```yaml
variables:
  DOCKER_IMAGE_TEST: registry.forge.hefr.ch/klagarge/mse2425-grp09/python-pdm:latest
  DOCKER_IMAGE_APP: registry.forge.hefr.ch/klagarge/mse2425-grp09/devsecops-app:latest

default:
  image: $DOCKER_IMAGE_TEST

stages:
  - build-docker
  - lint
  - test

.setup_env: &setup_env
  before_script:
    - cd src
    - cp -r /app/__pypackages__ .
    - export "PYTHONPATH=/builds/Klagarge/mse2425-grp09/src:/builds/Klagarge/mse2425-grp09/src/__pypackages__/3.9/lib"
    - export "PATH=/builds/Klagarge/mse2425-grp09/src/__pypackages__/3.9/bin:$PATH"
    - export "FLASK_APP=app"

test job:
  stage: test
  <<: *setup_env
  script:
    # Set environment variables for the tests
    - export FLASK_SECRET_KEY=$FLASK_SECRET_KEY

    # launch tests
    - pdm run pytest tests --cov --cov-report term --cov-report html

  artifacts:
    paths:
      - src/htmlcov/

lint job:
  stage: lint
  <<: *setup_env
  dependencies: []
  script:
    - pdm run flake8 --config=../tox.ini
  allow_failure: true # Linter can fail, fixing it is for now outside of the projects scope

pages:
  stage: test
  dependencies:
    - test job
  needs: ["test job"]
  script:
    - mv src/htmlcov/ public/
  artifacts:
    paths:
      - public
```

```
    expire_in: 7 days
  only:
    - main

# This job runs only when Dockerfile changes
docker-build-test:
  image: docker:latest
  stage: build-docker
  services:
    - docker:dind
  script:
    - docker build -t $DOCKER_IMAGE_TEST -f Dockerfile .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin $CI_REGISTRY
    - docker push $DOCKER_IMAGE_TEST
  rules:
    - if: $GITLAB_CI == 'false' # Only run in GitLab CI
      when: never
    - changes:
        - Dockerfile
        - src/pyproject.toml
        - src/pdm.lock

docker-build-app:
  image: docker:latest
  stage: build-docker
  services:
    - docker:dind
  script:
    - docker build -t $DOCKER_IMAGE_APP -f src/Dockerfile .
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER --password-stdin $CI_REGISTRY
    - docker push $DOCKER_IMAGE_APP

include:
  - template: Jobs/SAST.gitlab-ci.yml

dast:
  stage: test
  image: ghcr.io/zaproxy/zaproxy:stable
  services:
    - name: $DOCKER_IMAGE_APP
      alias: app
  script:
    - echo "Waiting for the app to start on http://app:5000"
    - timeout 60 bash -c 'until curl -s http://app:5000; do echo "Waiting..."; sleep 3; done'
    - zap-full-scan.py -t http://app:5000 -I

gitleaks:
  stage: test
  image:
    name: zricethezav/gitleaks:latest
    entrypoint: [""]
  script:
```

```
    - gitleaks dir -v --redact=75 .
```

Questions - Part 1

Unit tests - Secret keys - Linter - optimized CI/CD pipeline

- **Q1.1**: Provide additional unit tests to ensure that wrong input (type, values, ...) doesn't crash the application, but gets intercepted and produce a controlled error. It is possible that you have also to adjust slightly the application
- **Q1.2**: The secret key for flask is hard coded. Is this good practice? What are the dangers? How could this be fixed?
- **Q1.3**: Give a short description of *Linter*. Integrate a basic linter like Flake8 or Ruff in the existing CI/CD pipeline
- **Q1.4 (optional)**: The run of the current CI/CD pipeline takes some time. Especially the time to setup the docker with the update and installation of all the dependencies is quite time consuming compared to the real testing time. Do you see any alternatives to speed up this process? Describe and try to implement it in your pipeline.

