**Practical DevSecOps**

**Exam Report**

**for**

**DevSecOps Professional (CDP)**

**By Joshua Dallimore**

Table of Contents

[Challenge 1 2](#_Toc122197587)

[Challenge 2 7](#_Toc122197588)

[Challenge 3: 15](#_Toc122197589)

[Challenge 4 18](#_Toc122197590)

[Challenge 5 25](#_Toc122197591)

## Challenge 1: Create a CI/CD pipeline to embed the brakeman tool with the following steps (25 points) In this challenge, you will use Gitlab CI to implement a CI/CD pipeline with the following details:

* First, create a new repo on GitLab (based on https://github.com/OWASP/railsgoat), then create a job called sast under the build stage to set up brakeman tool (https://brakemanscanner.org)
* The sast job with brakeman should run on every commit on every branch, and should run for each pull/merge requests for the railsgoat repository
* sast job must implement all the applicable DevSecOps Gospel (best practices), and must save the scan results on the CI server for further processing in a machine-readable format such as JSON, CSV, XML, etc.
* Ensure you mark false positives as false positives using brakeman's inbuilt features, and ensure that the false positives are not reported as issues in the next scans
* Also, explain why an issue that is marked as false positive is indeed a false positive and not a real finding

### Solution

Following best practices, a Brakeman scan was run locally first before embedding into the CICD Pipeline in GitLab.

Ruby needs to be downloaded as well as the tool, therefore the following code downloads Ruby and the Brakeman tool.

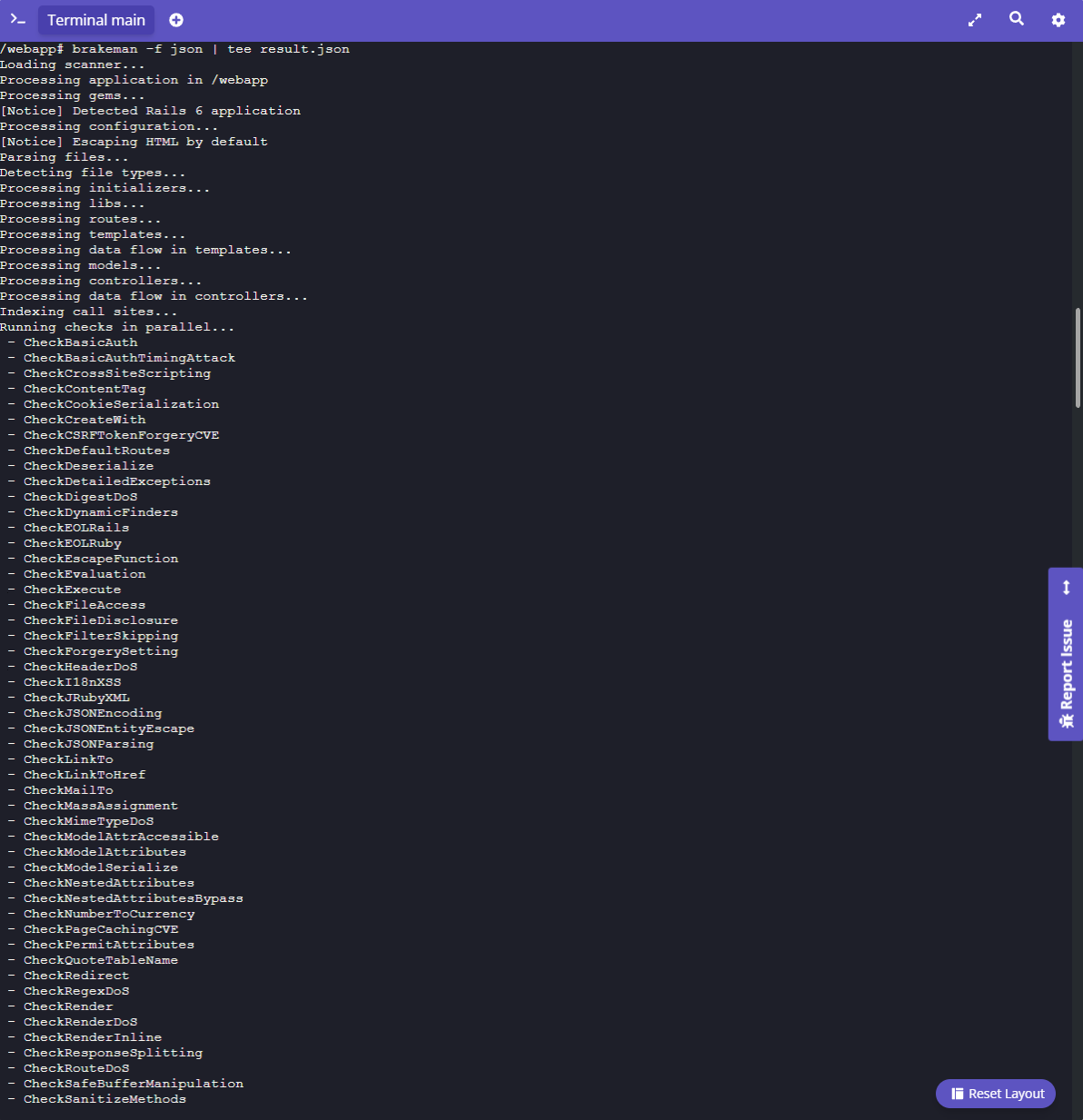
apt update

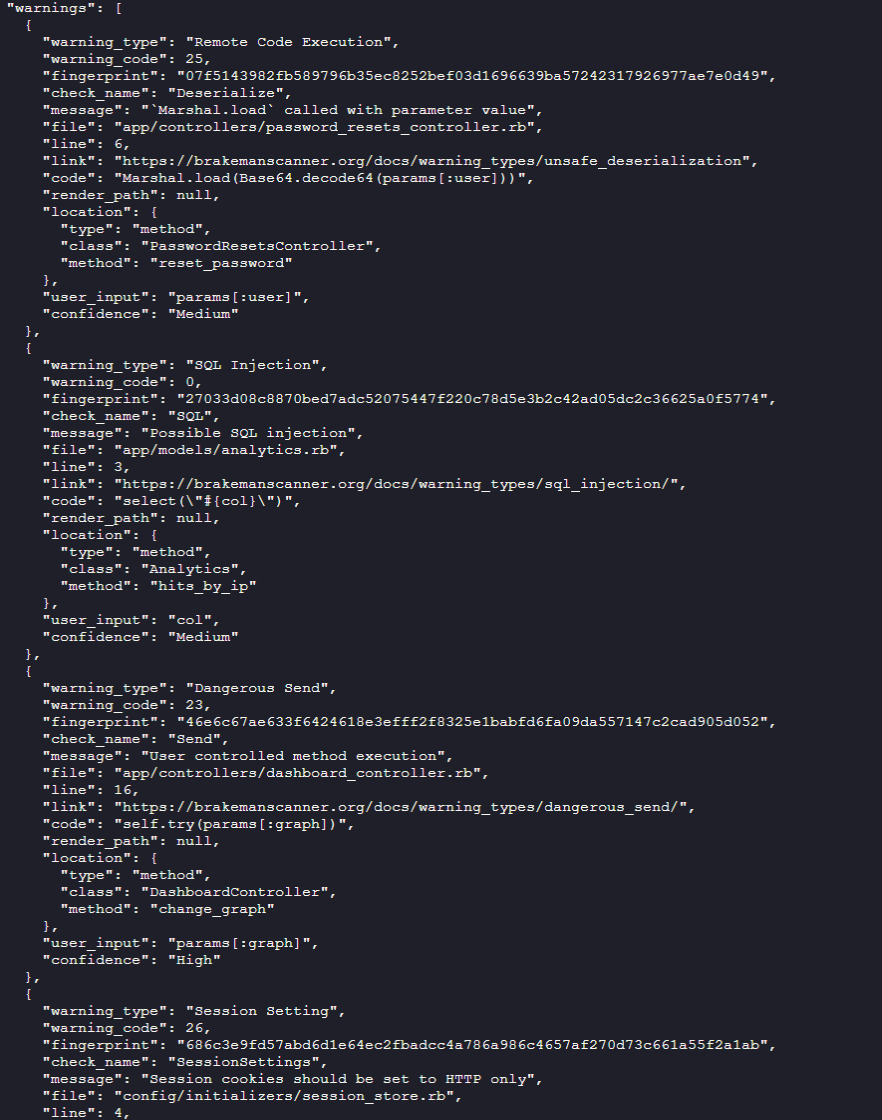
apt install ruby-full -y

gem install brakeman -v 5.2.1

Now that this has been successfully installed, Brakeman scan can be completed. Following best practices, it is being stored in a JSON format, as it is a machine readable format.

brakeman -f json | tee result.json





Next, it is good practice to remove known false positives from the file. This is done using the brakeman.ignore file. Any warning found within this file is deemed as a false positive and therefore will not be shown in the output of the next scan.

For example:

cat > brakeman.ignore <<EOF

{

"ignored\_warnings": [

{

"fingerprint": "febb21e45b226bb6bcdc23031091394a3ed80c76357f66b1f348844a7626f4df",

"note": "ignore Cross-Site Scripting"

}

]

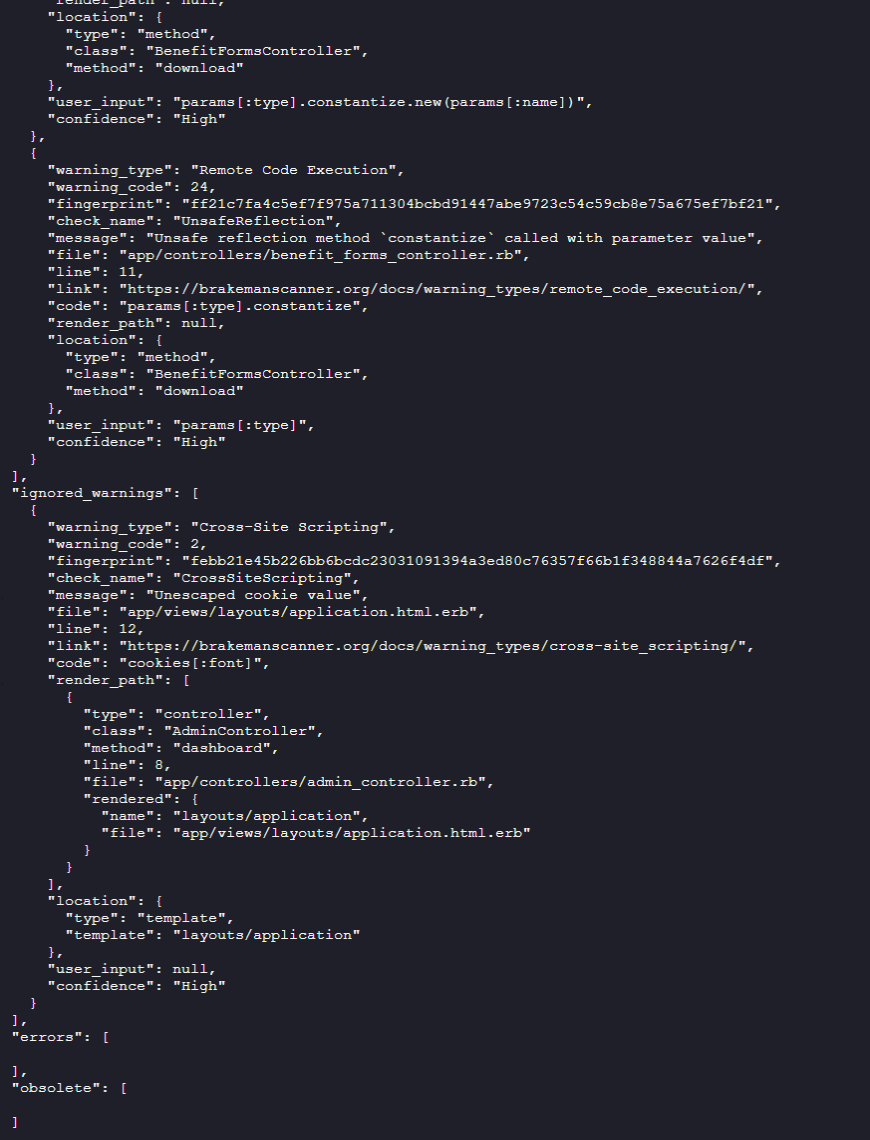
}

EOF

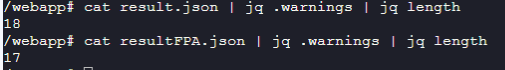
This is a warning that is a false positive and is therefore stored in the brakeman.ignore file. This is a false positive as it is a remote code execution issue.

