Web Hacking – Exercise 04

A04:2025 Cryptographic Failures:
Break and Repair the Crypto of a Notes Service

Duration	90 minutes
Organisation	Teams of two
Tools	Python 3.12 or Docker, an editor, `curl`
Material	Folder `app/` (source code, database, test skeleton)

Learning Objectives

At the end of this exercise you are able to:

- recognise that cryptographic failures are about **how** crypto is used, not about the maths: the algorithms here are all "real";
- **demonstrate** the consequences of weak choices – unsalted fast hashes, ECB mode, a fixed key, a predictable token – against a running application;
- **fix** each one with the right primitive: salted, slow, adaptive password hashing; authenticated encryption with a unique nonce; a key kept out of the code; a CSPRNG for tokens;
- **prevent regressions** with automated tests that assert on the *properties* (salt, non-determinism, unpredictability), not on a value.

Rules of engagement. VaultNotes is a training application that uses cryptography incorrectly on purpose. Run it **on your own machine only**, bound to `127.0.0.1`. Work only with the accounts and data of your local instance. Never test any system you are not explicitly authorised to test (Swiss Criminal Code, art. 143bis).

1 Setup (before the lesson, 5 minutes)

Option A – Python virtual environment

```
cd app
python3 -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
flask --app app:create_app run          # http://127.0.0.1:5000
pytest -v                               # runs the test skeleton
```

Option B – Docker

```
cd app
docker build -t vaultnotes .
docker run --rm -p 127.0.0.1:5000:5000 vaultnotes
```

The SQLite database is re-created at every start, so you can restart to get back to the initial state. There is no front end: drive the API with `curl` or the Python `requests` library.

macOS: port 5000 is often taken by the *AirPlay Receiver* (it answers `403` to every request, which is easy to mistake for the application). Either switch it off (System Settings → General → AirDrop & Handoff) or use another port, e.g. `flask ... run --port 5001` / `-p 127.0.0.1:5001:5000`, and adapt the URLs below.

2 Scenario

VaultNotes lets users register, log in and store private notes that are "encrypted in the database". The developer was proud to have used AES and SHA-256 – "strong, industry-standard algorithms". Your team is asked to check whether the *use* of these algorithms actually protects the users, and to repair whatever does not.

Test Accounts

User	Password	Note
`alice`	`sunshine`	
`bob`	`sunshine`	same password as alice, on purpose
`carol`	`correct horse`	

API Overview

Method	Path	Description
POST	`/api/register`	Create an account
POST	`/api/login`	Log in
POST	`/api/logout`	Log out
POST	`/api/reset/request`	Ask for a password-reset token
POST	`/api/reset/confirm`	Set a new password with the token
GET	`/api/notes`	List your notes (ciphertext)
POST	`/api/notes`	Store a note (returns the ciphertext)
GET	`/api/notes/{id}`	Read one of your notes (decrypted)

3 Tasks

Task 1 – Password Storage (20 min)

Look at how passwords are stored. You can read the database directly (from the `app/` folder, where `flask run` created it):

```
sqlite3 vaultnotes.db 'SELECT username, password_hash FROM users;'
# without the sqlite3 tool:
python3 -c "import sqlite3; print(sqlite3.connect('vaultnotes.db')
  .execute('SELECT username, password_hash FROM users').fetchall())"
# with Docker, the database is inside the container:
docker exec <container> python -c "import sqlite3; print(sqlite3.connect(
  '/tmp/vaultnotes.db').execute('SELECT * FROM users').fetchall())"
```

(a) What algorithm is used? Is there a **salt**? Is it **slow**?

(b) `alice` and `bob` chose the same password. What do you notice about their stored hashes, and what does that tell an attacker who steals the database?

(c) Take `alice`'s hash and "crack" it: the password is a common word. A one-line Python loop over a small word list, or an online hash-lookup for the exact digest, is enough – *do this only with your local database.* How long did it take? Why is this so fast?

(d) Which two properties (from the lecture) must a password hash have, and which vetted algorithms provide them?

