Auditing and Testing Web Applications

Web Hacking

Fall Term 2026/27

Emmanuel Benoist | BFH-TI

Table of Contents

- Introduction
- The WSTG as a Methodology
- Phase 1 – Reconnaissance
- Phase 2 – Systematic Testing
- Rating the Risk of a Finding
- Reporting
- Conclusion

Introduction

From Knowing Attacks to Running an Assessment

- So far we studied **single weaknesses**: access control, injection, crypto, configuration, …
- A real engagement asks a different question:
  - “Here is a system. **Is it secure enough?** Tell us what is wrong, how bad it is, and what to do first.”
- That needs a **method**, not a bag of tricks:
  1. **Reconnaissance** – know what you are testing
  2. **Systematic testing** – cover the whole attack surface
  3. **Risk rating** – turn findings into priorities
  4. **Reporting** – make the result actionable
- Our reference: the **OWASP Web Security Testing Guide (WSTG)** and the **OWASP Risk Rating Methodology**
- **Repeatability is the point**: two testers following the method should reach comparable conclusions

A Scan is not a Test

Engagement	What it answers	Typical limits
Vulnerability scan	“Do I have known, detectable flaws?”	No logic flaws, false positives, no chaining
Penetration test	“Can an attacker reach my data?”	Time-boxed, scope-bound, point in time
Security audit / review	“Do we meet a standard or policy?”	Compliance $\neq$ security
Red team	“Would we detect and respond?”	Expensive, not exhaustive
Bug bounty	“What do many eyes find over time?”	No coverage guarantee

- This chapter is about the **penetration test / application assessment**, the activity the WSTG describes
- All of them are **point-in-time** statements – say so in the report

How Much do we Know? Black, Grey, White Box

Mode	Information given	Consequence
Black box	URL only	Realistic for an outsider; recon eats the budget; blind spots stay blind
Grey box	Accounts, roles, docs, architecture	Best value for the money: authorization and logic get tested
White box	Sources, configuration, build pipeline	Finds causes, not only symptoms; needs code-review skills

- **Rule of thumb**: the attacker has *unlimited* time, you have five days – so **buy information with the budget you save**
- Always ask for **two accounts per role**: it is the only way to test horizontal access control (A01)
- Ask for a **non-production instance** when tests may be destructive

No Single Technique is Sufficient (WSTG)

Technique	Strength	Weakness
Manual inspection, design review	Policies, architecture, process	Slow, needs senior people
Threat modeling	Finds design flaws *early*	Expertise-dependent; misses new threats
Source code review	Subtle logic, crypto, race conditions	Needs access and deep knowledge
Penetration testing	Proves *exploitability* and impact	Late, costly to fix what it finds
Automation (SAST/DAST/SCA)	Known patterns, at scale, cheap to repeat	No understanding of business context

- A tool cannot replace “the reasoning and creativity of an experienced security tester” (WSTG)
- **Combine** techniques, and combine **people, process and technology** – a finding that nobody fixes was not worth finding

The OWASP Testing Framework: Test Across the Lifecycle

Phase	Activities
1. Before development	SDLC with security built in; policies and secure coding standards; metrics – “people can only do the right thing if they know what the right thing is”
2. Definition & design	Review security requirements; review architecture; **threat modeling** – the cheapest place to find flaws
3. Development	Code walkthroughs; static review against checklists (Top 10, language rules, regulation)
4. Deployment	**Application penetration test**; configuration management review
5. Maintenance & operations	Operational reviews; periodic health checks (monthly/quarterly); **change verification**

- The pen test in phase 4 is **one checkpoint**, not the whole programme
- Waiting for a finished build to start testing is the expensive way

Before You Touch Anything: Rules of Engagement

- **Written authorization** – who may test, what, when, from where. No authorization, no test.
  - In Switzerland, unauthorized access to a data processing system is a criminal offence (Art. 143^{bis} StGB/CP); a signed scope is your only protection
- **Scope**: exact domains, IP ranges, APIs, mobile clients, accounts; **and what is out of scope** (third-party SaaS, payment provider, …)
- **Limits**: no denial of service, no spam to real users, no social engineering unless explicitly agreed
- **Data handling**: what you may extract, where evidence is stored, encryption, deletion date
- **Emergency contacts** and a stop condition: whom to call when you break production or find an *active* intrusion
- **Timing window** and source IPs, so the SOC does not chase you (or so that it does – if detection is part of the test)
- Record all of this: it becomes the *Scope* and *Limitations* chapters of the report

