Nice — I turned your journal + the screenshot into a compact, viva-ready theory sheet. Each experiment gets a tiny “flashcard” you can memorise: aim, core theory, quick viva answers and one-line mitigation/commands where useful. Short, plain language — speak these aloud and you’ll be fine.

General tips before viva

- Answer clearly: say the aim, tools, one example vuln, and one fix.
- Use key words: validate/sanitize, prepared statements, HTTPS, HttpOnly, CSP, least privilege, SAST/DAST.
- If asked for an example, give a *defensive* example (prepared statement, escaping), not exploit code.

Experiment 1 — Study various cybersecurity standards

Aim: Learn laws, standards and how they guide secure development.

Key points

- Standards: ISO 27001 (info security), NIST, GDPR (privacy), PCI-DSS (payment).
- Purpose: define controls, risk management, compliance.
- Viva lines: “Standards help set minimum security controls and audit requirements.”
- When asked: name one control = encryption at rest & in transit; periodic audits.

Experiment 2 — SDLC Case Study (Secure SDLC)

Aim: Show how security fits into each SDLC phase.

Per phase — one sentence each

- Investigation: identify assets, compliance needs, stakeholders.
- Analysis: threat modelling, attack surface mapping.
- Logical design: define auth, data flow, trust boundaries.
- Physical design: servers, network segmentation, secure hosting.
- Implementation: secure coding, SAST, code reviews.
- Maintenance: patching, monitoring, incident response.

Viva one-liner: “SecSDLC integrates security controls early — shift left.”

Experiment 3 — Demonstrate SAST tools

Aim: Use static scanners to find code vulnerabilities.

Key points

- SAST = static code analysis (before running app). Tools: Bandit (Python), SpotBugs, Semgrep.
- Finds: insecure patterns (eval, insecure deserialization, hard-coded secrets).
- Viva: “Run SAST in CI to catch vulnerabilities before deployment.”
- Fix flow: run scanner → review findings → patch code → re-scan.

Experiment 4 — OWASP methodologies (pick 5)

Aim: Study OWASP Top 10 + implement mitigations.

Pick 5 quick examples & fixes

- Injection (SQLi) — use prepared statements / ORM, validate inputs.

- Broken Auth — use strong password storage, MFA, session controls.
- XSS — escape output, CSP, input sanitization.
- Insecure Components — keep libs updated, run dependency scanning.
- Security Misconfig — use secure defaults, remove debug endpoints.

Viva: “OWASP provides common web risks and mitigation guidance.”

ZAP mention: “ZAP is an open-source DAST scanner for runtime testing.”

Experiment 5 — DVWA OS Command Injection

Aim: Show command injection and how to prevent it.

Key points

- Vulnerability: unsanitized input used in shell commands.
- Risk: attacker can execute arbitrary OS commands.
- Fixes: avoid shell calls; if necessary, whitelist inputs, use safe APIs and escape inputs.
- Viva Qs quick:
    - How to detect: DAST scanners, suspicious input echoing.
    - Prevention: input validation + least privilege; never pass raw user input to shell.

Experiment 6 — Registration page data validation (Frontend + Backend)

Aim: Implement robust input validation and authentication basics.

Core theory

- Client-side (JS/HTML5) = immediate UX checks (required, pattern).
- Server-side = authoritative checks (regex, length, sanitization, encoding).
- Best practices: whitelist inputs, central validation library, limit lengths, show friendly errors.
- Viva answers:
    - Why both? Client = UX, Server = security.
    - Good practice = canonicalise input, then validate & sanitize.

Experiment 7 — Session management for web app

Aim: Implement secure sessions (tokens, timeouts, cookie flags).

