OWASP Top 10:2025 in Code: Ten Flaws, Their Attacks, and Tested Fixes

Bogdan Itsam Dorantes-Nikolaev and Hasan Coskun
Faculty of Computer Science and Engineering (FINKI)
Ss. Cyril and Methodius University in Skopje (UKIM)
Skopje, North Macedonia

***Abstract*—The Top 10:2025 of the Open Worldwide Application Security Project (OWASP) lists the most critical security risks to web applications. We show each of its ten categories as a short Python program with a realistic flaw, an attack that exploits it, and a fixed version of the same code. Each of the 13 attack scenarios is checked three ways: the attack must succeed against the vulnerable code, the same attack must fail against the fix, and normal use must still work with the fix. All pass, and 38 tests keep them so. A faceted classification of the ten demos, by where the flaw enters, what goes wrong, what is lost and how it is found, shows that six of the ten enter purely as code, that no demo breaks availability, and that seven kinds of mistake occur, none dominant; six kinds of repair cover all ten fixes. Every fix still leaves something open, and we state what. Applying OWASP's published ranking formula to its score tables reproduces the lowest score OWASP states but not the highest, and OWASP's pages contradict each other or themselves in ten places. The same code drives an interactive lab that runs Python in the browser and refuses nine plausible half-fixes.**

***Index Terms*—Common Weakness Enumeration (CWE), OWASP Top 10, secure coding, security education, software vulnerabilities, web application security**

I. Introduction

The OWASP Top 10 is the awareness document of the Open Worldwide Application Security Project (OWASP) [1] on the most critical security risks to web applications [2]. Its 2025 edition, the eighth installment, is the current one [3]. Each of its ten categories names a kind of weakness that ends up in a program, so we show each one as a program: for every category, a short Python module with a realistic flaw, an attack function that exploits it, and a fixed version of the same code. The text explains the code; the code and its output are the evidence.

We set out to answer three research questions:

- **RQ1:** Can each of the ten 2025 categories be shown as a flaw in code, with an attack that works against it and a fix that the same attack fails against, without breaking the feature?
- **RQ2:** What do the ten have in common as programming problems: where they enter, what goes wrong, what is lost, and how they are found?
- **RQ3:** What do the fixes still not do?

Prepared for Prof. Hristina Mihajloska, UKIM FINKI, Skopje, 26 November 2026. The interactive web version of this report, with the same code, is at https://seclab.bogdan.tr/report.html.

Our contributions are:

- ten demonstrations, one for each category, with 13 attack scenarios, each run against a vulnerable and a fixed version and checked in three ways (Sections III and IV);
- a faceted classification of the ten demos, what it shows, and how it relates to Microsoft's STRIDE threat model (Section V);
- for every fix, a statement of what it does not do (Section IV);
- a check of OWASP's published figures: the ranking formula applied to the score tables, and ten places where OWASP's pages disagree (Section VI);
- an interactive lab that turns the same code into exercises and runs them in the browser (Section VII).

Section II introduces the 2025 list and related work, and Section III our method. Section IV takes the ten categories one by one and Section V shows what they have in common. Section VI checks OWASP's figures, Section VII describes the lab, Section VIII lists the limitations and Section IX answers the three questions. Appendix A gives the code, a sequence diagram and the program output of every scenario; Appendix B the risk-score computation; Appendix C screenshots of the lab; and Appendix D how to run the code. This paper is generated from the same code as the interactive web version of the report, which also lists every Common Weakness Enumeration (CWE) entry that OWASP maps to each category.

II. Background and Related Work

A. The OWASP Top 10:2025

Table I lists the ten categories. Compared with the 2021 edition [16], there are two new categories, Software Supply Chain Failures (A03) and Mishandling of Exceptional Conditions (A10), and one consolidation: server-side request forgery (SSRF), a category of its own in 2021, is now part of Broken Access Control (A01). The names of three categories changed (Fig. 1).

Category names, positions, score-table figures and the mapping of CWEs to categories are OWASP's, read from its pages on 26 November 2026; OWASP publishes the Top 10 under the Creative Commons Attribution Share-Alike 4.0 license [1]. The summaries of OWASP's advice in Appendix A are ours, as are the code, the tests, the lab, the drawings, the explanations and the classification.

TABLE I
THE OWASP TOP 10:2025, WITH THE FIGURES FROM OWASP'S SCORE TABLES

	Category	In 2021	Chosen by	CWEs	Avg. incidence	Occurrences	CVEs
A01	Broken Access Control	= A01 in 2021; now also holds SSRF, A10 in 2021	Data	40	3.74%	1,839,701	32,654
A02	Security Misconfiguration	↑ A05 in 2021	Data	16	3.00%	719,084	1,375
A03	Software Supply Chain Failures	new, widens A06 in 2021, Vulnerable and Outdated Components	Survey	6	5.72%	215,248	11
A04	Cryptographic Failures	↓ A02 in 2021	Data	32	3.80%	1,665,348	2,185
A05	Injection	↓ A03 in 2021	Data	37	3.08%	1,404,249	62,445
A06	Insecure Design	↓ A04 in 2021	Data	39	1.86%	729,882	7,647
A07	Authentication Failures	= A07 in 2021, as Identification and Authentication Failures	Data	36	2.92%	1,120,673	7,147
A08	Software or Data Integrity Failures	= A08 in 2021; "and" became "or" in the name	Data	14	2.75%	501,327	3,331
A09	Security Logging and Alerting Failures	= A09 in 2021, as Security Logging and Monitoring Failures	Survey	5	3.91%	260,288	723
A10	Mishandling of Exceptional Conditions	new, not in the 2021 list	Data	24	2.95%	769,581	3,416

Figures from the score table on each OWASP category page [4]–[13], read on 26 November 2026; the meanings follow OWASP's methodology page [14]. =, ↑, ↓: same place as in 2021, moved up, moved down. *Chosen by*: ranked by the data, or chosen by OWASP's community survey. *Avg. incidence*: OWASP's average, over the category's CWEs, of the share of tested applications that had the CWE; OWASP does not say exactly how it averages. *Occurrences*: applications found to have the category's CWEs, not individual findings. *CVEs*: publicly known vulnerabilities, each with a Common Vulnerabilities and Exposures (CVE) identifier, that the US National Vulnerability Database [15] maps to those CWEs. A category with few CVEs is not a small risk: A03, which has the fewest, has the highest exploit and impact scores of the ten.

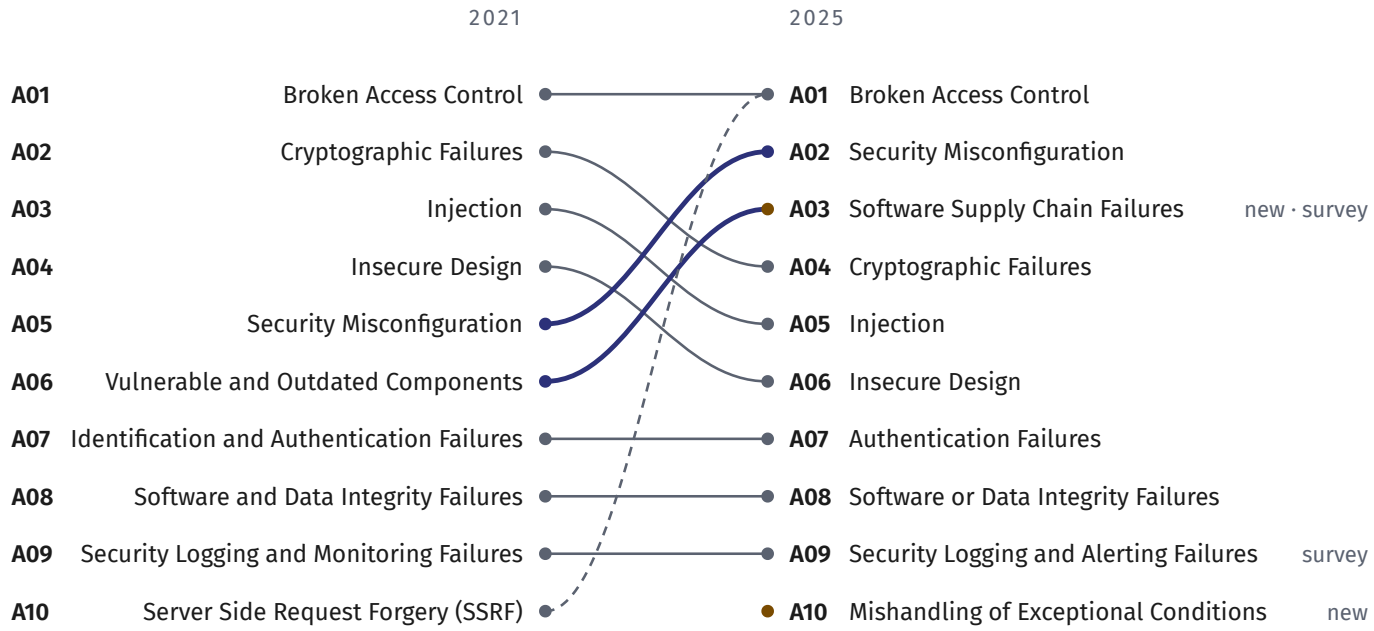


Fig. 1. From 2021 to 2025. Each 2021 category, left, joined to the 2025 category it became, right. Thick lines: moved up, by three places each. Dashed line: rolled into another category (SSRF into A01). Vulnerable and Outdated Components was widened into Software Supply Chain Failures. On the right, "new" marks the new categories and "survey" the two places chosen by the community survey. Sources: OWASP's 2021 list [16] and the 2025 introduction [3].

B. How OWASP Builds the List

OWASP's contributors sent test results for more than 2.8 million applications [3]. The results name weaknesses by their CWE number [17]; 589 different CWEs occur in this edition's data. The team grouped CWEs into categories of at most 40 CWEs each and ranked 12 categories by the data; the ten on the list hold fewer than half of the 589 CWEs. Eight of the ten places come from that ranking; the other two, A03 and A09, were chosen by a community survey of practitioners, because some risks are hard to find with automated tests and so stay rare in test data. OWASP calls the list "data-informed, but not blindly data-driven" [3].

The ranking uses a risk score that OWASP's methodology page defines as a weighted sum of the figures in each category's score table [14]:

$$S = 1000\, I_{\max} + 100\, C_{\max} + 10\, \bar{E} + 20\, \bar{H} + \frac{O}{10\,000}, \quad (1)$$

where $I_{\max}$ is the highest incidence rate among the category's

CWEs, $C_{\max}$ the highest coverage, $\bar{E}$ and $\bar{H}$ the average weighted exploit and impact scores, and O the number of occurrences. OWASP defines a CWE's coverage as the share of the applications in the data that were tested for it, and the exploit and impact scores as the matching sub-scores of the Common Vulnerability Scoring System (CVSS) of the CVEs mapped to those CWEs, on a scale of 10 [14]. OWASP writes the two rates as percentages; we take them as fractions (20.15% as 0.2015), the only reading that gives scores of the size OWASP states. Section VI applies the formula.

C. Three Risks That Came Close

OWASP names three risks that did not make the list [18]:

- **X01 Lack of Application Resilience**: OWASP calls it a renaming of the Denial of Service entry from 2021, which was not on the 2021 list, and says its score was very close to A10's.
- **X02 Memory Management Failures**: it has the third-most CVEs, but came lowest in the community survey and low in the data; OWASP puts this down to web applications outnumbering desktop software.
- **X03 Inappropriate Trust in AI Generated Code ('Vibe Coding')**: there are no CWEs or CVEs for it yet; OWASP gives advice for reviewing code an AI wrote.

D. Related Work

The CWE list [17] names weaknesses at a finer grain than OWASP's categories; each 2025 category maps between 5 and 40 of them (Table I). OWASP's Application Security Verification Standard [19] turns such risks into testable requirements, and the OWASP Cheat Sheet Series gives defensive advice topic by topic [20]–[24]; our fixes apply its central advice, and Section IV says where they fall short of it. Deliberately vulnerable applications such as OWASP WebGoat [25], OWASP Juice Shop [26] and Damn Vulnerable Web Application (DVWA) [27] teach the same risks by letting learners attack a complete running system. Our demos take the opposite route: each is a minimal program in which the flaw sits next to its fix, and tests check that the fix stops the attack without breaking the feature. For classification, Microsoft's STRIDE [28] names six kinds of threat (spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege), each against one security property [29], and Saltzer and Schroeder [30] set out design principles for protection; Section V relates our classification to the first and our repairs to the second.

III. Method

A. Ten Demonstrations

Each category has one module in the `demos` folder, 1,297 lines in all. For each of its scenarios, a module names a vulnerable implementation, a fixed implementation, an attack function and a normal-use function, and its docstring names the CWEs that the demo shows. Comments mark the regions shown in this paper (`# [vulnerable]`, `# [attack]`, `# [fixed]`, `# [end]`); the listings in Appendix A are cut out of the files between them.

TABLE II
THE THREE CHECKS EVERY SCENARIO MUST PASS

	Vulnerable version	Fixed version
The attack	**must succeed**; otherwise the demo shows no flaw	**must fail**; otherwise the fix does not work
Normal use	not checked	**must still work**; otherwise the fix broke the feature

Each demo runs inside one Python process with the standard library only. The databases are real SQLite databases held in memory, queried with the Structured Query Language (SQL). The network in A01 and the package indexes in A03 are Python dictionaries, so no request leaves the machine and nothing is installed. The pickle payload in A08 only sets a marker variable, so that the test can see that it ran. All people and account numbers in the demos are fictitious, and no address in them is ever contacted.

B. The Three-Way Check

A fix only counts if it stops the attack without breaking the feature. Every scenario is therefore run three times, and the tests require all three outcomes in Table II. The third rule matters most: deleting the feature, or rejecting every input, also "stops" an attack. The test file `test_demos.py` checks the 13 scenarios this way in three tests, one for each rule. A fourth test checks that every category has a demo, and 34 more check details such as lockout timing, log format and non-ASCII input. Neither version of the report is built unless all 38 tests pass.

C. A Faceted Classification

To see what the ten share as programming problems, we classified our ten demos by four questions (Table III). Strictly, this is a faceted classification rather than a hierarchy: four independent questions, each with a closed list of answers, and each answer with a one-line definition. The rule: a category gets every answer that at least one of its scenarios shows, so the classification describes the demo, not everything OWASP files under the category. It was made by the two authors together, from the code; OWASP does not publish one.

D. A Report Generated From the Code

The report exists in two versions, a web page and this paper, both built from the code (Fig. 2). The tables, the counts, the listings, the program output and the texts on each category come from one source for both versions; the rest of the prose is written separately for each. The build refuses to produce either version if a test fails or if the texts no longer match the code: every function, table or address that a sequence diagram takes from the code must occur in the demo's source, each demo's docstring must name the same CWEs as the report, and every category must have answers in the classification, a

TABLE III
OUR CLASSIFICATION: FOUR QUESTIONS, EACH WITH A CLOSED LIST OF ANSWERS

Question and answers	Definition
Where it enters	*At which point of building and running the software is the flaw introduced?*
Design	The flaw is in what was decided before coding: a rule the feature needed was never planned.
Code	The flaw is in the program text itself.
Configuration	The flaw is in the settings, or in the values used when nobody sets them.
Dependencies and build	The flaw arrives with components, packages or the build that fetches them.
What goes wrong	*What is the underlying mistake, independent of language or framework?*
A check is missing	The server does not enforce a rule that it owns.
Data is read as code or structure	Input crosses into the syntax of a query, a serialized object or a log entry.
Outside input is trusted without proof	Something from outside the trust boundary is accepted as genuine with no proof of its origin or integrity.
A mechanism unfit for the job	A security mechanism is too weak or not meant for the task, such as a fast hash for passwords or a counter as a token.
The default is unsafe	A setting that nobody changed takes an unsafe value.
A failure leaves an unsafe state	After an error the program carries on with half-done work.
A failure goes unnoticed	Nobody is told: an error is swallowed, or an attack raises no alert.
What is lost	*Which security properties does the demonstrated attack break?*
Confidentiality	Information reaches someone who may not read it.
Integrity	Data, money or code is changed without permission.
Availability	The system or its data cannot be used when needed.
Authenticity	A user or a piece of data is not what it claims to be: its identity or its origin is false.
Accountability	Actions can no longer be traced to whoever did them.
How it is found	*Which check is most likely to find it before an attacker does?*
Static analysis	A tool or a reviewer reading the source can flag the faulty line.
Scanning	A tool checks the dependencies, the settings or the running application from outside.
Misuse-case tests	A test does what an attacker would and checks the result.
Attack drills	A simulated attack checks that someone is alerted.

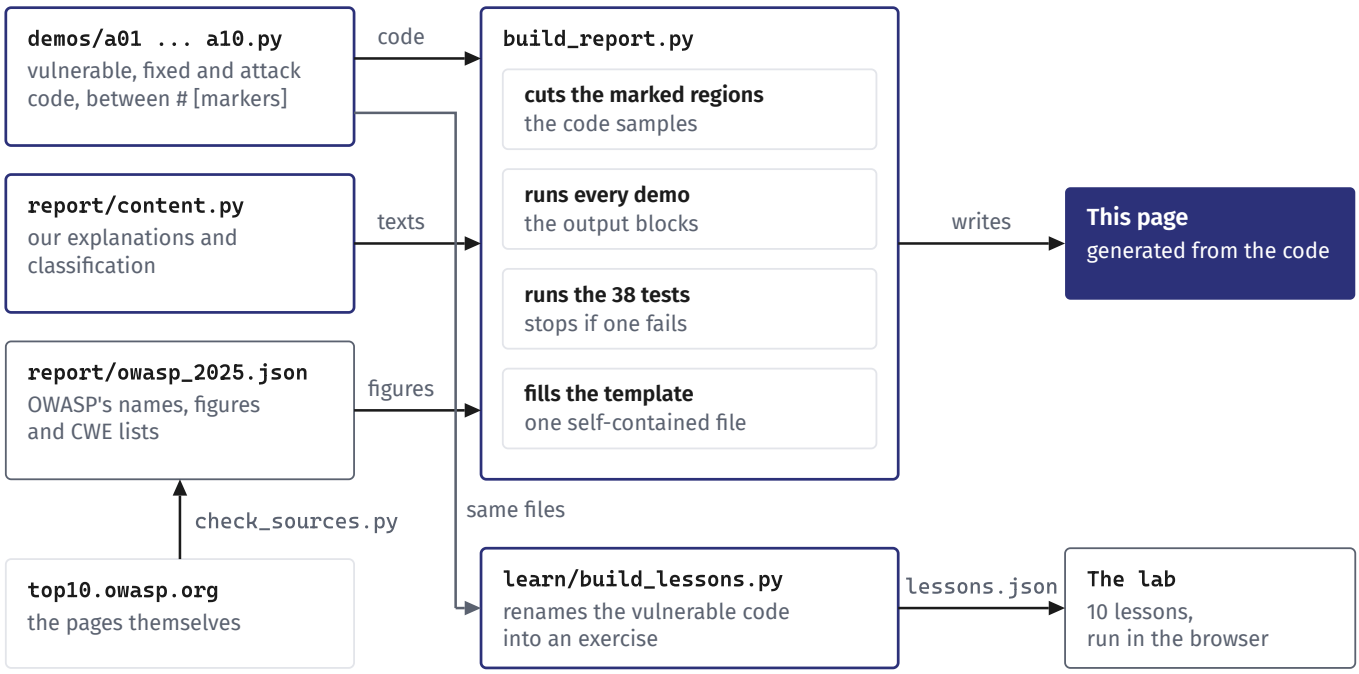


Fig. 2. From the code to the report. The code samples are cut out of the demo files between marker comments, the output blocks are captured while the build runs the demos, and OWASP's figures come from a data file that a separate script fills from OWASP's pages and checks again on request. "This page" is the web version of the report; this paper is typeset from the same data by `build_paper.py`. The lab of Section VII is generated from the same demo files.

place on the map of Fig. 3 and a row in the table of repairs. This paper's drawings are exported from the web page, and the build checks that they were drawn from the current data. It also refuses to finish if LaTeX reports a line that sticks out of its column.

TABLE IV
THE THREE CHECKS OF EVERY SCENARIO, AS RUN WHILE THIS PAPER WAS BUILT

		Attack against		Normal
	Scenario	vulnerable	fixed	use
A01	Read another user's invoice (IDOR)	succeeds	blocked	works
	Make the server fetch an internal address (SSRF)	succeeds	blocked	works
A02	Probe a configured deployment	succeeds	blocked	works
	Deploy with nothing configured	succeeds	blocked	works
A03	Dependency confusion	succeeds	blocked	works
	Tampered file on a mirror	succeeds	blocked	works
A04	Stolen password table	succeeds	blocked	works
A05	UNION-based SQL injection	succeeds	blocked	works
A06	Abuse the checkout rules	succeeds	blocked	works
A07	Guess the session ID, then guess the password	succeeds	blocked	works
A08	Send a crafted cart cookie	succeeds	blocked	works
A09	Forge a log entry and brute-force unnoticed	succeeds	blocked	works
A10	Make a transfer fail halfway	succeeds	blocked	works

IV. THE TEN CATEGORIES IN CODE

Table IV shows the three outcomes of each scenario, recorded while this paper was built, with Python 3.12.10 and SQLite 3.49.1. In 13 of 13 scenarios the attack succeeded against the vulnerable code, failed against the fix, and normal use still worked with the fix, which answers RQ1 for the scenarios we built.