The WSTG as a Methodology

The OWASP Web Security Testing Guide

- The reference **methodology** for web application testing, free and community-maintained
- Structure
  1. Foreword / About
  2. Introduction (techniques, principles)
  3. The OWASP Testing Framework (SDLC)
  4. **Web Application Security Testing** – the tests themselves
  5. **Reporting**
  6. Appendices (tools, reading, fuzz vectors, checklist)
- **Stable version: v4.2**; the *latest* version adds a WebAssembly category
- `https://wstg.owasp.org/latest/`

- Each test has a **stable ID**: `WSTG-<cat>-<nn>`
- Preferred in reports: `WSTG-v42-ATHN-01` (the version pins the meaning)
- Use the IDs to show **coverage**: what you tested, not only what you found

The 13 Testing Categories (Chapter 4)

§	v4.2 ID	Category	Tests
4.1	INFO	Information Gathering	10
4.2	CONF	Configuration and Deployment Mgmt	15
4.3	IDNT	Identity Management	5
4.4	ATHN	Authentication	11
4.5	ATHZ	Authorization (incl. OAuth)	5+
4.6	SESS	Session Management	11
4.7	INPV	Injection	23
4.8	ERRH	Error Handling	2
4.9	CRYP	Weak Cryptography	4
4.10	BUSL	Business Logic	10
4.11	CLNT	Client-side	15
4.12	APIT	API Testing	5
4.13	–	WebAssembly (new, *latest* only)	2

- Around **120 scenarios**: in five days you **prioritise** by attack surface and by business risk

Anatomy of a WSTG Test

Every scenario has the same shape:

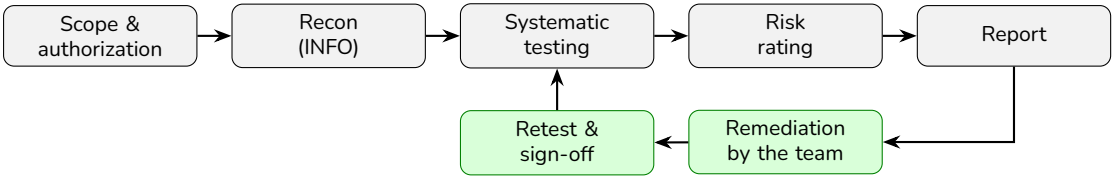
- **ID** and title
- **Summary** – what the weakness is
- **Test Objectives** – what you must establish
- **How to Test** – concrete procedure, requests, tools
- **Remediation** – what the fix looks like
- **References** / tools

Example: WSTG-v42-ATHN-01
Credentials Transported over an Encrypted Channel

- **Objective**: no credential or session token ever travels in clear
- **How to test**: log in, create an account, reset a password, all through a proxy; watch for `http://` on login forms, form actions, redirects and cookies without `Secure`
- **Remediation**: HTTPS everywhere plus HSTS

- The *Test Objectives* are what you copy into your checklist – they are written as verifiable statements

The Engagement in One Picture



- Testing is **iterative**: every finding feeds new recon (a new parameter, a new role, a new host)
- The loop only closes with a **retest**: an unverified fix is a hope, not a control
- Keep **notes and evidence as you go** – reconstructing them at the end costs twice as much

Phase 1 – Reconnaissance

Why Recon Decides the Quality of the Test

- **You cannot test what you have not found.** Most missed vulnerabilities sit on pages, parameters or hosts the tester never saw.
- Goals of the phase
  - Inventory of **hosts, endpoints, parameters, roles, technologies**
  - Understanding of the **business** the application runs
  - A **prioritised** list: where would *you* attack?
- **Passive** recon (no packet to the target: search engines, archives, certificate logs, code repositories) vs **active** recon (requests to the target: fingerprinting, crawling, content discovery)
- Passive first: it is free, silent, and legal even before the window opens
- Output is a **document**, not a feeling – the attack-surface inventory drives the test plan

WSTG 4.1 – Information Gathering (WSTG-INFO)

ID	Test
INFO-01	Search engine reconnaissance for information leakage
INFO-02	Fingerprint web server
INFO-03	Review webserver metafiles for information leakage
INFO-04	Attack surface identification (enumerate applications)
INFO-05	Review web page content for information leakage
INFO-06	Identify application entry points
INFO-07	Map execution paths through the application
INFO-08	Fingerprint web application framework
INFO-09	Fingerprint web application
INFO-10	Map application architecture

- Ten tests, and the last three are the ones people skip – and then test the wrong application

