Faraday

Integrating Faraday in the software development process

Part Two Jenkins

This white paper produced by our technical team, shares important information to attack vulnerabilities from the first stage when developing software.

About Us

Faraday's mission is to **make security simple and accessible to everyone**, using our experience and passion to enable SMB/SME companies reducing their gap between exposure and remediation.
We truly believe that a clear understanding of your security posture is the primary key to reduce your attack surface, allowing you to make smarter decisions to protect your most valuable assets.
Security is a world-class engineering challenge and we want to help. We are a passionate leading team that wants to transform the way security works.

Outstanding research results published

Constant contribution to the global security community

+15 years working with F500

Speakers at the best security conferences

+60 employees worldwide

Get to know us at **faradaysec.com**

Integrating Faraday in the software development process - Part Two

Introduction

Last time we explained how easily you can integrate your Faraday instance into the software development process of an application written in Python and deployed on **Heroku**. Iin that post, we used **Github Actions** as a CD/CI tool.

Now we are going to perform the same task but instead of using **Github Actions**, we are going to use **Jenkins** with pipelines since this is one of the most extended CD/CI tools in the community.

Previous considerations

As we've covered all the theory and requirements in the previous post, we are going to assume that you already have an application written in Python, uploaded to a Git repository, and it is easily deployable to Heroku. If this is not the case, please visit the mentioned post to have more context.

In addition, it's required for you to have a running instance of **Jenkins** with **Blue Ocean** plugin installed which supports docker images. If this is not the case, we recommend making sure that all this configuration is done before starting to read this post, since trying to follow this post at the same time you're setting up Jenkins could be a little bit messy.

Lastly, we are not going to explain how to configure **Blue Ocean** and how to create **Jenkins** jobs because we understand that you already count with a minimum knowledge of **Jenkins**.

1. Creating the *Jenkinsfile* file

Our first step is to create a file in the root of our repository called **Jenkinsfile.**
This file will be readed by our Jenkins instance and this file indicates how the pipeline workflow will be.

```
faraday-vmpipeline
  > dbs
  > setup
  > static
  > templates
  > venv
      .gitignore
      import_scan.py
      import_scan.sh
      Jenkinsfile
```

Once this file is created, we need to write the following:

```
pipeline {
    agent none
    stages {
        stage('Build') {

        }
        stage('Deploy') {

        }
        stage('Scan') {

        }
        stage('Upload') {

        }
    }
```

Let's explain what each line means:

1. **Jenkins** pipelines let us use Groovy language to run tasks. In addition, the first line of every Jenkinsfile must start with **pipeline {}** closure.

2. Inside of the pipeline closure we can declare a lot of things, but **agents** and **stages** are mandatory. **Agents** indicate where they are going to be executed at different stages and could have different configurations (for example, they can run on Linux,

on Windows, etc). **Stages** are groups of steps that could have their own configuration set. In our example, we've defined **agent none** because we're going to use different **agents** in each **stage**. Because of **agent none**, we need to declare the agent in each **stage** as mandatory.

3. By default, all the stages will run sequentially, unless you specify that they should run in parallel. In our example, we are running only this pipeline in a sequential order. In addition, each stage will have a label to easily recognize when it's running on Jenkins**.**

2. Declaring the *Build* stage

Our first declaration will be the **Build** stage. This stage will install the required dependencies and will run **Bandit** to perform the static code check:

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:3.9.1'
                    args '-u root:root'
                }
            }
            steps {
                sh 'pip install -r requirements.txt'
                sh 'bandit -r . -f xml -o flaskapp_faraday_bandit.xml || true'
                stash name: 'flaskapp_faraday_ban-
dit.xml', includes: 'flaskapp_faraday_bandit.xml'
            }
        }
        stage('Deploy') {