The subsections below take the categories in order. Each says what the flaw is, what is wrong in the code, how the attack works and what the fix changes, and then what the fix does not do, which answers RQ3 for that category. Appendix A has the sequence diagram, the code of all three parts and the program's output for every scenario, with a summary of OWASP's advice for each category.

A. A01:2025 Broken Access Control

Users act outside their permissions: they read or change other people's data, or use functions above their role, because the server does not enforce who may do what. Server-side request forgery is the network form of the same failure: the server lends its own network access to the user [4], [21].

Scenario 1: Insecure direct object reference (IDOR). **What is wrong.** The function knows who is calling and then ignores it. Any invoice number that exists is returned, and a missing one raises an error, so the answers also reveal which numbers exist.

The attack. Mallory, logged in as herself, asks for invoice numbers 1000 to 1005. In a web application that means changing a number in the address; no tool is needed. This is OWASP's example scenario 1 for A01.

The fix. Only the invoice's owner, or a user with the auditor role, gets the invoice; everyone else is refused, and refusal is the default. An unknown number and somebody else's number get the same answer, so the reply does not reveal which numbers exist.

Scenario 2: Server-side request forgery (SSRF). SSRF has been part of A01 since the 2025 edition. A URL parameter lets any user make the server send requests to addresses that only the server can reach, such as the cloud metadata service at 169.254.169.254. The fix allows only https and only listed names, then checks the address the name points to, because whoever controls an allowed name can point it at an internal address. The attack's fourth request does exactly that.

What the fix does not do. The check has to be on every route that touches invoices: one forgotten endpoint reopens the hole, which is why OWASP recommends one shared mechanism over a check written by hand in each handler. The check is only as good as `current_user`, which must come from the server's own session (A07), and refusals should be logged and alerted on (A09). The SSRF fix is sound here because it connects to the very address it checked, in a simulated network with one address per name and no redirects. Real code must not look the name up a second time, must refuse or re-check redirects, and should also block internal addresses at the network level.

Weaknesses shown. CWE-639, Authorization Bypass Through User-Controlled Key; CWE-918, Server-Side Request Forgery (SSRF). Appendix A-A has the sequence diagrams (Figs. A1 and A2), the code (Listings 1–6) and the output (Listing 7).

B. A02:2025 Security Misconfiguration

The software may be sound, but it runs with insecure settings: debug mode, default accounts, error pages that print internals, no security headers [5].

What is wrong. Every setting falls back to a development value when nobody sets it: debug on, `admin/admin`, one secret key for every installation. Nothing forces anyone to change them, so they reach production as they are, and an error sends the stack trace and every setting to whoever caused it.

The attack. Against a deployment where the operator set a password and a key but left debug alone: send a malformed parameter and read the stack trace, the secret key and the admin password from the error page (OWASP's example scenario 3 for A02). Against a deployment with nothing set: log in with `admin/admin`.

The fix. Debug is off unless asked for, and there is no default password or key. At start-up, `load_config` refuses to run with a known default password, a key shorter than 32 characters, or debug in production. Errors return only a reference number and the details go to the application's logger. Every response carries three security headers: Content Security Policy (CSP), HTTP Strict Transport Security (HSTS) and X-Content-Type-Options: nosniff.

What the fix does not do. The start-up check knows only what it was told. Its list of default passwords has four

entries, the key test measures length and not randomness, and only the exact value "production" counts as production. A real check uses a list of known weak passwords and treats every environment except an allowed development one as production. Cloud permissions, open ports and headers set by a proxy need their own automated checks, and OWASP prefers credentials issued by the platform to static secrets kept in the environment.

Weaknesses shown. CWE-489, Active Debug Code; CWE-547, Use of Hard-coded, Security-relevant Constants; CWE-1393, Use of Default Password (OWASP maps it to A07); CWE-209, Generation of Error Message Containing Sensitive Information (OWASP maps it to A10). Appendix A-B has the sequence diagrams (Figs. A3 and A4), the code (Listings 8–10) and the output (Listing 11).

C. A03:2025 Software Supply Chain Failures

The application is built from code, tools and services its authors do not control. A weakness or a compromise anywhere in that chain reaches production: a component with a known vulnerability, a substituted package, a tampered file or build [6]. Our demo shows two of these routes.

What is wrong. The resolver asks two indexes and takes the highest version number, wherever it comes from; on an equal number the public copy even wins. It never asks which index a name belongs to, and it installs whatever bytes arrive.

The attack. Publish a package under the company's internal name, with version 99.0.0, on the public index (dependency confusion) [31]. Separately, change a file on a mirror after its version was approved.

The fix. A lock file pins the exact version, the index it must come from, and the SHA-256 of the file. Anything missing from the lock file, on another index or with another hash is refused. With pip the same idea is the `--require-hashes` mode [32] together with a single index [33].

What the fix does not do. A hash proves that the file is the one that was reviewed, not that the reviewed file is harmless: a malicious release that was pinned on purpose still passes. A lock file also freezes known vulnerabilities, so it has to be checked against vulnerability feeds and updated deliberately, and every change to it reviewed like code, because whoever can edit it can re-point any package and its hash.

Weaknesses shown. CWE-1357, Reliance on Insufficiently Trustworthy Component; CWE-494, Download of Code Without Integrity Check (OWASP maps it to A08); CWE-829, Inclusion of Functionality from Untrusted Control Sphere (OWASP maps it to A08). Appendix A-C has the sequence diagrams (Figs. A5 and A6), the code (Listings 12–14) and the output (Listing 15).

D. A04:2025 Cryptographic Failures

Sensitive data is protected with cryptography that is missing, outdated or used for the wrong job [7].

What is wrong. MD5 is built to be fast, and the hash has no salt. Equal passwords give equal hashes, and one table of hashed common passwords works against every stolen database.

The attack. The attacker has a copy of the password table. She hashes a list of common passwords once and looks up every stored value; with no salt, three of the four passwords are found at once. This is OWASP's example scenario 2 for A04.

The fix. Each password is hashed with scrypt and its own random 16-byte salt: the table is useless, equal passwords are stored differently, and every guess costs time and about 16 MB of memory [34], [35]. The salt comes from the `secrets` module, a cryptographic random source. Hashes are compared in constant time, and the scheme and its parameters are stored with each hash so that they can be raised later.

What the fix does not do. Slow hashing raises the price of each guess; it does not rescue a password that is on the attacker's list. Trying the same thirteen words against each salted hash still finds the three listed passwords, at one scrypt per word and user. Online, A07's attempt limit slows guessing at the login form; against a stolen table, only passwords that are on no such list survive, which is why new passwords should be checked against lists of breached ones. The scrypt cost here (`n=16384`, `r=8`, `p=1`) keeps the demo fast and is below the minimum in OWASP's Password Storage Cheat Sheet [20]. The 1 ms bar that decides what counts as a cheap guess is a teaching simplification.

Weaknesses shown. CWE-327, Use of a Broken or Risky Cryptographic Algorithm; CWE-759, Use of a One-Way Hash without a Salt; CWE-916, Use of Password Hash With Insufficient Computational Effort. Appendix A-D has the sequence diagram (Fig. A7), the code (Listings 16–18) and the output (Listing 19).

E. A05:2025 Injection

Text supplied by the user becomes part of a command, and the interpreter cannot tell the data from the instructions [8].

What is wrong. The f-string builds the SQL text out of the input. A single quote in the input ends the string, and what follows is read as SQL, as part of the same statement. The same flaw breaks an honest search: a customer called `o'brien` gets a syntax error.

The attack. Type `x' UNION SELECT username, password_hash FROM users --` into the customer field. The quote ends the string, the UNION adds a second query that must select two columns like the first, and `--` turns the closing quote into a comment. The query meant to list orders returns the users table, password hashes included. Here the injection can only read: Python's `sqlite3` runs one statement per call [36], so DROP or UPDATE cannot be appended. Where the injected query is itself an UPDATE, or the driver runs several statements, the same flaw changes data.

The fix. The statement is constant text with a ? placeholder, and the value is passed separately. The database compiles the statement first and then compares the value with the column; the value is never parsed as SQL [22].

What the fix does not do. A placeholder protects only the statement it is in: a query built as text elsewhere, inside a stored procedure or by hand for an object-relational mapper (ORM), is still injectable. Placeholders work for values only; table names, column names and sort directions cannot be bound and need an allow-list. The demo shows one of the 37 weaknesses OWASP maps to A05. OWASP counts more than 30,000 CVEs for cross-site scripting against more than 14,000 for SQL injection, and cross-site scripting needs a browser to demonstrate.

Weaknesses shown. CWE-89, Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). Appendix A-E has the sequence diagram (Fig. A8), the code (Listings 20–22) and the output (Listing 23).

F. A06:2025 Insecure Design

The feature works as it was designed, and the design is what is unsafe: a control it needed was never planned [9].

What is wrong. The checkout was designed to read the price from the form the browser sends back, to accept any quantity, and to apply the coupon without recording its use. The business wanted single use, but the checkout's contract had no place to record it, so the control was never designed; no single line is a coding slip.

The attack. Edit the price field to one cent, order minus forty mice to receive a credit, and use the single-use coupon twice.

The fix. A different contract. The browser names products and quantities only; the server looks up the price, accepts 1 to 10 of an item per line and records each coupon use. Our attack function is the misuse-case test that OWASP asks for. No scan of the code for a dangerous call would find these rules: they come from asking how the feature can be abused before it is written, which is what threat modeling is for.

What the fix does not do. The quantity limit applies per order line, so sixty lines of ten laptops pass in one order; limits per order and per customer, and rate limits, are separate design rules. Coupon use is remembered in one process's memory, so a restart or a second server forgets it, and a new account gets a new coupon. In a real system the record must be durable and shared, and written in one atomic step, for example under a unique constraint, or two simultaneous orders both get the discount.

Weaknesses shown. CWE-501, Trust Boundary Violation; CWE-472, External Control of Assumed-Immutable Web Parameter; CWE-799, Improper Control of Interaction Frequency. Appendix A-F has the sequence diagram (Fig. A9), the code (Listings 24–26) and the output (Listing 27).

G. A07:2025 Authentication Failures

The system accepts someone as a user without good enough proof of who they are [10].

What is wrong. The login allows unlimited guesses, and it hands out session numbers that count upwards.

The attack. Log in normally and subtract one from your own session number: you are now the previous user. Then try a list of passwords against Alice: four common ones, then `winter1995` to `winter2029`. Her password, `winter2019`, falls on guess 29. OWASP's example scenario 1 for A07 describes this habit of attackers, who step passwords such as `Winter2025` to `Winter2026`.

The fix. Five failed logins lock the account for fifteen minutes, and the session token is 256 random bits instead of the next number in a sequence.

What the fix does not do. The lock slows guessing but does not stop it: five guesses every fifteen minutes is 480 a day, so a password on the attacker's list still falls. What protects Alice is refusing weak and breached passwords, and multi-factor authentication, which OWASP lists first. Locking per account also lets an attacker lock a victim out, and does nothing against one password tried across many accounts. The counters live in one process's memory and are updated after the slow password check, so parallel requests and a second server each get their own five; a real service records each attempt atomically in a shared store, or uses a vetted framework, as OWASP advises. Neither version ends a session at logout or after a timeout.

Weaknesses shown. CWE-307, Improper Restriction of Excessive Authentication Attempts; CWE-340, Generation of Predictable Numbers or Identifiers (OWASP maps it to A04). Appendix A-G has the sequence diagram (Fig. A10), the code (Listings 28–30) and the output (Listing 31).

H. A08:2025 Software or Data Integrity Failures

Code or data is accepted from outside without checking that it is what the server wrote or approved [11], [23].

What is wrong. The cart cookie is a pickle, and unpickling calls whatever functions the bytes name [37]. The client wrote the bytes, and nothing checks whether the server did.

The attack. Send a pickle that tells the loader to call `exec` on a statement of the attacker's choice. The `Payload` class's `__reduce__` method is what writes that call into the bytes, on the attacker's side; it never runs on the server. Our statement only sets a marker; a real one could run any command. A forged cart with `role=admin`, sent as a pickle, is accepted as well. The attack also tries the cart as plain JavaScript Object Notation (JSON), unsigned or with a made-up tag: the vulnerable loader refuses JSON, but a fix that only switched to JSON would accept it.

The fix. Two repairs, each closing one hole. The cookie is JSON, which holds data only, so loading it runs nothing. And it carries a tag, a hash-based message authentication code (HMAC) [38] computed with SHA-256, which is checked in constant time [39] before anything is parsed, so a forged cart is refused. JSON alone would still accept a forged `role=admin`.

What the fix does not do. The tag proves that the server wrote these bytes once, not when or for whom: an old cart or an old `role=admin` cookie is accepted again indefinitely, because no time and no user is inside the signed data. In the demo the key is a constant in the source, so that the run is reproducible; a real server loads it from a secret store and plans for changing it. The client can still read the cart: a tag gives integrity, not secrecy. The software half of this category,

code downloaded without an integrity check, is what the A03 demo shows.

Weaknesses shown. CWE-502, Deserialization of Untrusted Data; CWE-345, Insufficient Verification of Data Authenticity; CWE-565, Reliance on Cookies without Validation and Integrity Checking. Appendix A-H has the sequence diagram (Fig. A11), the code (Listings 32–34) and the output (Listing 35).

I. A09:2025 Security Logging and Alerting Failures

An attack happens and nobody can see it: events are not recorded, the record can be falsified, nobody is alerted, or the log itself exposes secrets [12], [24].

What is wrong. The handler writes user input into the log unchanged, writes the typed password next to it, records nothing when a login succeeds and never notes where a request came from, and treats the twentieth failure exactly like the first.

The attack. Put a line break and a fake "`login ok user=admin`" entry into the username field. Then guess the password twenty times in twenty seconds.

The fix. One JSON object per line, so a line break is escaped inside the value and cannot start an entry. The password is never written. Successes and failures are both recorded with the user and the address, and an alert is raised when five failures come from one address within a minute.

What the fix does not do. The alert detects the guessing but does not stop it; limiting attempts is the A07 fix. The rule fires when the count reaches exactly five, so a steady pace of one guess every fifteen seconds never alerts (5,760 guesses a day), and guessing spread over many addresses goes unseen; real rules also count per account and in total. An alert helps only if someone receives it and acts, and the log must be stored where the application cannot rewrite it. Tools that display the log must still encode it for their own output.

Weaknesses shown. CWE-117, Improper Output Neutralization for Logs; CWE-532, Insertion of Sensitive Information into Log File; CWE-778, Insufficient Logging. Appendix A-I has the sequence diagram (Fig. A12), the code (Listings 36–38) and the output (Listing 39).

J. A10:2025 Mishandling of Exceptional Conditions

The program meets a situation it did not plan for, such as a failed step, a missing value or an exhausted resource, and carries on in an unsafe state [13].

What is wrong. The transfer credits one account, then fails to debit the other, because the table refuses a negative balance. It catches the exception and answers "error", but each statement was committed at once [36], so the credit stays. It also never checks that both accounts exist or that the amount is positive.

The attack. Mallory has 10.00 and sends 1,000.00 to her own savings account, three times. Every failed transfer creates 1,000.00.

The fix. The same two updates, in the same order, now run in one transaction: when the debit fails, the rollback undoes the credit that was already made. Before that, an amount that is not a positive whole number is refused, and the transfer fails unless each update touched exactly one account. Expected failures get one answer, "rejected", which does not reveal which accounts exist; unexpected ones are raised again for a central handler to log.

What the fix does not do. A transaction protects one database; a transfer between two services needs compensating steps. Conservation of money is the first test, not the only one: a refused transfer must also answer "rejected" and leave every balance as it was, and an amount that is not positive must be refused. The function starts and ends its own transaction, so it cannot be called inside one that is already open: there it fails, and its rollback also undoes the caller's uncommitted changes.

Weaknesses shown. CWE-460, Improper Cleanup on Thrown Exception; CWE-636, Not Failing Securely ('Failing Open'); CWE-754, Improper Check for Unusual or Exceptional Conditions; CWE-396, Declaration of Catch for Generic Exception. Appendix A-J has the sequence diagram (Fig. A13), the code (Listings 40–42) and the output (Listing 43).

V. What the Ten Have in Common

A. The Classification

Table V gives our answers for each demo, and Fig. 3 places each demo where its flaw sits in one web application. Three findings stand out:

- **Six of ten enter purely as code.** In five of them (A04, A05, A08, A09, A10) the faulty line can be flagged by reading the source; in A01 the missing check is only visible to someone who knows the rule it should enforce. The other four begin in the design (A06, A07), the configuration (A02) and the dependencies and build (A03), where a review of the code alone does not reach.
- **No demo breaks availability.** Every other property is lost in at least one demo. None of the ten is a denial-of-service attack; that risk is OWASP's near miss X01.
- **Seven kinds of mistake, none dominant.** The most frequent, at three demos each, are "a check is missing" (A01, A06, A07), "data is read as code or structure" (A05, A08, A09) and "outside input is trusted without proof" (A03, A06, A08). Five demos (A06, A07, A08, A09, A10) combine two mistakes.

B. Kinds of Repair

Our ten fixes use six kinds of repair (Table VI). Two design principles of Saltzer and Schroeder [30] meet in the first row: complete mediation, checking every access on the server, and fail-safe defaults, refusing unless access is allowed. The fifth and the last rows carry fail-safe defaults over from access decisions to settings and to failures. The repairs carry over better than the fixes: a fix belongs to its demo, while a kind of repair applies to every category in its row.

C. Next to STRIDE

Our question "what is lost" looks at the same thing as Microsoft's threat model STRIDE [28], from the other side:

TABLE V
THE TEN DEMOS, CLASSIFIED

Category	Where it enters	What goes wrong	What is lost	How it is found
A01 Broken Access Control	Code	A check is missing	Confidentiality	Misuse-case tests
A02 Security Misconfiguration	Configuration; Code	The default is unsafe	Confidentiality; Authenticity	Scanning
A03 Software Supply Chain Failures	Dependencies and build	Outside input is trusted without proof	Integrity; Authenticity	Scanning
A04 Cryptographic Failures	Code	A mechanism unfit for the job	Confidentiality	Static analysis
A05 Injection	Code	Data is read as code or structure	Confidentiality	Static analysis
A06 Insecure Design	Design	A check is missing; Outside input is trusted without proof	Integrity	Misuse-case tests
A07 Authentication Failures	Design; Code	A check is missing; A mechanism unfit for the job	Authenticity	Misuse-case tests
A08 Software or Data Integrity Failures	Code	Data is read as code or structure; Outside input is trusted without proof	Integrity; Authenticity	Static analysis; Misuse-case tests
A09 Security Logging and Alerting Failures	Code	A failure goes unnoticed; Data is read as code or structure	Accountability; Confidentiality	Static analysis; Attack drills
A10 Mishandling of Exceptional Conditions	Code	A failure goes unnoticed; A failure leaves an unsafe state	Integrity	Misuse-case tests; Static analysis

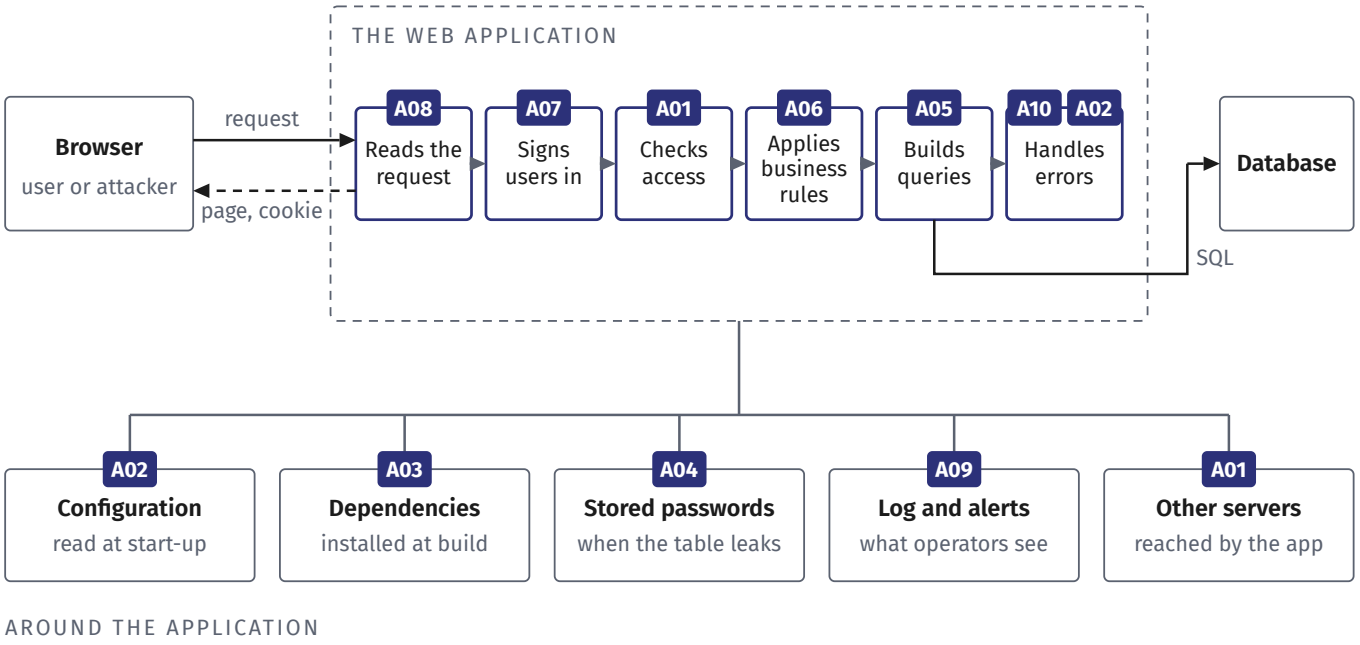


Fig. 3. The ten demos placed in one web application. Six stages handle every request; five more parts surround it. A01 appears twice, for its two scenarios, and A02 twice, for its settings and its error page.