INFO-01/05 – What the Internet Already Knows

- **Search engines and archives**: indexed admin pages, backups, directory listings, error pages with stack traces

```
site:example.com inurl:admin | inurl:login | inurl:backup
site:example.com ext:sql | ext:log | ext:bak | ext:old | ext:zip
site:example.com intitle:"index of" | intext:"Fatal error"
# archives and certificate transparency reveal forgotten hosts
web.archive.org/web/*/example.com/*
crt.sh/?q=%25.example.com
```

- **Code and paste sites**: a developer's repository, a CI log or a Stack Overflow question with a real connection string
- **The pages themselves** (INFO-05): HTML comments, disabled form fields, source maps, `/*.js.map`, API keys in bundles, author names for the username scheme
- **Ask for removal** of what should not be public – a cached backup is a finding in its own right

INFO-03 – Metafiles and Well-Known Locations

```
curl -s https://example.com/robots.txt
curl -s https://example.com/sitemap.xml
curl -s https://example.com/.well-known/security.txt
curl -s https://example.com/crossdomain.xml
curl -s https://example.com/.git/HEAD        # exposed repository?
curl -s https://example.com/.env             # exposed secrets?
```

- `robots.txt` is a **map of what the owner wants hidden**: every `Disallow` is a hint, never a control
- `sitemap.xml` gives you URLs the crawler would not reach
- `security.txt` tells you *how to report* – and whether the organisation is prepared for findings at all
- `.git/`, `.env`, `.DS_Store`, `*.bak`, `#file#` and editor backups are configuration findings (CONF) that often hand you the source code

INFO-02/08/09 – Fingerprinting the Stack

```
HTTP/1.1 200 OK
Server: Apache/2.4.58 (Debian)
X-Powered-By: PHP/8.2.4
Set-Cookie: PHPSESSID=...; path=/
X-Generator: Drupal 10 (https://www.drupal.org)
```

- **Signals**: response headers, cookie names (`PHPSESSID`, `JSESSIONID`, `csrftoken`), error page style, favicon hash, default files (`/wp-login.php`), JS bundle names, HTTP method behaviour, TLS certificate and its SANs
- **Tools**: `curl -I`, `whatweb`, `nmap -sV`, Wappalyzer, `testssl.sh` for the TLS side
- Why it matters: the stack tells you **which attacks are plausible** and which **known CVEs** to check (chapter 3, supply chain)
- Record versions **with evidence** – a banner is a claim, a working exploit is proof

INFO-04/06/07 – Entry Points and Execution Paths

- **Entry points** (INFO-06): every place input enters – URL parameters, form fields, JSON bodies, headers, cookies, file uploads, WebSocket messages, webhooks
- **Method**: browse the application *through a proxy* (Burp / ZAP) with the browser doing the work, then read the proxy history: it *is* your entry-point list
- **Content discovery** for what the crawler misses:

```
ffuf -u https://example.com/FUZZ -w /usr/share/wordlists/dirb/common.txt \
     -mc 200,204,301,302,401,403 -ac
# per-parameter discovery once you know a page
ffuf -u 'https://example.com/report?FUZZ=1' -w params.txt -fs 4242
```

- **Execution paths** (INFO-07): walk the *workflows* – register, verify mail, log in, change password, order, pay, refund, delete account
- Each state transition is a place where authorization and logic can fail

INFO-10 – Map the Architecture

- Who is actually answering? **CDN / WAF, reverse proxy, load balancer, application server, API gateway, database, object storage, third-party services**
- Why you care
  - A **WAF** changes how you test (encode, fragment, slow down) and what your findings mean
  - A **reverse proxy** invites path confusion and header smuggling
  - **Cloud metadata** and internal services turn an SSRF into a takeover (chapter 2, Capital One)
- Cheap probes: `dig`, `nslookup`, response-header differences between endpoints, timing, error pages from different components, behaviour of unusual methods and of `Host:` / `X-Forwarded-For` headers
- Ask the client for the real diagram – then **verify it**, because diagrams age badly

Deliverable of the Phase: the Attack Surface Inventory

Endpoint	Method	Params	Auth / role	First guesses
/login	POST	user, pass	none	ATHN, lockout, SESS
/invoice	GET	id	customer	**ATHZ (IDOR)**, INPV
/admin/users	GET	q	admin	ATHZ vertical, INPV
/api/v1/orders	GET POST	filter, body	token	APIT, mass assignment
/upload	POST	file	customer	CONF, file execution
/report	GET	from, to, fmt	customer	INPV (SQL), BUSL

