before the round)

- Is every user input validated & canonicalized?
- Are DB calls parameterized?
- Are passwords hashed using bcrypt/argon2?
- Are session & auth tokens protected (HttpOnly, Secure, SameSite)?
- Are error pages generic (no stack trace)?

- Are third-party components up-to-date?
- Is deserialization avoided for untrusted data?
- Are appropriate HTTP security headers present?
- Are logs scrubbed for PII and contain auth & error events?
- Are file accesses protected against path traversal?

---

12 Practice code-review problems with short answers (great for timed practice)

1. **Java:**

```
response.getWriter().println("Welcome " + request.getParameter("user"));
```

   Bug: XSS (reflecting raw input). **Fix:** HTML-encode.
2. **Python:**

```
os.system("convert " + filename + " out.png")
```

   Bug: Command injection. **Fix:** use subprocess with arg list + validate filename.
3. **C:**

```
char buf[128];
read(fd, buf, 256);
```

   Bug: Buffer overflow (reading more than buffer size). **Fix:** bounds check.
4. **Java:**

```
Cookie c = new Cookie("session", sessionId);
c.setHttpOnly(false); // in prod
```

   Bug: cookie not HttpOnly. **Fix:** setHttpOnly(true), setSecure(true).
5. **Python (Flask):**

```
token = user.email + ":" + user.id
```

   Bug: predictable token. **Fix:** use cryptographically random tokens, short expiry.
6. **C++:**

```
sprintf(buf, "%s", user_input);
```

Bug: possible overflow if no size check. **Fix:** use `snprintf` with size.

7. **Java:**

```
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
Object o = in.readObject();
```

Bug: insecure deserialization. **Fix:** avoid or validate classes (use not deserializing untrusted streams).

8. **Python:**

```
data = json.loads(request.data)
eval(data['expr'])
```

Bug: arbitrary code execution via `eval`. **Fix:** remove eval; use a safe expression evaluator.

9. **C:**

```
FILE *f = fopen("/var/www/uploads/" + filename, "r");
```

Bug: path traversal if filename contains ../. **Fix:** sanitize filename and canonicalize path.

10. **Java:**

```
if (userParam.equals("admin")) { allow(); }
```

Bug: trusting user param. **Fix:** check server-side identity/role.

11. **Python:**

```
app = Flask(__name__)
app.config['DEBUG'] = True
```

Bug: debug in prod. **Fix:** disable debug, use config per env.

12. **C++ (network):**

```
// server logs raw request body to disk, including Authorization header
logfile << requestBody;
```

Bug: logging sensitive tokens. **Fix:** redact auth tokens, PII.

One-page Cheat sheet (copyable)

- Use **PreparedStatement / parameterized queries** (SQLi).

- Encode output for **HTML/JS/URL** (XSS).
- Use **CSP**, **HttpOnly**, **Secure**, **SameSite** cookies.
- **Never** `eval`, `pickle.loads`, `ObjectInputStream` on untrusted data.
- **Regenerate session ID** after login. Use short session expiry.
- **Whitelist** hosts for any server-side URL fetch (SSRF).
- **Hash** passwords with bcrypt/argon2 + salt.
- **Keep dependencies updated**, run SCA.
- **Disable debug** in production. Remove default creds.
- **Log** auth attempts, but **do not log secrets**.