Answers - Part 1

Q1.2

- It's a very bad practice. The secret key will be exposed in the codebase and can be easily accessed by anyone who has access to the codebase. This can lead to security vulnerabilities and compromise the integrity of the application.
- To fix this, you can use environment variables to store the secret key.

Q1.3

- A linter is a tool to statically analyse code for readability and improving code quality. It is usually executed from a standalone tool.
- It is used to check for errors, vulns, code smells or general issues but also to enforce a coding style over the whole project.

Q1.4

The minimum is to change the image on the CI to put an Alpine or a basic python image. Another option (which we implemented) is to create a custom Docker image that includes all the necessary dependencies for the application. This can significantly reduce the time required to set up the environment and speed up the CI/CD pipeline.

Questions

Part 2

- **Q2.1**: Every commit triggers the CI/CD pipeline. Find out a way to trigger the pipeline only if specific commits (e.g. commit in a development branch) are made. Where can this be configured. Describe your solution and implement it in your pipeline.
- **Q2.2**: Take the CIS controls and give some examples (minimum 5) of controls from this standard that are not or not enough implemented in the calculator app. Provide a short description and a possible remediation. Implement at least two of the controls in the app / pipeline.
- **Q2.3 (optional)**: The linter from question 1.3 is a good start. It is only executed in your pipeline. But what if you would also integrate it directly in your local development environment (e.g. IDE)? Can you do the linting before you commit? Describe your solution and implement it in your (local) pipeline. Describe the advantages and disadvantages of this approach.

Answers - Part 2

Q2.1

Solution is to add a `rule` section to add condition to trigger the pipeline. It's what is implemented for the `docker-build` job. Another option is to use an `only` section to trigger the pipeline only if the change is made in a specific branch. It's what is implemented for the pages job.

Q2.2

Example 1 - 3.6 Encrypt Data on End-User Devices

description Sensible data is everywhere. It is also on the end user's device. It is primordial to keep it secure.

mitigation There is multiple ways to secure data. CIS suggests the following : Windows BitLocker®, Apple FileVault®, Linux® dm-crypt.

=> is this applicable for this use case?

Example 2 - 4.3 Configure Automatic Session Locking on Enterprise Assets

description A logged in computer doesn't check permanantly for the user's identity. A user could by mistake leave his computer open and logged in, give way for anyone ill-intentionned with physical access to use the computer with it's permissions.

mitigation Forcing an auto-logout after a few minutes. => dito ?

Example 3 - 5.2 Use Unique Passwords

description If a user uses the same password everywhere, it only needs one to get compromised and everything is equally compromised.

mitigation Usage of unique passwords and for users use 2FA at least.

~

Example 4 & 5 with implementation

Unfortunately, due to the amount of work we both had, from work and from school, we didn't have enough time to do those last two points. We did the rest (except for optionals) though.

oh

1/2

Q2.3

We can use a pre-commit that runs the linter before committing. This ensures that the code is linted before it is committed, which can help catch errors and improve code quality. However, this approach can be time-consuming and may require additional setup.

✓

Questions

Part 3

- **Q3.1**: Setup your CI/CD pipeline with an additional SAST solution. I propose that you use `semgrep` for this task. Get your inspiration here: https://semgrep.dev/for/gitlab and https://docs.gitlab.com/ee/user/application_security/sast/
- **Q3.2**: Describe the found problems (alerts) in the `calculator` app (in the original code, git tag `v3.0`)
- **Q3.3**: Install DAST OWASP ZAP on your host or in a Docker. Play with OWASP ZAP, analyze the calculator code
- **Q3.4**: Implement a DAST solution in your pipeline. Get some inspiration here https://docs.gitlab.com/ee/user/application_security/dast/ . Describe what you have integrated in your pipeline. *Note: you must ensure that your application is running while you are testing!*
- **Q3.5 (optional)**: Normally, the provided code has some bugs, which are discovered by SAST solution. Describe the found bugs (in the original code, git tag `v3.0`) and provide solution to remediate the problems. Indicate which commit/tag contains the corrected code
- **Q3.6 (optional)**: Describe the found bugs (in the original code, git tag `v3.0`) with DAST and provide solution to remediate the problems. Indicate which commit/tag contains the corrected code. Do corrections only in the provided code (no libraries)

Answers - Part 3

Q3.2

For some reasons, semgrep works locally, but not on GitLab. Here is the report when runned locally.