        }
        stage('Scan') {

        }
        stage('Upload') {

        }
    }
}
```

As you can see, we've defined the **agent** inside the **Build** stage. As we've explained before, this is required here because we declared **agent none** globally. Our agent will be a docker container with the **python:3.9.1** image and with root perm issions.

Then, we defined the steps. Each step will be executed sequentially.

The first step installs the dependencies using the **requirements.txt** file (Bandit dependency is located there). The second step runs **Bandit** and saves the output in the **flaskapp_faraday_bandit.xml** file.

Some importants notes about this:

1. In our example, we are ignoring the **Bandit** execution result just to avoid making this file more complex.

2. A real application requires more steps, but this example is simplified to only install dependencies.

Lastly, since each **stage** could use different **agents**, we need to share the result file among them just as we did when we used **GitHub Actions**. To do that, we used the function **stash** which we'll combine with **unstash** later.

3. Declaring *Deploy* stage

The **Deploy** stage will upload the application to **Heroku** as shown here:

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:3.9.1'
                    args '-u root:root'
                }
            }
            steps {
                sh 'pip install -r requirements.txt'
                sh 'bandit -r . -f xml -o flaskapp_faraday_bandit.xml || true'
                stash name: 'flaskapp_faraday_ban-
dit.xml', includes: 'flaskapp_faraday_bandit.xml'
            }
        }
        stage('Deploy') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'HERO-
KU_API_KEY', variable: 'HEROKU_API_KEY'),
                    string(credentialsId: 'HERO-
KU_APP_NAME', variable: 'HEROKU_APP_NAME'),
                ])
```

```
                    {
                    script {
                        try {
                            sh 'git remote add heroku https://her-
oku:$HEROKU_API_KEY@git.heroku.com/$HEROKU_APP_NAME.git'
                        }
                        catch(Exception e) {
                            echo 'Remote heroku already exists'
                        }
                    }
                    sh 'git push heroku HEAD:master -f'
                }
            }
        }
        stage('Scan') {

        }
        stage('Upload') {

        }
    }
}
```

There are some interesting points to mention here:

1. We've used the credentials plugin when we called the **withCredentials** function. This plugin allows us to store secrets in **Jenkins** and retrieve them in our pipelines just as we did with **GitHub Actions**. We'll explain how to store secrets in **Jenkins** later in this post.

2. As you can see, in the first step we are adding the remote **Heroku** in Git and that is where we're using the secrets.

3. Lastly, we deploy the application to **Heroku** calling **git push**.

4. Declaring the Scan stage

The **Scan** stage will execute **zap** over our recently deployed application. This will be very similar to what we did with **Github Actions**.

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:3.9.1'
                    args '-u root:root'
                }
            }
            steps {
                sh 'pip install -r requirements.txt'
                sh 'bandit -r . -f xml -o flaskapp_faraday_bandit.xml || true'
                stash name:
                  'flaskapp_faraday_bandit.xml', includes: 'flaskapp_faraday_bandit.xml'
            }
        }
        stage('Deploy') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'HEROKU_API_KEY',
                     variable: 'HEROKU_API_KEY'),
                    string(credentialsId: 'HEROKU_APP_NAME',
                     variable: 'HEROKU_APP_NAME'),
                ]) {
                    script {
                        try {
                            sh 'git remote add heroku
                   https://heroku:$HEROKU_API_KEY@git.heroku.com/$HEROKU_APP_NAME.git'
                        }
                        catch(Exception e) {
                            echo 'Remote heroku already exists'
                        }
                    }
                    sh 'git push heroku HEAD:master -f'
                }
            }
        }
        stage('Scan') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'ZAP_SCAN_URL', variable: 'ZAP_SCAN_URL')
                ]) {
                    sh 'docker run -v $WORKSPACE:/zap/wrk/:rw --network=host -t owasp/
zap2docker-stable zap-baseline.py -t $ZAP_SCAN_URL -x flaskapp_faraday_zap.xml ||echo 0'
                    stash name: 'flaskapp_faraday_zap xml',
                     includes: 'flaskapp_faraday_zap.xml'
                }
            }
        }
        stage('Upload') {