To run the new scanner with the false positive removed, the following code is used. Unfortunately, I suffered some problems downloading files from GitLab and therefore cannot submit the file, but this is what it looked like and was submitted onto GitLab too.

Brakeman -f json -I brakeman.ignore | tee resultFPA.json



Using the following code, we can compare the two different scans and see the number of reported warnings



As you can see the FPA scan has one less warning compared to the original. Next, the code is embedded into the railsgoat repository on GitLab. The railsgoat repository was created by creating a new project into GitLab based off of the OWASP Railsgoat on GitHub. This was done by copying the GitHub over to the repository using the copy link function in create new project, using this link

Git clone [http:// gitlab-ce-hh3rj0pi.lab.practical-devsecops.training/root/railsgoat.git](http://gitlab-ce-acsrq8h9.lab.practical-devsecops.training/root/railsgoat.git) webapp

Git remote rename origin old-origin

Git remote add origin http://gitlab-ce- hh3rj0pi.lab.practical-devsecops.training/root/railsgoat.git

Git push -u origin –all

Next this can be embedded into GitLab as a YAML file. The YAML file can be found attached in the email sent along with this document.

CICD Pipeline

image: docker:latest

services:

- docker:dind

stages:

- build

- test

- release

- preprod

- integration

- prod

sast:

stage: build

script:

- docker pull hysnsec/brakeman

- docker run –rm -v $(pwd):/src hysnsec/brakeman -t https://prod-hh3rj0pi.lab.practical-devsecops.training -f json /src | tee brakeman-result.json

artifacts:

paths: [brakeman-output.json]

when: always

expire\_in: one week

allow\_failure: true

To see the results, we visit <https://gitlab-acsrq8h9/root/django.nv/pipelines> to see the results of the new pipeline. We can also see the result file on the server, in the pipelines side panel or above the commit file.

Unfortunately, I could not get this code to work in the repository copied and going against best practices some of the jobs associated with the GitLab repository railgoat meant that the job runs were over 10 minutes and therefore the build was cancelled. As there were problems with the output, it would be expected that the sast job would run and complete a brakeman scan against the prod machine, creating a json file of the results named brakeman-output, similar to that shown in the screenshots of the local terminal. It would also be expected that the job run would fail the build due to vulnerabilities being found, therefore the allow\_failure: true allows the build to pass even with a failure.

It would also be worthwhile implementing the brakeman.ignore file into the code and committing the brakeman.ignore file to the Gitlab to allow false positives to be ignored in the scan and thus the output.

Best Practices followed:

* Allow build failure
* Run scan locally first
* Save results of scan to GitServer (expected but not executed)
* Job scan runs for under 10 minutes (expected but not executed)
* Save scan in a machine-readable format (done locally but not on GitLab)

## Challenge 2: Create an Ansible Playbook to harden the prod server (25 points) In this challenge, you will use Gitlab CI to implement a CI/CD pipeline with the following details:

* Create a job called os-hardening under the deploy stage to run ansible os hardening script from dev-sec (https://github.com/dev-sec/ansible-os-hardening), the hardening script should harden the production machine
* Ensure the os-hardening job runs only on the master branch in the django.nv repository, and the os-hardening job must not fail the build
* The os-hardening job must save the results on the CI server for further processing in a machine-readable format like JSON, CSV, XML, etc.
* Explain the need to save the output in machine-readable formats like JSON, CSV, XML, etc.
* Run the ansible playbook in the dry mode before making changes to the production machine.
* As always, test everything locally on the DevSecOps-box machine before integrating it into the CI/CD pipeline

### Solution

Following best practices, Ansible Hardening was run locally first before embedding into the CICD Pipeline in GitLab.

Download file

pip3 install ansible==6.4.0 ansible-lint==6.8.1

Create Inventory file

cat > inventory.ini <<EOL

> # DevSecOps Studio Inventory

> [devsecops]

> devsecops-box-acsrq8h9

> [prod]

> prod-acsrq8h9

> EOL

Prevent SSH yes/no prompt

ssh-keyscan -t rsa prod-6jnpiomi devsecops-box-6jnpiomi >> ~/.ssh/known\_hosts

Find bash version installed

ansible -i inventory.ini all -m command -a "bash --version"

Harden the production environment by going into harden directory and copy inventory profile

mkdir /hardening && cd /hardening

cp ../inventory.ini .

Create the ansible hardening playbook named hardening (unable to save from GitLab)

cat > /hardening/ansible-hardening.yml <<EOL

---

- name: Playbook to harden Ubuntu OS.

hosts: prod

remote\_user: root

become: yes

roles:

- dev-sec.os-hardening

EOL

Install the hardening role using ansible-galaxy.

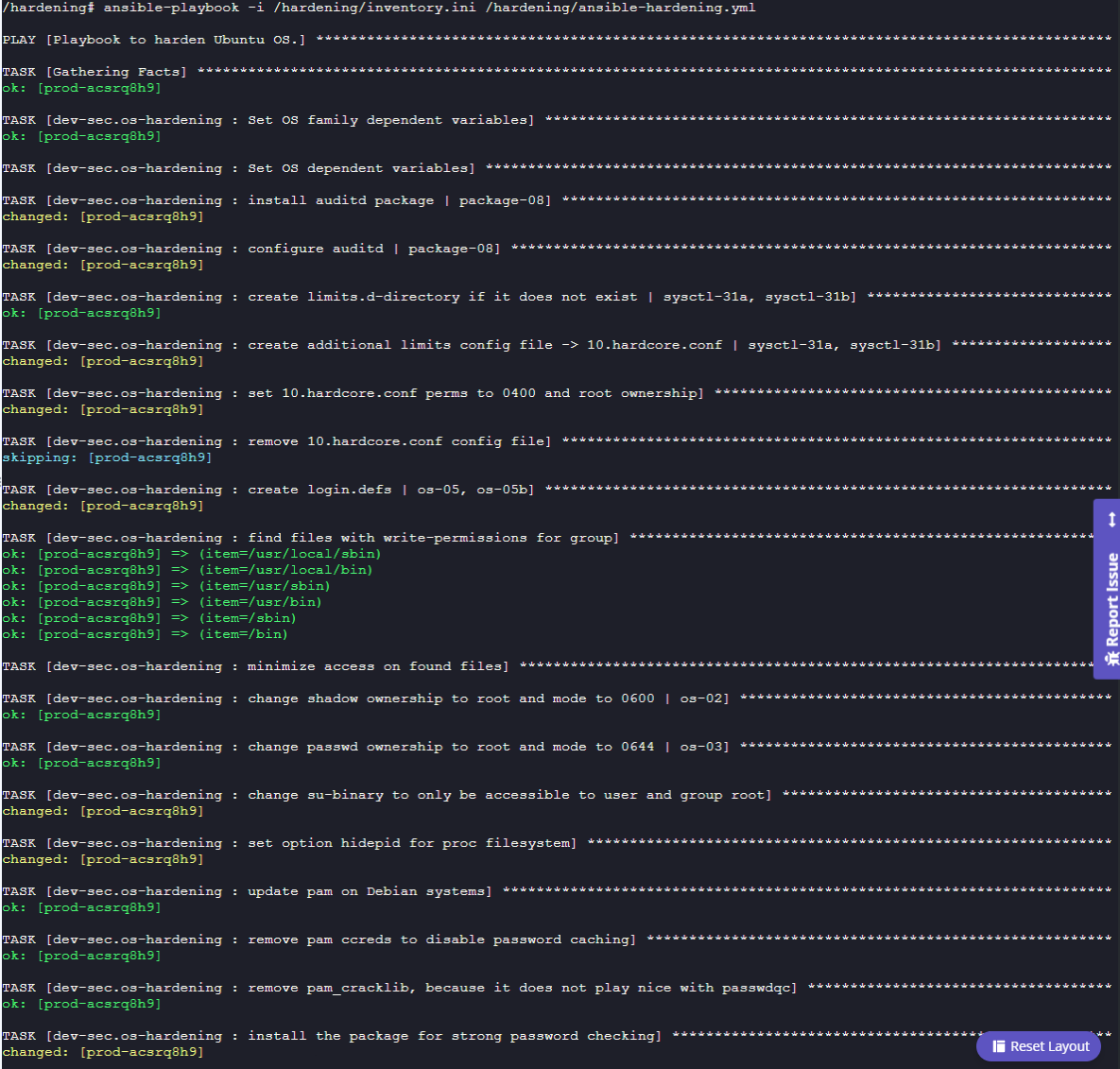
ansible-galaxy install dev-sec.os-hardening

Execute the hardening playbook in dry mode (using check option)

ansible-playbook -i /hardening/inventory.ini /hardening/ansible-hardening.yml --check

Execute the playbook on production machine

ansible-playbook -i /hardening/inventory.ini /hardening/ansible-hardening.yml





Now embed into CICD Pipeline

(After running locally, the machine connection ended and was unable to run GitLab locally, therefore restarted the lab environment and the prod-machine changed. The machine was hardened locally again however due to time restraint this was not documented)

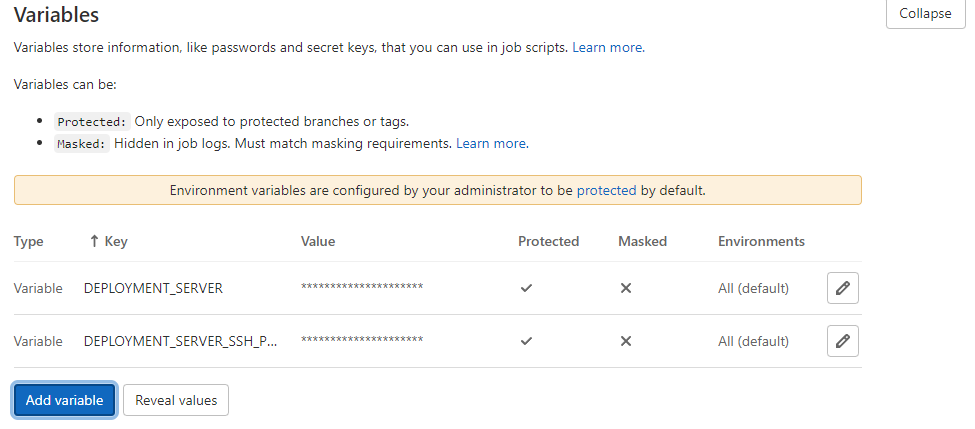
Create variables for Deployment Server and SSH Privkey

|  |  |
| --- | --- |
| Name | Value |
| Key | DEPLOYMENT\_SERVER |
| Value | prod-bilz50ig |