1/2

Q3.3

After performing a scan, we can see a few alerts as seen on this screenshot :

✓

Q3.4

The integrate DAST in Github doesn't work on our version, we need the *Ultimate* version of GitLab selfhosted.

↳ it worked for most of the sips 1/2

should be 3

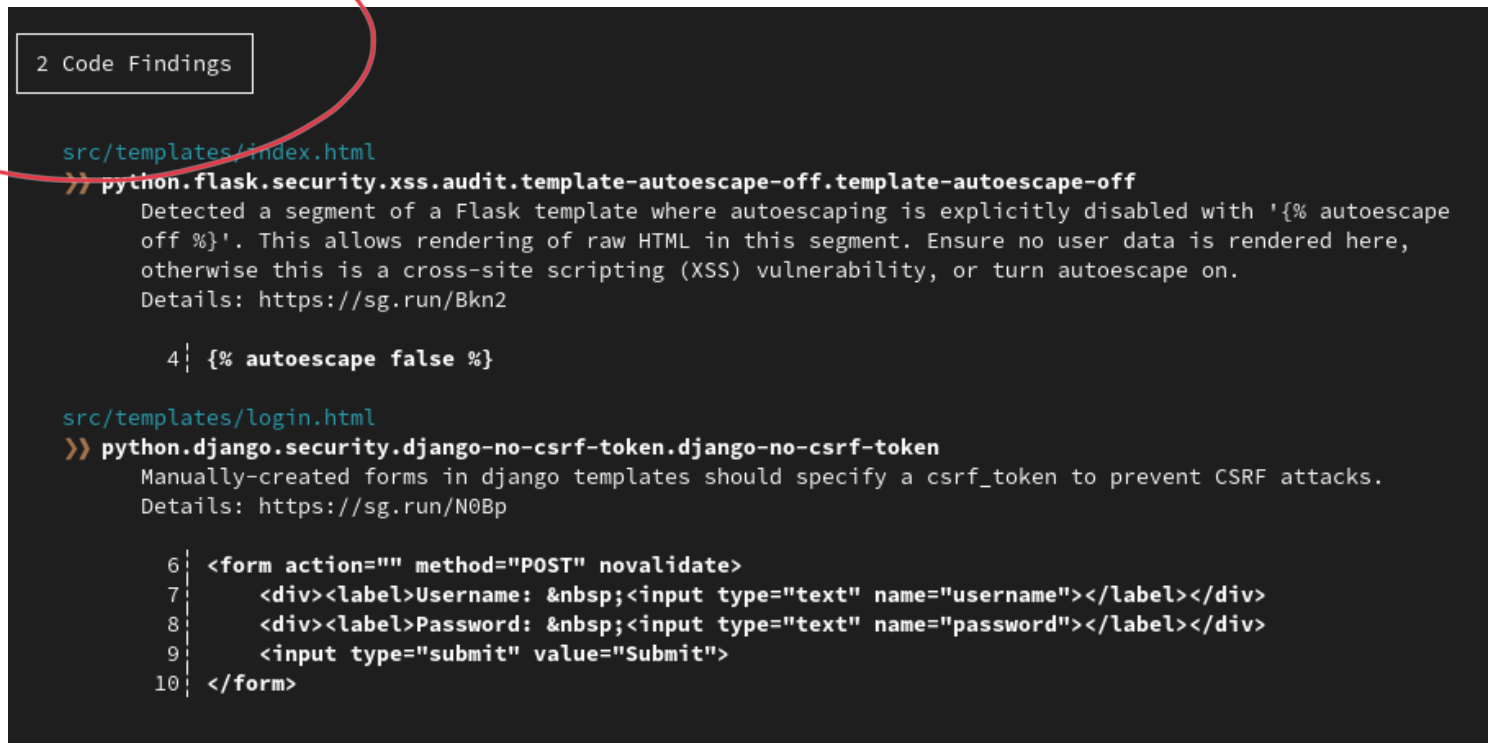

Figure 1: SAST-report

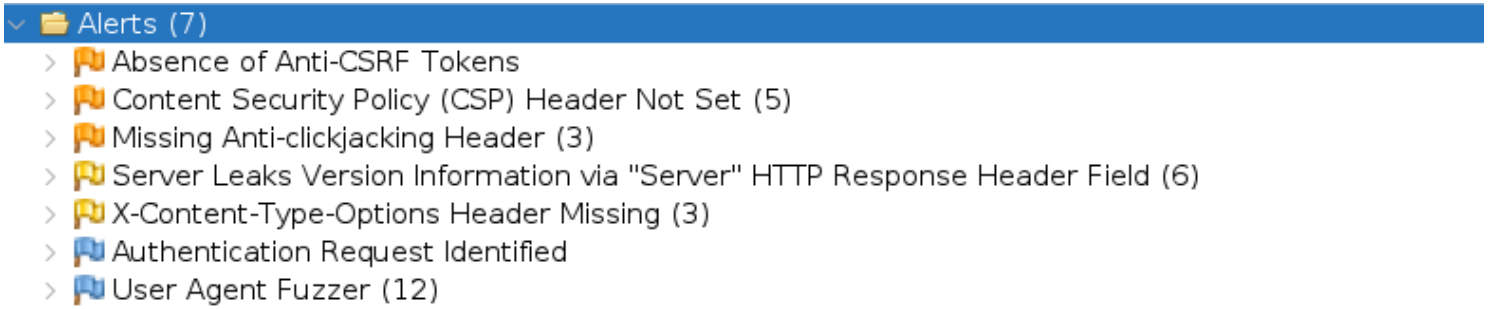

Figure 2: alt text

We create a new Docker image for the application. This image auto launch the flask app when the container is started.

We used this image as a service for the DAST stage on our CI. The stage use zaproxy to test the application. Warning do not return wailure, so the stage pass if no error is found by the OWASP ZAP.

We don't understand why the stage fail when we try to provide the html report as artifact. So if the stage fail, we can see the error in the logs.

Questions

Part 4

- **Q4.1**: Often secrets are committed in a repository. Different research tools exist and help to detect this kind of dangerous forgotten credentials. Integrate a check in your pipeline for these kinds of problems. Have a look at https://github.com/zricethezav/gitleaks. What kind of leaked secrets can you find in the git repo? Did the tool not find something that it should have found? Why? What possibilities exist to prevent this kind of leakage?
- **Q4.2**: Try to find any possible problems in our used libraries (e.g. flask). The `pyproject.toml` describes all the additional libraries used by the application. You can use a dependency scanning (have a look here: https://docs.gitlab.com/ee/user/application_security/dependency_scanning/) to see if all imported libraries

are safe. Do you find any problems? Integrate the scanning in your pipeline.

- **Q4.3 (optional)**: API Fuzzing (and other kinds of DAST) is described at this page: https://docs.gitlab.com/ee/user/application_security/api_fuzzing/. Choose one of the different description possibilities for your *calculator* API. Integrate it in your pipeline.

Answers - Part 4

Q4.1

GitLeaks can find strings like API keys, passwords, and other sensitive information that might be accidentally committed to a repository. This tool can only recognize them if they look like sensitive information.

The scan of the git repository didn't detect the previous flask key because it was not in the format that GitLeaks recognizes.

Usually, the best practice is to use environment variables to store sensitive information. This way, the information is not exposed in the code.

Q4.2

The Dependency scanning tool from GitLab linked by the teacher in the exercise cannot be used as it is limited to Gitlab Ultimate. I am looking into using an open-source solution

After using three different scanning tools, no known vulnerabilities were found.

Tools used : - pyscan - safety - pip-audit