Core theory

- Session lifecycle: create (login) → assign session ID → store server-side or use token (JWT) → send cookie/token → validate on each request → expire/terminate on logout or timeout.
- Session ID: long, random, unpredictable (use secure generator).
- Cookie flags: `HttpOnly`, `Secure`, `SameSite=Strict/Lax`.
- Token vs stateful sessions: JWT = stateless (careful with revocation), server sessions = stateful (easier revoke).
- Viva Qs:
    - How set/retrieve? Server sends `Set-Cookie`; browser sends Cookie header.
    - Alternatives: URL rewriting or hidden form fields (less secure, avoid cookies only if unavoidable).

Quick secure checklist: rotate ID on privilege change, short timeout, use HTTPS, store minimal session data.

Experiment 8 — Burp Suite proxy to test web applications

Aim: Intercept and analyze HTTP(S) traffic to identify issues.

Flash answers

1. **Primary function of Proxy tool:** intercept and inspect HTTP(S) requests and responses between browser and server for analysis and modification.

2. **Configure browser:** set browser proxy to Burp host (default `127.0.0.1:8080`) or use Burp's helper extension; disable browser's automatic proxy settings as needed.

3. **Proxy listener:** a Burp process that listens for incoming proxy connections (default `127.0.0.1:8080` for HTTP).

4. **Install Burp CA certificate:** to intercept HTTPS you must trust Burp's CA in browser so TLS won't break (Burp acts as MITM for analysis).

5. **Troubleshooting:** ensure browser proxy set correctly, Burp listener enabled, certificate installed, no conflicting proxy extensions, firewall allowing port 8080, and site not using certificate pinning.

Quick viva line: "Burp helps find insecure headers, cookies, input fields, and replay/modify attacks."

Supporting theory & short answers (common viva Qs)

XSS (Cross Site Scripting)

- Attacker injects script into pages users view. Prevent by encoding output, CSP, server-side sanitization.

SQL Injection

- Occurs when user input changes SQL logic. Fix with prepared statements and parameterized queries.

Authentication vs Authorization

- Auth = verify identity. Authz = decide permitted actions.

HTTP vs HTTPS

- HTTPS = HTTP over TLS (encrypted). Always use HTTPS in production.

API / REST API

- API: interface for apps. REST uses stateless HTTP verbs (GET/POST/PUT/DELETE) and resources (URLs).

SAST / DAST

- SAST = static code scanning (pre-run). DAST = dynamic testing (running app). Both complement each other.

Repository (Git)

- Repo stores code, commits, branches. Common commands: `git add`, `git commit -m`, `git push`, `git pull`.

How to commit changes (short)

- `git status` → `git add file` → `git commit -m "msg"` → `git push origin branch`.

MFA

- Multi-factor auth uses 2+ proofs (password + OTP or hardware). Significantly lowers account takeover risk.

Encryption / Decryption

- Encryption uses key to convert plain text into ciphertext; decryption reverses. Symmetric (AES) vs asymmetric (RSA).

SADL / Secure SDLC phases

- Requirements → Design → Implementation → Testing (SAST/DAST) → Deployment → Maintenance + monitoring.

How to secure an application (summary)

- Validate & sanitize inputs, use HTTPS, secure auth, use prepared statements, limit privileges, update dependencies, logging & monitoring, run SAST/DAST.

Short viva phrases you can memorise (one-liners)

- “Always validate on server; client-side only for user convenience.”
- “Use prepared statements to stop SQL injection.”
- “Set cookie flags HttpOnly and Secure to protect session cookies.”
- “Run SAST in CI and DAST before release.”
- “Burp Proxy listens on localhost:8080 and needs its CA cert for HTTPS.”
- “Session ID must be random, rotated on privilege change, and short-lived.”
- “Use HTTPS and certificates to protect data in transit.”
- “OWASP Top 10 lists the most common web risks and fixes.”

If you want I can:

- Convert this into a 1-page printable PDF (compact).
- Make a 2-minute viva script per experiment (what to say).
- Make flashcards (Q + one-line A) for quick memorization.

Which one next?