STRIDE names six kinds of threat, and each one threatens one security property [29] (Table VII). Five of the six line up with our answers. The sixth, elevation of privilege, threatens authorization, which our list leaves out on purpose: we record what the attack costs, so A01, where Mallory reads other people's invoices, counts as a loss of confidentiality.

VI. CHECKING OWASP'S FIGURES

A. The Ranking Formula, Applied

We applied (1) to the score tables of the ten categories [4]–[13] and of the two near misses that have one [18] (Fig. 4). The two categories chosen by the survey score lowest of the ten, which is why the survey was needed to put them on the list. X01 Lack of Application Resilience scores above A03, A08 and A09. A08 is placed above A10 although it scores lower. OWASP's pages explain neither. Our arithmetic reproduces the lowest score that OWASP's methodology page states [14], 271.08 for X02, which that page calls Memory Management Errors (ours 271.16), but not the highest, 621.60 for A01 (ours 632.67). OWASP's calculation therefore used inputs that are not all in the tables, and the scores should be

TABLE VI
THE KINDS OF REPAIR OUR FIXES USE

Repair	In our code	Categories
Decide on the server and refuse by default	Ownership check; allow-list of destinations and a check of the address; server-side price, quantity and coupon rules; attempt limit	A01, A06, A07
Keep data apart from code	SQL placeholder; JSON in place of pickle; one JSON object per log line	A05, A08, A09
Verify before trusting	Lock file with versions, indexes and hashes; HMAC tag on the cookie; start-up check of the settings	A02, A03, A08
Use a mechanism built for the job	scrypt with a salt for passwords; random 256-bit session tokens	A04, A07
Make the safe value the default	Debug off unless asked for; no default password or key; security headers on every response	A02
Fail closed and make the failure visible	Generic error page with a reference number; alert on repeated failures; rollback and re-raise	A02, A09, A10

TABLE VII
OUR PROPERTY FACET NEXT TO STRIDE

STRIDE threat	Property it threatens [29]	Our value	Our categories with that value
Spoofing	Authentication	Authenticity	A02, A03, A07, A08
Tampering	Integrity	Integrity	A03, A06, A08, A10
Repudiation	Non-repudiation	Accountability	A09
Information disclosure	Confidentiality	Confidentiality	A01, A02, A04, A05, A09
Denial of service	Availability	Availability	–
Elevation of privilege	Authorization	none	–

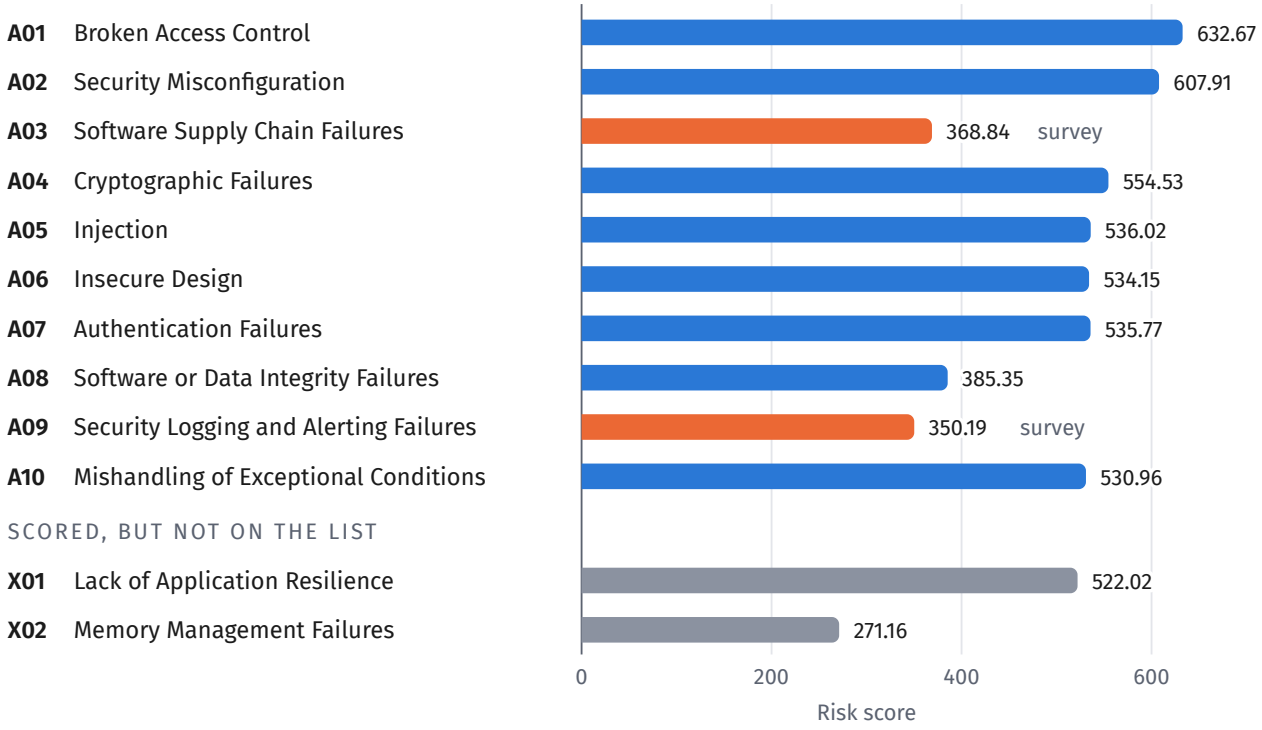


Fig. 4. OWASP's ranking formula (1) applied to the published score tables: our arithmetic, not OWASP's. Blue: on the list, ranked by the data. Orange, marked "survey": on the list, chosen by the survey. Gray: scored, but not on the list. Table IX in Appendix B gives every term.

read as approximate.

B. Where OWASP's Pages Disagree

While checking every figure against OWASP's pages, we found ten places where the pages disagree, with each other or within one page (Table VIII): figures in the introduction that differ from the category page they describe, pages whose text, score table and list of CWEs do not agree, and names that differ. Where they differ, this report uses the category pages, because they carry the full score table and list each CWE. The script that reads OWASP's pages into the report's data file, `report/check_sources.py`, lists these places again every time it runs.

TABLE VIII
PLACES WHERE OWASP'S OWN PAGES DISAGREE

About	What	One place says	Another place says
A01	Average incidence rate	introduction: **3.73%**	A01 score table: **3.74%**
A03	Mapped CWEs	introduction: **5**	A03 score table and list: **6**
A05	Mapped CWEs	introduction: **38**	A05 score table and list: **37**
All ten	Mapped CWEs	introduction: **248**	the ten lists, counted: **249**
A03	Average incidence rate	A03 background: **5.19%**	A03 score table: **5.72%**
A03	Number of "Use of Obsolete Function"	A03 background: **CWE-477**	A03 list: **CWE-447**
A07	Names in the list	A07 list, numbers: **CWE-297, CWE-298 and CWE-299**	A07 list, names: **all called "Improper Validation of Certificate with Host Mismatch"**
X01	Mapped CWEs	X01 score table: **16**	X01 list: **39, the same list as A06**
X02	Mapped CWEs	X02 score table: **24**	X02 list: **20**
X02	Category name	methodology page: **Memory Management Errors**	Next Steps page: **Memory Management Failures**

Pages: introduction [3]; methodology page, "What are Application Security Risks?" [14]; category pages, with their background text, score table and list [4]–[13]; Next Steps page, with the near misses X01 to X03 [18]. All read on 26 November 2026.

VII. THE INTERACTIVE LAB

The same ten demos also became exercises. In the lab, a lesson explains the flaw, the learner runs the attack against the starting code, changes the code in an editor, and the page checks the change with the two rules of Table II for the fixed version: the attack must fail, and normal use must still work. There are ten lessons, one for each category, and their code is generated from the demo files.

The Python runs inside the visitor's browser, in a background worker, through Pyodide [40] 314.0.8 with Python 3.14.2, so the website itself is only static files: there is no server that runs anyone's code (Fig. 5). In Pyodide, `hashlib` has no algorithms that depend on OpenSSL [41], so it lacks `hashlib.scrypt` and `hashlib.pbkdf2_hmac` [35]. There the A04 fix stores passwords with PBKDF2 (Password-Based Key Derivation Function 2) over HMAC-SHA-256 [42], computed by a version written out in Python, with 20,000 rounds: far fewer than the 600,000 that OWASP recommends [20], so that the lesson stays responsive. Fig. 6 shows the A05 lesson as it opens and a half-fix that the check refuses.

What was tested:

- `learn/test_runner.py` runs the exercise checker under ordinary Python: every lesson's starting code must lose to its attacks, every solution must pass, broken code must be reported with its line, and nine half-fixes must be refused (the list below).
- The page has a self-test that runs every lesson and all ten demos in the browser, tests the syntax check, undo and redo, the colors and the font, and reports the results to the test server. After the corrections of 26 November 2026 it passed in Firefox 161 and in Microsoft Edge 158.
- The screenshots in Fig. 6 and Appendix C come from a scripted run in Microsoft Edge on 26 November 2026, after the corrections listed in Section VIII: the attack, a refused half-fix, the passing fix, the syntax check and all ten demos.
- Not tested: Safari, phones, slow connections, and a real hosting account.

The check refuses these half-fixes. Each is a test in `learn/test_runner.py`: a plausible fix that stops part of the problem, and that the lab must not accept.

- **A02**: debug switched off but default password kept.
- **A03**: private index first but no hash check.
- **A04**: a newer hash without a salt.
- **A04**: salted but still fast.
- **A06**: server price but coupon reusable.
- **A07**: random tokens but unlimited guesses.
- **A07**: lockout that also locks out honest typos.
- **A08**: JSON but no signature.
- **A09**: structured log but nobody is told.

VIII. LIMITATIONS

- Each demo shows between one and four weaknesses, while each OWASP category maps between 5 and 40 CWEs. The demos show 28 CWEs in all; 23 of them are mapped by OWASP to the category they illustrate, and 5 to another category. Each category's subsection says which.
- The demos are Python only. Weaknesses that need a browser to show, such as cross-site scripting (part of A05) and cross-site request forgery (part of A01), are not demonstrated.
- The network in A01 and the package indexes in A03 are simulated. The attacks show the logic of the flaw, not how it is exploited on a real network.
- Each fix is the smallest change that makes the attack fail. None is a complete defense; each category's subsection says what its fix does not do.
- The OWASP figures were copied from the pages on 26 November 2026. The script `report/check_sources.py` compares them with the pages again on request; it is the only part of the project that reads OWASP's pages. The risk scores in Section VI are our own arithmetic and do not match OWASP's stated scores exactly.
- The classification was made by the two authors together, without an independent second coding.
- The tests were written by the same people as the fixes, so they check what we thought of. The lab's half-fix tests and the normal-use checks reduce this; they do not remove it.
- The lab is a prototype. It has been tested in a Chromium-based browser and in Firefox, not in Safari, on phones or on a real hosting account.

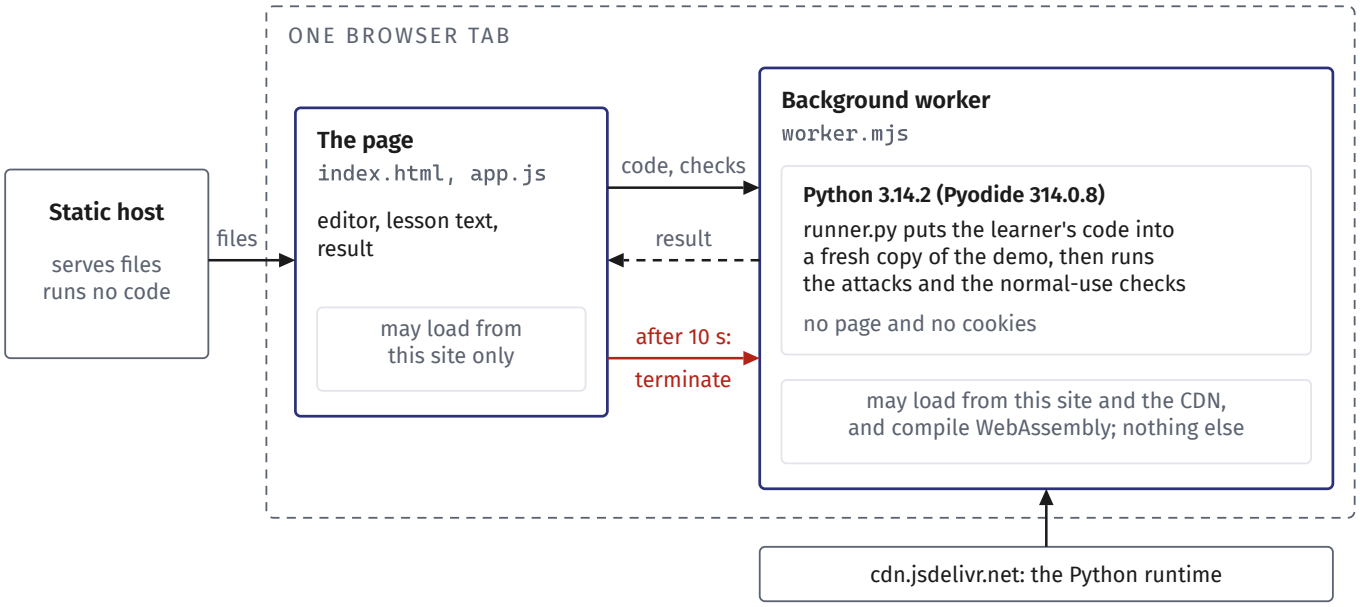
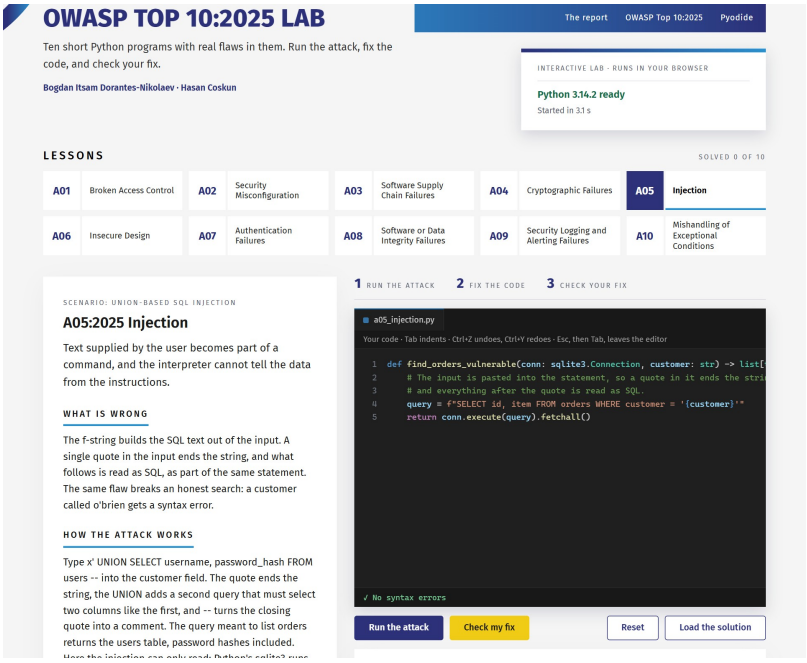


Fig. 5. How the lab runs a learner's code. Nothing is executed on the server. The page carries its own content security policy; the worker gets a separate one only from a header that the server sends, as `learn/serve.py` does and a real host must be set up to do. A run that takes longer than 10 seconds is stopped by terminating the worker.



(a) The A05 lesson as a learner first sees it

(b) A fix that breaks normal use is refused

Fig. 6. The lab in Microsoft Edge 158, 26 November 2026. (a) The A05 lesson as it opens. The lesson explains the flaw on the left; the editor on the right holds the vulnerable function from `demos/a05_injection.py`, unchanged. The start-up time is a second visit, with the Python runtime in the browser's cache; a first visit downloads about 6 MB. (b) A tempting half-fix: delete every quote. The attack is blocked, but the check still says no, because the search for the customer `o'brien` now returns nothing. The check runs the attack and the normal-use test every time. All six screenshots, at a larger size, are in Appendix C.

A review of the demos against the text of the report, on 26 November 2026, found places where the code did less than the text said. Each was fixed; where tests were added with a fix, their number is given in brackets:

- A02, A08 and A09: the fixed code crashed on input with non-ASCII characters, because `hmac.compare_digest` refuses such text [39]. It now compares bytes (three tests). In

A08 one character in a cookie's tag was enough to cause a server error.

- A09: an unknown user could log in with an empty password (one test), and the printed count of log lines with a typed password counted one guess instead of all of them.
- A04: the attacker's lookup table was always built with MD5, so the fix's result did not depend on the salt. The table is now built with the scheme under attack.
- A01: the attack never reached the fix's second check, of the address. A fourth request now re-points an allowed name (three tests).
- A10: the fix debited before it credited, so the transaction was not what stopped the attack. It now keeps the original order, and the rollback undoes the credit.
- A02: the fix wrote the request path into the log unescaped, the same log forging that A09 shows. A03: the fixed resolver now refuses an unknown index instead of falling back to the public one (one test). A06: each misuse case is now tested on its own (two tests).
- Five of the CWEs listed are mapped by OWASP to another category; each category's subsection now says which. The lab's syntax check now also reports errors that Python finds only when compiling, such as `return` outside a function (one test of the lab).

IX. Conclusion

RQ1: every category can be shown in code. In 13 of 13 scenarios the attack succeeded against the vulnerable code, failed against the fix, and normal use still worked, every time the report was built (Table IV). The lab turns the same code into exercises and refuses nine plausible half-fixes (Section VII).

RQ2: what the ten share. Six of ten enter purely as code. In five of them (A04, A05, A08, A09, A10) the faulty line can be flagged by reading the source; in A01 the missing check is only visible to someone who knows the rule it should enforce. The other four begin in the design (A06, A07), the configuration (A02) and the dependencies and build (A03), where a review of the code alone does not reach. Seven kinds of mistake, none dominant. The most frequent, at three demos each, are "a check is missing" (A01, A06, A07), "data is read as code or structure" (A05, A08, A09) and "outside input is trusted without proof" (A03, A06, A08). Five demos (A06, A07, A08, A09, A10) combine two mistakes. No demo breaks availability (Section V).

RQ3: every fix leaves something open. An attempt limit slows guessing without stopping it, a lock file freezes known vulnerabilities, a signed cookie can be replayed, and an alert does not stop the attack it reports; each category's subsection in Section IV says what is left. The repairs carry over better than the fixes: six kinds of repair cover all ten, and three of them rest on Saltzer and Schroeder's principles of complete mediation and fail-safe defaults.

What the demos cannot show. They leave out flaws that need a browser, such as cross-site scripting, and anything that depends on a real network or on concurrency. The next steps would be lessons in JavaScript for the flaws on the browser's side, and demos over a real HTTP server.

APPENDIX A
CODE, DIAGRAM AND OUTPUT OF EVERY SCENARIO

For each category: the sequence diagram of each scenario, the vulnerable code, the attack and the fixed code exactly as they stand in the demo file, the output of running the demo, and a summary of OWASP's advice for the category. Lines too long for the page continue after an arrow.

A. A01:2025 Broken Access Control

File `demos/a01_broken_access_control.py`, 164 lines, two scenarios; explained in Section IV-A.

