Zoho Round 1 AppSec Cheatsheet (No Quants)

Java

OWASP Top 10 Vulnerabilities

A01: Broken Access Control (IDOR)

- **Vulnerability:** The application trusts user-provided parameters to make access control decisions, allowing users to access unauthorized data or functionality.
- **Vulnerable Code:**

```
// Trusts a client parameter to grant admin rights
if (request.getParameter("isAdmin").equals("true")) {
    showAdminPage();
}

// Fetches a user record based on a user-provided ID without checking
authorization
String userId = request.getParameter("userId");
User u = db.getUser(userId);
```

- **Fix:** Authorization decisions must be made server-side based on the user's session or a verified token (e.g., JWT).
  Always verify that the logged-in user has the permission to access or modify the requested resource. [cite: 3, 57, 60, 63, 64, 132, 192, 194, 308, 392, 902, 1099, 1102]

A02: Cryptographic Failures

- **Vulnerability:** Storing passwords using weak or broken hashing algorithms like MD5 or SHA1, or storing secrets like API keys directly in the code.
- **Vulnerable Code:**

```
// Using a weak hashing algorithm
MessageDigest md = MessageDigest.getInstance("MD5");
md.update(password.getBytes());

// Hardcoding secrets
String dbUser = "admin";
String dbPass = "password123";
```

- **Fix:**
  - Use strong, modern hashing algorithms with a salt, such as **Bcrypt** or **Argon2**. [cite: 78, 81, 133, 381, 382, 385, 918, 922, 1031, 1032, 1036, 1039]
  - Store secrets in environment variables or a secure vault, not in source code. [cite: 881, 884, 1034]

A03: Injection (SQLi, XSS)

SQL Injection (SQLi)

- **Vulnerability:** User-controlled input is concatenated directly into a SQL query, allowing an attacker to alter the query's logic. [cite: 24, 26, 128]
- **Vulnerable Code:**

```
String sql = "SELECT * FROM users WHERE id=" + request.getParameter("id");
ResultSet rs = stmt.executeQuery(sql);
```

- **Fix:** Use `PreparedStatement` (parameterized queries). [cite: 32, 36, 160, 373, 802, 878, 1028]

Cross-Site Scripting (XSS)

- **Vulnerability:** Unsanitized user input is rendered directly on a web page. [cite: 52, 130, 848]
- **Vulnerable Code (JSP):**

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

- **Fix:** Escape user-provided output using OWASP Java Encoder or `<c:out>`. [cite: 53, 56, 378, 849, 887, 1043]

A05: Security Misconfiguration

- **Vulnerability:** Exposing sensitive information through stack traces, enabling directory listing, or leaving debugging features enabled. [cite: 94, 135, 838]
- **Vulnerable Code:**

```
e.printStackTrace(response.getWriter());
```

- **Fix:**
  - Show generic error pages. [cite: 212, 456, 844, 1058]
  - Disable directory listing & debugging in production. [cite: 97, 513]
  - Secure Spring Boot Actuator endpoints. [cite: 99]

A06: Vulnerable and Outdated Components

- **Vulnerability:** Using libraries/frameworks with known vulnerabilities (e.g., old Log4j). [cite: 17, 101, 136, 515, 836, 1112]
- **Fix:** Maintain SBOM, scan dependencies with Snyk/Dependabot, keep updated. [cite: 102, 516, 1113]

A07: Identification and Authentication Failures

- **Vulnerability:** Poor session management (session fixation, insecure cookies). [cite: 18, 104, 134]
- **Vulnerable Code:**

```
Cookie c = new Cookie("sid", sessionId);
response.addCookie(c);
```

- **Fix:**
  - Invalidate old session (`request.changeSessionId()`). [cite: 107, 415, 1068]
  - Set **HttpOnly** and **Secure** cookie flags. [cite: 107, 425, 891, 1090]

A08: Software and Data Integrity Failures (Insecure Deserialization)

- **Vulnerability:** Deserializing untrusted data can lead to RCE. [cite: 19, 111, 113, 129, 829]
- **Vulnerable Code:**

```
ObjectInputStream in = new ObjectInputStream(request.getInputStream());
Object obj = in.readObject();
```