—----------

Viva questions asked

---

1. How to integrate a live API

- **What**: Connect your app to a remote service (endpoint) that returns data (JSON/XML).
- **Steps**: get endpoint URL + method (GET/POST), read docs, obtain API key if needed → call endpoint (e.g., `fetch` in JS or `requests` in Python) → handle response (parse JSON) → show/save data.
- **Extras**: handle errors, retries, rate limits, and CORS in browsers; secure API keys (don't store in client).

---

2. Cross-Site Scripting (XSS)

- **Definition**: Attacker injects malicious scripts into pages viewed by other users.
- **Types**: Stored (in DB), Reflected (in URL), DOM-based (client-side).
- **Prevention**: escape/encode output (`html.escape`), use Content Security Policy (CSP), validate/strip `<script>` on input, use frameworks' templating that auto-escapes.

---

3. Client and Server Validation — kaise hota hai

- **Client-side**: JavaScript checks before submit (UX). Example: required fields, format checks.
- **Server-side**: Mandatory security checks on server — never trust client. Example: regex/email, length limits, sanitization.

- **Why both**: client = good UX; server = security and data integrity.

---

4. Authentication — kaise karte hain

- **Definition**: Verify who the user is.
- **Common methods**: username/password, OAuth (Google/GitHub), tokens (JWT), API keys, sessions (cookie-based).
- **Flow (session)**: user logs in → server validates → server creates session id → sends cookie → cookie used on subsequent requests.

---

5. SQL Injection and OS Command Injection

- **SQL Injection**: attacker inserts SQL into inputs → modifies DB queries.
    - **Example**: `' OR '1'='1'`
    - **Fix**: prepared statements / parameterized queries, ORM, strong input validation, least DB privileges.
- **OS Command Injection**: attacker runs system commands via unsafe concatenation.
    - **Fix**: avoid shelling out, use safe APIs, validate/whitelist inputs, escape inputs.

---

6. How a session is established

- **Cookie session (classic)**:
    1. User logs in → server creates session record (id + data).
    2. Server sends `Set-Cookie: session=SESSIONID`.

3. Browser sends cookie on requests → server looks up session data.

- **Token-based (stateless)**: server returns JWT → client stores (localStorage or cookie) → sends in `Authorization: Bearer …`
- **Secure tips**: set `HttpOnly`, `Secure`, `SameSite` flags for cookies.

---

7. How to secure an application (quick checklist)

- Validate + sanitize inputs (server-side).
- Use HTTPS everywhere.
- Use prepared statements for DB.
- Escape output to prevent XSS.
- Apply secure auth (hash passwords with bcrypt).
- Least privilege for DB and services.
- Keep dependencies updated.
- Use CSP, rate limiting, logging & monitoring, and secure config (no secrets in code).
- Pen-test & code review (SAST/DAST).

---

8. What is privacy?

- **Definition**: Control over personal data — who collects it, how it's used, stored, and shared.
- **Practices**: minimize data collection, use encryption at rest/in transit, get consent, follow laws (e.g., GDPR basics).

---

9. What is SAST / DAST?

- **SAST (Static Application Security Testing)**: analyze source code or binaries for vulnerabilities before running app (e.g., linters, code scanners).
- **DAST (Dynamic Application Security Testing)** (you wrote DST — likely DAST): test running application to find runtime issues (e.g., scanning for SQLi, XSS).
- **Use both**: SAST finds coding errors early; DAST finds runtime issues and misconfigurations.

---

10. What is HTTP / HTTPS?

- **HTTP**: Hypertext Transfer Protocol — request/response protocol for web.
- **HTTPS**: HTTP over TLS — encrypted connection, prevents eavesdropping and tampering.
- **Note**: HTTPS uses certificates; always use HTTPS in production.

---

11. Authentication vs Authorization

- **Authentication**: “Who are you?” (login)
- **Authorization**: “What can you do?” (permissions/roles)
- Example: Alice logs in (auth), then allowed to access admin page only if she has admin role (authz).