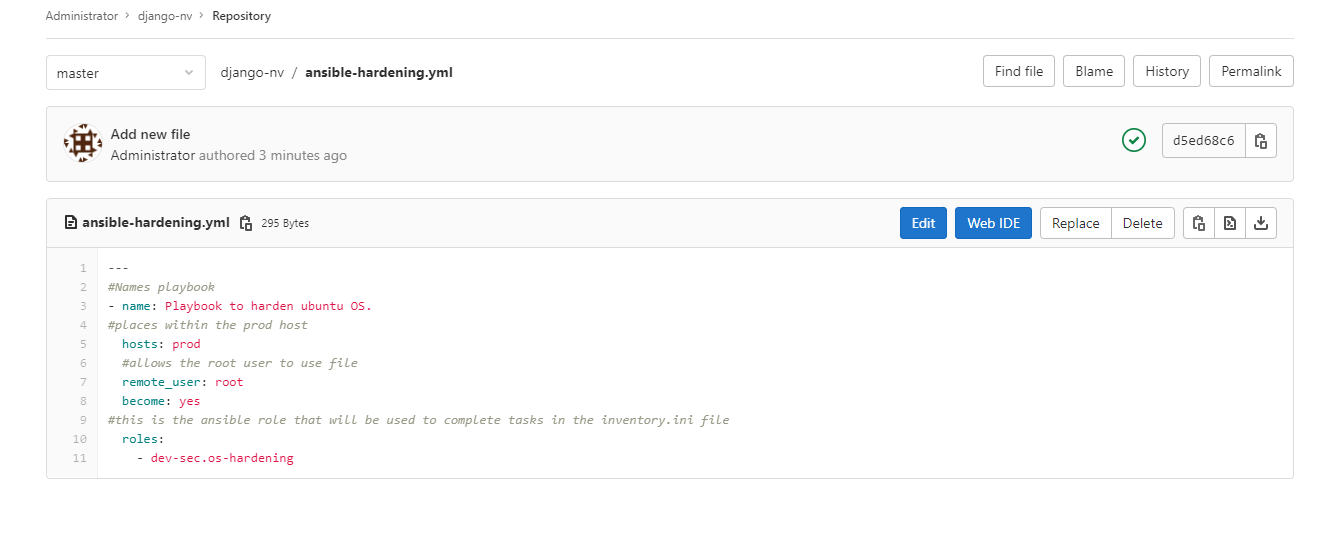
The deployment server SSH key is obtained by ssh into production machine and then requesting the ssh privkey.

|  |  |
| --- | --- |
| Name | Value |
| Key | DEPLOYMENT\_SERVER\_SSH\_PRIVKEY |
| Value | -----BEGIN RSA PRIVATE KEY-----  MIIEpAIBAAKCAQEAwNpZcFqfOVT0Q486JHr7kQO7HGD2zsGC4iodBxc6bdU/S1zn  OUM633CIM0UYSqRTaOIRAi+HowZ5W0/8ZKW3okMQTQNjjY7DLiT3lRxRUly1PEi2  dAimRGlgm1ycE/ucKXShiH9iSSeIrtkUiHWxvRssxSjdT0lGhB/z7PN2hu0mpj2k  CDoeSXRM+EtFBkc45Mbyc7VCIyYcW2gUmFCLr8js/vUuGJYu/FQp8yZDvVg8+bBx  5tHjKd+qNCs0HhytWbhVWDwl46uigIrvwk6gaQXgVv1djrPNh9eYYLoXSwTfo//c  DrPZspBia9bxShKbL+zGcyoVgAy/yvdM3peI6QIDAQABAoIBAQCubpCJFB6CT7nj  xY+UYXxa7OH3yChUWClXATpiKHtbzo7CTpSBcbK1WOaIYQ2YrcsXyaoSrQTkyr1H  fzBNpKpU5I3A6rjH2AHoId2iDAvuEBaJIUeN6ijhJeMQgxJU7LaRtIFKodU3T7/M  TmLJDpMl9YdoCQ8rYJ6ccP5DKu7hF9fQA/8V5/OAkj2XKjQg7APEYV+nFjYvcpl6  zw/P5DnRTzKQPG58u2TMb8cPKmi9MAJUAeFd1k2uBxBWXdIXHqDH7fHNFKlb5vaP  iYsNQtDRiVLDu1+CJ5ddo8IEFBRJVk8xUD9vjAocplINgc9rRIlIKQDs4tHxoceU  wv2ZJG1RAoGBAP4tdWTj2ds7TS4yBoOGLHdXDIPHHBHV8g7V39TBCMO4aRfB2Xf+  tHR5ykp7o2OqSH+sVDzK4tGy50dWZzw+m4lR3714C4BWx89OZJw/5RJ0j2OSLpIp  OPAykJtrKBYPJZ3Lan90HnVkU5MD4blmRrdHFrq6idYaDXCHgBvC1undAoGBAMI8  VGw9C1iDPz1PcLOmV3uVpNJFLud0gSaTtUyouGJPbauvwcnVIxgeXjRO3sPMHEvI  700OvMDH/YUCE5YB0VtZ0y/55s/qGEp8LRzYFUUxHB38YnekNW9PCUZ5Fmbfcv5a  2veoxdT4FO/U6Stg0aLXQxakyFj7AcDKl1WcCzh9AoGAEHuuMz67cAYmeSpxVbIr  zAlvHFSbM2Tmb6PbAhcKlHavCgVeLvPri+oh/jaKX/o4/V6Vj+OwVdz+NpgZ1cRR  ndQbaFQSmt4F0yHIUIGsP0gjzFc8gen+cUU2L34BeXy9+b+pRl6nYwGAkfYce0Nw  ro4DoVRbf/DskjGXUzWNblkCgYAMWBMxccu3y1eIiPTrpeWnaAI6jsUFVqUik36R  KaPWM6APqjLRpeb+EGgCQQTtQpqFwnZa2lXqlospGdGu1dy9Rn8ibGpbyk/S5ANl  8uGfLRjRWwnS+q+erFI1lVp0HT1Mpu+Fj8dK2p1SBKDw7c1E4RNVbBGDfihFXVqy  ySD5bQKBgQDJt2QT2oxFP/UYU1OQO1Vu38U82Yw5F7XCFE5GaUWv2n+ID/4kg6IR  sCXOqlg8u/Hat/tsAI4WNjyjXSLLdCvF85d6Sx2/pbPMfNBttDr6Oj0+CikklQCi  1PSx8Bw50vWJVZFlHSC0thG7psyrflbOoLBKdwfvBkrfSrpfyNeQGQ==  -----END RSA PRIVATE KEY----- |

Add variables to the CICD Settings



Add the hardening role to the repository (saved onto GitLab however could not download from GitLab therefore included screenshot)



Add the following code to the .gitlab-ci.yaml file located in the Django.nv repository.

ansible-hardening:

  stage: prod

  image: willhallonline/ansible:2.9-ubuntu-18.04

  before\_script:

    - mkdir -p ~/.ssh

    - echo "$DEPLOYMENT\_SERVER\_SSH\_PRIVKEY" | tr -d '\r' > ~/.ssh/id\_rsa

    - chmod 600 ~/.ssh/id\_rsa

    - eval "$(ssh-agent -s)"

    - ssh-add ~/.ssh/id\_rsa

    - ssh-keyscan -t rsa $DEPLOYMENT\_SERVER >> ~/.ssh/known\_hosts

  script:

    - echo -e "[prod]\n$DEPLOYMENT\_SERVER" >> inventory.ini

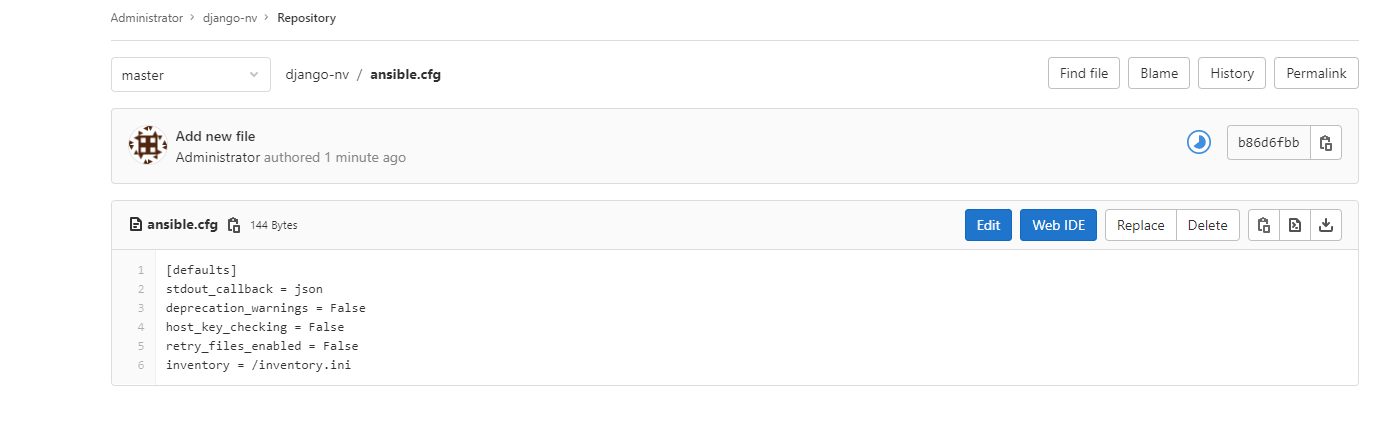
    - ansible-galaxy install dev-sec.os-hardening

    - ansible-playbook -i inventory.ini ansible-hardening.yml

To run the playbook in dry mode (check mode): this line of code must be edited slightly:

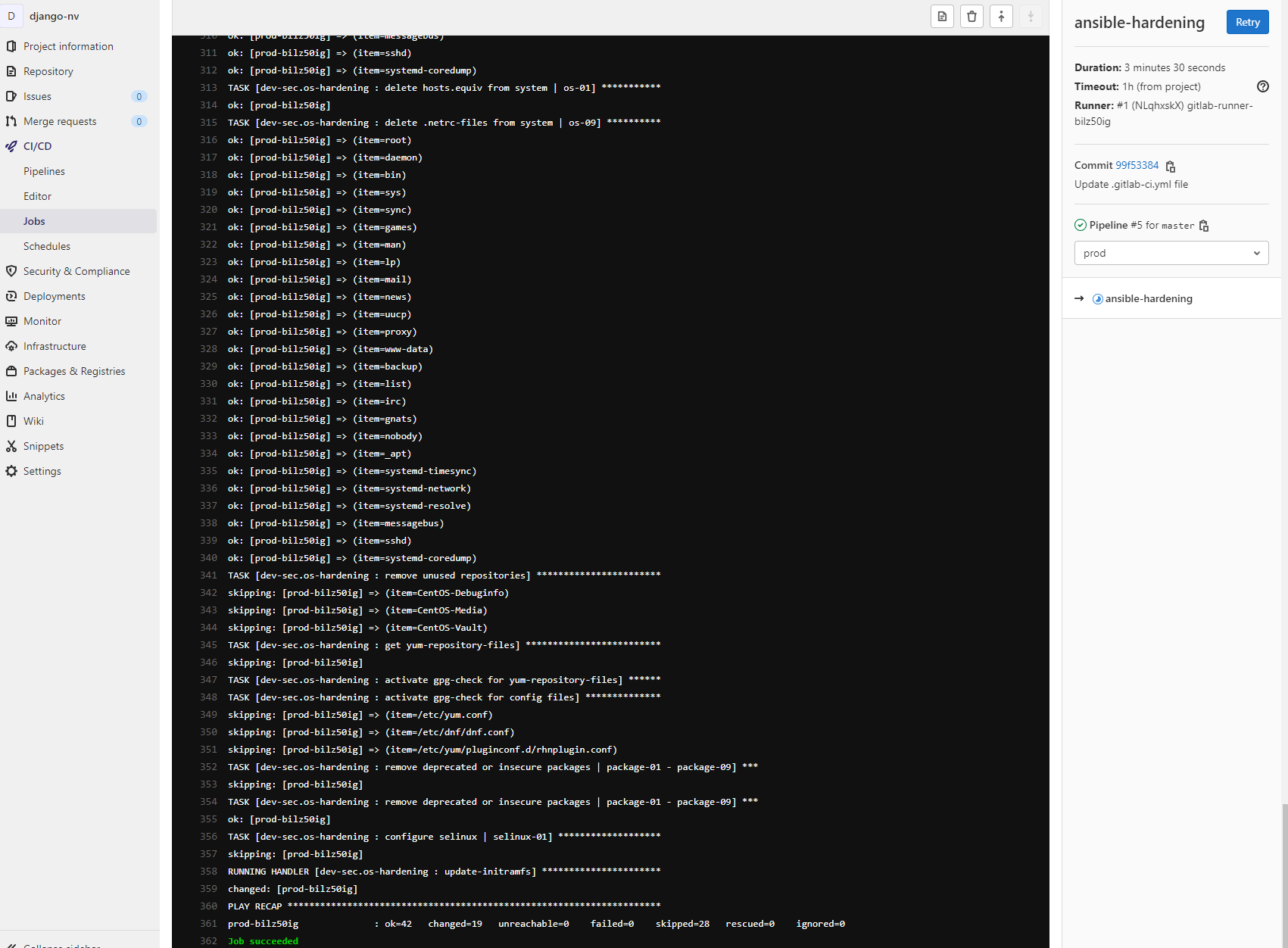
- ansible-playbook -i inventory.ini ansible-hardening.yml --check

To output the file as json, first the ansible config file needs to be added to the repository. (Again, struggled to save therefore screenshot added here). This config sets the output file in a machine readable format (json file). Output of scans and reports need to be saved in a machine-readable format because a large amount of data can be produced and it is difficult/impossible for a person to read everything. It allows for computers to understand the output in a standardised format and therefore can be shared and utilised by other machines if there is a need for it. It allows for collaboration between machines and can allow them to evaluate and produce further analysis.