- **Fix:** Avoid deserialization of untrusted data. Use JSON + schema validation. [cite: 116, 406, 830, 900, 1053]

A09: Security Logging and Monitoring Failures

- **Vulnerability:** Not logging important events, logging sensitive data. [cite: 12, 119]
- **Vulnerable Code:**

```
logger.info("Password entered: " + pwd);

try {
    login(user, pass);
} catch(Exception e) {
    // ignore
}
```

- **Fix:**
  - Log auth attempts, access failures, validation errors. [cite: 120]
  - Never log sensitive info. [cite: 460, 911, 1108]
  - Monitor logs for anomalies. [cite: 120, 866]

A10: Server-Side Request Forgery (SSRF)

- **Vulnerability:** Server fetches attacker-controlled URLs. [cite: 20, 124]
- **Vulnerable Code:**

```
URL url = new URL(request.getParameter("url"));
InputStream in = url.openStream();
```

- **Fix:** Use allow-list for domains/IPs/protocols. [cite: 125, 916]

Other Vulnerabilities

- **Path Traversal**
  Vulnerable Code:
  ```
  File f = new File("/app/data/" + request.getParameter("file"));
  ```
  Fix: Sanitize input, canonicalize path. [cite: 74, 400, 1047]
- **XML External Entity (XXE)**
  Vulnerable Code:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
```

Fix: Disable DTDs and external entities. [cite: 444, 807]

- **Open Redirect**
  Vulnerable Code:

```
String target = request.getParameter("url");
response.sendRedirect(target);
```

  Fix: Use allow-list or relative paths. [cite: 434, 618, 1097]
- **Insecure Randomness**
  Vulnerability: Using `java.util.Random` for security-critical tokens.
  Fix: Use `SecureRandom`. [cite: 438, 895]

Perfect 👍 I'll continue with **Python** in the same clean, structured style I used for Java. Here's the reformatted **OWASP Top 10 Vulnerabilities in Python** (keeping all your content intact but better organized for readability):

Python

OWASP Top 10 Vulnerabilities

A01: Broken Access Control (IDOR)

- **Vulnerability:** Application relies on client-supplied parameters for access control, enabling unauthorized access.
- **Vulnerable Code (Flask):**

```
@app.route('/profile')
def profile():
    user_id = request.args.get('user_id')
    user = db.get_user(user_id)
    return render_template('profile.html', user=user)
```

- **Fix:** Use session-based user IDs, verify ownership on server. [cite: 3, 57, 60, 63, 64, 132, 192, 194, 308, 392, 902, 1099, 1102]

A02: Cryptographic Failures

- **Vulnerability:** Using insecure algorithms like MD5 or SHA1 for password storage, or hardcoding secrets in code.
- **Vulnerable Code:**

```
import hashlib
hashlib.md5(password.encode()).hexdigest()

API_KEY = "1234567890abcdef"
```

- **Fix:**
  - Use **bcrypt**, **argon2**, or **scrypt** for password hashing. [cite: 78, 81, 133, 381, 382, 385, 918, 922, 1031, 1032, 1036, 1039]
  - Load secrets from environment variables or secret managers. [cite: 881, 884, 1034]

A03: Injection (SQLi, XSS, Command Injection)

SQL Injection (SQLi)

- **Vulnerability:** Unsanitized user input is directly used in SQL queries.
- **Vulnerable Code:**

```
cursor.execute("SELECT * FROM users WHERE id = " + request.args['id'])
```

- **Fix:** Use parameterized queries. [cite: 32, 36, 160, 373, 802, 878, 1028]

Cross-Site Scripting (XSS)

- **Vulnerability:** Rendering unsanitized user input in templates.
- **Vulnerable Code (Flask/Jinja):**

```
{{ request.args['msg'] }}
```

- **Fix:** Use auto-escaping in templates. [cite: 53, 56, 378, 849, 887, 1043]

Command Injection

- **Vulnerability:** Passing user input directly to system commands.
- **Vulnerable Code:**