Scenario 1: Insecure direct object reference (IDOR)

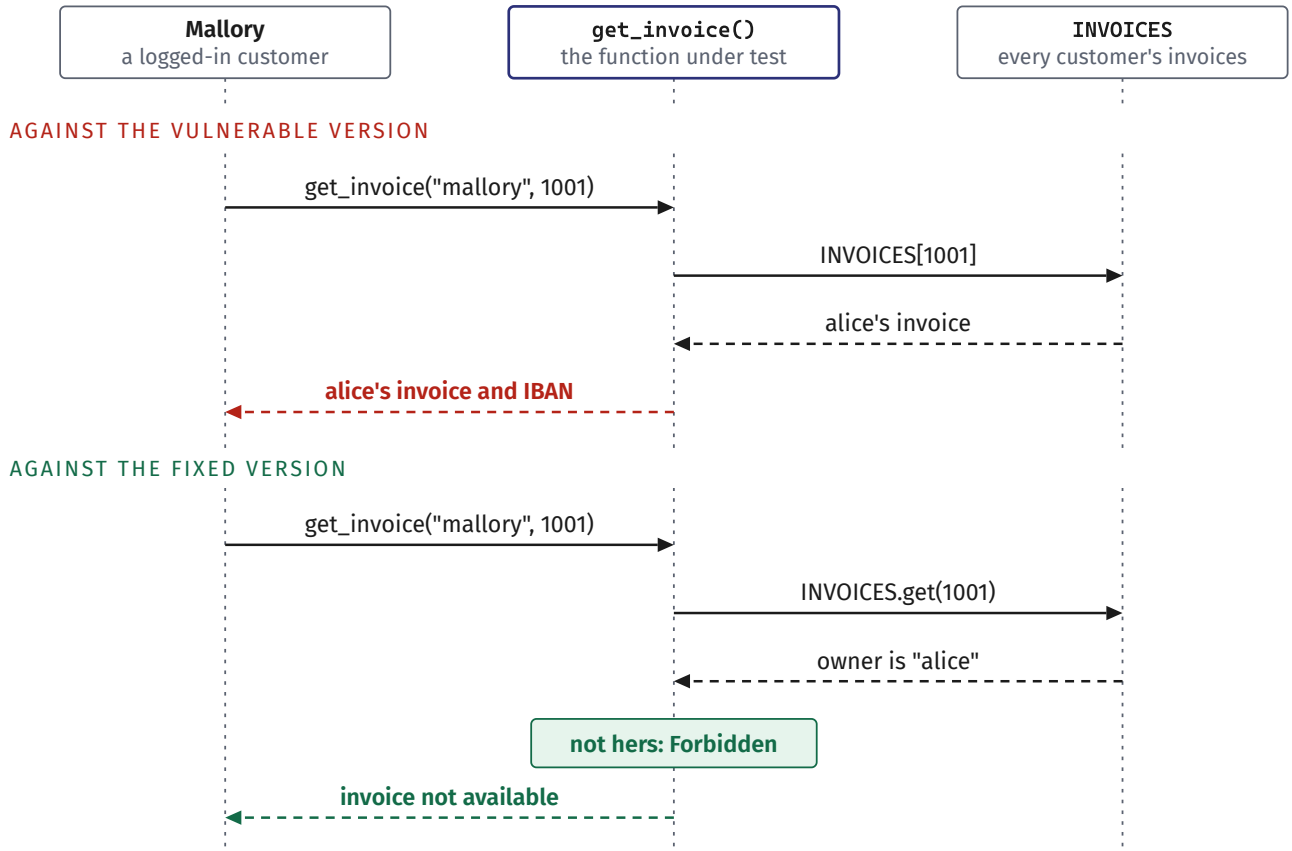


Fig. A1. A01, scenario 1: read another user's invoice (IDOR). The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a01_broken_access_control.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 1. A01, scenario 1: the vulnerable version

```python
def get_invoice_vulnerable(current_user: str, invoice_id: int) -> dict:
    # The caller is logged in, but nothing checks that the invoice is theirs.
    return INVOICES[invoice_id]
```

Listing 2. A01, scenario 1: the attack

```
def attack_idor(get_invoice) -> Result:
    """Mallory is logged in as herself and walks through the invoice numbers."""
    stolen = []
    for invoice_id in range(1000, 1006):
        try:
            invoice = get_invoice("mallory", invoice_id)
        except (KeyError, Forbidden):
            continue
        if invoice["owner"] != "mallory":
            stolen.append(f"{invoice_id} ({invoice['owner']}, {invoice['iban']})")
    return Result(bool(stolen), "read: " + ", ".join(stolen) if stolen else "no foreign invoice returned")
```

Listing 3. A01, scenario 1: the fixed version

```
def get_invoice_fixed(current_user: str, invoice_id: int) -> dict:
    invoice = INVOICES.get(invoice_id)
    # Deny by default. An unknown id and somebody else's id get the same answer,
    # so the reply does not reveal which invoice numbers exist.
    allowed = invoice is not None and (
        invoice["owner"] == current_user or ROLES.get(current_user) == "auditor"
    )
    if not allowed:
        raise Forbidden("invoice not available")
    return invoice
```

Scenario 2: Server-side request forgery (SSRF)

Listing 4. A01, scenario 2: the vulnerable version

```
def fetch_preview_vulnerable(url: str) -> str:
    host = urlsplit(url).hostname
    return _connect(DNS.get(host, host))  # goes wherever the user points it
```

Listing 5. A01, scenario 2: the attack

```
@contextmanager
def repointed_name():
    """Whoever controls an allowed name can point it at an internal address."""
    DNS["images.example.org"] = "10.0.0.5"
    try:
        yield
    finally:
        DNS["images.example.org"] = "93.184.216.34"


def attack_ssrf(fetch_preview) -> Result:
    """Ask the image-preview feature to fetch addresses only the server can reach."""
    reached = []

    def fetch(url: str, label: str) -> None:
        try:
            body = fetch_preview(url)
        except Forbidden:
            return
        if "credentials" in body or "admin console" in body:
            reached.append(label)

    fetch("http://169.254.169.254/latest/meta-data/", "169.254.169.254")   # the cloud metadata service
    fetch("http://10.0.0.5/admin", "10.0.0.5")                             # an internal admin console
    fetch("https://evil.example.net/logo.png", "evil.example.net")         # the attacker's name, pointing inside
    with repointed_name():                                                  # an allowed name, re-pointed
        fetch("https://images.example.org/cat.png", "images.example.org (re-pointed)")
    return Result(bool(reached), "internal answers from: " + ", ".join(reached) if reached else "every internal destination
     ↪ refused")
```

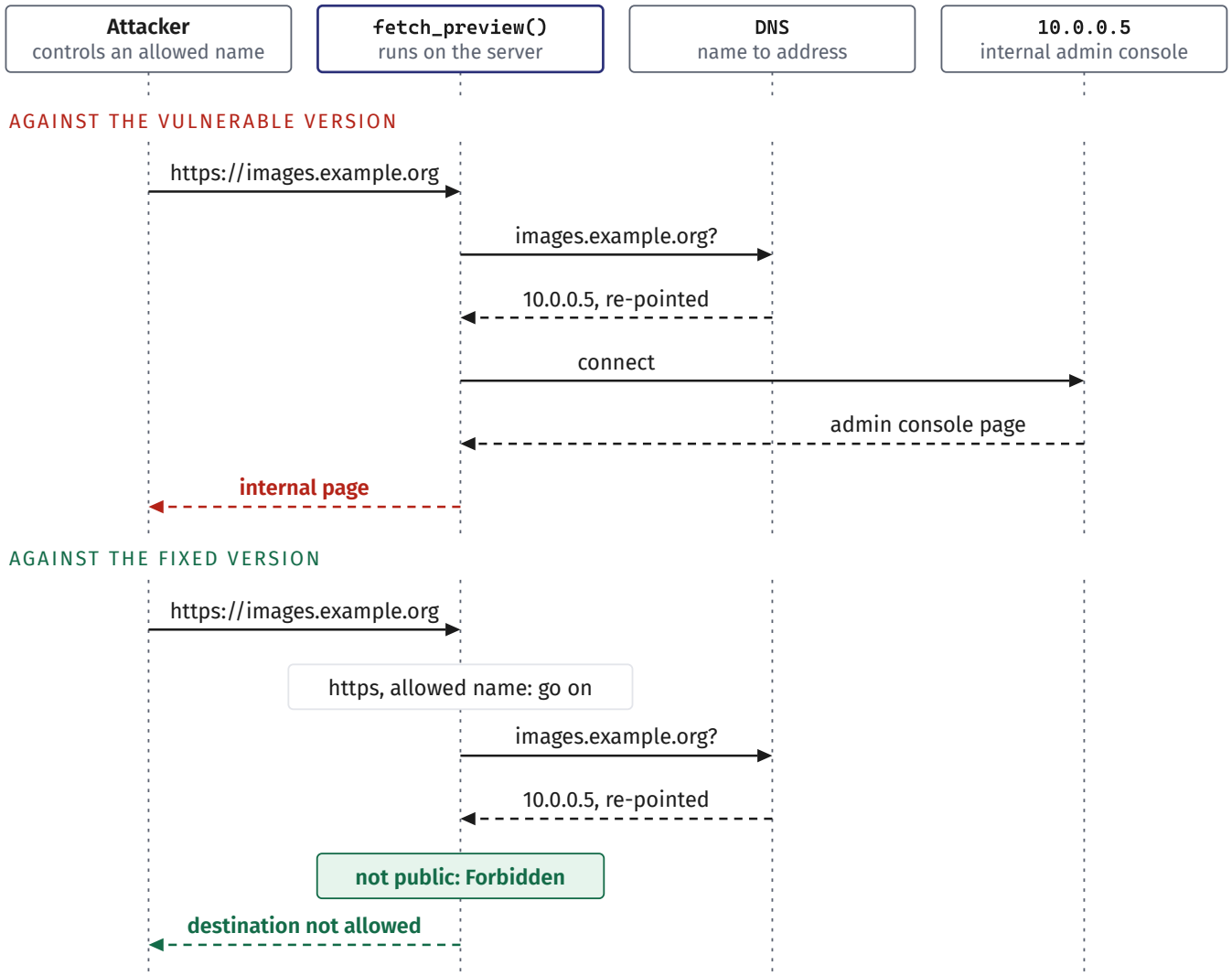


Fig. A2. A01, scenario 2: make the server fetch an internal address (SSRF). The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a01_broken_access_control.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 6. A01, scenario 2: the fixed version

```python
def fetch_preview_fixed(url: str) -> str:
    parts = urlsplit(url)
    if parts.scheme != "https" or parts.hostname not in ALLOWED_HOSTS:
        raise Forbidden("destination not allowed")
    address = ipaddress.ip_address(DNS[parts.hostname])
    # Check the address we are about to connect to, not only the name:
    # an allowed name can later be re-pointed at an internal address.
    if not address.is_global:
        raise Forbidden("destination not allowed")
    return _connect(str(address))
```

Listing 7. A01: output of `python -m demos.a01_broken_access_control`

```
A01:2025 Broken Access Control
  scenario: read another user's invoice (IDOR)
    vulnerable ATTACK SUCCEEDED read: 1001 (alice, MK07 2501 2000 0058 984), 1002 (bob, MK07 3000 0000 0042 425)
    fixed      attack blocked   no foreign invoice returned
    normal use still works with the fix: yes
  scenario: make the server fetch an internal address (SSRF)
    vulnerable ATTACK SUCCEEDED internal answers from: 169.254.169.254, 10.0.0.5, evil.example.net, images.example.org
     ↪ (re-pointed)
    fixed      attack blocked   every internal destination refused
    normal use still works with the fix: yes
```

OWASP's advice, summarized [4]:

- Deny by default; only public resources are open.
- Implement access control once, in server-side code, and reuse it everywhere.
- Enforce record ownership, not only that the user is logged in.
- Log access-control failures and alert on repeats.
- Include functional access-control checks in unit and integration tests.

B. A02:2025 Security Misconfiguration

File `demos/a02_security_misconfiguration.py`, 165 lines, two scenarios; explained in Section IV-B.

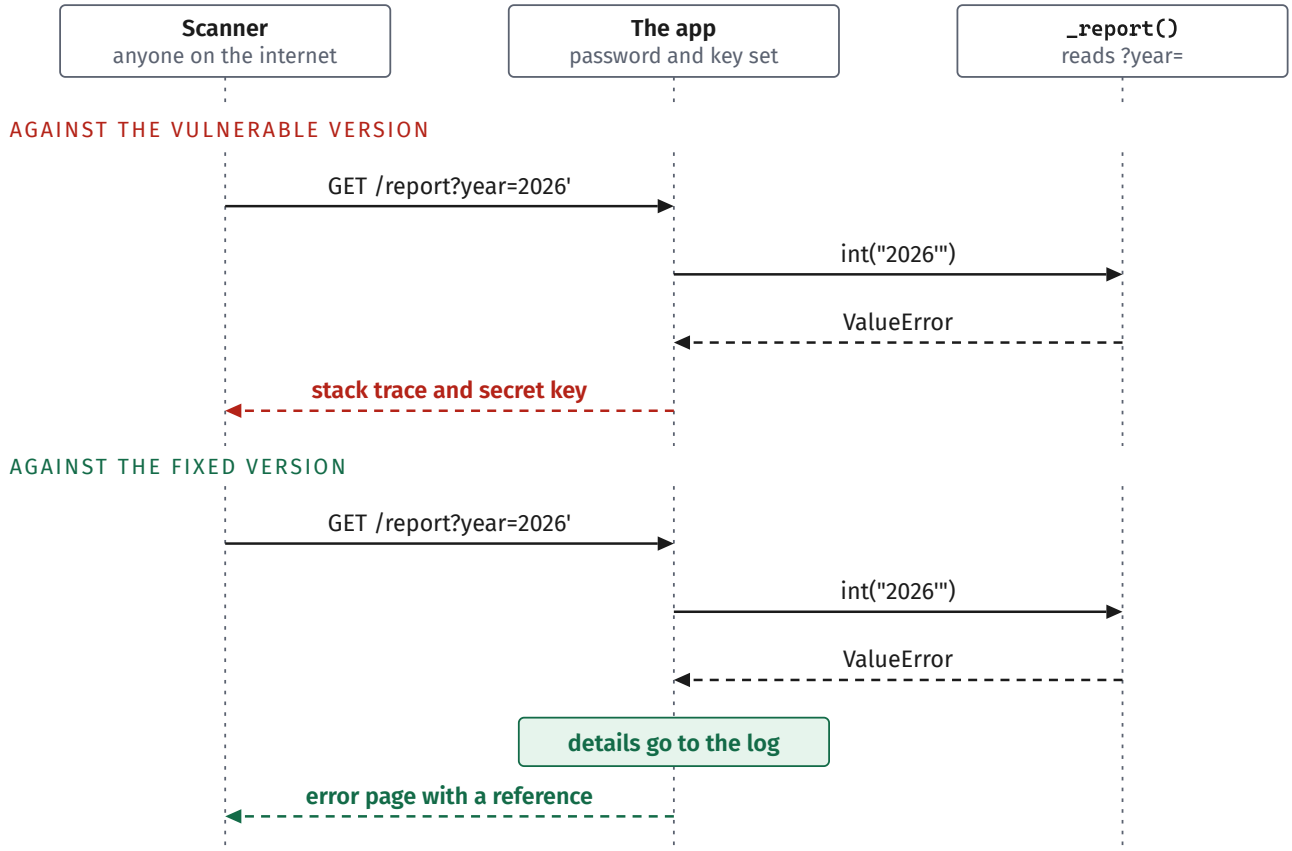


Fig. A3. A02, scenario 1: probe a configured deployment. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a02_security_misconfiguration.py`. Red: where the attack succeeds. Green: where the fix stops it.

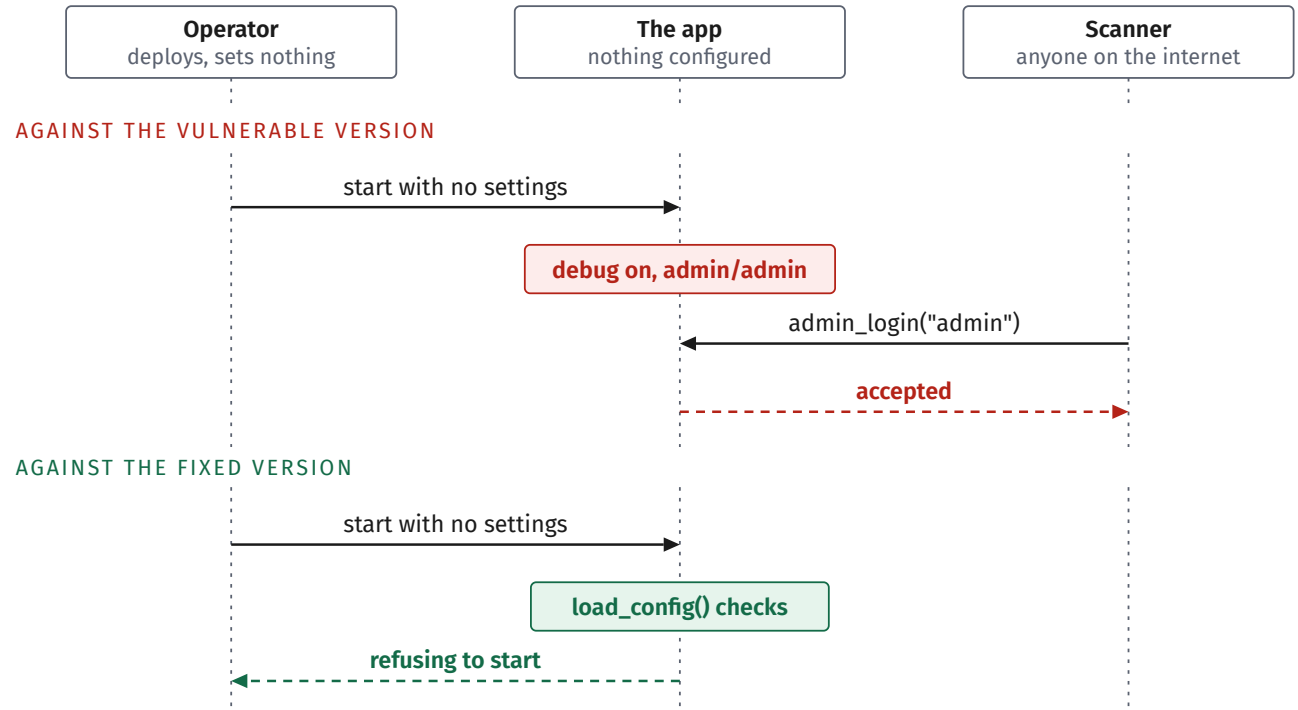


Fig. A4. A02, scenario 2: deploy with nothing configured. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a02_security_misconfiguration.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 8. A02: the vulnerable version

```python
class VulnerableApp:
    def __init__(self, env: dict) -> None:
        # Every setting falls back to a development value when nobody sets it.
        self.debug = env.get("APP_DEBUG", "1") == "1"                  # on unless switched off
        self.admin_password = env.get("APP_ADMIN_PASSWORD", "admin")    # account from the install guide
        self.secret_key = env.get("APP_SECRET_KEY", "dev-secret-key")   # same in every installation

    def handle(self, path: str, query: dict) -> tuple[int, dict, str]:
        try:
            return 200, {}, ROUTES[path](query)
        except Exception:
            if self.debug:
                # Stack trace and settings are sent to whoever caused the error.
                return 500, {}, traceback.format_exc() + f"\nsettings: {vars(self)}"
            return 500, {}, "error"

    def admin_login(self, password: str) -> bool:
        return password == self.admin_password
```

Listing 9. A02: the attack

```python
def probe(app) -> list[str]:
    """What a scanner tries against any deployed application."""
    findings = []
    if app.admin_login("admin"):
        findings.append("admin/admin accepted")
    status, headers, body = app.handle("/report", {"year": "2026'"})
    if "Traceback" in body:
        findings.append("stack trace in the error page")
    if "secret_key" in body or GOOD_ENV["APP_SECRET_KEY"] in body:
        findings.append("secret key in the error page")
    if "admin_password" in body:
        findings.append("admin password in the error page")
    if "Content-Security-Policy" not in headers:
```

```
        findings.append("no security headers")
    return findings


def attack_configured_app(app_class) -> Result:
    """The operator set a password and a key, and left everything else alone."""
    findings = probe(app_class(GOOD_ENV))
    return Result(bool(findings), "; ".join(findings) if findings else "nothing exposed; the error page gives only a reference
     ↪ number")


def attack_unconfigured_app(app_class) -> Result:
    """The operator deployed without setting anything."""
    try:
        app = app_class({})
    except RuntimeError as refusal:
        return Result(False, str(refusal))
    findings = probe(app)
    return Result(bool(findings), "started anyway; " + "; ".join(findings) if findings else "started with safe settings; nothing
     ↪ exposed")
```

Listing 10. A02: the fixed version

```
SECURITY_HEADERS = {
    "Content-Security-Policy": "default-src 'self'",
    "Strict-Transport-Security": "max-age=31536000; includeSubDomains",
    "X-Content-Type-Options": "nosniff",
}
KNOWN_DEFAULTS = {"", "admin", "password", "changeme"}


@dataclass(frozen=True)
class Config:
    debug: bool
    admin_password: str
    secret_key: str


def load_config(env: dict) -> Config:
    """Read settings from the environment and refuse to start on unsafe ones."""
    config = Config(
        debug=env.get("APP_DEBUG", "0") == "1",           # off unless asked for
        admin_password=env.get("APP_ADMIN_PASSWORD", ""),
        secret_key=env.get("APP_SECRET_KEY", ""),
    )
    problems = []
    if config.debug and env.get("APP_ENV", "production") == "production":
        problems.append("debug is on in production")
    if config.admin_password in KNOWN_DEFAULTS:
        problems.append("admin password is missing or a known default")
    if len(config.secret_key) < 32:
        problems.append("secret key is missing or shorter than 32 characters")
    if problems:
        raise RuntimeError("refusing to start: " + "; ".join(problems))
    return config


class FixedApp:
    def __init__(self, env: dict) -> None:
        self.config = load_config(env)    # an unsafe deployment fails here, loudly

    def handle(self, path: str, query: dict) -> tuple[int, dict, str]:
        try:
            return 200, SECURITY_HEADERS, ROUTES[path](query)
        except Exception:
            incident = uuid.uuid4().hex[:8]
            log.error("incident %s on %r", incident, path, exc_info=True)  # details stay on the server; %r escapes line breaks
            return 500, SECURITY_HEADERS, f"Something went wrong (reference {incident})."

    def admin_login(self, password: str) -> bool:
        # Compare bytes, because compare_digest raises an error for non-ASCII text.
        return hmac.compare_digest(password.encode(), self.config.admin_password.encode())
```

Listing 11. A02: output of `python -m demos.a02_security_misconfiguration`

```
A02:2025 Security Misconfiguration
  scenario: probe a configured deployment
    vulnerable ATTACK SUCCEEDED stack trace in the error page; secret key in the error page; admin password in the error page;
     ↪ no security headers
    fixed      attack blocked   nothing exposed; the error page gives only a reference number
    normal use still works with the fix: yes
  scenario: deploy with nothing configured
    vulnerable ATTACK SUCCEEDED started anyway; admin/admin accepted; stack trace in the error page; secret key in the error
     ↪ page; admin password in the error page; no security headers
    fixed      attack blocked   refusing to start: admin password is missing or a known default; secret key is missing or
     ↪ shorter than 32 characters
    normal use still works with the fix: yes
```

OWASP's advice, summarized [5]:

- Harden with a repeatable, automated process.
- Same configuration in every environment, different credentials in each.
- Install nothing that is not used; send security headers.
- Intercept excessive error messages centrally.
- Verify the configuration automatically and regularly.