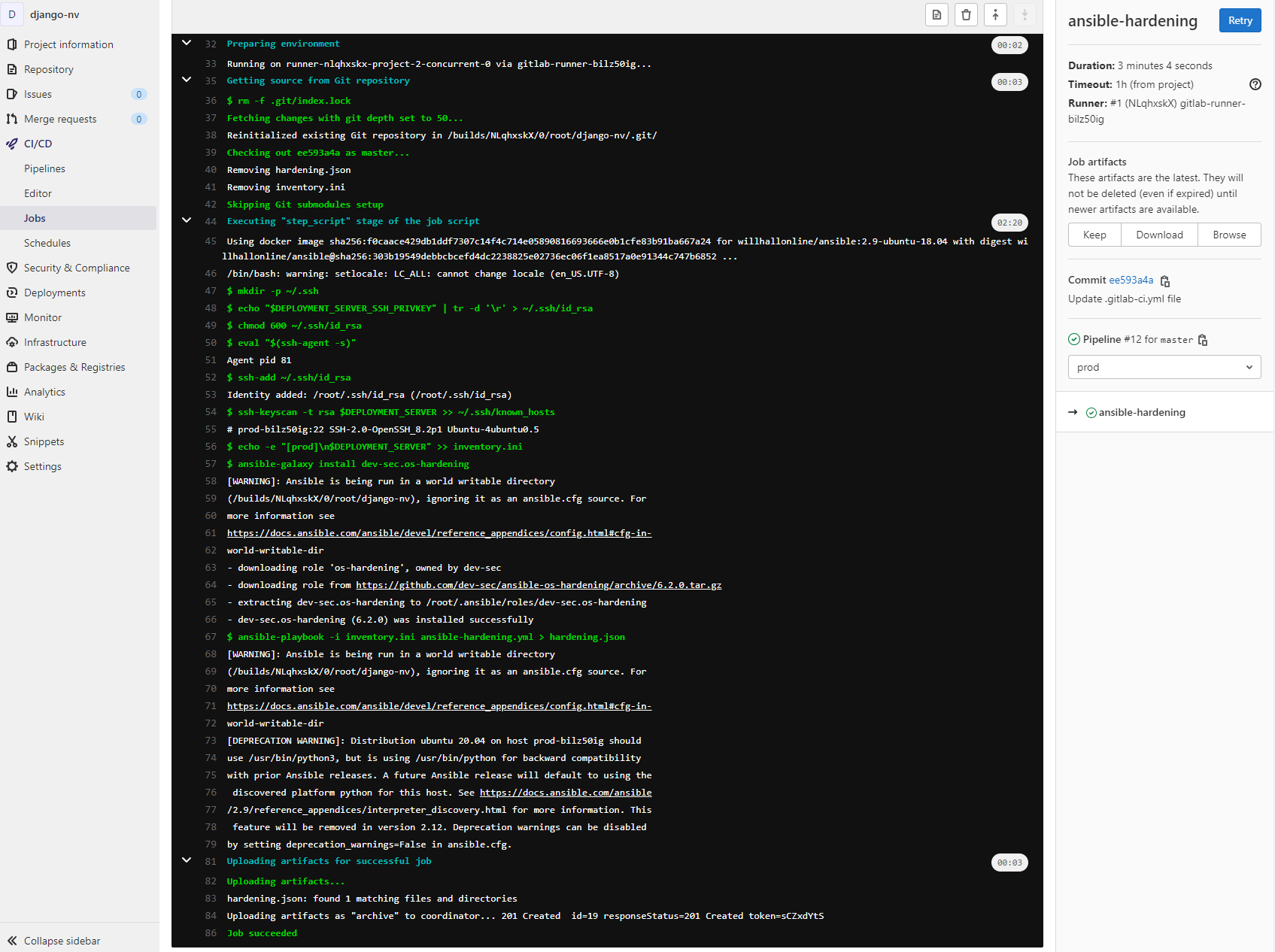


To see the results, we visit <https://gitlab-bilz50ig/root/django.nv/pipelines> to see the results of the new pipeline. We can also see the result file on the server, in the pipelines side panel or above the commit file.

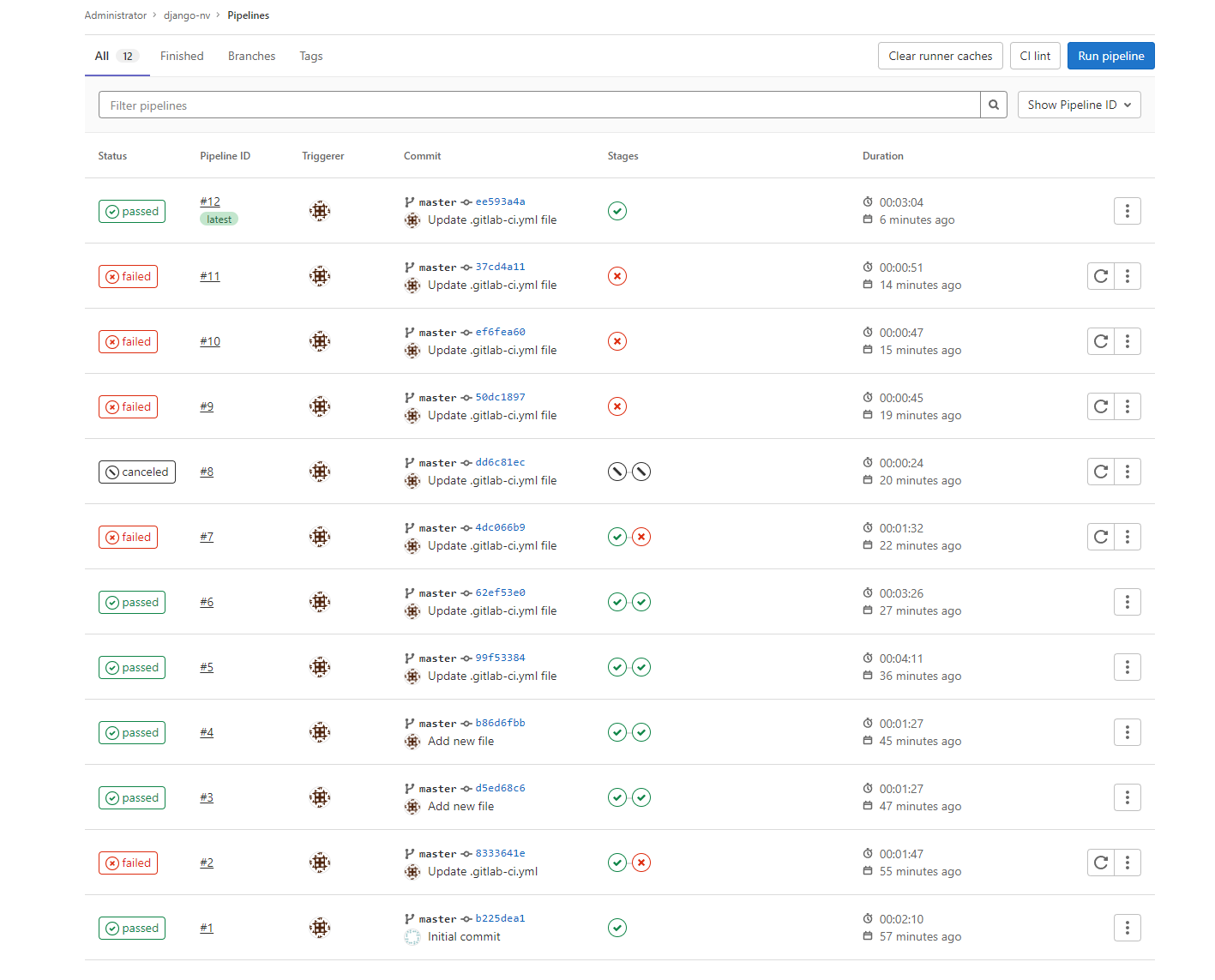
This is the output for the CICD Pipeline Job, showing the production environment has been hardened. There have been 19 changes made to the production machine.



This shows the results being saved to the repository in artifacts. This was saved to the CIServer for further processing.



Here is a list of the pipelines that were run and are shown below:



## Challenge 3: Scan for secrets in django.nv repository (15 points) In this challenge, you will use Gitlab CI to implement a CI/CD pipeline using a secret scanning tool named detect-secrets with the following details:

* Create a job called secrets-scanning under the build stage to run a tool named detect-secrets using docker (example: hysnsec/detect-secrets)
* The secrets-scanning job must fail the build if issues are found (please do not just use exit 1 to achieve job failure)
* The secrets-scanning job must save the detect-secrets results/output on the CI server for further processing in a machine-readable format like JSON, CSV, XML, etc.
* secrets-scanning job must implement all the applicable DevSecOps Gospel (best practices)
* As always, test everything locally on the DevSecOps-box machine before integrating it into the CI/CD pipeline

### Solution

Following best practices, the detect-secrets scan was run locally first before embedding into the CICD Pipeline in GitLab. The source code was downloaded from the Django.nv webapp and entered into the webapp

git clone https://gitlab.practical-devsecops.training/pdso/django.nv webapp

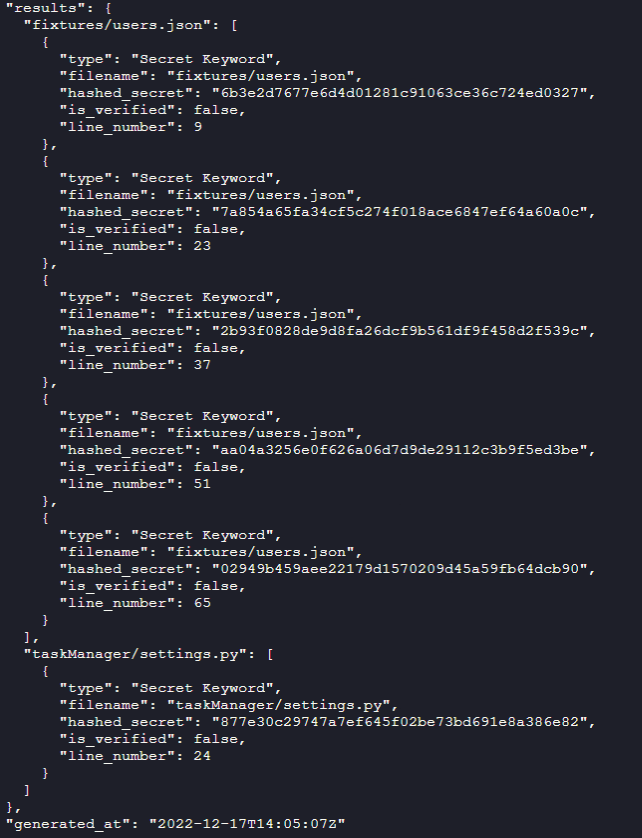
cd webapp

Install detect.secrets scanner by YELP was downloaded using the pip3 install function.