Task 2 – Notes "Encryption" (20 min)

Store a few notes and look at the returned ciphertext:

```
# log in (keep the cookie), then store a note with a repeating pattern:
curl -c jar -X POST http://127.0.0.1:5000/api/login \
     -H 'Content-Type: application/json' -d '{"username":"alice","password":"sunshine"}'
curl -b jar -X POST http://127.0.0.1:5000/api/notes \
     -H 'Content-Type: application/json' \
     -d '{"text":"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}'
```

(a) Which cipher and which **mode** are used (read `app.py`)? Split the ciphertext into 16-byte (32 hex-char) blocks. What do you see for a repeating plaintext, and why (recall the "ECB penguin")?

(b) Store the *same* note twice. Are the two ciphertexts identical? What does that leak to someone reading the database?

(c) Where does the encryption key come from? What happens to the confidentiality of every note the day the *source code* leaks?

(d) Name the mode you would use instead and the extra guarantee it adds beyond confidentiality.

Task 3 – The Reset Token (10 min)

```
curl -X POST http://127.0.0.1:5000/api/reset/request \
     -H 'Content-Type: application/json' -d '{"username":"carol"}'
```

(a) How is the token generated (function and range)? How many possible values are there? Could you **guess** it, and roughly how many tries would it take?

(b) The response returns the token "to ease testing". Why is that, by itself, a vulnerability, and what should happen instead?

(c) Which function must replace the current generator, and how long should the token be?

Task 4 – Repair the Crypto (30 min)

Fix every finding in `app.py`. Use vetted, high-level primitives; do not invent your own. Expected:

- **Passwords**: a salted, slow, adaptive hash (Argon2id, scrypt, bcrypt or PBKDF2-HMAC-SHA-512 with a high work factor). The standard library offers `hashlib.scrypt`. Verify in constant time.
- **Notes**: authenticated encryption (AES-GCM or ChaCha20-Poly1305) with a **fresh, unique nonce** per note, stored next to the ciphertext. The key must come from the **environment**, not the source.
- **Reset token**: a CSPRNG (`secrets.token_urlsafe`); store only a *hash* of the token; never return it; make it single-use; compare in constant time.
- **In transit** (bonus): give the session cookie the `Secure` and `SameSite` flags, and send an HSTS header.

The existing notes cannot be decrypted after you change the scheme – that is expected; the database is re-created at start. Keep the API unchanged so the tests still pass.

Task 5 – Lock it Down with Tests (10 min)

Turn your findings into tests in `app/tests/test_crypto.py` (a skeleton and the fixtures `client` and `logged_in` are provided). Assert on **properties**, not values: two equal passwords give *different* stored hashes; the hash is not a bare SHA-256; two equal notes give *different* ciphertext; the reset response contains *no* token. Your suite must fail on the original code and pass on your fixed code.

4 Wrap-up and Self-Check

This exercise is **not graded** and there is **nothing to hand in**. Keep your notes, cracked hashes, fixes and tests to prepare the exam. Before the debriefing, check your work:

- **Passwords:** salted and slow; equal passwords no longer share a hash; constant-time verification. Did you think about re-hashing on login when the work factor increases?
- **Notes:** authenticated encryption, unique nonce, key from the environment; equal plaintext no longer gives equal ciphertext.
- **Token:** unpredictable, hashed at rest, single-use, never returned.
- **Reflection:** which real-world case of the lecture (Adobe, LinkedIn, WEP, Heartbleed, Firesheep…) matches each of your findings, and which single measure would have prevented it? We discuss this together at the end.

References

- Slides of chapter 04, in particular *Storing Passwords*, *Why ECB Leaks*, *Randomness and Entropy* and *Primitives and Modes*
- OWASP Cheat Sheets – Password Storage, Cryptographic Storage `https://cheatsheetseries.owasp.org/`
- OWASP Top 10:2025 – A04 Cryptographic Failures `https://top10.owasp.org/2025/A04_2025-Cryptographic_Failures/`
- Python `secrets`, `hashlib.scrypt`; `cryptography` AES-GCM `https://cryptography.io/`