C. A03:2025 Software Supply Chain Failures

File `demos/a03_supply_chain_failures.py`, 126 lines, two scenarios; explained in Section IV-C.

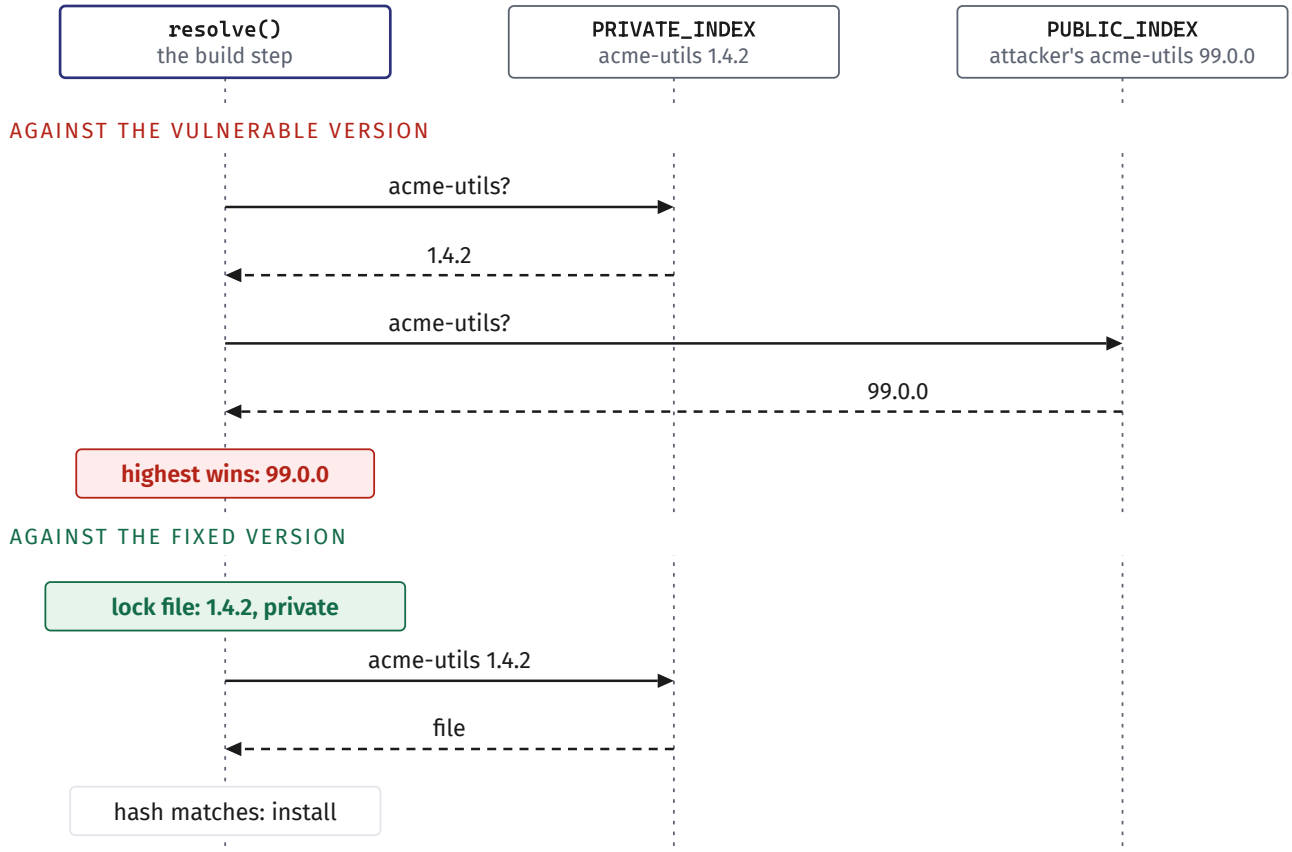


Fig. A5. A03, scenario 1: dependency confusion. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a03_supply_chain_failures.py`. Red: where the attack succeeds. Green: where the fix stops it.

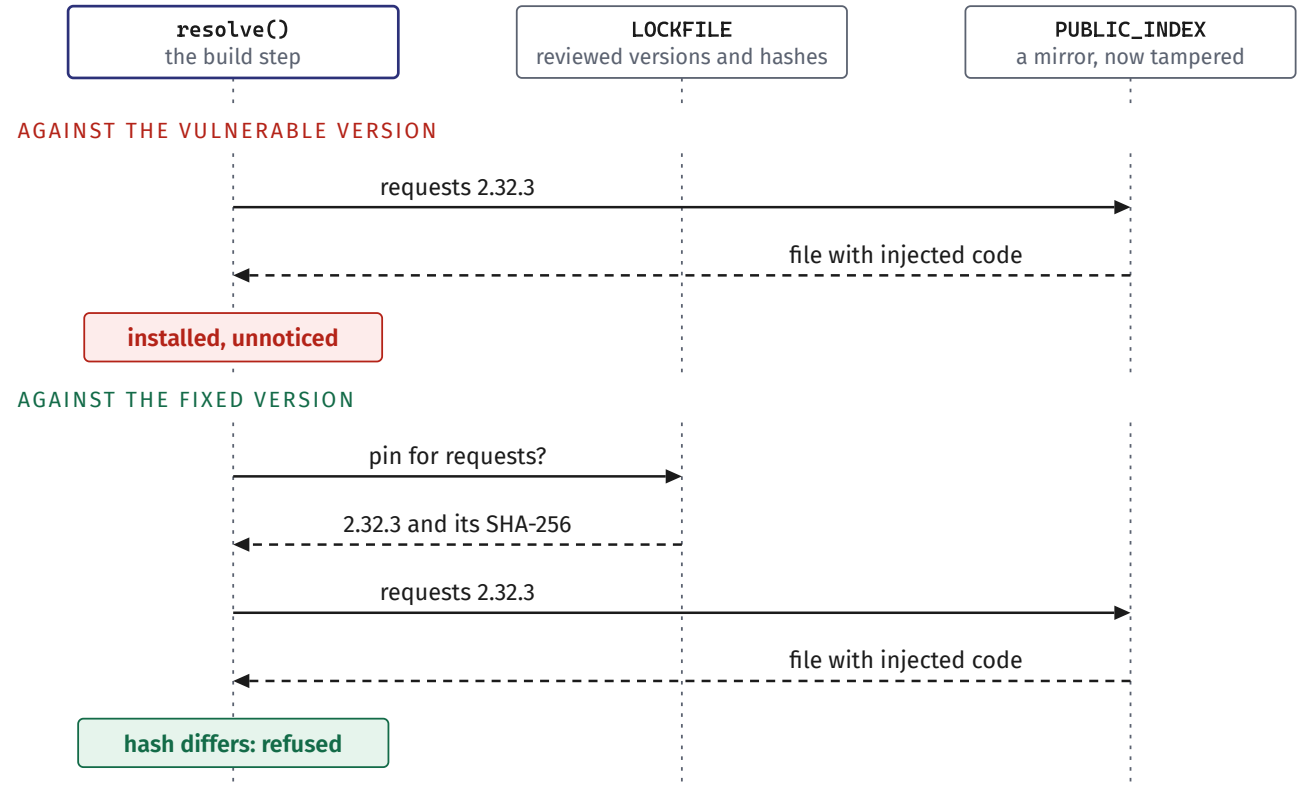


Fig. A6. A03, scenario 2: tampered file on a mirror. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a03_supply_chain_failures.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 12. A03: the vulnerable version

```python
def resolve_vulnerable(name: str) -> tuple[str, bytes]:
    # Both indexes are searched and the highest version wins, wherever it comes from.
    candidates = {**PRIVATE_INDEX.get(name, {}), **PUBLIC_INDEX.get(name, {})}
    version = max(candidates, key=_version_key)
    return version, candidates[version]  # installed as received
```

Listing 13. A03: the attack

```python
# The attacker learned the internal package name and published it with a higher version.
PUBLIC_INDEX["acme-utils"] = {"99.0.0": b"send_environment_to('attacker.example')  # attacker code"}


@contextmanager
def tampered_mirror():
    """Replace the genuine requests file for a while, as a compromised mirror would."""
    PUBLIC_INDEX["requests"]["2.32.3"] = GENUINE_REQUESTS + b" + injected code"
    try:
        yield
    finally:
        PUBLIC_INDEX["requests"]["2.32.3"] = GENUINE_REQUESTS


def attack_dependency_confusion(resolve) -> Result:
    """Install the internal package and see whose code arrives."""
    version, artifact = resolve("acme-utils")
    hijacked = artifact != GENUINE_ACME_UTILS
    return Result(hijacked, f"acme-utils resolved to {version}" + (" from the public index (attacker's file)" if hijacked else "
     ↪ from the private index"))


def attack_tampered_artifact(resolve) -> Result:
```

```
    """Change the file on the mirror after the version was approved."""
    with tampered_mirror():
        try:
            version, artifact = resolve("requests")
        except SupplyChainError as refusal:
            return Result(False, str(refusal))
    return Result(artifact != GENUINE_REQUESTS, f"requests {version} installed with {len(artifact) - len(GENUINE_REQUESTS)}
     ↪ extra bytes, unnoticed")
```

Listing 14. A03: the fixed version

```
# Lock file: regenerated and reviewed at every update, then committed. Each entry pins
# the exact version, the index it must come from, and the SHA-256 of the file.
LOCKFILE = {
    "acme-utils": {
        "version": "1.4.2",
        "index": "private",
        "sha256": "dafdc5fb6f81525ffb4f19a830199062c1bba65131965b0baaa2b2246eaa421a",
    },
    "requests": {
        "version": "2.32.3",
        "index": "public",
        "sha256": "be72a9fc3bcabe33139c25a1a5a0f8ae5df533896b0f045cbd66ab0cd91f9995",
    },
}


INDEXES = {"private": PRIVATE_INDEX, "public": PUBLIC_INDEX}


def resolve_fixed(name: str) -> tuple[str, bytes]:
    pin = LOCKFILE.get(name)
    if pin is None:
        raise SupplyChainError(f"{name} is not in the lock file")
    # Only the pinned version, only from the pinned index; anything else is refused.
    artifact = INDEXES.get(pin["index"], {}).get(name, {}).get(pin["version"])
    if artifact is None:
        raise SupplyChainError(f"{name} {pin['version']} is not on the {pin['index']} index")
    if hashlib.sha256(artifact).hexdigest() != pin["sha256"]:
        raise SupplyChainError(f"{name} {pin['version']}: file does not match the locked hash")
    return pin["version"], artifact
```

Listing 15. A03: output of `python -m demos.a03_supply_chain_failures`

```
A03:2025 Software Supply Chain Failures
  scenario: dependency confusion
    vulnerable ATTACK SUCCEEDED acme-utils resolved to 99.0.0 from the public index (attacker's file)
    fixed      attack blocked   acme-utils resolved to 1.4.2 from the private index
    normal use still works with the fix: yes
  scenario: tampered file on a mirror
    vulnerable ATTACK SUCCEEDED requests 2.32.3 installed with 16 extra bytes, unnoticed
    fixed      attack blocked   requests 2.32.3: file does not match the locked hash
    normal use still works with the fix: yes
```

OWASP's advice, summarized [6]:

- Keep a software bill of materials, transitive dependencies included.
- Choose dependency versions deliberately; upgrade only when there is need.
- Take components from official sources over secure links; prefer signed packages.
- Monitor vulnerability feeds and scan dependencies.

D. A04:2025 Cryptographic Failures

File `demos/a04_cryptographic_failures.py`, 149 lines, one scenario; explained in Section IV-D.

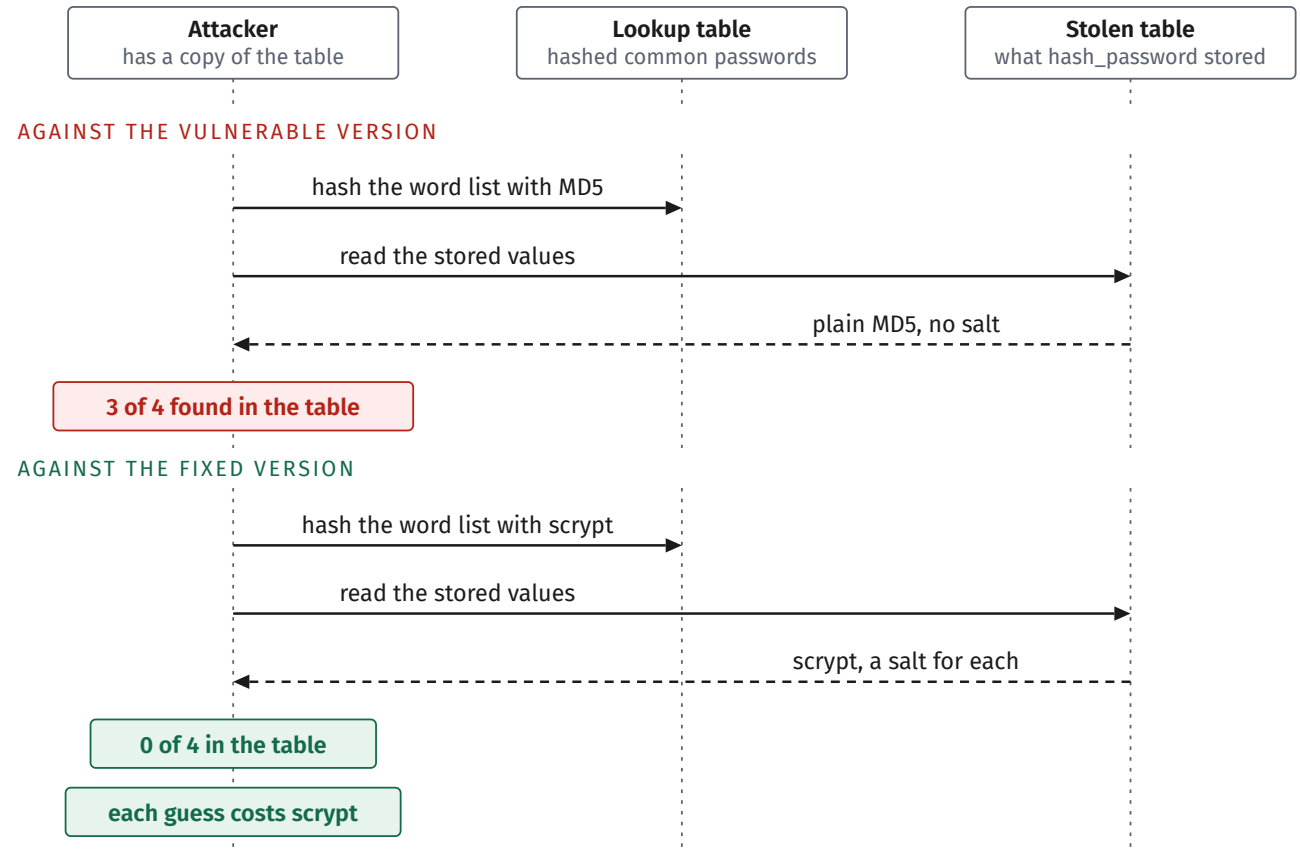


Fig. A7. A04, scenario 1: stolen password table. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a04_cryptographic_failures.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 16. A04: the vulnerable version

```python
def hash_password_vulnerable(password: str) -> str:
    # MD5 is fast and there is no salt: equal passwords give equal hashes,
    # and one precomputed table works against every user and every site.
    return hashlib.md5(password.encode()).hexdigest()


def verify_password_vulnerable(password: str, stored: str) -> bool:
    return hash_password_vulnerable(password) == stored
```

Listing 17. A04: the attack

```python
MIN_GUESS_SECONDS = 0.001   # a scheme that hashes faster than this can be brute-forced cheaply


def attack_stolen_table(scheme) -> Result:
    """The attacker has a copy of the password table. Three things make it easy to use."""
    hash_password, _ = scheme
    start = time.perf_counter()
    stolen = {user: hash_password(password) for user, password in USERS.items()}
    seconds_per_guess = (time.perf_counter() - start) / len(USERS)

    # 1. A table of common passwords, hashed once with the same function and reused
    #    for every stolen database. A salt makes every entry useless.
    lookup = {hash_password(word): word for word in WORDLIST}
    cracked = {user: lookup[stored] for user, stored in stolen.items() if stored in lookup}
    # 2. No salt: equal passwords are stored identically, so such a table can always be built.
```

```
    shared = stolen["bob"] == stolen["carol"]
    # 3. A fast hash: even without a table, billions of guesses are affordable.
    fast = seconds_per_guess < MIN_GUESS_SECONDS

    evidence = f"{len(cracked)} of {len(stolen)} passwords read from the table"
    if cracked:
        evidence += " (" + ", ".join(f"{user}={password}" for user, password in cracked.items()) + ")"
    evidence += f"; bob and carol share a hash: {'yes' if shared else 'no'}"
    evidence += f"; one guess costs {'under 1 ms' if fast else 'more than 1 ms'}"
    return Result(bool(cracked) or shared or fast, evidence)
```

Listing 18. A04: the fixed version

```
def _stretch(scheme: str, params: str, password: str, salt: bytes) -> bytes:
    """Slow, salted key derivation: the opposite of a fast hash."""
    if scheme == "scrypt":
        n, r, p = (int(value) for value in params.split(","))
        return hashlib.scrypt(password.encode(), salt=salt, n=n, r=r, p=p, dklen=32)
    if scheme == "pbkdf2-sha256":
        return pbkdf2_sha256(password.encode(), salt, int(params))
    raise ValueError(f"unknown password scheme: {scheme}")


def hash_password_fixed(password: str) -> str:
    salt = secrets.token_bytes(16)              # new random salt for every password
    key = _stretch(SCHEME, PARAMS, password, salt)   # SCHEME, PARAMS: scrypt, n=16384, r=8, p=1
    # Scheme and parameters are stored with the hash, so they can be raised later
    # without locking out existing accounts.
    return f"{SCHEME}${PARAMS}${salt.hex()}${key.hex()}"


def verify_password_fixed(password: str, stored: str) -> bool:
    scheme, params, salt_hex, key_hex = stored.split("$")
    key = _stretch(scheme, params, password, bytes.fromhex(salt_hex))
    return hmac.compare_digest(key.hex(), key_hex)   # constant-time comparison
```

Listing 19. A04: output of `python -m demos.a04_cryptographic_failures`

```
A04:2025 Cryptographic Failures
  scenario: stolen password table
    vulnerable ATTACK SUCCEEDED 3 of 4 passwords read from the table (alice=skopje2026, bob=letmein, carol=letmein); bob and
     ↪ carol share a hash: yes; one guess costs under 1 ms
    fixed      attack blocked   0 of 4 passwords read from the table; bob and carol share a hash: no; one guess costs more than
     ↪ 1 ms
    normal use still works with the fix: yes
    cost of one guess on this machine, one call from Python: MD5 1.4 microseconds, scrypt 84 milliseconds
    the fix makes each guess about 62,000 times more expensive; a weak password is still guessable
```

OWASP's advice, summarized [7]:

- Classify data and keep only what is needed.
- Encrypt sensitive data at rest, and all data in transit with Transport Layer Security (TLS) 1.2 or later and HSTS.
- Store passwords with an adaptive, salted hash such as Argon2, yescrypt, scrypt or PBKDF2-HMAC-SHA-512.
- Avoid deprecated functions such as MD5 and SHA-1.