pip3 install detect-secrets

The detect secrets scan was run locally on the prod machine, with the tee function creating a JSON file with the results of the scan and outputting the scan locally. There was several secrets found and produces the file name, its verification status, the type of secret and the line number it was found on

detect-secrets scan | tee secrets.json



Since this was run locally, it can now be embedded into the CI/CD Pipeline located in the Django.nv repository.

image: docker:latest

services:

- docker:dind

stages:

- build

- test

- release

- preprod

- integration

- prod

build:

stage: build

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 check

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

secrets-scanning:

stage: build

script:

- docker pull hysnsec/detect-secrets

- docker run --user $(id -u):$(id -g) -v $(pwd):/src --rm hysnsec/detect-secrets file:///src scan | tee secrets.json

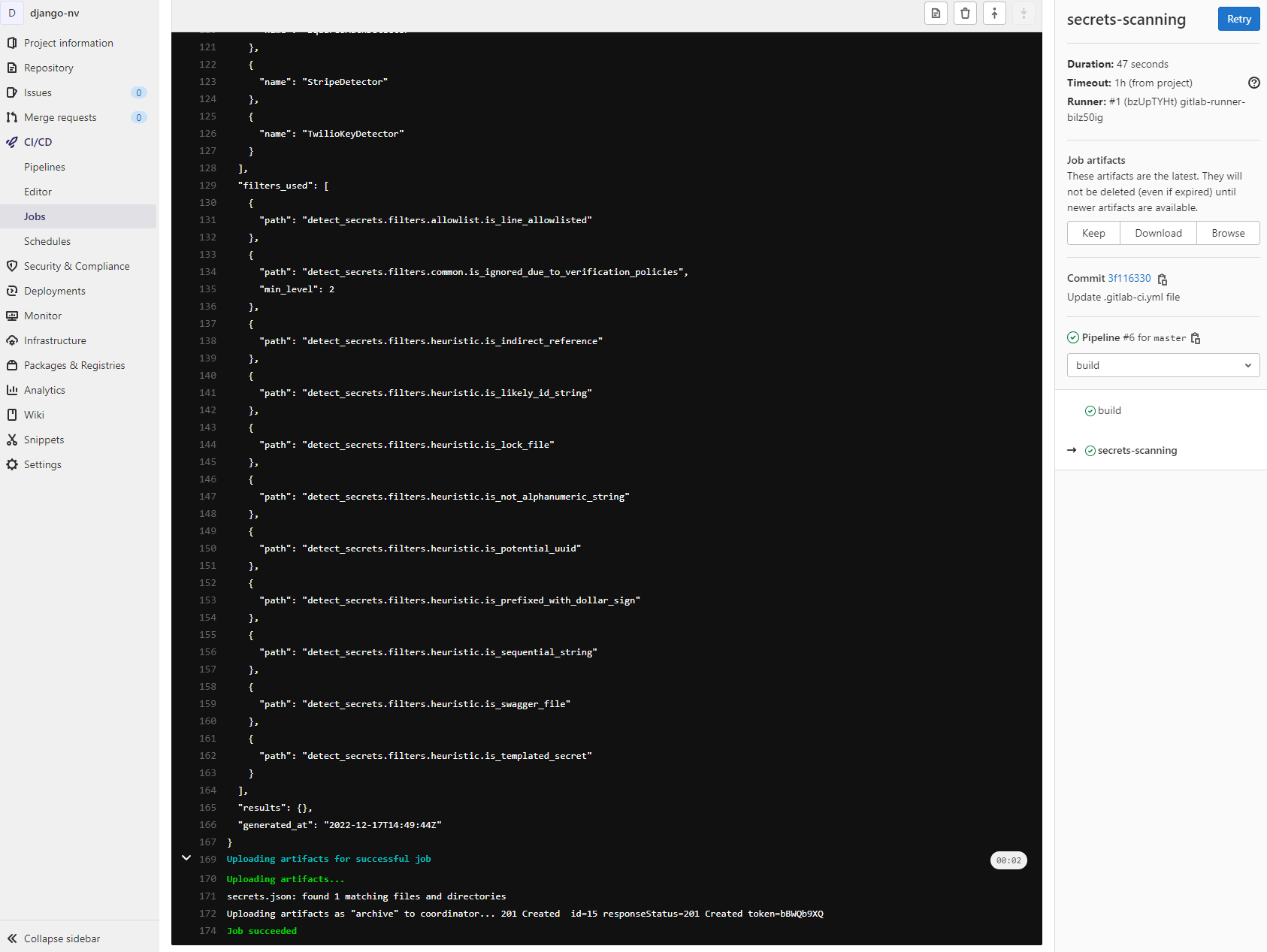
artifacts:

paths: [secrets.json]

when: always

allow\_failure: true

To see the results of the pipeline and the detect-secrets job, we visit <https://gitlab-bilz50ig/root/django.nv/pipelines> to see the results of the new pipeline. We can also see the result file on the server, in the pipelines side panel or above the commit file.



Unsure as to why there are no results, considering the local scan found several secrets. Therefore, there is a problem with the Gitlab file however I ran out of time to check where this problem could be. There are 0 secrets found within this scan, and the artifacts would show this scan. There was again a problem with downloading from GitLab so the scan was saved but cannot be shared within the exam report. Cannot find how to implement into CICD Pipeline. I believe it is not scanning any git repository, indicated by the code.

## Challenge 4: Run ZAP Scan against the django.nv application (production) and upload results to Defect Dojo (20 points) In this challenge, you will run a ZAP baseline scan on the django.nv application(production) from the DevSecOps Box, then integrate baseline scan into CI/CD pipeline with the following details:

* Run ZAP Scan on the django.nv app, and upload the results of the ZAP baseline scan into defect dojo's engagement id 1, using upload-results.py python script
* Ensure the ZAP Scan identifies more application urls for scanning using various available spidering options
* Create a zap job in the integration stage to scan the production app (django.nv), and automatically upload the ZAP Scan results to Defect Dojo on every scan