- One row per endpoint, with the **WSTG categories to apply** – this table *is* your test plan and later your coverage evidence
- Add: technologies and versions, roles and accounts, data classification of what each endpoint touches
- Keep it in the report appendix: it shows **what you looked at**

Phase 2 – Systematic Testing

From Inventory to Checklist

- **Work the matrix**: endpoints × relevant WSTG categories, each cell marked *not applicable / passed / finding / blocked*
- **Prioritise** – with limited days, start where impact lives:
  1. Authentication, session, authorization (who can do what)
  2. Anything touching money, personal data or files
  3. Injection on every reachable parameter
  4. Configuration and error handling (cheap, fast, often fruitful)
  5. Business logic (slow, needs understanding – but nobody else finds it)
- **Time-box** each area and keep a running “to revisit” list
- Track **coverage, not activity**: “ATHN 1–11 executed, ATHN-10 blocked (no SMS channel in test env)” is a result; “tested authentication” is not
- The WSTG checklist (appendix of the guide) is a ready-made spreadsheet – the filled copy goes into the report

The Proxy-Centred Workflow

1. **Browse** the feature like a user, through the proxy
2. **Read** the request: parameters, tokens, headers, cookies
3. **Form a hypothesis**: "`id` is a database key and nobody checks the owner"
4. **Modify and replay** (repeater): one variable at a time
5. **Observe**: status, length, timing, error text, side effects
6. **Confirm** twice, from a clean session, and **capture evidence**

- Keep **two browsers / profiles** open: *victim* and *attacker* sessions
- Scope the proxy to the target, or your history fills with noise
- Name your repeater tabs after the finding – they become the reproduction steps
- Log **timestamps**: the client will ask what you did at 14:37

Automated Scanning: Useful, Insufficient

What tools do well

- Known CVEs and outdated components (SCA)
- Missing headers, weak cookie flags, TLS configuration
- Reflected XSS, obvious SQLi, directory listings
- Breadth: thousands of requests while you read code
- Regression checks in CI

What they essentially never find

- **Authorization** flaws – the tool does not know who *should* see the invoice
- **Business logic** and workflow abuse
- Multi-step and chained attacks
- Design flaws, missing controls
- Context: which data is actually sensitive

- Run the scanner **and** verify every hit by hand: an unverified scanner line in a report destroys your credibility
- Never paste raw tool output as findings – clean, deduplicated, **confirmed** results only (an appendix may hold the cleaned output)

Testing CONF, IDNT, ATHN, ATHZ in Practice

Cat.	Concrete checks
CONF	Allowed HTTP methods; admin interfaces reachable from the Internet; default credentials; old versions side by side (`/v1`, `/beta`); backup and source files; directory listing; file-upload handling; security headers
IDNT	Can you enumerate users (registration, login, reset, timing)? Role definitions; self-registration into privileged roles; account provisioning and de-provisioning
ATHN	Credentials over TLS only; lockout vs brute force; password policy; password reset tokens (guessable, reusable, long-lived?); “remember me”; cache of authenticated pages; MFA bypass through an alternative channel
ATHZ	Directory traversal; **IDOR with the second account**; privilege escalation by changing a role field; forced browsing to admin URLs; OAuth scope and redirect-URI handling

- **The second account is your most productive tool**: replay A’s requests with B’s session

Testing SESS, INPV, ERRH, CRYP in Practice

Cat.	Concrete checks
SESS	Cookie attributes (`Secure`, `HttpOnly`, `SameSite`, scope, lifetime); token entropy and predictability; is the session *new* after login (fixation)?; does logout invalidate server-side?; concurrent sessions; CSRF tokens bound to the session
INPV	Every parameter, every sink: SQL, NoSQL, command, LDAP, XPath, template, header and log injection; XSS reflected / stored / DOM; SSRF; file inclusion; HTTP request smuggling; test with *one* marker payload first, then escalate
ERRH	Stack traces, SQL errors, debug pages, framework error templates; differences between 401/403/404 that leak existence
CRYP	TLS version, ciphers, certificate chain, HSTS; sensitive data sent or stored in clear; weak hashes for passwords; padding oracles; keys in repositories or configuration

Business Logic (BUSL): Where Humans Beat Tools

- **Nothing is malformed** – every request is valid; the *sequence* or the *quantity* is the attack
- Classic abuse cases
  - **Skip a step**: jump from basket to confirmation without payment
  - **Replay**: use a voucher, a refund or a one-time token twice
  - **Negative and extreme values**: quantity `-1` credits the account; `0.001` rounds to free
  - **Race conditions**: two parallel withdrawals on one balance
  - **Time**: order in the past, change the price between display and payment, use an expired discount
  - **Capacity abuse**: reserve thousands of seats without deposit (Ao6 example)
- **Method**: write the *intended* workflow down, then write the **rule** each step enforces, then try to break that rule
- Ask “what would a dishonest *customer* do?” – not “what would a hacker do?”

