Web Hacking – Exercise 03

A03:2025 Software Supply Chain Failures:
Audit and Harden the Supply Chain of a Service

Duration	90 minutes
Organisation	Teams of two
Tools	Python 3.12, an editor, Internet access (PyPI, OSV, GitHub)
Material	Folder `app/` (source, lockfile, CI workflow, Dockerfile, pull request to review)

Learning Objectives

At the end of this exercise you are able to:

- build the **inventory** of what an application really ships (direct and transitive dependencies) and produce an **SBOM**;
- match components against vulnerability databases and **triage** the results: which vulnerabilities are reachable, which fix is needed, how urgent;
- evaluate a **new dependency** before it enters the project;
- find the weaknesses of a **CI/CD pipeline** and of a container build;
- **harden** the chain: lockfile with hashes, controlled upgrades, pinned actions and images, automated checks in the pipeline.

Rules of engagement. You audit and fix *your own copy* of the project. Do **not install** the package proposed in the pull request of Task 3: answer from the material provided. Do **not publish** anything to a public registry (PyPI, Docker Hub, ...), not even "to test" a dependency confusion: uploading packages that imitate someone else's is forbidden by the registries' terms and may be illegal.

1 Setup (before the lesson, 10 minutes)

Use **two** virtual environments: one for the application, one for the audit tools, so that the tools do not end up in the inventory of the application.

```
cd app
# 1. the application, exactly as the team runs it today
python3.12 -m venv .venv
.venv/bin/pip install -r requirements.txt pytest
.venv/bin/python -m pytest -q                       # 9 passed

# 2. the audit tools
python3.12 -m venv ../.tools
../.tools/bin/pip install pip-audit pipdeptree cyclonedx-bom zizmor pip-tools
```

The application needs Python 3.12 or 3.13: it does *not* start on Python 3.14. Keep this in mind, it is already part of the audit. You can start it with `.venv/bin/flask --app app:create_app run` (on macOS add `--port 5001` if the AirPlay Receiver holds port 5000), but you do not need to: in this exercise the weaknesses are in the files *around* the code.

2 Scenario

RateWatch is a small service that publishes exchange rates and receives rate alerts from a partner system (authenticated with a signed token). It was written in early 2023 and has been running since then without much attention. A new security officer asks your team to **review the software supply chain** of the service before the next release: what does it contain, where do its components come from, how is it built and deployed, and how can this be made trustworthy?

What is in the Folder

File	Description
`app.py`, `templates/`	The application (about 70 lines). It is not the subject of the review.
`tests/test_app.py`	Functional tests: they tell you whether the application still works after you change its dependencies.
`requirements.txt`	Dependency list, generated with `pip freeze` in 2023.
`.github/workflows/ci.yml`	The CI/CD pipeline (GitHub Actions): tests on every push and pull request, deployment from `main`.
`Dockerfile`	The production image. It uses the company's internal package index, so it does not build outside the company network: *review it*, do not try to build it.
`review/PR-31.md`	A pull request waiting for review, with the registry page of the package it adds.

Context Given by the Team

- The company runs an internal package index (`pkgs.ratewatch.internal`) for its own packages, e.g. `ratewatch-lint-rules` and `ratewatch-metrics`. These names have never been registered on the public PyPI.
- In production, RateWatch runs behind a reverse proxy, in a network where outgoing HTTPS goes through an authenticated corporate proxy.
- The repository is public on GitHub and accepts pull requests from external contributors.
- The secrets `REGISTRY_TOKEN` (push to the image registry) and `DEPLOY_SSH_KEY` (shell access to the production host) are stored in the repository settings.

3 Tasks

Task 1 – Inventory and SBOM (15 min)

```
../.tools/bin/pipdeptree --python .venv/bin/python
../.tools/bin/cyclonedx-py environment .venv/bin/python -o sbom.cdx.json
```

(a) Which packages are **direct** dependencies (used by the code) and which are **transitive**? Compare with `app.py`.

(b) Compare the packages installed in `.venv` with `requirements.txt`. Is the file really a complete *lock*? What is missing, and what does it mean for two installations made one month apart?

(c) Open `sbom.cdx.json`. What does an SBOM give you that `requirements.txt` does not? Who would use it, and when?

(d) Is any of the components an obstacle to updating the *runtime* (Python version)? Which CWE of the lecture describes this situation?

Task 2 – Known Vulnerabilities and Triage (20 min)

```
../.tools/bin/pip-audit -r requirements.txt --no-deps --disable-pip --desc
```

The scanner gives you a long list. A list is not a decision. For **at least six advisories in different packages**, fill in a triage table:

Package	Advisory	Reachable in RateWatch? (why / under which condition)	Impact if exploited	Priority
...	...	...	...	...