### Solution

Download source code from the Django.nv repository on gitlab and cd into webapp

git clone https://gitlab.practical-devsecops.training/pdso/django.nv webapp

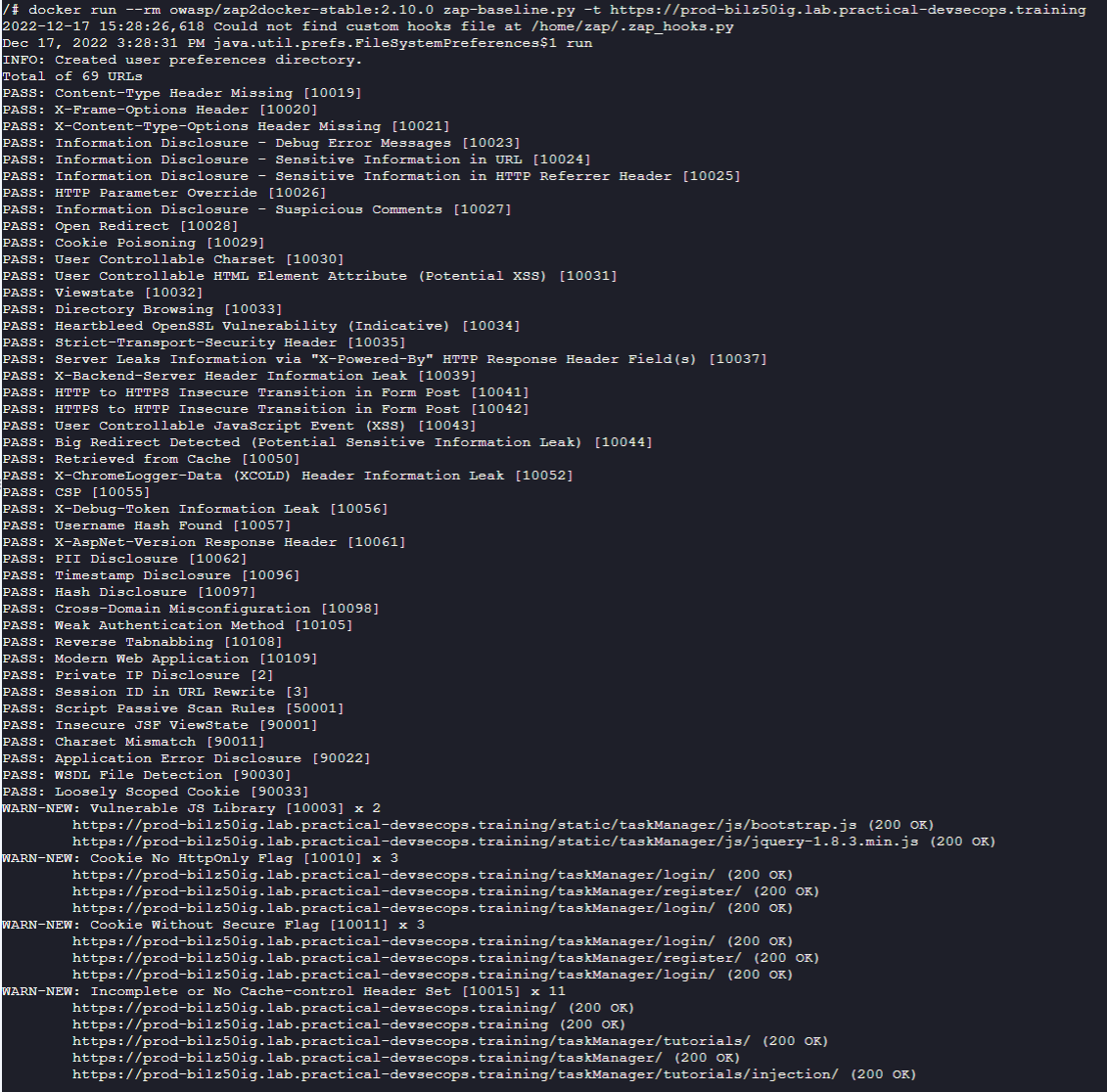
cd webapp

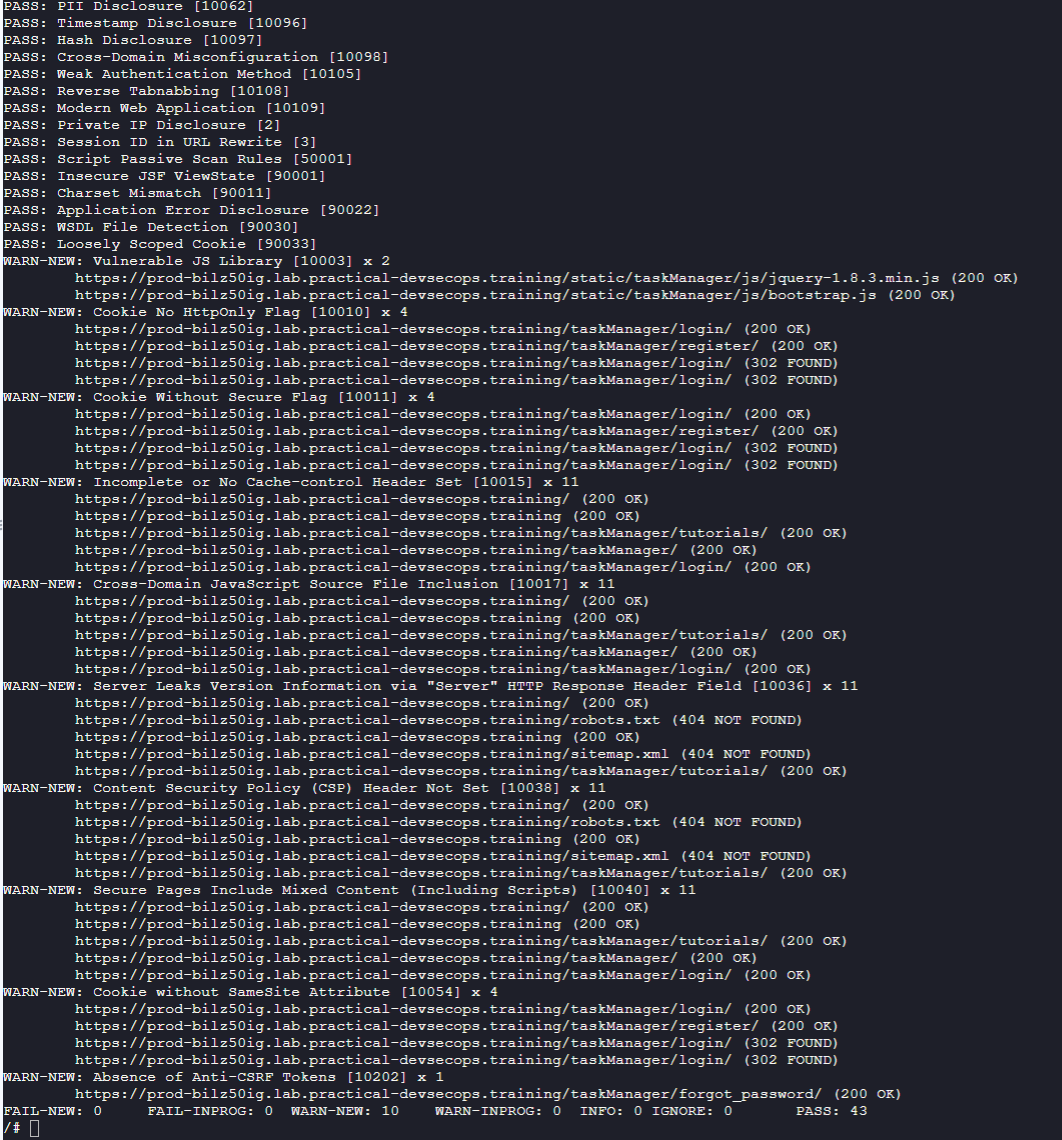
Download stable ZAP docker image using docker pull

docker pull owasp/zap2docker-stable:2.10.0

Run the zap baseline against the production machine and save as JSON (machine readable format)

docker run --user $(id -u):$(id -g) -w /zap -v $(pwd):/zap/wrk:rw --rm owasp/zap2docker-stable:2.10.0 zap-baseline.py -t https://prod-bilz50ig.lab.practical-devsecops.training -J zap-output.json





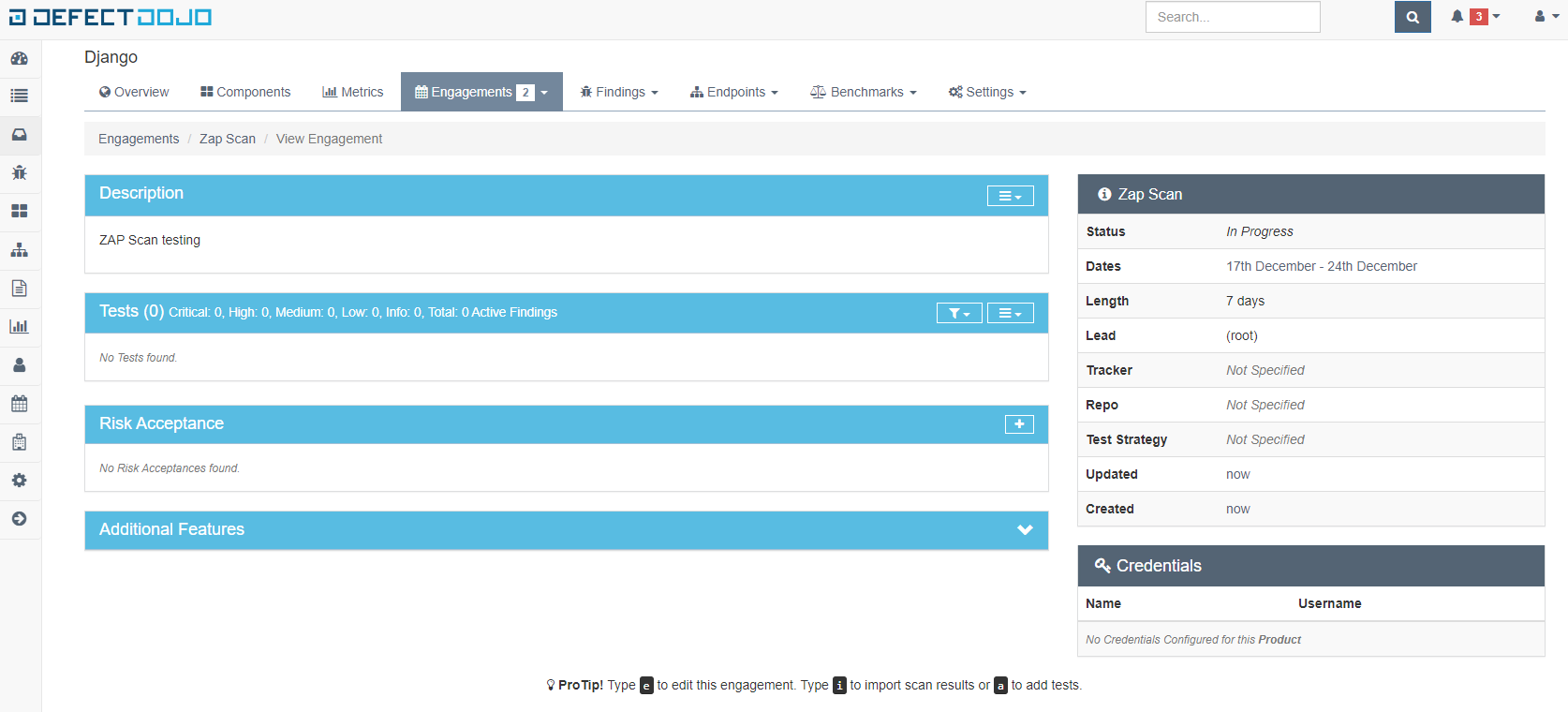
As we can see from the scan, there are 10 new warnings found within the production machine, with each warning showing the contributing reason and https:// related to the production machine.

To view in JSON format, use cat and the zap-output.json

Cat zap-output.json



Create a new interactive engagement on Defect Dojo within Django called Zap Scan. This should be set to a time period within the exam lifecycle. The status of the engagement should also be set to in progress



Download upload results locally into the DevSecOps box.

curl [https://gitlab.practical-devsecops.training/-/snippets/3/raw -o upload-results.py](https://gitlab.practical-devsecops.training/-/snippets/3/raw%20-o%20upload-results.py)

Install requests module as well.

Pip3 install requests

Create API\_KEY locally in the Linux terminal.

export API\_KEY=$(curl -s -XPOST -H 'content-type: application/json' https://dojo-bilz50ig.lab.practical-devsecops.training/api/v2/api-token-auth/ -d '{"username": "root", "password": "pdso-training"}' | jq -r '.token' )

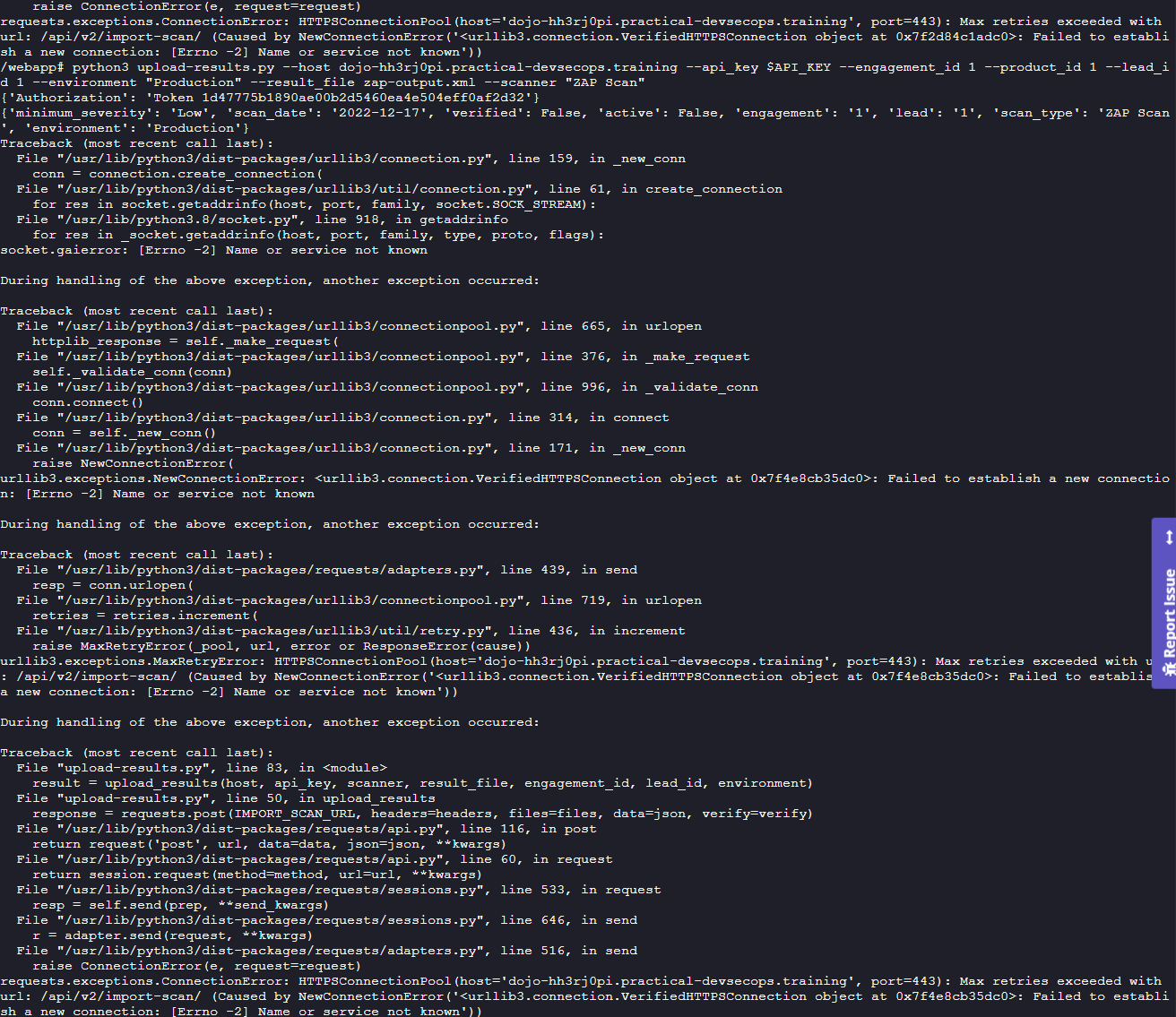
The api key can be validified by echoing the API\_KEY.

echo API\_KEY

Upload results to Dojo using the upload-results.py function

python3 upload-results.py --host dojo-hh3rj0pi.practical-devsecops.training --api\_key $API\_KEY --engagement\_id 1 --product\_id 1 --lead\_id 1 --environment "Production" --result\_file zap-output.xml --scanner "ZAP Scan"

Unknown error was unsure how to fix. Tried changing the id’s of the environment and the machine format of the ZAP Scanner but still not fixed. The same issue was found when trying to run in GitLab.