Client-side (CLNT) and API Testing (APIT)

Client-side

- DOM XSS: sources (`location`, `postMessage`) to sinks (`innerHTML`, `eval`)
- `postMessage` origin checks
- CORS: `Access-Control-Allow-Origin` reflected plus credentials
- Clickjacking, open redirect, tabnabbing
- Local/session storage holding tokens
- Third-party scripts (A03) and CSP strength

APIs

- Get the **schema** (OpenAPI/GraphQL introspection) – it is the perfect endpoint list
- Object-level authorization per *id* (the API version of IDOR)
- **Mass assignment**: send `"role":"admin"` and see
- Missing rate limits, no pagination caps
- Verb tampering, undocumented versions
- GraphQL: depth/complexity abuse, batching

- APIs often have **more** endpoints than the UI shows – the UI is only one client

Testing Without Breaking the Client

- **Use test data** you created yourself; never exfiltrate real personal data “to prove it works” – a screenshot of *your own* record and a count are enough
- **Mask** what you do capture (names, card numbers, tokens) in notes and in the report
- **Destructive tests** (deletion, mass e-mail, bulk import, fuzzing a payment flow) only on agreement, preferably on a staging copy
- **No stress/DoS** unless explicitly in scope; watch your own concurrency on fragile targets
- **Stop and call** when you reach production data you should not see, or find traces of a *real* attacker
- Keep an **action log** (time, request, effect) – it protects you if something breaks for an unrelated reason

Rating the Risk of a Finding

Why Rate, and Why a Method

- A list of 40 findings without priorities is **useless** to a team that can fix five things this month
- Ratings must be **defensible and repeatable**: the developer will argue, and “it feels high to me” loses that argument
- **OWASP Risk Rating Methodology**

$$\textbf{Risk} = \textbf{Likelihood} \times \textbf{Impact}$$

- Six steps
  1. Identify the risk
  2. Estimate **likelihood** (threat agent + vulnerability factors)
  3. Estimate **impact** (technical + business factors)
  4. Determine the **severity**
  5. Decide **what to fix**
  6. **Customize** the model for the organisation
- The scores are a **structured conversation**, not a measurement: their value is that every factor is written down

Step 1 – Identify the Risk

- A finding is not “XSS on /search”. A **risk** is a sentence with four parts:
  - **Threat agent** – who? (anonymous Internet user, authenticated customer, insider, partner, automated bot)
  - **Attack** – how? (reflected payload in a link sent by mail)
  - **Weakness** – what is missing? (no output encoding, no CSP)
  - **Impact** – so what? (session theft of a logged-in customer, access to their invoices)
- **Use the worst-case scenario** that is realistic for this application – and write down which one you assumed
- The same technical bug gets **different ratings in different applications**: a stored XSS in a public guestbook is not a stored XSS in an online bank
- If you cannot fill the four parts, you do not yet understand your own finding

Step 2 – Likelihood: Threat Agent Factors

Factor	Options (value)
Skill level	no technical skills (1), some technical skills (3), advanced computer user (5), network and programming skills (6), security penetration skills (9)
Motive	low or no reward (1), possible reward (4), high reward (9)
Opportunity	full access or expensive resources required (0), special access or resources required (4), some access or resources required (7), no access or resources required (9)
Size	developers (2), system administrators (2), intranet users (4), partners (5), authenticated users (6), anonymous Internet users (9)

- Read them as: **how big and how motivated is the group that could do this, and how easy is it for them to get into position?**
- Pick the *group that matters*, not the most dangerous imaginable one

Step 2 – Likelihood: Vulnerability Factors

Factor	Options (value)
Ease of discovery	practically impossible (1), difficult (3), easy (7), automated tools available (9)
Ease of exploit	theoretical (1), difficult (3), easy (5), automated tools available (9)
Awareness	unknown (1), hidden (4), obvious (6), public knowledge (9)
Intrusion detection	active detection in application (1), logged and reviewed (3), logged without review (8), not logged (9)

- **Intrusion detection** is why chapter A09 matters here: a weakness nobody would notice being exploited is *more* likely to be exploited
- **Likelihood** = average of these eight numbers (four agent, four vulnerability); report the two group averages too, they explain the result
- Each value is a **judgement you must justify in one line** – that line is worth more than the number