E. A05:2025 Injection

File `demos/a05_injection.py`, 80 lines, one scenario; explained in Section IV-E.

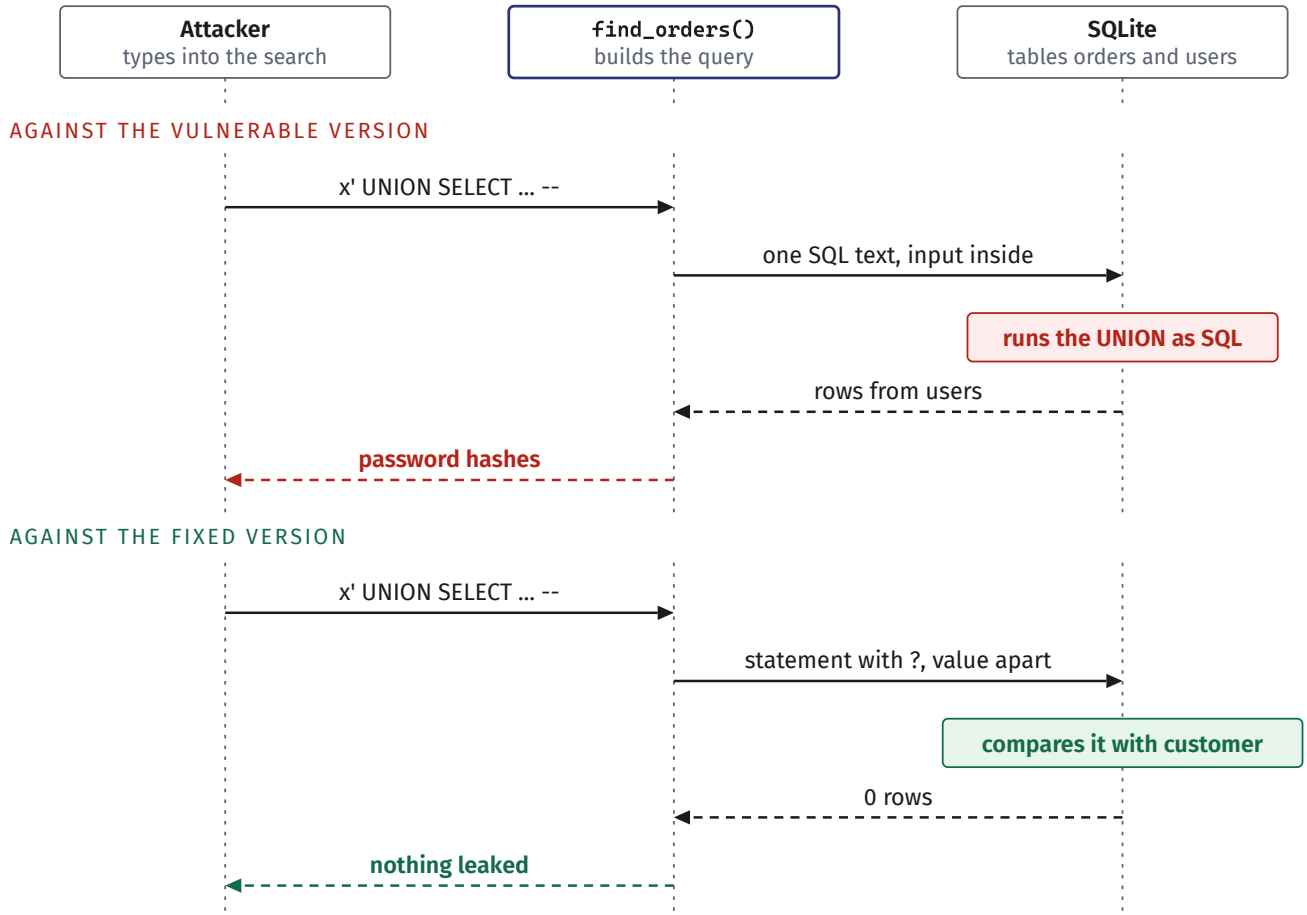


Fig. A8. A05, scenario 1: UNION-based SQL injection. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a05_injection.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 20. A05: the vulnerable version

```python
def find_orders_vulnerable(conn: sqlite3.Connection, customer: str) -> list[tuple]:
    # The input is pasted into the statement, so a quote in it ends the string
    # and everything after the quote is read as SQL.
    query = f"SELECT id, item FROM orders WHERE customer = '{customer}'"
    return conn.execute(query).fetchall()
```

Listing 21. A05: the attack

```python
PAYLOAD = "x' UNION SELECT username, password_hash FROM users --"


def attack_union(find_orders) -> Result:
    """Type SQL into the customer field to read a table the query never mentions."""
    rows = find_orders(make_db(), PAYLOAD)
    leaked = [row for row in rows if row[0] in ("admin", "alice")]
    if leaked:
        return Result(True, f"{len(leaked)} rows from the users table: {leaked}")
    return Result(False, f"{len(rows)} rows; the payload was treated as a customer name")
```

Listing 22. A05: the fixed version

```python
def find_orders_fixed(conn: sqlite3.Connection, customer: str) -> list[tuple]:
    # The statement is fixed text. The value travels separately and the database
    # only ever compares it with the column; it is never parsed as SQL.
    query = "SELECT id, item FROM orders WHERE customer = ?"
    return conn.execute(query, (customer,)).fetchall()
```

Listing 23. A05: output of `python -m demos.a05_injection`

```
A05:2025 Injection
  scenario: UNION-based SQL injection
    vulnerable ATTACK SUCCEEDED 2 rows from the users table: [('admin', 'scrypt$16384,8,1$5c0e$9f2c'), ('alice',
     ↪ 'scrypt$16384,8,1$a7d3$41ab')]
    fixed      attack blocked   0 rows; the payload was treated as a customer name
    normal use still works with the fix: yes
```

OWASP's advice, summarized [8]:

- Use a safe API that keeps data apart from commands, such as parameterized queries.
- Where that is impossible, validate input on the server against an allow-list; OWASP notes this is not a complete defense.
- Escape what remains with the interpreter's own syntax; escaping is error-prone.

F. A06:2025 Insecure Design

File `demos/a06_insecure_design.py`, 111 lines, one scenario; explained in Section IV-F.

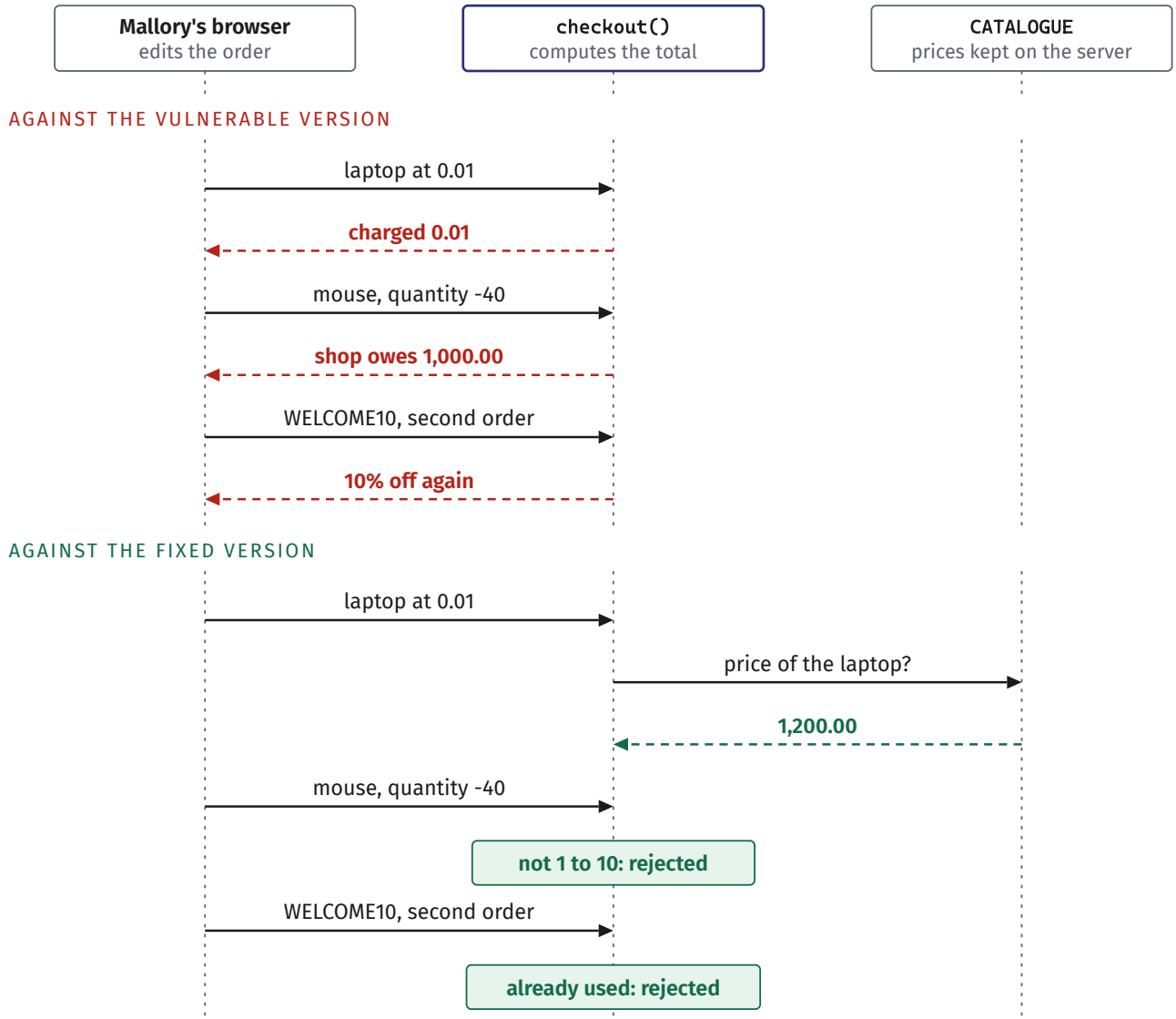


Fig. A9. A06, scenario 1: abuse the checkout rules. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a06_insecure_design.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 24. A06: the vulnerable version

```python
class VulnerableShop:
    def checkout(self, customer: str, order: dict) -> int:
        total = 0
        for line in order["lines"]:
            # The price comes back from the browser, and any quantity is accepted.
            total += line["unit_price_cents"] * line["quantity"]
        if order.get("coupon") in COUPONS:
            total -= total * COUPONS[order["coupon"]] // 100   # nothing records that it was used
        return total
```

Listing 25. A06: the attack

```python
def attack_business_logic(shop_class) -> Result:
    """Three misuse cases that a threat model of the checkout would list."""
    shop = shop_class()
    wins = []

    def try_order(order: dict):
        try:
            return shop.checkout("mallory", order)
        except OrderRejected:
            return None

    # 1. Edit the price field before the form is sent.
    paid = try_order({"lines": [{"sku": "laptop", "unit_price_cents": 1, "quantity": 1}]})
    if paid is not None and paid < CATALOGUE["laptop"]:
        wins.append(f"laptop charged at {paid / 100:.2f}")

    # 2. A negative quantity turns a purchase into a credit.
    paid = try_order({"lines": [{"sku": "mouse", "unit_price_cents": 2500, "quantity": -40}]})
    if paid is not None and paid < 0:
        wins.append(f"shop owes the customer {-paid / 100:.2f}")

    # 3. Use the single-use coupon on two orders.
    order = {"lines": [{"sku": "mouse", "unit_price_cents": 2500, "quantity": 1}], "coupon": "WELCOME10"}
    first, second = try_order(order), try_order(order)
    if first is not None and second is not None and second < CATALOGUE["mouse"]:
        wins.append("coupon accepted twice")

    return Result(bool(wins), "; ".join(wins) if wins else "price ignored, quantity and coupon rules enforced")
```

Listing 26. A06: the fixed version

```python
class FixedShop:
    MAX_PER_LINE = 10

    def __init__(self) -> None:
        self.used_coupons: set[tuple[str, str]] = set()

    def checkout(self, customer: str, order: dict) -> int:
        total = 0
        for line in order["lines"]:
            price = CATALOGUE.get(line["sku"])            # the server owns the price
            quantity = line["quantity"]
            if price is None or type(quantity) is not int or not 1 <= quantity <= self.MAX_PER_LINE:
                raise OrderRejected("invalid order line")
            total += price * quantity
        coupon = order.get("coupon")
        if coupon is not None:
            if coupon not in COUPONS or (customer, coupon) in self.used_coupons:
                raise OrderRejected("coupon not valid")
            self.used_coupons.add((customer, coupon))     # single use is part of the design
            total -= total * COUPONS[coupon] // 100
        return total
```

Listing 27. A06: output of `python -m demos.a06_insecure_design`

```
A06:2025 Insecure Design
  scenario: abuse the checkout rules
    vulnerable ATTACK SUCCEEDED laptop charged at 0.01; shop owes the customer 1000.00; coupon accepted twice
    fixed      attack blocked   price ignored, quantity and coupon rules enforced
    normal use still works with the fix: yes
```

OWASP's advice, summarized [9]:

- Threat-model critical flows: authentication, access control, business logic.
- Integrate plausibility checks at each tier, the server included.
- Write unit and integration tests for misuse cases.
- Reuse secure design patterns and vetted components.

G. A07:2025 Authentication Failures

File `demos/a07_authentication_failures.py`, 144 lines, one scenario; explained in Section IV-G.

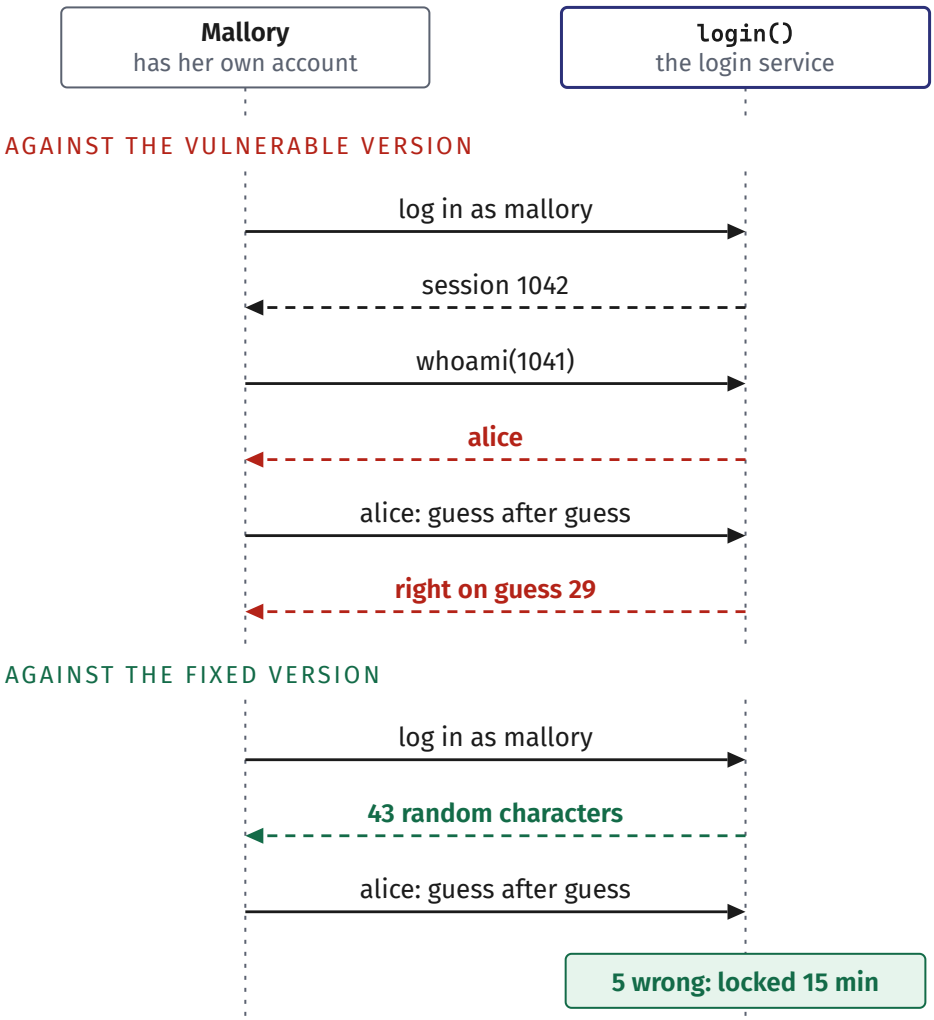


Fig. A10. A07, scenario 1: guess the session ID, then guess the password. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a07_authentication_failures.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 28. A07: the vulnerable version

```python
class VulnerableAuth:
    def __init__(self, users: dict) -> None:
        self.users = users
        self.sessions: dict[str, str] = {}
        self._counter = 1040

    def login(self, username: str, password: str) -> str | None:
        if not _password_ok(self.users, username, password):
            return None                     # guess again, as often as you like
        self._counter += 1
        token = str(self._counter)          # 1041, 1042, 1043 ...
        self.sessions[token] = username
        return token

    def whoami(self, token: str) -> str | None:
        return self.sessions.get(token)
```

Listing 29. A07: the attack

```
def attack_login(auth_class) -> Result:
    auth = auth_class(make_users())
    notes = []

    # 1. Session prediction. Alice is logged in; Mallory logs in after her.
    auth.login("alice", PASSWORDS["alice"])
    mine = auth.login("mallory", PASSWORDS["mallory"])
    if mine.isdigit() and auth.whoami(str(int(mine) - 1)) == "alice":
        notes.append(f"session {int(mine) - 1} guessed from {mine}: logged in as alice")

    # 2. Password guessing against Alice with a list of common passwords.
    guesses = 0
    try:
        for guess in WORDLIST:
            guesses += 1
            if auth.login("alice", guess):
                notes.append(f"password found after {guesses} guesses")
                break
    except TooManyAttempts:
        pass

    if notes:
        return Result(True, "; ".join(notes))
    return Result(False, f"the session token is not a number to count from; account locked after {guesses - 1} wrong guesses")
```

Listing 30. A07: the fixed version

```
class FixedAuth:
    MAX_FAILURES = 5
    LOCK_SECONDS = 15 * 60

    def __init__(self, users: dict, clock=time.monotonic) -> None:
        self.users = users
        self.sessions: dict[str, str] = {}
        self.clock = clock
        self._failures: dict[str, int] = {}
        self._locked_until: dict[str, float] = {}

    def login(self, username: str, password: str) -> str | None:
        now = self.clock()
        if self._locked_until.get(username, 0.0) > now:
            raise TooManyAttempts("try again later")      # same answer whatever the password
        if not _password_ok(self.users, username, password):
            self._failures[username] = self._failures.get(username, 0) + 1
            if self._failures[username] >= self.MAX_FAILURES:
                self._locked_until[username] = now + self.LOCK_SECONDS
                self._failures[username] = 0
            return None
        self._failures.pop(username, None)
        token = secrets.token_urlsafe(32)    # 256 random bits, new at every login
        self.sessions[token] = username
        return token

    def whoami(self, token: str) -> str | None:
        return self.sessions.get(token)
```

Listing 31. A07: output of `python -m demos.a07_authentication_failures`

```
A07:2025 Authentication Failures
  scenario: guess the session ID, then guess the password
    vulnerable ATTACK SUCCEEDED session 1041 guessed from 1042: logged in as alice; password found after 29 guesses
    fixed      attack blocked   the session token is not a number to count from; account locked after 5 wrong guesses
    normal use still works with the fix: yes
```

OWASP's advice, summarized [10]:

- Use multi-factor authentication.
- Ship no default credentials; check new passwords against breached lists.
- Limit or increasingly delay failed logins; log and alert on them.
- Issue a new random session ID at login; invalidate it at logout and after idle and absolute timeouts.

H. A08:2025 Software or Data Integrity Failures

File `demos/a08_integrity_failures.py`, 121 lines, one scenario; explained in Section IV-H.