Seeing as the scan was successful but could not be uploaded, it can still be embedded into the Gitlab Django.nv repository using the following code. The virtual environments are created and the ZAP scan is embedded.

image: docker:latest

services:

- docker:dind

stages:

- build

- test

- release

- preprod

- integration

- prod

build:

stage: build

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 check

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

dast-zap:

  stage: integration

  before\_script:

    - apk add py-pip py-requests

  script:

    - docker pull owasp/zap2docker-stable:2.10.0

    - docker run --user $(id -u):$(id -g) -w /zap -v $(pwd):/zap/wrk:rw --rm owasp/zap2docker-stable:2.10.0 zap-baseline.py -t https://prod-hh3rj0pi.lab.practical-devsecops.training -J zap-output.json

  after\_script:

    - - python3 upload-results.py --host $DOJO\_HOST --api\_key $DOJO\_API\_TOKEN --engagement\_id 1 --product\_id 1 --lead\_id 1 --environment "Production" --result\_file zap-output.json --scanner "ZAP Scan"

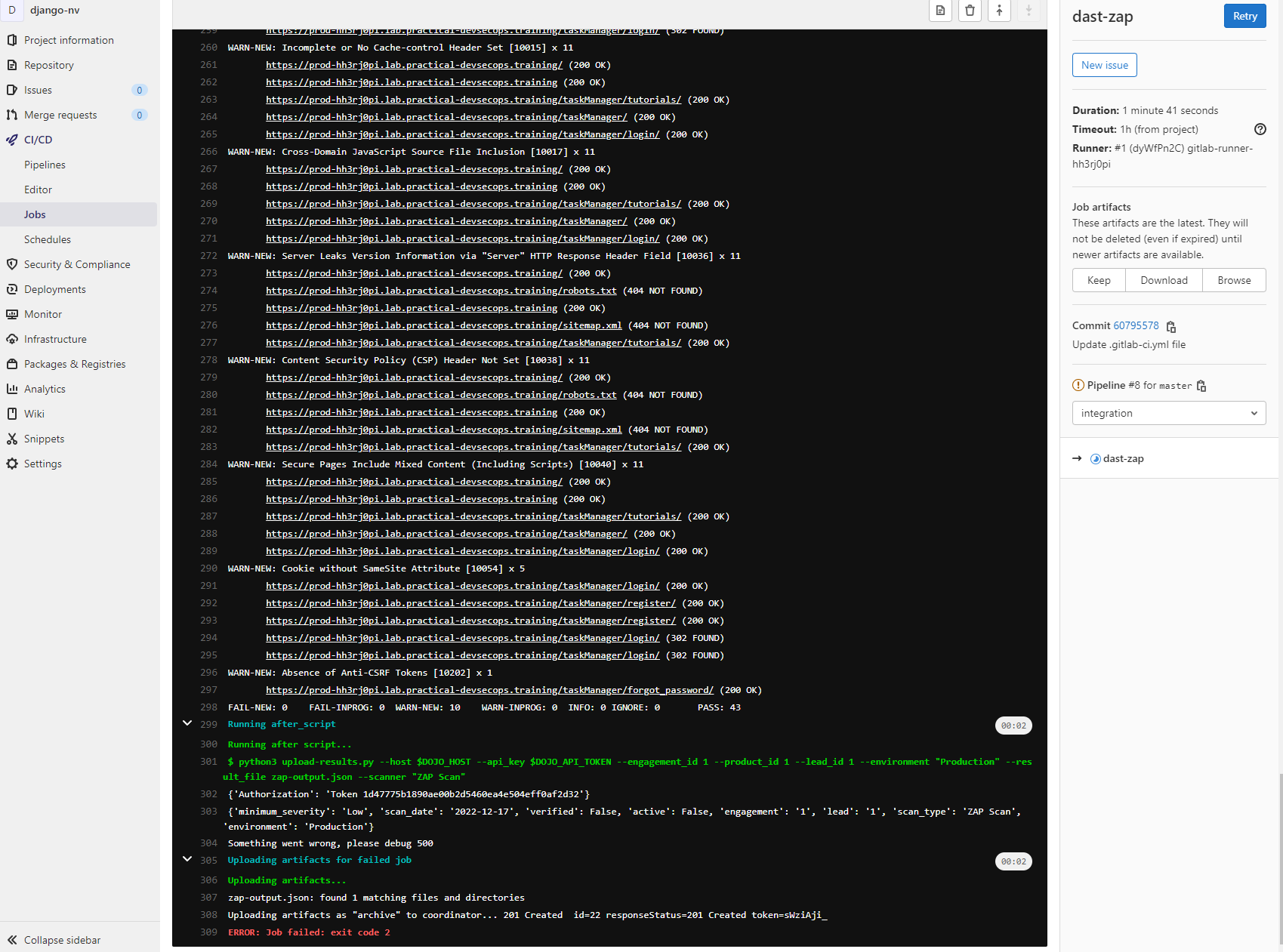
  artifacts:

    paths: [zap-output.json]

    when: always

  allow\_failure: **true**

To see the results of the pipeline and the detect-secrets job, we visit <https://gitlab-hh3rj0pi/root/django.nv/pipelines> to see the results of the new pipeline. We can also see the result file on the server, in the pipelines side panel or above the commit file.



Hit with error 500 and was unable to rectify, therefore decided not to add to DefectDojo and just complete a ZAP Scan in the GitLab server. However, the expected output would be a scan that is saved to the CI Server and the results uploaded to Defect Dojo every time the pipeline is run.

image: docker:latest

services:

- docker:dind

stages:

- build

- test

- release

- preprod

- integration

- prod

build:

stage: build

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 check

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

dast-zap:

stage: integration

script:

- docker pull owasp/zap2docker-stable:2.10.0

- docker run --user $(id -u):$(id -g) -w /zap -v $(pwd):/zap/wrk:rw --rm owasp/zap2docker-stable:2.10.0 zap-baseline.py -t https://prod-hh3rj0pi.lab.practical-devsecops.training -J zap-output.json

after\_script:

- docker rmi owasp/zap2docker-stable:2.10.0

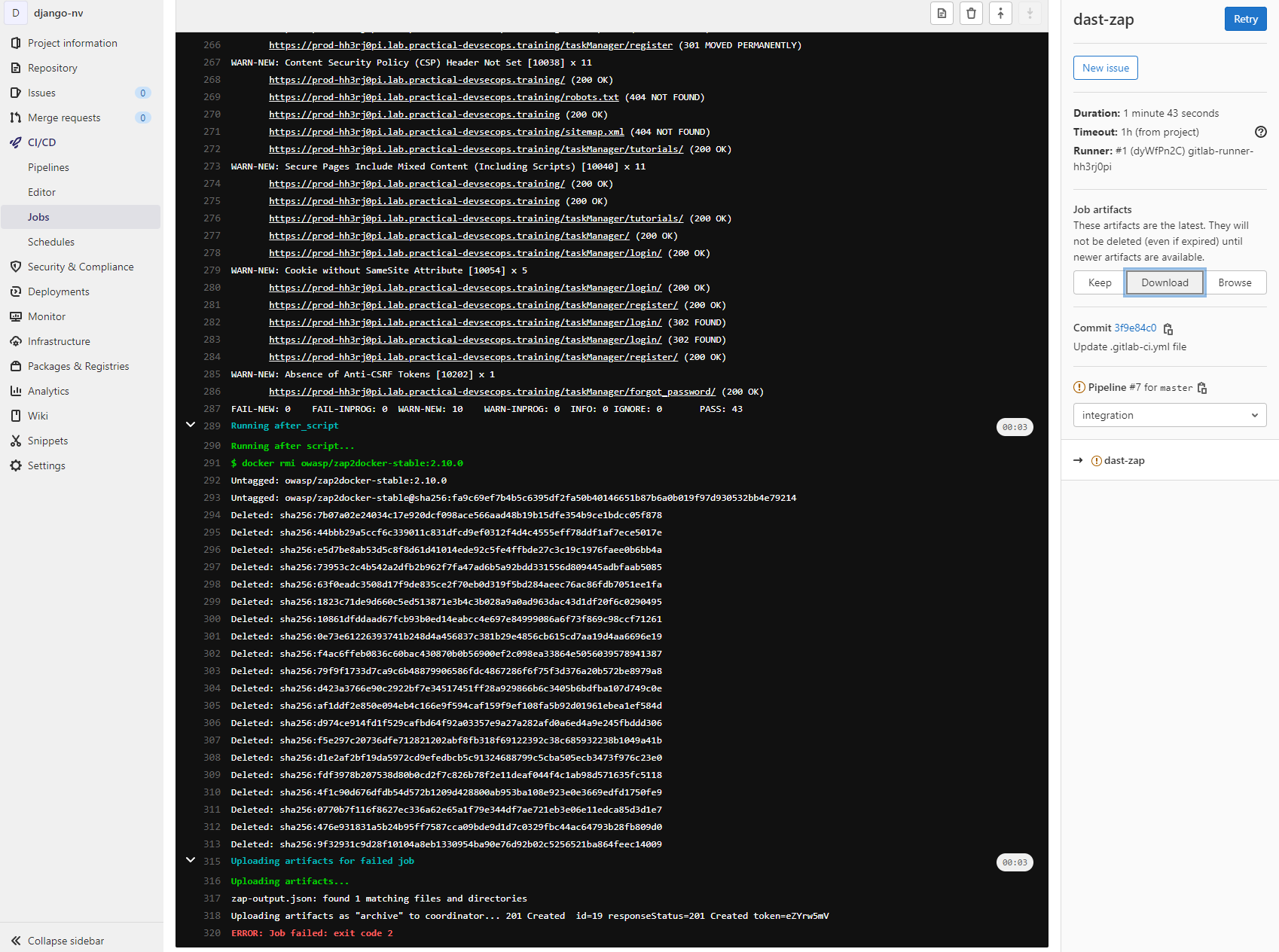
artifacts:

paths: [zap-output.json]

when: always

allow\_failure: true

To see the results of the pipeline and the detect-secrets job, we visit <https://gitlab-hh3rj0pi/root/django.nv/pipelines> to see the results of the new pipeline. We can also see the result file on the server, in the pipelines side panel or above the commit file.



The resulting output is saved to the CI Server and we cleaned up the image to save the disk space in GitLab. The results are saved in a machine readable format and the build is allowed to fail as per the level 1 and 2 of the DevSecOps best practices.

## Challenge 5: Run inspec linux-baseline profile on the production machine (15 points) In this challenge, you will run linux-baseline inspec profile on the production machine, and fix failed tests/errors on the production machine as outlined below:

* Run linux-baseline inspec profile on production machine from the DevSecOps-Box
* Fix the inspec control failures on the production machine manually (not using any automation), you can log in to the production machine using the ssh command
* Explain how you manually (not using any automation) fixed inspec control failures
* Ensure the linux-baseline profile checks for the presence of Antivirus software on the production machine (if the presence of the Antivirus software is not being checked in the linux-baseline profile, please include the Antivirus check)
* 10 Bonus points will be awarded if the manual fixes are added to the ansible role os-hardening (of dev-sec)

### Solution

Download inspec Debian package which has the linux-baseline also in it

wget https://packages.chef.io/files/stable/inspec/4.37.8/ubuntu/18.04/inspec\_4.37.8-1\_amd64.deb