Step 3 – Impact: Technical Factors

Factor	Options (value)
Loss of confidentiality	minimal non-sensitive data disclosed (2), minimal critical data disclosed (6), extensive non-sensitive data disclosed (6), extensive critical data disclosed (7), all data disclosed (9)
Loss of integrity	minimal slightly corrupt data (1), minimal seriously corrupt data (3), extensive slightly corrupt data (5), extensive seriously corrupt data (7), all data totally corrupt (9)
Loss of availability	minimal secondary services interrupted (1), minimal primary services interrupted (5), extensive secondary services interrupted (5), extensive primary services interrupted (7), all services completely lost (9)
Loss of accountability	fully traceable (1), possibly traceable (7), completely anonymous (9)

- Technical impact is what **you** can observe as a tester

Step 3 – Impact: Business Factors

Factor	Options (value)
Financial damage	less than the cost to fix (1), minor effect on annual profit (3), significant effect on annual profit (7), bankruptcy (9)
Reputation damage	minimal damage (1), loss of major accounts (4), loss of goodwill (5), brand damage (9)
Non-compliance	minor violation (2), clear violation (5), high profile violation (7)
Privacy violation	one individual (3), hundreds of people (5), thousands of people (7), millions of people (9)

- **Business impact wins the argument** – but you usually cannot fill this table alone: **ask the client** (data classification, user counts, regulations, contracts)
- When you must guess, **say that you guessed** and give the assumption; let the business correct it
- In Europe/Switzerland, remember GDPR / revDSG for the last two rows

Step 4 – From Numbers to a Severity

Scale for each average

0 to <3	LOW
3 to <6	MEDIUM
6 to 9	HIGH

- Average the likelihood factors, average the impact factors, map each to LOW/MEDIUM/HIGH

Overall severity

	Impact		
Likelihood	LOW	MEDIUM	HIGH
LOW	Note	Low	Medium
MEDIUM	Low	Medium	High
HIGH	Medium	High	**Critical**

- **Note** = minimal, informational
- **Critical** = drop everything
- Which impact? Use the **business** average when you have the data, the technical one otherwise – and state your choice

Worked Example A – IDOR on `/invoice?id=`

Risk: any authenticated customer changes `id` and downloads every other customer's invoices (names, addresses, amounts). Not logged.

Likelihood	
Skill level	3
Motive	4
Opportunity	7
Size (authenticated users)	6
Ease of discovery	7
Ease of exploit	5
Awareness	6
Intrusion detection	8
Average	**5.75**

⇒ **MEDIUM** (agent 5.0, vuln 6.5)

Impact	
Confidentiality	7
Integrity	1
Availability	1
Accountability	1
Financial damage	3
Reputation damage	5
Non-compliance	5
Privacy violation	7
Technical avg	2.5 (LOW)
Business avg	5.0 (MEDIUM)

Severity: MEDIUM × MEDIUM ⇒ **Medium**; with the worst-case technical impact (confidentiality alone, 7 = HIGH) ⇒ **High**

Worked Example B – Stack Trace on Every Error

Risk: any anonymous visitor triggers an unhandled error and reads the framework version, file paths and an SQL fragment.

Likelihood	
Skill level	3
Motive	1
Opportunity (none needed)	9
Size (anonymous)	9
Ease of discovery (tools)	9
Ease of exploit	5
Awareness (public)	9
Intrusion detection	9
Average	**6.75**

⇒ **HIGH**

Impact	
Confidentiality	2
Integrity	1
Availability	1
Accountability	1
Financial damage	1
Reputation damage	1
Non-compliance	2
Privacy violation	3
Technical avg	1.25 (LOW)
Business avg	1.75 (LOW)

Severity: HIGH × LOW ⇒ **Medium** by the matrix. Most teams would report *Low*: trivially exploitable, but nearly harmless on its own. Its real value is as a **step in a chain** – say so instead of inflating it.