---

12. What is a repository

- **Repo**: Storage for code + history (e.g., Git repository).

- **Contains**: commits, branches, tags, commit messages.
- **Purpose**: version control, collaboration, rollback.

---

13. MFA (Multi-Factor Authentication)

- **Definition**: Use 2+ methods to verify identity (something you know + have + are).
- **Examples**: password + OTP (SMS / authenticator app), password + hardware key.
- **Benefit**: mitigates stolen passwords.

---

14. Maintenance phase

- **Definition**: Post-deployment phase: bug fixes, updates, security patches, performance tuning, and feature enhancements.
- **Tasks**: monitoring, backups, incident response, dependency updates.

---

15. API (Application Programming Interface)

- **Definition**: Contract that lets programs talk to each other (endpoints, methods).
- **Formats**: REST (HTTP/JSON), GraphQL, SOAP.
- **Elements**: endpoint, method (GET/POST/PUT/DELETE), headers, body, status codes.

---

16. REST API

- **Principles**: stateless, resources (URLs), HTTP verbs (GET/POST/PUT/DELETE), use JSON typically.
- **Example**: `GET /users/123` returns user data.
- **Status codes**: 200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 404 Not Found.

---

17. PHP (short)

- **What**: Server-side scripting language often used for web apps.
- **Use**: handle form submissions, sessions, interact with DB.
- **Note**: secure PHP by using parameterized queries, proper escaping, and up-to-date PHP versions.

---

18. Git

- **What**: distributed version control system.
- **Key concepts**: clone, branch, commit, push, pull, merge.
- **Local vs remote**: local repo vs GitHub/GitLab as remote.

---

19. How to commit changes (basic commands)

1. `git status` — see changes
2. `git add file` or `git add .` — stage
3. `git commit -m "message"` — record snapshot

4. `git push origin branch` — push to remote

---

20. Can you make changes in a git repo?

- **Yes**: clone → create branch → make edits → `git add` → `git commit` → `git push` → create Pull Request (PR) → code review → merge.
- **Permissions**: you need push access or fork and PR.

---

21. What is encryption / decryption

- **Encryption**: convert plain data to unreadable form using a key.
- **Decryption**: reverse using key.
- **Types**:
    - Symmetric (same key): AES — fast, used for data at rest.
    - Asymmetric (public/private): RSA — used for key exchange & signatures.
- **Use-cases**: TLS uses both (asymmetric to exchange symmetric key).

---

22. SADL lifecycle phases (Secure Application Development Lifecycle)

- **1. Requirements & Threat Modeling**: identify assets, threats, security requirements.
- **2. Design**: design architecture with security controls (auth, encryption).
- **3. Implementation**: secure coding, SAST, code reviews.

- **4. Testing**: DAST, penetration tests, security test cases.
- **5. Deployment**: secure config, secrets management, use HTTPS.
- **6. Monitoring & Maintenance**: logging, patching, incident response.
- **(Extra)**: Training & governance across all phases.

Quick viva cheat-sheet (one-liners to memorize)

- **Integrate API**: call endpoint, parse JSON, secure keys.
- **XSS**: input → output; prevent by escaping output and CSP.
- **Client vs Server validation**: UX vs security — both required.
- **Auth**: verify identity; **Authz**: grant access.
- **SQLi**: fix with prepared statements.
- **Session**: cookie or token; secure cookie flags.
- **Secure app**: validate, escape, patch, HTTPS, least privilege.
- **SAST/DAST**: static vs dynamic testing.
- **HTTP/HTTPS**: plain vs encrypted.
- **Repo**: versioned code store.
- **MFA**: multiple proofs of identity.
- **Maintenance**: update, monitor, fix.
- **API/REST**: endpoints + HTTP verbs + status codes.
- **Git commit**: add → commit → push.
- **Encryption**: symmetric/asymmetric.