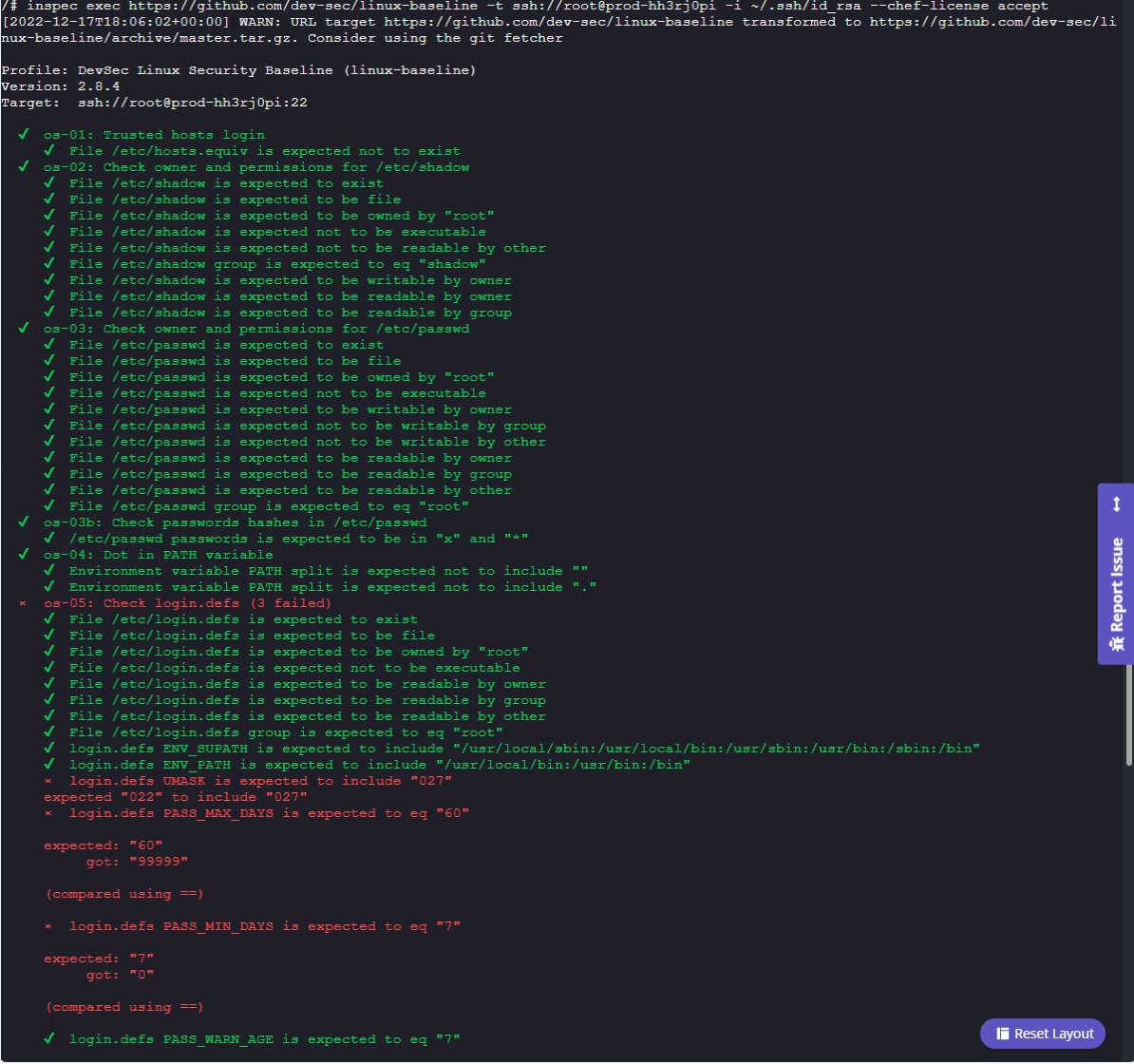
Install the downloaded inspec Debian package

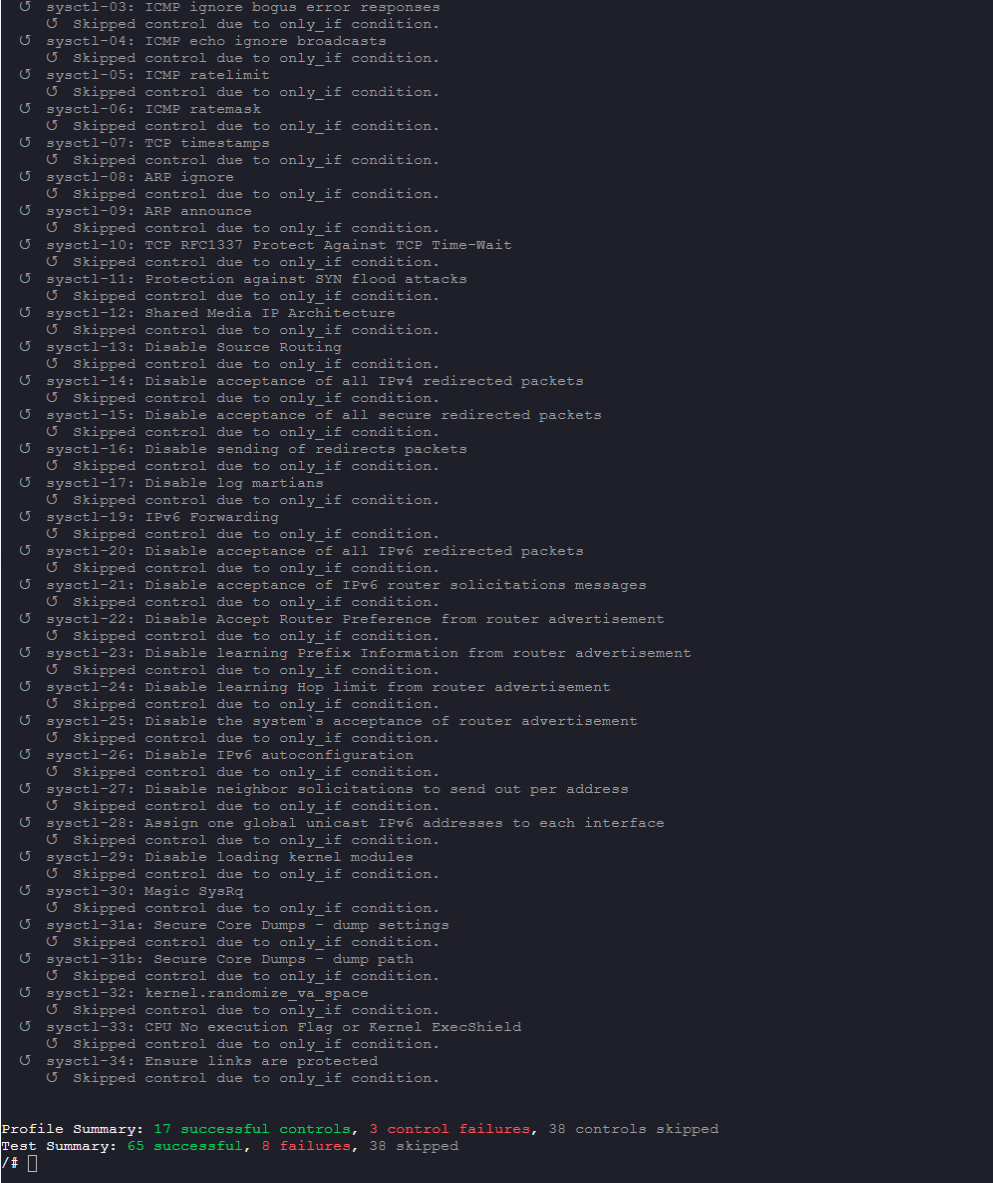
dpkg -i inspec\_4.37.8-1\_amd64.deb

Run inspec against the prod machine using linux-baseline

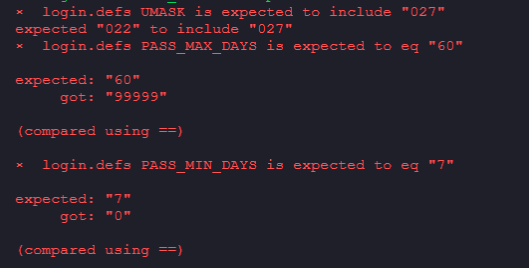
inspec exec https://github.com/dev-sec/linux-baseline -t ssh://root@prod-hh3rj0pi -i ~/.ssh/id\_rsa --chef-license accept

This creates a total of 3 control failures and 8 overall failures





Fixing the issue can be done by entering into the production machine and taking steps to fix the control failures. This could be a number of things, for example installing software updates, modifying configuration files and making other changes to the system. For example, this is a failure highlighted below:

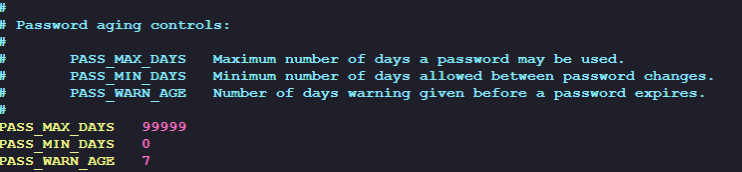


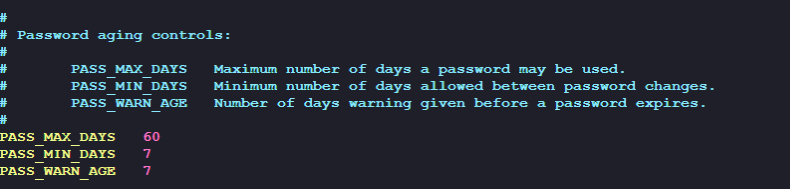
To fix this enter into the prod machine and file ound with failures

ssh prod-hh3rj0pi

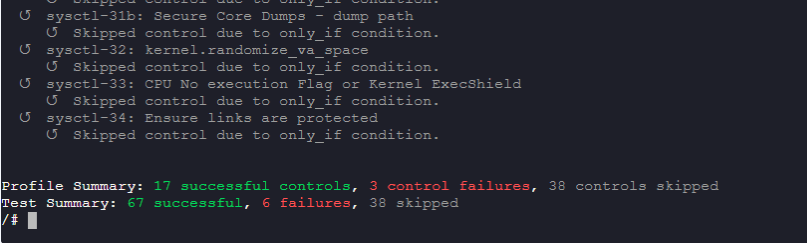
vi /etc/login.defs

As you can see these values do not match the expected.

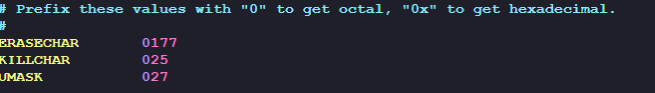


Edit to the expected values and save

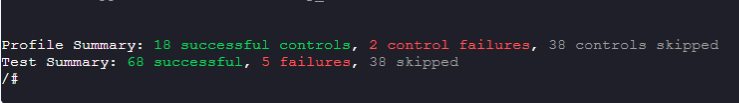
Run the inspec profile again



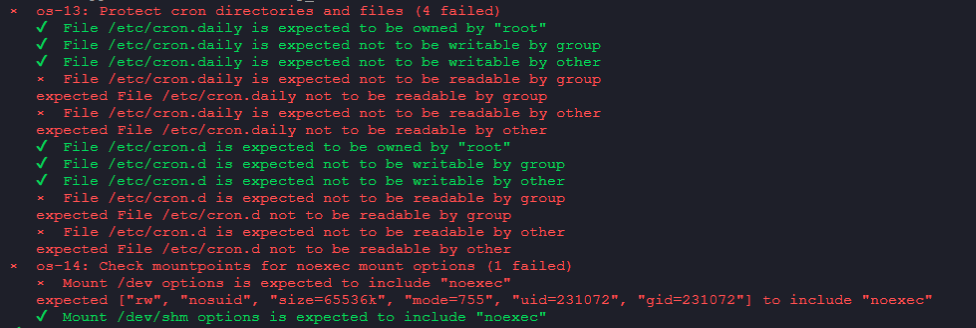
As you can see the number of failures has gone down. Fixing the entire control will reduce the number of control failures



Run the scan again



\fix the rest as follow:



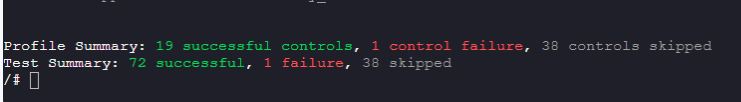
To remove permissions for cron.d and cron.daily use the code below

~# chmod o-rwx /etc/cron.d

~# chmod o-rwx /etc/cron.daily

~# chmod g-rwx /etc/cron.daily

~# chmod g-rwx /etc/cron.d



Cannot edit fstab file and therefore cannot rectify final failure. However, the etc/fstab file in unconfigured, and therefore may need to be configured to allow for the control to pass. This could also be fixed by mounting no-exec to the fstab file if it is configured. This should be close to fixing the /Dev issue found within the linux-baseline scan.