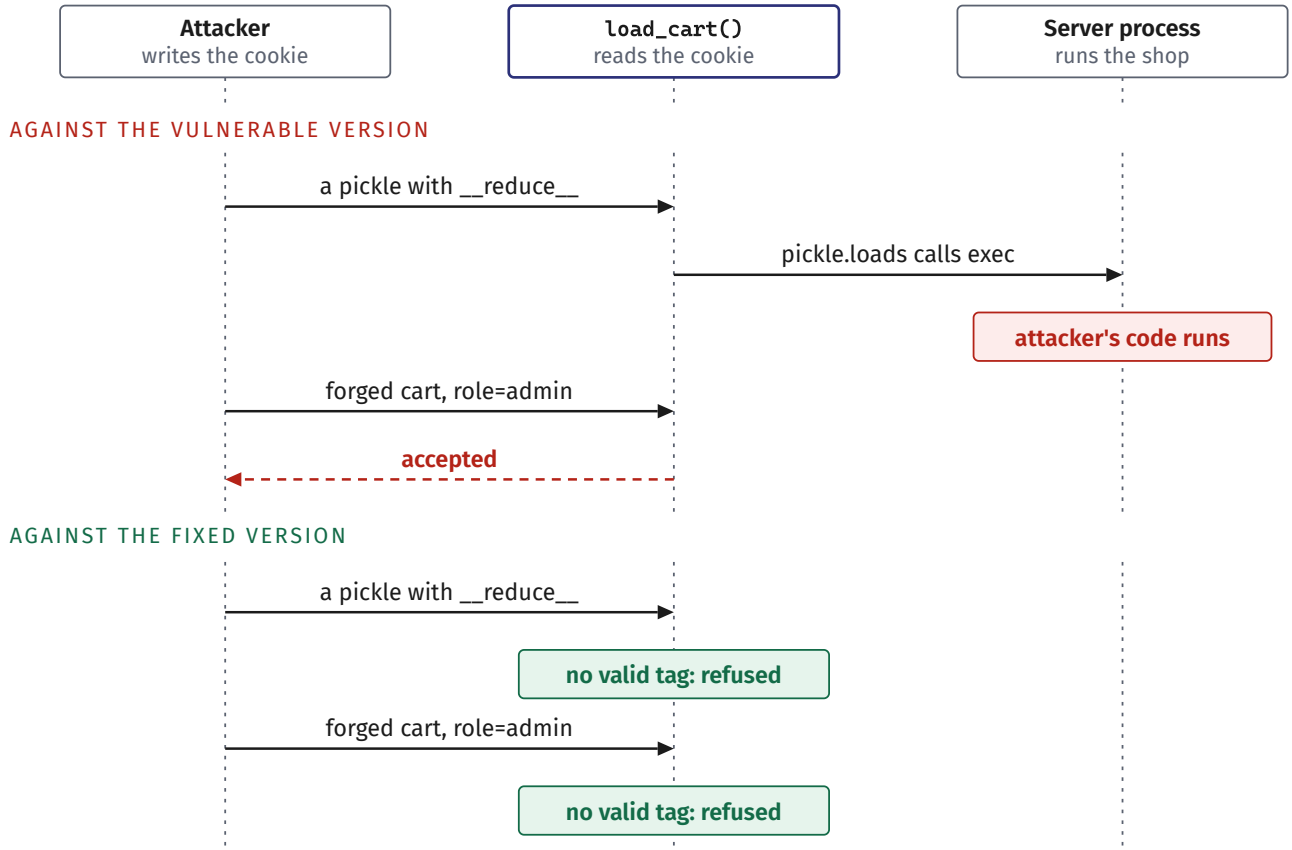


Fig. A11. A08, scenario 1: send a crafted cart cookie. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a08_integrity_failures.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 32. A08: the vulnerable version

```python
def dump_cart_vulnerable(cart: dict) -> str:
    return base64.b64encode(pickle.dumps(cart)).decode()


def load_cart_vulnerable(cookie: str) -> dict:
    # pickle rebuilds objects by calling whatever the bytes name.
    # The client wrote those bytes.
    return pickle.loads(base64.b64decode(cookie))
```

Listing 33. A08: the attack

```
class Payload:
    def __reduce__(self):
        # Harmless stand-in for attacker code: it sets a marker and nothing else.
        return (exec, (f"import builtins; builtins.{MARKER} = True",))


def attack_cookie(codec) -> Result:
    _, load_cart = codec
    notes = []

    # 1. Send a pickle that runs code when it is loaded.
    setattr(builtins, MARKER, False)
    try:
        load_cart(base64.b64encode(pickle.dumps(Payload())).decode())
    except Exception:
        pass
    if getattr(builtins, MARKER):
        notes.append("attacker's code ran inside the server process")
    delattr(builtins, MARKER)

    # 2. Write a cookie from scratch: a cheaper cart, and the role of an admin.
    forged = {"items": [{"sku": "laptop", "unit_price_cents": 1}], "role": "admin"}
    as_json = json.dumps(forged).encode()
    for cookie in (
        base64.b64encode(pickle.dumps(forged)).decode(),              # a pickle
        base64.b64encode(as_json).decode(),                           # plain JSON, no signature
        base64.urlsafe_b64encode(as_json).decode(),                   # the same, URL-safe
        base64.urlsafe_b64encode(as_json).decode() + "." + "0" * 64,  # JSON with a made-up signature
    ):
        try:
            if load_cart(cookie).get("role") == "admin":
                notes.append("forged cookie accepted with role=admin")
                break
        except Exception:
            continue

    return Result(bool(notes), "; ".join(notes) if notes else "pickles, unsigned and wrongly signed cookies all rejected")
```

Listing 34. A08: the fixed version

```
def dump_cart_fixed(cart: dict, key: bytes = SERVER_KEY) -> str:
    body = base64.urlsafe_b64encode(json.dumps(cart, separators=(",", ":")).encode()).decode()
    tag = hmac.new(key, body.encode(), hashlib.sha256).hexdigest()
    return f"{body}.{tag}"


def load_cart_fixed(cookie: str, key: bytes = SERVER_KEY) -> dict:
    body, _, tag = cookie.partition(".")
    expected = hmac.new(key, body.encode(), hashlib.sha256).hexdigest()
    # Compare bytes, because compare_digest raises an error for non-ASCII text,
    # and the client can put any text in the tag.
    if not hmac.compare_digest(tag.encode(), expected.encode()):   # verify first, parse second
        raise IntegrityError("cookie was not signed by this server")
    return json.loads(base64.urlsafe_b64decode(body))   # JSON holds data only, never code
```

Listing 35. A08: output of `python -m demos.a08_integrity_failures`

```
A08:2025 Software or Data Integrity Failures
  scenario: send a crafted cart cookie
    vulnerable ATTACK SUCCEEDED attacker's code ran inside the server process; forged cookie accepted with role=admin
    fixed      attack blocked   pickles, unsigned and wrongly signed cookies all rejected
    normal use still works with the fix: yes
```

OWASP's advice, summarized [11]:

- Verify digital signatures or similar integrity checks on software and data.
- Take packages from trusted repositories; consider a vetted internal mirror.
- Review code and configuration changes; protect the continuous integration and delivery (CI/CD) pipeline.
- Accept no unsigned serialized data from untrusted clients; check it for tampering and replay.

I. A09:2025 Security Logging and Alerting Failures

File `demos/a09_logging_alerting_failures.py`, 120 lines, one scenario; explained in Section IV-I.

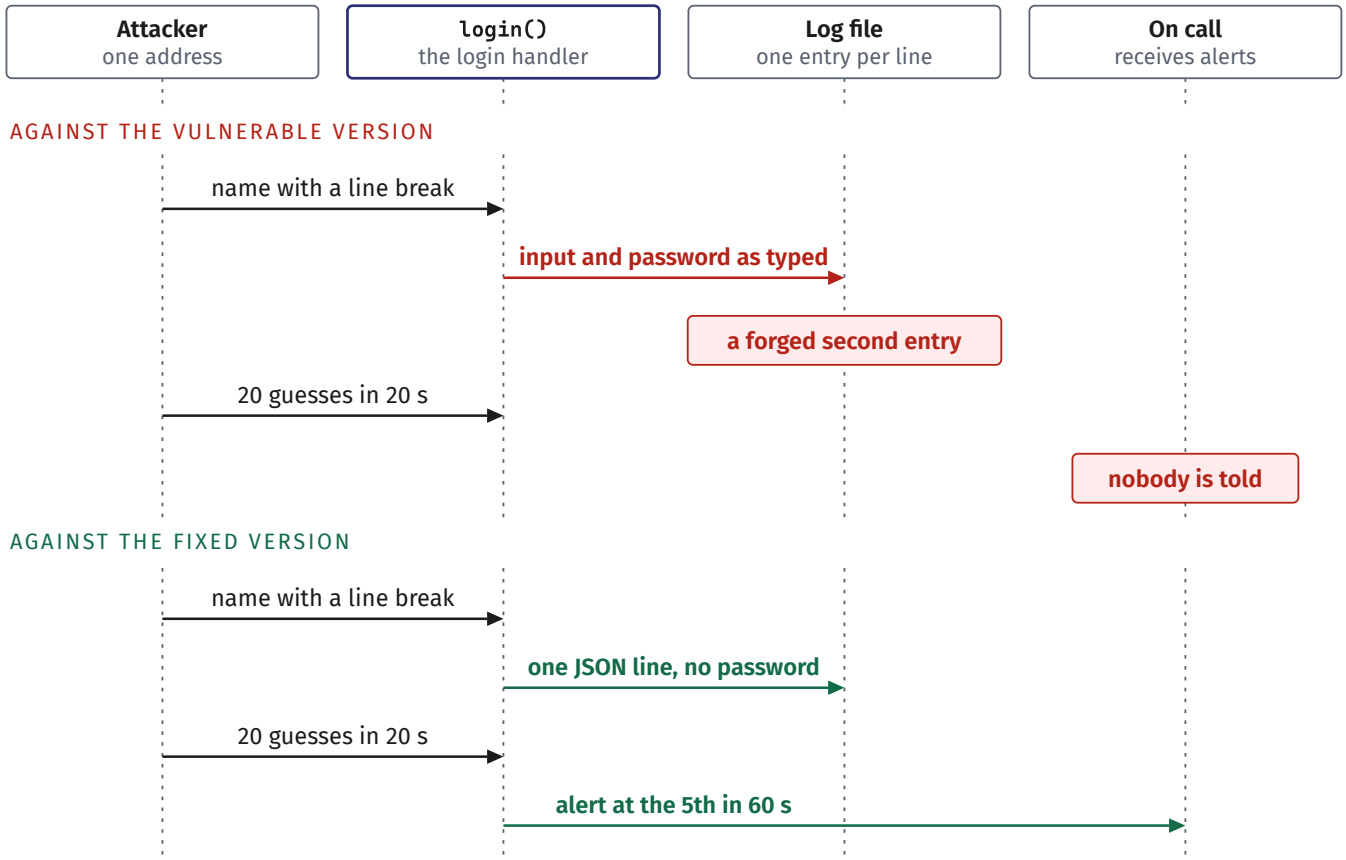


Fig. A12. A09, scenario 1: forge a log entry and brute-force unnoticed. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a09_logging_alerting_failures.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 36. A09: the vulnerable version

```python
class VulnerableLoginService:
    def __init__(self) -> None:
        self.log_lines: list[str] = []    # one entry per line, like a log file
        self.alerts: list[str] = []       # messages sent to whoever is on call

    def _write(self, text: str) -> None:
        self.log_lines.extend(text.split("\n"))   # a file starts a new line at every line break

    def login(self, ip: str, username: str, password: str, now: float = 0.0) -> bool:
        ok = _check(username, password)
        if not ok:
            # Input goes into the log as typed, the password is written down,
            # and however often this happens, nobody is told.
            self._write(f"{NOW} WARN login failed user={username} password={password}")
        return ok
```

Listing 37. A09: the attack

```python
def attack_cover_tracks(service_class) -> Result:
    service = service_class()
    attacker_ip = "203.0.113.7"

    # 1. A username with a line break in it, followed by a fake "success" entry.
    forged_name = "bob\n" + NOW + " INFO login ok user=admin"
    service.login(attacker_ip, forged_name, "x")

    # 2. Twenty password guesses in twenty seconds.
    for second in range(20):
        service.login(attacker_ip, "admin", f"guess-{second}", now=float(second))

    forged = [line for line in service.log_lines if line.startswith(NOW + " INFO login ok user=admin")]
    passwords = [line for line in service.log_lines if "password=" in line or "guess-" in line]
    evidence = (
        f"forged entries: {len(forged)}; log lines containing a typed password: {len(passwords)}; "
        f"alerts raised: {len(service.alerts)}"
    )
    return Result(bool(forged) or bool(passwords) or not service.alerts, evidence)
```

Listing 38. A09: the fixed version

```python
class FixedLoginService:
    THRESHOLD = 5     # failed logins ...
    WINDOW = 60       # ... within this many seconds, from one address

    def __init__(self) -> None:
        self.log_lines: list[str] = []
        self.alerts: list[str] = []
        self._failures: dict[str, list[float]] = {}

    def _event(self, kind: str, **fields) -> None:
        # One JSON object per line. json.dumps escapes line breaks,
        # so input cannot begin a new entry.
        self.log_lines.append(json.dumps({"ts": NOW, "event": kind, **fields}, sort_keys=True))

    def login(self, ip: str, username: str, password: str, now: float = 0.0) -> bool:
        ok = _check(username, password)
        # Successes and failures are both recorded, with who and from where.
        # The password is never written.
        self._event("login_ok" if ok else "login_failed", ip=ip, user=username)
        if not ok:
            recent = [t for t in self._failures.get(ip, []) if now - t < self.WINDOW] + [now]
            self._failures[ip] = recent
            if len(recent) == self.THRESHOLD:
                self._event("alert", rule="repeated_login_failures", ip=ip, count=len(recent))
                self.alerts.append(f"{len(recent)} failed logins from {ip} within {self.WINDOW} s")
        return ok
```

Listing 39. A09: output of `python -m demos.a09_logging_alerting_failures`

```
A09:2025 Security Logging and Alerting Failures
  scenario: forge a log entry and brute-force unnoticed
    vulnerable ATTACK SUCCEEDED forged entries: 1; log lines containing a typed password: 21; alerts raised: 0
    fixed      attack blocked   forged entries: 0; log lines containing a typed password: 0; alerts raised: 1
    normal use still works with the fix: yes
```

OWASP's advice, summarized [12]:

- Log login, access-control and validation failures with enough user context.
- Encode log data so that it cannot be injected.
- Protect audit trails from tampering, for example with append-only tables.
- Define alerting use cases and response playbooks.

J. A10:2025 Mishandling of Exceptional Conditions

File `demos/a10_exceptional_conditions.py`, 117 lines, one scenario; explained in Section IV-J.

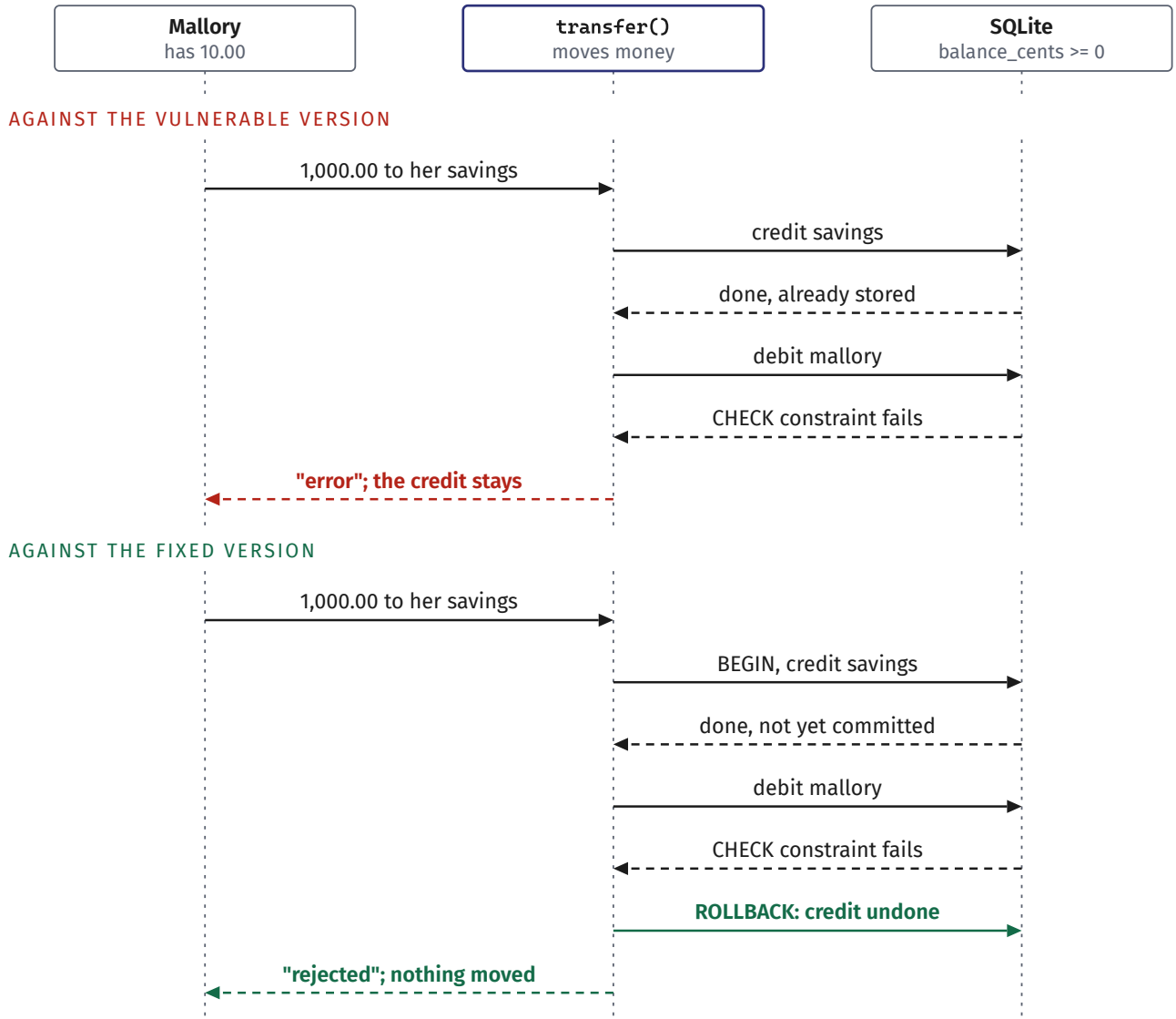


Fig. A13. A10, scenario 1: make a transfer fail halfway. The same attack against the vulnerable version (upper half) and the fixed version (lower half) of `demos/a10_exceptional_conditions.py`. Red: where the attack succeeds. Green: where the fix stops it.

Listing 40. A10: the vulnerable version

```python
def transfer_vulnerable(conn: sqlite3.Connection, source: str, target: str, cents: int) -> str:
    try:
        conn.execute("UPDATE accounts SET balance_cents = balance_cents + ? WHERE owner = ?", (cents, target))
        conn.execute("UPDATE accounts SET balance_cents = balance_cents - ? WHERE owner = ?", (cents, source))
        return "ok"
    except Exception:
        # The debit failed (not enough money), but the credit above is already stored.
        # Swallowing the error hides that the two halves no longer match.
        return "error"
```

Listing 41. A10: the attack

```python
def attack_partial_failure(transfer) -> Result:
    """Mallory has 10.00 and sends 1,000.00 to her own savings account, three times."""
    conn = make_bank()
    before = total(conn)
    answers = [transfer(conn, "mallory", "mallory_savings", 100000) for _ in range(3)]
    after = total(conn)
    created = after - before
    evidence = (
        f"answers: {answers}; money in the bank {before / 100:,.2f} -> {after / 100:,.2f}; "
        f"mallory_savings = {balance(conn, 'mallory_savings') / 100:,.2f}"
    )
    return Result(created != 0, evidence)
```

Listing 42. A10: the fixed version

```python
def transfer_fixed(conn: sqlite3.Connection, source: str, target: str, cents: int) -> str:
    if type(cents) is not int or cents <= 0:
        return "rejected"
    try:
        conn.execute("BEGIN IMMEDIATE")            # both updates, or neither
        credited = conn.execute(                   # same order as before: the credit comes first
            "UPDATE accounts SET balance_cents = balance_cents + ? WHERE owner = ?", (cents, target)
        ).rowcount
        debited = conn.execute(
            "UPDATE accounts SET balance_cents = balance_cents - ? WHERE owner = ?", (cents, source)
        ).rowcount
        if debited != 1 or credited != 1:
            raise LookupError("unknown account")
        conn.execute("COMMIT")
        return "ok"
    except Exception as error:
        if conn.in_transaction:
            conn.execute("ROLLBACK")               # undo the credit that already happened
        if isinstance(error, (sqlite3.IntegrityError, LookupError)):
            log.warning("transfer refused: %s", type(error).__name__)
            return "rejected"                      # expected failure: closed, no detail to the caller
        raise                                      # unexpected failure: let the global handler log and alert
```

Listing 43. A10: output of `python -m demos.a10_exceptional_conditions`

```
A10:2025 Mishandling of Exceptional Conditions
  scenario: make a transfer fail half-way
    vulnerable ATTACK SUCCEEDED answers: ['error', 'error', 'error']; money in the bank 510.00 -> 3,510.00; mallory_savings =
     ↪ 3,000.00
    fixed      attack blocked   answers: ['rejected', 'rejected', 'rejected']; money in the bank 510.00 -> 510.00;
     ↪ mallory_savings = 0.00
    normal use still works with the fix: yes
```

OWASP's advice, summarized [13]:

- Catch errors where they occur and handle them meaningfully.
- Roll back incomplete transactions; fail closed.
- Use one central exception handler; log and alert.
- Apply rate limits and quotas; validate input strictly.

