How to Plant Safety into GitLab CI/CD

We will understand how to integrate a Safety tool into GitLab, enabling the job to fail even when multiple issues are detected by the tool.

Considering your DevOps team created a simple CI pipeline with the following contents. Please add the Safety scan to this pipeline.

```
image: docker:20.10  # To run all jobs in this pipeline, use a latest docker image

services:
  - docker:dind       # To run all jobs in this pipeline, use a docker image which contains a docker
daemon running inside (dind - docker in docker). Reference: https://forum.gitlab.com/t/why-
services-docker-dind-is-needed-while-already-having-image-docker/43534

stages:
  - build
  - test
  - release
  - preprod
  - integration
  - prod

build:
  stage: build
  image: python:3.6
  before_script:
   - pip3 install --upgrade virtualenv
  script:
   - virtualenv env                       # Create a virtual environment for the python application
   - source env/bin/activate              # Activate the virtual environment
   - pip install -r requirements.txt      # Install the required third party packages as defined in
requirements.txt
   - python manage.py check               # Run checks to ensure the application is working fine

test:
  stage: test
  image: python:3.6
  before_script:
   - pip3 install --upgrade virtualenv
  script:
   - virtualenv env
   - source env/bin/activate
   - pip install -r requirements.txt
   - python manage.py test taskManager
```

There are some jobs in the pipeline, and we need to embed the OAST tool in this pipeline.

Let’s login into GitLab using the following details and execute this pipeline once again.

Next, we need to create a CI/CD pipeline by replacing the **.gitlab-ci.yml** file content with the above CI script. Click on the **Edit** button to replace the content (use **Control+A** and **Control+V**).

Save changes to the file using the **Commit changes** button.

Verifying the Pipeline Run

As soon as a change is made to our repository, the pipeline starts to execute the defined jobs.

Check out the results of this pipeline by visiting https://gitlab-ce-1jtmsiog.lab.productsecurity.ai/root/django-nv/pipelines and click on the appropriate job name under the pipeline to see the job output.

As discussed in the **Software Component Analysis using the Safety** exercise, we can embed the Safety tool in our CI/CD pipeline. However, do remember you need to run the command manually before you embed OAST in the pipeline.

Most of the code (up to 95%) in any software project is open-source/third-party components. It makes sense to perform SCA scans before static analysis.

```
image: docker:20.10  # To run all jobs in this pipeline, use a latest docker image

services:
  - docker:dind       # To run all jobs in this pipeline, use a docker image which contains a docker
daemon running inside (dind - docker in docker). Reference: https://forum.gitlab.com/t/why-
services-docker-dind-is-needed-while-already-having-image-docker/43534

stages:
  - build
  - test
  - release
  - preprod
  - integration
  - prod

build:
  stage: build
  image: python:3.6
  before_script:
   - pip3 install --upgrade virtualenv
  script:
   - virtualenv env                      # Create a virtual environment for the python application
   - source env/bin/activate             # Activate the virtual environment
   - pip install -r requirements.txt      # Install the required third party packages as defined in
requirements.txt
   - python manage.py check               # Run checks to ensure the application is working fine

test:
  stage: test
  image: python:3.6
  before_script:
   - pip3 install --upgrade virtualenv
  script:
   - virtualenv env
   - source env/bin/activate
   - pip install -r requirements.txt
   - python manage.py test taskManager

oast:
  stage: test
  script:
    - docker pull prodsecai/safety  # We are going to pull the prodsecai/safety image to run the safety
scanner
    # third party components are stored in requirements.txt for python, so we will scan this particular
```

```
file with safety.
    - docker run --rm -v $(pwd):/src prodsecai/safety check -r requirements.txt --json > oast-
results.json
  artifacts:
    paths: [oast-results.json]
    when: always # What does this do?

integration:
  stage: integration
  script:
    - echo "This is an integration step."
    - exit 1
  allow_failure: true # Even if the job fails, continue to the next stages

prod:
  stage: prod
  script:
    - echo "This is a deploy step."
  when: manual # Continuous Delivery
```

Copy the above CI script and add it to the **.gitlab.ci.yml** file

As discussed, any change to the repo kick starts the pipeline.

We can see the results of this pipeline

Remember!

You some times definitely notice the job failed because of **non zero exit code(here, exit code 255)**?

Why did the job fail? Because the tool found security issues. This is the expected behavior of any DevSecOps friendly tool.

You need to either fix the security issues or add **allow_failure: true** to let the job finish without blocking other jobs.

Please refer to the **Linux Exit Code** from the Google Search Engine

You will notice that the **oast** job stores the output to a file **oast-results.json**. We need the output file to process the results further either via APIs or vulnerability management systems like **DefectDojo**.

Allow The Job Failure

In DevSecOps Maturity Levels 1 and 2, we do not want to fail the builds, jobs, or scans. This is because security tools often generate a significant number of false positives.

```
image: docker:20.10  # To run all jobs in this pipeline, use a latest docker image

services:
  - docker:dind       # To run all jobs in this pipeline, use a docker image which contains a docker daemon running inside (dind - docker in docker). Reference: https://forum.gitlab.com/t/why-services-docker-dind-is-needed-while-already-having-image-docker/43534

stages:
  - build
  - test
  - release
  - preprod
  - integration
  - prod

build:
  stage: build
  image: python:3.6
  before_script:
   - pip3 install --upgrade virtualenv
  script:
   - virtualenv env                       # Create a virtual environment for the python application
   - source env/bin/activate              # Activate the virtual environment
   - pip install -r requirements.txt      # Install the required third party packages as defined in requirements.txt
   - python manage.py check               # Run checks to ensure the application is working fine

test:
  stage: test
  image: python:3.6
  before_script:
   - pip3 install --upgrade virtualenv
  script:
   - virtualenv env
   - source env/bin/activate
   - pip install -r requirements.txt
   - python manage.py test taskManager

oast:
  stage: test
  script:
    # We are going to pull the prodsecai/safety image to run the safety scanner
    - docker pull prodsecai/safety
    # third party components are stored in requirements.txt for python, so we will scan this particular file with safety.
    - docker run --rm -v $(pwd):/src prodsecai/safety check -r requirements.txt --json > oast-results.json
  artifacts:
    paths: [oast-results.json]
    when: always # What does this do?
  allow_failure: true

integration:
  stage: integration
  script:
    - echo "This is an integration step"
    - exit 1
  allow_failure: true # Even if the job fails, continue to the next stages

prod:
  stage: prod
  script:
    - echo "This is a deploy step."
  when: manual # Continuous Delivery
```

Go ahead and add it to the **.gitlab-ci.yml** file to run the pipeline.

You will notice that the **oast** job has failed but didn’t block the other jobs from running.

As mentioned before, any change to the repository triggers the pipeline.

To view the results of this pipeline