```
import os
os.system("ping " + request.args['host'])
```

- **Fix:** Use safe libraries (`subprocess.run` with list args, input validation).

A05: Security Misconfiguration

- **Vulnerability:** Debug mode left enabled in production.
- **Vulnerable Code:**

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

- **Fix:** Disable debug mode, configure proper error handlers. [cite: 94, 135, 838

A06: Vulnerable and Outdated Components

- **Vulnerability:** Using outdated Python packages with known CVEs. [cite: 17, 101, 136, 515, 836, 1112]
- **Fix:** Use tools like **pip-audit**, **safety**, or **dependabot** to scan dependencies. [cite: 102, 516, 1113]

A07: Identification and Authentication Failures

- **Vulnerability:** Weak session handling, predictable session IDs, missing cookie flags.
- **Vulnerable Code (Flask):**

```
resp.set_cookie("sid", session_id)
```

- **Fix:**
  - Use **Flask-Login** or JWT with strong secrets.
  - Set `HttpOnly` and `Secure` flags for cookies. [cite: 107, 425, 891, 1090]

A08: Software and Data Integrity Failures (Insecure Deserialization)

- **Vulnerability:** Loading pickled data from untrusted sources. [cite: 19, 111, 113, 129, 829]
- **Vulnerable Code:**

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

- **Fix:** Never use `pickle` with untrusted input. Use JSON with schema validation. [cite: 116, 406, 830, 900, 1053]

A09: Security Logging and Monitoring Failures

- **Vulnerability:** Missing or insecure logging.
- **Vulnerable Code:**

```
logger.info(f"User logged in with password: {pwd}")
```

- **Fix:**
  - Log login attempts, access denials, errors. [cite: 120]
  - Exclude sensitive data from logs. [cite: 460, 911, 1108]
  - Centralize and monitor logs. [cite: 120, 866]

A10: Server-Side Request Forgery (SSRF)

- **Vulnerability:** Fetching user-supplied URLs without validation. [cite: 20, 124]
- **Vulnerable Code:**

```
import requests
r = requests.get(request.args['url'])
```

- **Fix:** Restrict requests to trusted domains, validate URLs. [cite: 125, 916]

Other Vulnerabilities

- **Path Traversal**
  Vulnerable Code:
  ```
  open("/app/data/" + request.args['file'])
  ```
  Fix: Canonicalize path, enforce whitelist. [cite: 74, 400, 1047]
- **XML External Entity (XXE)**
  Vulnerable Code:

```
import lxml.etree as ET
ET.parse(request.data)
```

Fix: Disable external entity resolution. [cite: 444, 807]

- **Open Redirect**
  Vulnerable Code:

```
return redirect(request.args['url'])
```

  Fix: Restrict to allow-listed URLs. [cite: 434, 618, 1097]

- **Insecure Randomness**
  Vulnerability: Using `random` for tokens.
  Vulnerable Code:

```
import random
token = str(random.random())
```

  Fix: Use `secrets` or `os.urandom`. [cite: 438, 895]

C / C++

OWASP Top 10 Vulnerabilities

A01: Broken Access Control (IDOR)

- **Vulnerability:** Access control checks are missing or performed on client input.
- **Vulnerable Code:**

```
// Accessing files based on user-supplied input
std::string filename = argv[1];
std::ifstream file("/home/data/" + filename);
```

- **Fix:** Always enforce server-side authorization, validate inputs, and avoid direct reliance on client-supplied data. [cite: 3, 57, 60, 63, 64, 132, 192, 194, 308, 392, 902, 1099, 1102]

A02: Cryptographic Failures

- **Vulnerability:** Using weak cryptographic primitives (e.g., MD5, DES), or storing secrets in plain text.
- **Vulnerable Code:**

```
#include <openssl/md5.h>
MD5((unsigned char*)password, strlen(password), hash);
```

- **Fix:**
  - Use strong hashing algorithms like **bcrypt**, **argon2**, or **PBKDF2**. [cite: 78, 81, 133, 381, 382, 385, 918, 922, 1031, 1032, 1036, 1039]
  - Store secrets in secure storage, not in source code. [cite: 881, 884, 1034]

A03: Injection (SQLi, Command Injection, XSS in C++ Web Frameworks)

SQL Injection (SQLi)

- **Vulnerability:** Concatenating user input into SQL queries.
- **Vulnerable Code:**

