Zoho Practice Questions 2

Java (10)

Q1. SQL Injection

```
String query = "SELECT * FROM users WHERE id=" + request.getParameter("id");
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery(query);
```

Bug: SQL Injection.
Fix: Use `PreparedStatement` with placeholders.

Q2. Hardcoded credentials

```
String dbUser = "admin";
String dbPass = "password123";
Connection con = DriverManager.getConnection(url, dbUser, dbPass);
```

Bug: Hardcoded credentials.
Fix: Store secrets in environment variables or secure vault.

Q3. XSS in JSP

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

Bug: Reflected XSS (unsanitized input rendered).
Fix: Use `<c:out>` or OWASP Java Encoder.

Q4. Insecure cookie

```
Cookie c = new Cookie("session", sessionId);
c.setHttpOnly(false);
response.addCookie(c);
```

Bug: Cookie accessible via JS, can be stolen via XSS.
Fix: Set `HttpOnly=true`, `Secure=true`, `SameSite`.

Q5. Insecure Random

```
int otp = new Random().nextInt(999999);
```

Bug: `java.util.Random` is predictable.
Fix: Use `SecureRandom`.

Q6. Insecure Deserialization

```
ObjectInputStream in = new ObjectInputStream(request.getInputStream());
User u = (User) in.readObject();
```

Bug: Insecure deserialization, possible RCE.
Fix: Avoid raw deserialization, use JSON with schema validation.

Q7. Missing authorization

```
if (request.getParameter("role").equals("admin")) {
   showAdminPage();
}
```

Bug: Trusts client input for role.
Fix: Check role from server-side session/JWT claims.

Q8. Sensitive data exposure

```
System.out.println("User password: " + password);
```

Bug: Logging sensitive info.
Fix: Mask or avoid logging credentials.

Q9. SSRF

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

Bug: SSRF (attacker can hit internal services).
Fix: Whitelist allowed domains/IP ranges.

Q10. Weak password storage

```
MessageDigest md = MessageDigest.getInstance("MD5");
md.update(password.getBytes());
```

Bug: MD5 hashing (fast & broken).
Fix: Use bcrypt/argon2.

Python (10)

Q11. SQL Injection

```
cur.execute("SELECT * FROM users WHERE name='%s'" % username)
```

Bug: SQL Injection.
Fix: Use parameterized queries: `cur.execute("SELECT ... WHERE name=%s", (username,))`.

Q12. Command Injection

```
os.system("ping -c 1 " + ip)
```

Bug: OS Command Injection.
Fix: Use `subprocess.run(["ping", "-c", "1", ip])` + input validation.

Q13. Insecure Deserialization

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

Bug: Arbitrary code execution.
Fix: Use JSON with validation.

Q14. Debug mode enabled

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

Bug: Debug in production → RCE.
Fix: Disable debug in production.

Q15. Weak random

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

Bug: Predictable token.
Fix: Use `secrets.token_hex()`.

Q16. Path traversal

```
f = open("/var/www/uploads/" + filename)
```

Bug: Path traversal (`../../etc/passwd`).
Fix: Canonicalize path, restrict to allowed dirs.

Q17. Weak password storage

```
hash = hashlib.sha1(password.encode()).hexdigest()
```

Bug: SHA1 weak.
Fix: Use `bcrypt` / `argon2`.

Q18. Logging sensitive data

```
logging.info("Password entered: %s", password)
```

Bug: Logs sensitive info.
Fix: Mask/omit sensitive fields.

Q19. CSRF missing

```
@app.route('/transfer', methods=['POST'])
def transfer():
    # no CSRF token
    ...
```

Bug: No CSRF protection.
Fix: Implement CSRF tokens (Flask-WTF / Django CSRF middleware).

Q20. XSS in templates

```
return f"<h1>{request.args['msg']}</h1>"
```

Bug: Reflected XSS.
Fix: Use autoescaping template engines (Jinja2 with autoescape).

C / C++ (10)

Q21. Buffer Overflow

```
char buf[10];
strcpy(buf, input);
```

Bug: No bounds check → buffer overflow.
Fix: Use `strncpy(buf, input, sizeof(buf)-1)`.

Q22. Format String Vulnerability

```
printf(user_input);
```

Bug: User controls format string.
Fix: `printf("%s", user_input);` .

Q23. Command Injection

```
char cmd[100];
sprintf(cmd, "ls %s", user_input);
system(cmd);
```

Bug: OS command injection.
Fix: Avoid `system()` . Use library calls (e.g., `opendir`).

Q24. Use-after-free

```
char *p = malloc(10);
free(p);
strcpy(p, "test");
```

Bug: Use-after-free.
Fix: Avoid using freed pointers, set `p=NULL` .

Q25. Double Free

```
char *p = malloc(10);
free(p);
free(p);
```

Bug: Double free → memory corruption.
Fix: Set pointer to NULL after `free()` .

Q26. Integer Overflow

```
int size = user_input * 1000;
char *buf = malloc(size);
```

Bug: Integer overflow may allocate smaller buffer.
Fix: Validate ranges before allocation.

Q27. Hardcoded cryptographic key

```
char key[] = "mysecretkey";
```

Bug: Hardcoded secret.
Fix: Load from secure storage / env vars.

Q28. Race Condition

```
if (access(file, W_OK) == 0) {
    fd = open(file, O_WRONLY);
}
```

Bug: TOCTOU race.
Fix: Open file directly with secure flags.

Q29. Insecure Random

```
int token = rand();
```

Bug: `rand()` is predictable.
Fix: Use `RAND_bytes()` from OpenSSL or `/dev/urandom`.

Q30. Path Traversal

```
char path[256];
sprintf(path, "/var/www/%s", user_input);
fopen(path, "r");
```

Bug: Path traversal (`../../etc/passwd`).
Fix: Validate filename, canonicalize, restrict to safe directory.