Reading the Numbers Honestly

- **Averaging hides outliers.** In example A, one serious confidentiality loss (7) plus three harmless 1s became a technical LOW (2.5).
  - Fixes: report the **worst factor** next to the average, use the business table, or **weight** the factors (Step 6)
- **Likelihood is dominated by "is it easy and is nobody watching"**, which is why trivial information leaks score high – and why the matrix is deliberately conservative
- **Consistency inside a report beats absolute truth**: rate all findings with the same assumptions, so the *order* is right
- **Never silently override the model.** If you raise a Medium to a High because the endpoint is the payment flow, write that sentence down
- Give the **factor table** in the appendix for at least the top findings – it is what makes a rating reviewable

Steps 5 and 6 – Deciding and Customizing

- **Step 5: decide what to fix**
  - Most severe first – but **not all risks are worth fixing**: spending 100 000 to avoid a 2 000 annual loss is bad engineering
  - Also consider: cost of the fix, availability of a workaround (WAF rule, disabled feature), regulatory duty, whether one change closes several findings
  - Output: **owner + priority + deadline** per finding, agreed with the client, and accepted risks *documented* with a sign-off
- **Step 6: customize the model**
  - **Add factors** that matter in your domain (e.g. “loss of human life” for medical or military systems, safety, contractual penalties)
  - **Adapt the options** to the organisation’s own data classification and vocabulary
  - **Weight** the factors (weighted average) so business-critical aspects dominate
  - Then **freeze and publish** the model: the appendix of every report uses the same definitions

OWASP Risk Rating vs CVSS

	OWASP Risk Rating	**CVSS**
Designed for	*Your* application in *its* context	A product vulnerability, comparable worldwide
Inputs	Threat agent, business impact	Technical attack vector and impact (base), environment (env. metrics)
Strength	Business language, transparent, customizable	Standard, tool-supported, expected by many clients
Weakness	Subjective, not comparable across companies	Base score ignores context; often reported alone

- In practice: a **CVSS vector** for comparability *plus* an OWASP-style rating for prioritisation – the WSTG allows either, and many clients ask for CVSS explicitly
- Whatever you use, **define the levels in the appendix** and apply them uniformly

Reporting

The Report is the Product

- “The quality of the technical testing is completely irrelevant if the client can’t understand your findings.” (WSTG)
- **Three audiences, one document**
  - **Management**: what is the business exposure, what do we decide?
  - **Engineers**: what exactly is broken, how do I reproduce and fix it?
  - **Auditors / future testers**: what was in scope, what was tested, with which method?
- Consequences
  - Separate **executive summary** (no jargon) from **findings details** (full technical depth)
  - Every finding **reproducible by a developer** without you
  - Document **coverage and limitations** – silence about what you did *not* test is a lie by omission

WSTG Report Structure (1): Front Matter

- **Version control** – a table: version, description, date, author (reports get re-issued after a retest)
- **Table of contents**
- **The team** – who tested, with which qualifications
- **Scope** – “the boundaries and the needs of the engagement agreed upon with the organization”
- **Limitations** – out-of-bounds areas, broken functionality, missing credentials, lack of cooperation, time constraints
- **Timeline** – the test window
- **Disclaimer** – legally reviewed: a *point-in-time* assessment, no guarantee that everything was found

This part is boring and it is **what protects you** when someone asks six months later why a vulnerability was missed.

WSTG Report Structure (2): Content

- **Executive summary** – business need, what the test gave the organisation, key findings in business terms (compliance, reputation), strategic recommendations; figures only if they explain better than words
- **Findings summary** – one table: reference ID, title, risk level
- **Findings details** – one entry per finding (next slide)
- **Human factors** – phishing exposure in UI texts, confusing error messages, weak recovery UX, 2FA fallbacks, social-engineering angles; with recommendations mixing **technical controls, UX and training**
- **Reproducible test artifacts** – `curl` commands, proof-of-concept forms, HAR files, Selenium/Puppeteer scripts, redacted screenshots
- **Appendices** – test methodology, **severity definitions**, *cleaned* tool output (never raw dumps), and the **filled WSTG checklist** documenting every test conducted

Anatomy of a Finding

- **Reference ID** – so both sides can cite it (`WH-2026-004`)
- **Title** – the weakness, not the tool (“Any customer can download other customers’ invoices”)
- **Likelihood / exploitability** – ease, available exploit code, required access, attacker motivation
- **Impact** on the system and on the business
- **Risk rating** – Informational / Low / Medium / High / Critical, with the factor table or CVSS vector behind it
- **Affected assets** – URLs, parameters, versions, roles
- **Description** – the weakness, how it is exploited, what damage results; sensitive data masked
- **Reproduction** – numbered steps, requests, evidence
- **Remediation** – the concrete fix, plus the missing practice behind it (and references: CWE, WSTG ID, cheat sheet)

Example Finding (Condensed)

WH-2026-004 – Any customer can download other customers' invoices

Risk	**High** (likelihood MEDIUM 5.75, business impact MEDIUM 5.0, worst-case confidentiality HIGH)
WSTG	WSTG-v42-ATHZ-04 (IDOR) CWE-639 OWASP A01:2025
Asset	`GET /invoice?id=<n>` – role *customer*, production

Description. The invoice endpoint reads the `id` parameter and returns the matching record without checking that it belongs to the session's customer. Any authenticated customer can enumerate `id` from `1` and retrieve all invoices of all customers (name, address, items, amount). Access is not logged, so the extraction leaves no trace.