```
std::string query = "SELECT * FROM users WHERE id = " + userInput;
db.exec(query);
```

- **Fix:** Use parameterized queries or prepared statements. [cite: 32, 36, 160, 373, 802, 878, 1028]

Command Injection

- **Vulnerability:** Passing unsanitized input into `system()`.
- **Vulnerable Code:**

```
system(("ping " + userInput).c_str());
```

- **Fix:** Avoid `system()`. Use safe APIs like `execve()` with proper argument arrays.

A05: Security Misconfiguration

- **Vulnerability:** Exposing debug symbols, verbose error messages, or leaving unsafe compiler flags.
- **Examples:**
  - Shipping binaries with debugging info (`-g`).
  - Not disabling verbose error output.
- **Fix:**

- Strip debug symbols in production (`strip` or `-s`).
- Provide generic error messages. [cite: 94, 135, 838]

A06: Vulnerable and Outdated Components

- **Vulnerability:** Using outdated C libraries (e.g., old OpenSSL, outdated zlib) with known CVEs. [cite: 17, 101, 136, 515, 836, 1112]
- **Fix:** Keep dependencies updated, use vulnerability scanning tools. [cite: 102, 516, 1113]

A07: Identification and Authentication Failures

- **Vulnerability:** Storing passwords in plain text, insecure session handling.
- **Vulnerable Code:**

```
std::string password = "userpass"; // Stored as plain text
```

- **Fix:** Use salted password hashing (bcrypt/argon2 libraries). [cite: 107, 425, 891, 1090]

A08: Software and Data Integrity Failures (Insecure Deserialization / Unsafe Libraries)

- **Vulnerability:** Deserializing untrusted binary data or unsafe use of serialization libraries. [cite: 19, 111, 113, 129, 829]
- **Fix:** Use safer formats (JSON, Protocol Buffers) with schema validation. [cite: 116, 406, 830, 900, 1053]

A09: Security Logging and Monitoring Failures

- **Vulnerability:** Logging sensitive data, or failing to log critical events.
- **Vulnerable Code:**

```
std::cout << "Password: " << password << std::endl;
```

- **Fix:**
  - Never log sensitive info (passwords, keys). [cite: 460, 911, 1108]

- Log access attempts, failures, anomalies. [cite: 120, 866]

A10: Server-Side Request Forgery (SSRF)

- **Vulnerability:** Fetching URLs from user input without validation. [cite: 20, 124]
- **Vulnerable Code:**