Use the code, the description of the advisory and the context of the scenario (reverse proxy, corporate proxy, debug mode, templates, ...). Then answer:

- Which advisory would you fix **first**, and why?
- "Not reachable" – is it a reason not to upgrade? What does it change?
- How many advisories does the scan report? Why will your neighbours, or you next month, get a different number with the same `requirements.txt`?

Task 3 – Review an Incoming Dependency (10 min)

Read `review/PR-31.md`. **Do not install the package.**

(a) List every signal on this page that should worry a reviewer. Which concept of the lecture does each one illustrate?

(b) What exactly happens, and with which data, at the moment someone runs `pip install -r requirements.txt` with this change?

(c) The pull request says "CI status: all checks passed". Look at `ci.yml`: what may already have happened *before* anybody reviewed the pull request?

(d) Write your review decision in three lines, and the check that would let the team catch such a package automatically next time.

Task 4 – Pipeline and Image Review (20 min)

Review `.github/workflows/ci.yml` and `Dockerfile` line by line, **then** run the static analyser and compare its result with your list:

```
../.tools/bin/zizmor --offline .github/workflows/ci.yml
```

For each weakness, record it in a findings table:

ID	File / line	What an attacker can do	Fix	Severity
S1	...	...	...	...

Guiding questions: who can trigger this workflow, with which code and which secrets? Which code, downloaded from where, runs during the build? Which references can change without a commit in *your* repository? Where does `pip` look for `ratewatch-lint-rules`, and what if a public package with that name and version `99.0` appears? What does the analyser *not* find?

Task 5 – Harden the Chain (25 min)

Fix the project in place. Recommended order (do as much as you can, in this order):

1. **A real lockfile.** Create `requirements.in` with the direct dependencies only (with sensible version ranges), and generate the lockfile:

   ```
   ../.tools/bin/pip-compile --generate-hashes --strip-extras requirements.in
   ```

   This also upgrades every component to a current version. Install it in a *fresh* environment with `pip install --require-hashes --no-deps -r requirements.txt` and run the tests. Then re-run `pip-audit`: the goal is **no known vulnerabilities**.

2. **The workflow.** Fix at least the trigger, the permissions, the scope of the secrets and the pinning of the actions. To find the commit of a release:

   ```
   git ls-remote https://github.com/actions/checkout 'refs/tags/v7.0.1^{}'
   ```

   Replace the "curl | bash" and the `--extra-index-url`, and add a step that fails the build on known vulnerabilities. Check your result with `zizmor`.

3. **The image.** Pin the base image by digest (`docker buildx imagetools inspect python:3.12-slim`, or the Docker Hub page), install from the lockfile only, remove what the application does not use.

4. **If time allows:** publish the SBOM as a CI artifact, sign the image, protect the deployment, and describe how future updates should arrive (Dependabot / Renovate, review, delay before adopting a new release).

4 Wrap-up and Self-Check

This exercise is **not graded** and there is **nothing to hand in**. Its purpose is practice: keep your notes, tables and fixed project for yourself, they are a good basis to prepare the exam. Before the debriefing, check your work with the following questions:

- **Inventory:** can you say how many components RateWatch ships, which ones you chose, and where each one is downloaded from?
- **Triage:** for your most urgent advisory, can you explain the path from the attacker to the vulnerable code, in RateWatch?
- **Intake:** would your process stop pull request #31 *before* its code runs anywhere?
- **Pipeline:** after your fix, which secrets can code from a pull request reach? Which references in your files can still change without a commit?
- **Verification:** do the tests pass, is `pip-audit` clean, does `zizmor` report nothing, on the fixed project?
- **Reflection:** which real-world case of the lecture (Equifax, Log4Shell, dependency confusion, event-stream, Shai-Hulud, SolarWinds, XZ Utils…) is closest to one of your findings, and which single measure would have stopped it? We discuss this question together at the end of the lesson.

References

- Slides of chapter 03, in particular *Pinning and Verifying* and *Guardrails in the Pipeline*
- OWASP Top 10:2025 – A03 Software Supply Chain Failures
  `https://top10.owasp.org/2025/A03_2025-Software_Supply_Chain_Failures/`
- OWASP CycloneDX `https://cyclonedx.org/`
- pip – Secure installs (hash-checking mode)
  `https://pip.pypa.io/en/stable/topics/secure-installs/`
- GitHub Docs – Security hardening for GitHub Actions
  `https://docs.github.com/en/actions/security-for-github-actions`
- zizmor – static analysis for GitHub Actions `https://docs.zizmor.sh/`
- SLSA – Supply-chain Levels for Software Artifacts `https://slsa.dev/`