Reproduction. (account `test-a`, evidence `ev/004-*.png`)

```
# 1. log in as test-a and note the own invoice id (4711)
curl -s -b jar 'https://app.example.com/invoice?id=4711' -o own.pdf
# 2. request a neighbour's id with the SAME session
curl -s -b jar 'https://app.example.com/invoice?id=4710' -o other.pdf
# -> HTTP 200, invoice of customer "B. Muster" (masked in evidence)
```

Remediation. Resolve the record *through* the session owner (`WHERE id=? AND customer_id=?`), return 404 on mismatch, and log denied accesses. Add an authorization test per role to the test suite; review all endpoints taking an object identifier.

Writing the Executive Summary

Do

- Answer: *how exposed are we, and what should we decide?*
- Name the **business consequence**: customer data, fines, downtime, fraud
- Give **3–5 strategic** recommendations (“authorization tests in CI”, “retire the legacy admin console”)
- State the **scope and the window** in two lines

Avoid

- Jargon, payloads, tool names, HTTP headers
- A colour chart as the only content
- Blaming individuals
- “The application is secure” – you tested a point in time, with limits
- Padding with pages of scanner statistics

- One page. If the CEO reads only this page, the right decisions must still be possible.

Evidence, Delivery and Retest

- **Evidence**: request/response pairs, `curl` one-liners, HAR files, short screen recordings, redacted screenshots – enough for a developer to reproduce *without* the tester
- **Handle the report like the data it describes**: it is a map of how to break the system – encrypt it, deliver it over an agreed channel, agree a retention period, and never leave it in a ticket attachment forever
- **Walk through it** with the team: a one-hour debrief fixes more than a hundred pages sent by mail
- **Retest** after remediation
  - Verify each finding by its reference ID: *fixed / partially fixed / not fixed / risk accepted*
  - Check for **regressions and variants** – the same bug often exists on three other endpoints
  - Re-issue the report with a new version number and a remediation status column

Conclusion

A Five-Day Engagement, Concretely

Day	Work
0	Scope, authorization, accounts (two per role), contacts, test data, environment confirmed
1	Passive and active recon (INFO), attack-surface inventory, first automated baseline (CONF, CRYP, headers, components)
2	Identity, authentication, session (IDNT, ATHN, SESS) – plus every reset and alternative channel
3	Authorization with two accounts (ATHZ), injection on all entry points (INPV), error handling (ERRH)
4	Business logic and APIs (BUSL, APIT, CLNT), chaining of findings, filling coverage gaps, retrying the blocked tests
5	Risk rating of all findings, report writing, debrief with the team; agree retest date

- Reserve **a full day for rating and writing** – it is the deliverable, not the leftover

Conclusion

- **Method over tricks**: scope → recon → systematic testing → rating → report → retest. The WSTG gives you the catalogue and the IDs to prove coverage.
- **Recon decides the result**: untested surface is undefended surface
- **No single technique suffices** – automation for breadth, humans for authorization and business logic
- **Risk = Likelihood** × **Impact**, with the factors written down: transparent, reviewable, and arguable with the business
- **Averages lie**; keep the worst factor visible, and say when you overrule the model
- **The report is the product**: one page for decisions, full depth for engineers, honest limitations, verified fixes
- **Legality and care first**: written authorization, agreed limits, masked data, no surprises

References

- OWASP Web Security Testing Guide `https://wstg.owasp.org/latest/`
- WSTG – The Web Security Testing Framework
  `https://wstg.owasp.org/latest/3-The_OWASP_Testing_Framework/`
- WSTG – Web Application Security Testing (the 13 categories)
  `https://wstg.owasp.org/latest/4-Web_Application_Security_Testing/`
- WSTG – Reporting
  `https://wstg.owasp.org/latest/5-Reporting/01-Reporting_Structure`
- OWASP Risk Rating Methodology
  `https://community.owasp.org/OWASP_Risk_Rating_Methodology`
- OWASP ASVS (verification requirements), OWASP Cheat Sheet Series
- CVSS v4.0 specification `https://www.first.org/cvss/`
- PTES (Penetration Testing Execution Standard), NIST SP 800-115