```
std::string url = userInput;
fetchUrl(url);
```

- **Fix:** Restrict to allow-listed domains/IPs. [cite: 125, 916]

Other Vulnerabilities

- **Buffer Overflow**
  Vulnerable Code:
  ```
  char buf[10];
  strcpy(buf, userInput.c_str()); // no bounds check
  ```
  Fix: Use safer functions like `strncpy`, `snprintf`, or C++ strings. [cite: 70, 72, 137, 787]
- **Use-After-Free**
  Vulnerable Code:
  ```
  char* ptr = (char*)malloc(10);
  free(ptr);
  strcpy(ptr, "data"); // UAF
  ```
  Fix: Set pointer to `nullptr` after free. Use smart pointers. [cite: 75, 791]
- **Integer Overflow**
  Vulnerable Code:
  ```
  int size = a + b; // may overflow
  char* buf = new char[size];
  ```
  Fix: Check boundaries before arithmetic. Use safe integer libraries. [cite: 73, 789]
- **Race Conditions (TOCTOU)**
  Vulnerable Code:
  ```
  if (access("file.txt", W_OK) == 0) {
      fd = open("file.txt", O_WRONLY);
  }
  ```
  Fix: Open files with secure flags (`O_CREAT | O_EXCL`). [cite: 76, 793]

Security Based Questions

Q1. What is the difference between Authentication and Authorization?

- **Authentication:** Verifying *who* the user is (identity verification).
  - Example: Login with username & password, biometrics, OTP.
- **Authorization:** Determining *what* the authenticated user can do (permissions).
  - Example: Admins can delete users, normal users cannot.

Q2. What is the difference between Symmetric and Asymmetric Encryption?

- **Symmetric Encryption:** Same key used for encryption & decryption.
  - Faster but requires secure key sharing.
  - Example: AES.
- **Asymmetric Encryption:** Uses a **public key** for encryption and a **private key** for decryption.
  - Slower but solves key distribution problem.
  - Example: RSA, ECC.

Q3. What is the difference between Hashing and Encryption?

- **Hashing:**
  - One-way transformation (cannot be reversed).
  - Used for data integrity and password storage.
  - Example: SHA-256, bcrypt.
- **Encryption:**
  - Two-way transformation (can be decrypted with the correct key).
  - Used for confidentiality of data.
  - Example: AES, RSA.

Q4. What is the difference between Encoding, Encryption, and Hashing?

- **Encoding:**
  - Reversible, meant for data transport or storage.
  - Example: Base64, URL encoding.
- **Encryption:**
  - Reversible with a key, provides confidentiality.
  - Example: AES, RSA.
- **Hashing:**
  - Irreversible, provides integrity verification.
  - Example: SHA-256, bcrypt.

Q5. What is the difference between Vulnerability, Threat, and Risk?

- **Vulnerability:** A weakness in the system.
- **Threat:** A potential event/exploit that could harm the system.
- **Risk:** The likelihood that a threat exploits a vulnerability, causing damage.

Q6. Explain CIA Triad.

- **Confidentiality:** Ensuring data is accessible only to authorized users.
- **Integrity:** Ensuring data is accurate and not tampered with.
- **Availability:** Ensuring systems and data are available when needed.

Q7. What is SQL Injection and how to prevent it?

- **SQL Injection:** Attack where user input is injected into SQL queries to manipulate database.
- **Prevention:** Use prepared statements, parameterized queries, stored procedures, and input validation.

Q8. What is Cross-Site Scripting (XSS) and how to prevent it?

- **XSS:** Injection of malicious scripts into web pages viewed by users.
- **Prevention:**

- Escape/encode output.
- Use Content Security Policy (CSP).
- Input validation.

Q9. What is CSRF (Cross-Site Request Forgery)?

- **CSRF:** Attack where a user is tricked into performing actions on a site where they are authenticated.
- **Prevention:**
  - CSRF tokens.
  - SameSite cookies.
  - User re-authentication for sensitive actions.

Q10. Difference between Black Box, White Box, and Grey Box Testing.

- **Black Box:** Tester has no knowledge of internal code. Tests from external perspective.
- **White Box:** Tester has full access to source code and internal logic.
- **Grey Box:** Tester has partial knowledge of system internals.

Q11. What is SSL/TLS and why is it important?

- **SSL/TLS:** Protocols that secure communication over the internet by encrypting traffic.
- **Importance:** Provides confidentiality, integrity, and authenticity of data in transit (HTTPS).

Q12. What is the difference between IDS and IPS?

- **IDS (Intrusion Detection System):** Monitors traffic for malicious activity and alerts.
- **IPS (Intrusion Prevention System):** Detects and actively blocks malicious traffic.

Q13. What is the difference between Virus, Worm, and Trojan?

- **Virus:** Malicious code that attaches to programs/files and spreads when executed.
- **Worm:** Self-replicating malware that spreads over networks without user action.
- **Trojan:** Malware disguised as legitimate software.

Q14. What is a Zero-Day Vulnerability?

- **Definition:** A vulnerability unknown to vendors/defenders and exploited by attackers before a fix is available.

Q15. What is the difference between Static and Dynamic Application Security Testing (SAST vs DAST)?

- **SAST (Static):** Analyzes source code or binaries for vulnerabilities without running the program.
- **DAST (Dynamic):** Tests the application while it is running, simulating real-world attacks.