APPENDIX B
THE RISK-SCORE COMPUTATION

Table IX applies (1) to the score table of each category and of the two near misses that have one. Rates are taken as fractions.

TABLE IX
OWASP'S RANKING FORMULA APPLIED TO THE PUBLISHED SCORE TABLES, TERM BY TERM

Category	Basis	Max incidence $\times 1000$	Max coverage $\times 100$	Exploit $\times 10$	Impact $\times 20$	Occurrences $\div 10\,000$	Score
A01 Broken Access Control	data	201.50	100.00	70.40	76.80	183.97	**632.67**
A02 Security Misconfiguration	data	277.00	100.00	79.60	79.40	71.91	**607.91**
A03 Software Supply Chain Failures	survey	95.60	65.42	81.70	104.60	21.52	**368.84**
A04 Cryptographic Failures	data	137.70	100.00	72.30	78.00	166.53	**554.53**
A05 Injection	data	137.70	100.00	71.50	86.40	140.42	**536.02**
A06 Insecure Design	data	221.80	88.76	69.60	81.00	72.99	**534.15**
A07 Authentication Failures	data	158.00	100.00	76.90	88.80	112.07	**535.77**
A08 Software or Data Integrity Failures	data	89.80	78.52	71.10	95.80	50.13	**385.35**
A09 Security Logging and Alerting Failures	survey	113.30	85.96	71.90	53.00	26.03	**350.19**
A10 Mishandling of Exceptional Conditions	data	206.70	100.00	71.10	76.20	76.96	**530.96**
X01 Lack of Application Resilience	not listed	200.50	86.01	79.20	69.80	86.51	**522.02**
X02 Memory Management Failures	not listed	29.60	55.62	67.50	96.40	22.04	**271.16**

Appendix C
Screenshots of the Lab

The screenshots in Figs. C1–C3 were taken on 26 November 2026 in Microsoft Edge 158 at 1,280 pixels wide by `report/screenshots.mjs`, which puts the lab into each state and checks the page before it takes the picture. All but the last show the A05 lesson.

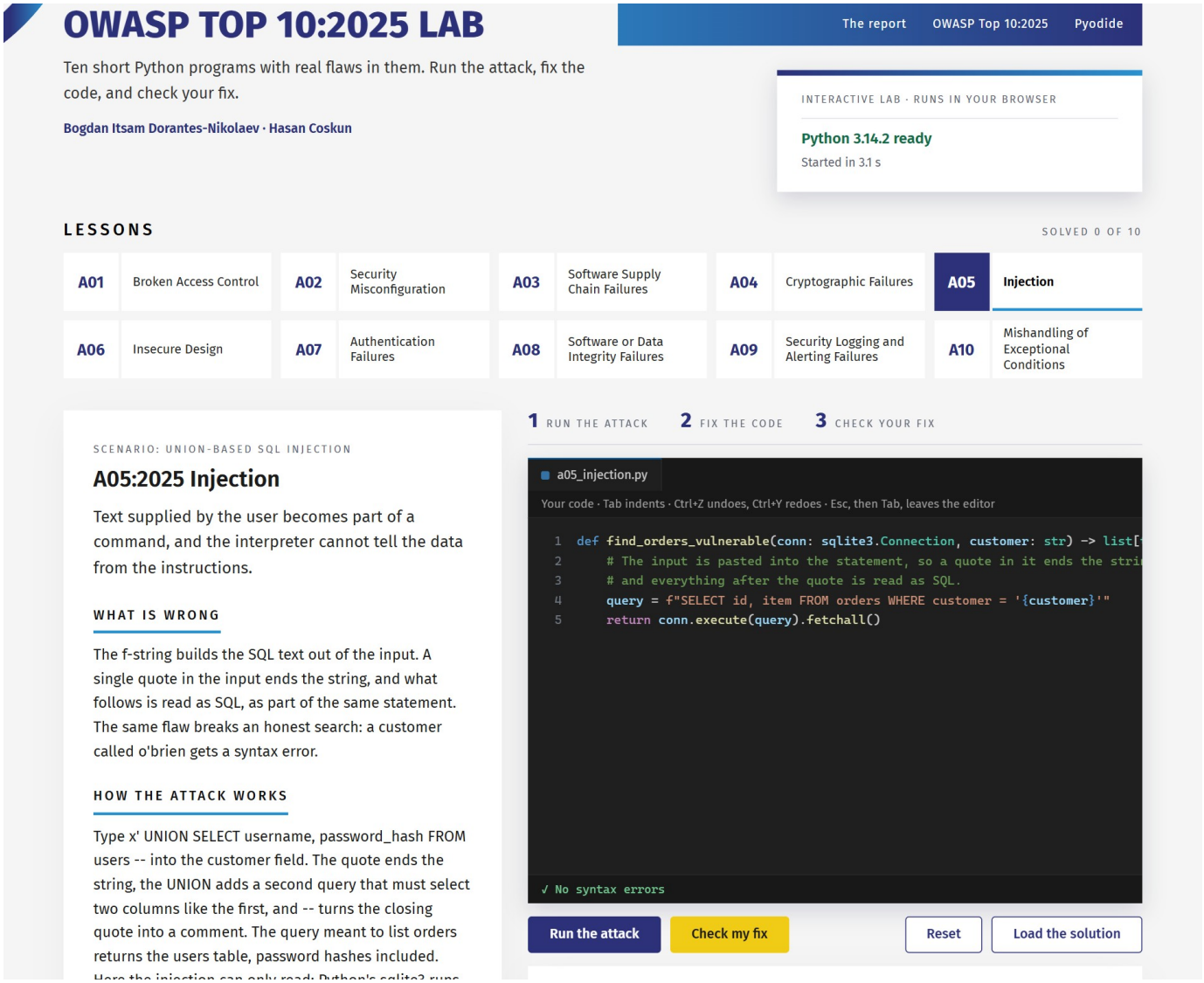
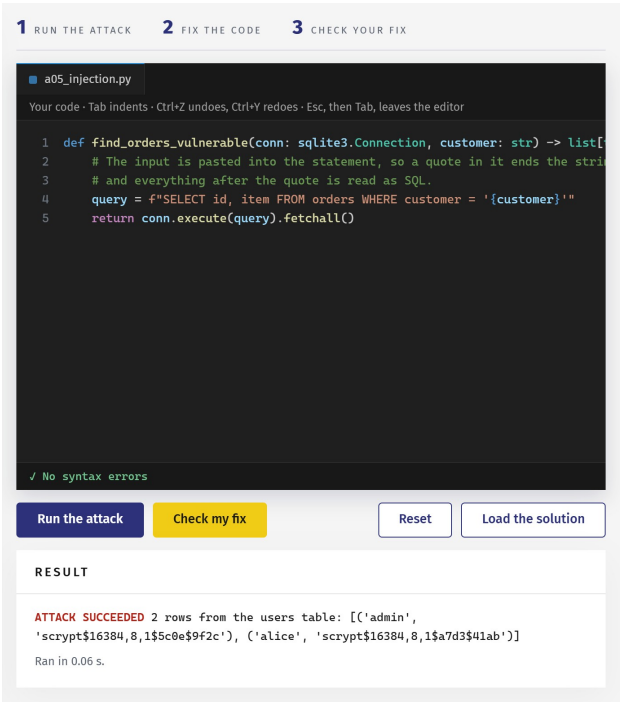
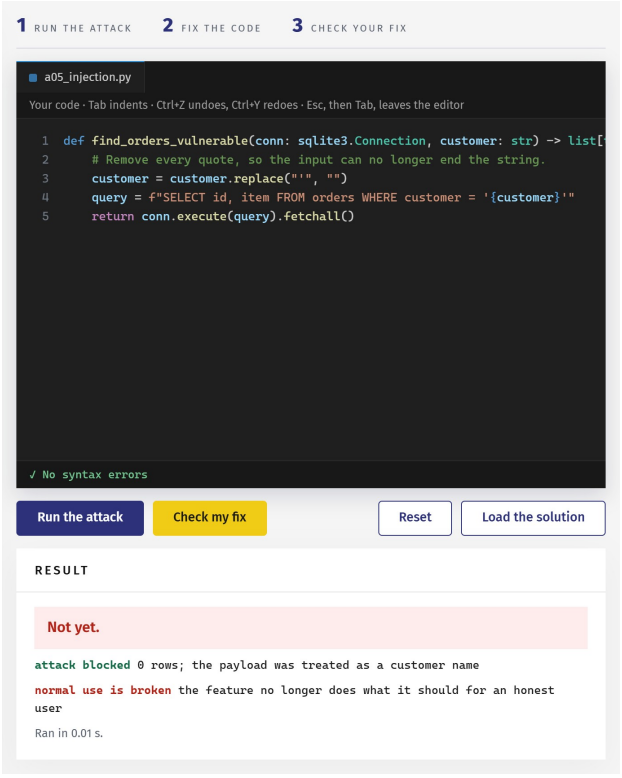


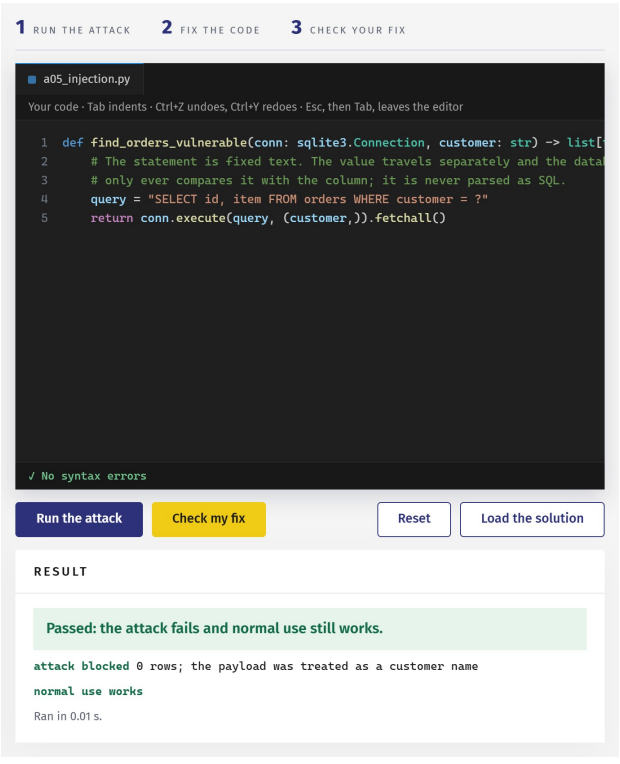
Fig. C1. The A05 lesson as it opens. The lesson explains the flaw on the left; the editor on the right holds the vulnerable function from `demos/a05_injection.py`, unchanged. The start-up time is a second visit, with the Python runtime in the browser's cache; a first visit downloads about 6 MB.



(a) Run the attack: it works against the starting code

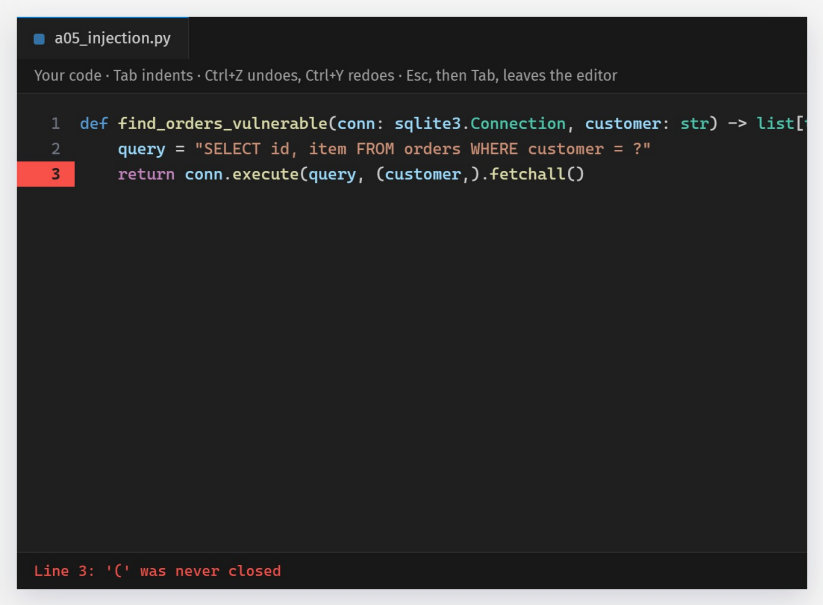


(b) A fix that breaks normal use is refused



(c) The parameterized query passes

Fig. C2. (a) Run the attack on the starting code. The UNION payload returns two rows of the users table, password hashes included. (b) A tempting half-fix: delete every quote. The attack is blocked, but the check still says no, because the search for the customer `o'brien` now returns nothing. The check runs the attack and the normal-use test every time. (c) The fix from `demos/a05_injection.py`: a constant statement with a ? placeholder. The attack is blocked and normal use still works, so the check passes.



(a) Errors are found while typing

Do all ten demos run in this browser?

Each lesson is built on a complete demo program with a vulnerable version, a fixed version and an attack. This check runs all ten of them, unchanged, in the same in-browser Python.

Run all ten — 10 of 10 ran with the expected result.

DEMO	RESULT	SECONDS	DETAIL
A01:2025 Broken Access Control	runs	0.10	attack works on the vulnerable version and fails on the fixed one
A02:2025 Security Misconfiguration	runs	0.04	attack works on the vulnerable version and fails on the fixed one
A03:2025 Software Supply Chain Failures	runs	0.01	attack works on the vulnerable version and fails on the fixed one
A04:2025 Cryptographic Failures	runs	3.98	attack works on the vulnerable version and fails on the fixed one
A05:2025 Injection	runs	0.01	attack works on the vulnerable version and fails on the fixed one
A06:2025 Insecure Design	runs	0.01	attack works on the vulnerable version and fails on the fixed one
A07:2025 Authentication Failures	runs	0.76	attack works on the vulnerable version and fails on the fixed one
A08:2025 Software or Data Integrity Failures	runs	0.08	attack works on the vulnerable version and fails on the fixed one
A09:2025 Security Logging and Alerting Failures	runs	0.01	attack works on the vulnerable version and fails on the fixed one
A10:2025 Mishandling of Exceptional Conditions	runs	0.01	attack works on the vulnerable version and fails on the fixed one

(b) All ten demos run in the browser

Fig. C3. (a) The syntax check runs while the learner types, without running the code. Here a closing parenthesis is missing; Python's own parser names the line. (b) The button under the lessons runs the ten complete demos, unchanged, in the browser's Python. All ten give the same result as on a computer.

APPENDIX D
RUNNING THE CODE

The demos, the tests, the lab and the scripts that build both versions of this report are available from the authors. The demos need Python 3.10 or later and nothing else, and they run offline; the lab and `check_sources.py` need the network. From the project folder:

```
python run_all.py                  # every attack against both versions
python -m unittest -v              # the demo tests
python -m demos.a05_injection      # one category on its own
python build_report.py             # rebuild the web version of the report
python build_paper.py              # rebuild this paper (needs Tectonic)
python learn/serve.py              # the lab, at http://127.0.0.1:8765/ (loads Python from cdn.jsdelivr.net)
python report/check_sources.py     # compare OWASP's figures with its pages (needs the network)
```

The script `build_paper.py` needs the drawings exported from the web version first, with `node report/paper/export_figures.mjs`, which needs Node.js 22 or later and Microsoft Edge or Google Chrome.

References

[1] OWASP Foundation, "About OWASP," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/0x01_2025-About_OWASP/

[2] ——, "OWASP Top 10:2025," 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/

[3] ——, "Introduction," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/0x00_2025-Introduction/

[4] ——, "A01:2025 Broken Access Control," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A01_2025-Broken_Access_Control/

[5] ——, "A02:2025 Security Misconfiguration," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A02_2025-Security_Misconfiguration/

[6] ——, "A03:2025 Software Supply Chain Failures," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A03_2025-Software_Supply_Chain_Failures/

[7] ——, "A04:2025 Cryptographic Failures," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A04_2025-Cryptographic_Failures/

[8] ——, "A05:2025 Injection," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A05_2025-Injection/

[9] ——, "A06:2025 Insecure Design," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A06_2025-Insecure_Design/

[10] ——, "A07:2025 Authentication Failures," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A07_2025-Authentication_Failures/

[11] ——, "A08:2025 Software or Data Integrity Failures," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A08_2025-Software_or_Data_Integrity_Failures/

[12] ——, "A09:2025 Security Logging and Alerting Failures," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A09_2025-Security_Logging_and_Alerting_Failures/

[13] ——, "A10:2025 Mishandling of Exceptional Conditions," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/A10_2025-Mishandling_of_Exceptional_Conditions/

[14] ——, "What are application security risks?" OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/0x02_2025-What_are_Application_Security_Risks/

[15] National Institute of Standards and Technology, "National Vulnerability Database (NVD)," Accessed: Nov. 26, 2026. [Online]. Available: https://nvd.nist.gov/

[16] OWASP Foundation, "OWASP Top 10:2021," 2021, Accessed: Nov. 26, 2026. [Online]. Available: https://owasp.org/Top10/2021/

[17] The MITRE Corporation, "Common Weakness Enumeration (CWE)," Accessed: Nov. 26, 2026. [Online]. Available: https://cwe.mitre.org/

[18] OWASP Foundation, "Next steps," OWASP Top 10:2025, 2025, Accessed: Nov. 26, 2026. [Online]. Available: https://top10.owasp.org/2025/X01_2025-Next_Steps/

[19] ——, "OWASP Application Security Verification Standard (ASVS)," Accessed: Nov. 26, 2026. [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/

[20] OWASP Cheat Sheet Series, "Password storage cheat sheet," Accessed: Nov. 26, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[21] ——, "Server-side request forgery prevention cheat sheet," Accessed: Nov. 26, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html

[22] ——, "SQL injection prevention cheat sheet," Accessed: Nov. 26, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[23] ——, "Deserialization cheat sheet," Accessed: Nov. 26, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

[24] ——, "Logging cheat sheet," Accessed: Nov. 26, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

[25] OWASP Foundation, "OWASP WebGoat," Accessed: Nov. 26, 2026. [Online]. Available: https://owasp.org/www-project-webgoat/

[26] ——, "OWASP Juice Shop," Accessed: Nov. 26, 2026. [Online]. Available: https://owasp.org/www-project-juice-shop/

[27] DVWA contributors, "DVWA: Damn Vulnerable Web Application," Accessed: Nov. 26, 2026. [Online]. Available: https://github.com/digininja/DVWA

[28] Microsoft, "Threats – Microsoft Threat Modeling Tool," Microsoft Learn, Accessed: Nov. 26, 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats

[29] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack, "Uncover security design flaws using the STRIDE approach," MSDN Magazine, Nov. 2006, Accessed: Nov. 26, 2026. [Online]. Available: https://learn.microsoft.com/en-us/archive/msdn-magazine/2006/november/uncover-security-design-flaws-using-the-stride-approach

[30] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, pp. 1278–1308, Sep. 1975, doi: 10.1109/PROC.1975.9939.

[31] A. Birsan, "Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies," Medium, Feb. 2021, Accessed: Nov. 26, 2026. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[32] The pip developers, "Secure installs," pip documentation, Accessed: Nov. 26, 2026. [Online]. Available: https://pip.pypa.io/en/stable/topics/secure-installs/

[33] ——, "pip install," pip documentation, Accessed: Nov. 26, 2026. [Online]. Available: https://pip.pypa.io/en/stable/cli/pip_install/

[34] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," RFC 7914, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7914

[35] Python Software Foundation, "hashlib — secure hashes and message digests," The Python Standard Library, Accessed: Nov. 26, 2026. [Online]. Available: https://docs.python.org/3/library/hashlib.html

[36] ——, "sqlite3 — DB-API 2.0 interface for SQLite databases," The Python Standard Library, Accessed: Nov. 26, 2026. [Online]. Available: https://docs.python.org/3/library/sqlite3.html

[37] ——, "pickle — Python object serialization," The Python Standard Library, Accessed: Nov. 26, 2026. [Online]. Available: https://docs.python.org/3/library/pickle.html

[38] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," RFC 2104, Feb. 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2104

[39] Python Software Foundation, "hmac — keyed-hashing for message authentication," The Python Standard Library, Accessed: Nov. 26, 2026. [Online]. Available: https://docs.python.org/3/library/hmac.html

[40] The Pyodide development team, "Pyodide," Pyodide documentation, version 314.0.8, Accessed: Nov. 26, 2026. [Online]. Available: https://pyodide.org/en/stable/

[41] ——, "Pyodide Python compatibility," Pyodide documentation, version 314.0.8, Accessed: Nov. 26, 2026. [Online]. Available: https://pyodide.org/en/stable/usage/wasm-constraints.html

[42] K. Moriarty, Ed., B. Kaliski, and A. Rusch, "PKCS #5: Password-based cryptography specification version 2.1," RFC 8018, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8018