        }
    }
}
```

As we mentioned, this is similar to what was done with **Github Actions**. However, there are some details to mention here:

1. We're using stored secrets again, in this case the stored secret ID is **ZAP_SCAN_URL**.

2. If you pay attention, we are using the **Jenkins** environment variable **$WORKSPACE**. There are a lot of environment variables you can see by entering **<jenkins-url>/env-vars.html** (in our case is **http://localhost:8080/env-vars.html**).

3. We used the **stash** function again since we need to use it in the **Upload** stage.

4. Again, the **zap** result is being ignored as we did with **Bandit** for this example, just to simplify the code.

5. Declaring the *Upload* stage

In the previous post we used the **faraday-cli** library to perform the upload of our results to our **Faraday** instance. We're going to do the same here. We will also show you an alternative using a custom docker image created by us.

```
pipeline {
    agent none
    stages {
        stage(‘Build’) {
            agent {
                docker {
                    image ‘python:3.9.1’
                    args ‘-u root:root’
                }
            }
            steps {
                sh ‘pip install -r requirements.txt’
                sh ‘bandit -r . -f xml -o flaskapp_faraday_bandit.xml || true’
                stash name: ‘flaskapp_faraday_bandit.xml’,
includes: ‘flaskapp_faraday_bandit.xml’
            }
        }
        stage(‘Deploy’) {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: ‘HEROKU_API_KEY’,
                    variable: ‘HEROKU_API_KEY’),
                    string(credentialsId: ‘HEROKU_APP_NAME’,
                    variable: ‘HEROKU_APP_NAME’),
                ]) {
                   script {
                        try {
                            sh ‘git remote add heroku
                   https://heroku:$HEROKU_API_KEY@git.heroku.com/$HEROKU_APP_NAME.git’
                        }
                        catch(Exception e) {
                            echo ‘Remote heroku already exists’
                        }
                    }
                    sh ‘git push heroku HEAD:master -f’
                }
            }
        }
        stage('Scan') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'ZAP_SCAN_URL', variable: 'ZAP_SCAN_URL')
                ]) {
                    sh 'docker run -v $WORKSPACE:/zap/wrk/:
rw--network=host -t owasp/zap2docker-stable zap-baseline.
py -t $ZAP_SCAN_URL -x flaskapp_faraday_zap.xml  || echo 0'
                    stash name: 'flaskapp_faraday_zap.xml',
 includes: 'flaskapp_faraday_zap.xml'
                }
            }
        }
        stage('Upload') {
            agent {
                docker {
                    image 'python:3.9.1'
                    args '-u root:root -v $WORKSPACE:/reports'
                }
            }
            steps {
                withCredentials([
                    string(credentialsId: 'FARADAY_URL', variable: 'FARADAY_URL'),
                    string(credentialsId: 'FARADAY_USERNAME',
 variable: 'FARADAY_USERNAME'),
                    string(credentialsId: 'FARADAY_PASSWORD',
 variable: 'FARADAY_PASSWORD')
                ]) {
                    unstash 'flaskapp_faraday_bandit.xml'
                    unstash 'flaskapp_faraday_zap.xml'
                    script {
                        CURRENT_DATE = (
                            script: "echo \${date + '%Y-%m-%d'}"
                            returnStdout: true
                        ).trim()
                        JOB_NAME = (
                            script: "echo $JOB_NAME| cut -d'/' -f1"
                            returnStdout: true
                        ).trim()
                    }
                    sh "pip install faraday-cli"
                    sh "faraday-cli auth -f $FARADAY_VMPIPELINES_FARADAY_URL -u
$FARADAY_VMPIPELINES_FARADAY_USERNAME -p $FARADAY_VMPIPELINES_FARADAY_PASSWORD"
                    sh "faraday-cli create_ws $JOB_NAME-$CURRENT_DATE-$BUILD_NUMBER"
                    sh "faraday-cli process_report -w  $JOB_NAME-$CUR-
RENT_DATE-$BUILD_NUMBER /reports/flaskapp_faraday_bandit.xml"
                    sh "faraday-cli process_report -w  $JOB_NAME-$CUR-
RENT_DATE-$BUILD_NUMBER /reports/flaskapp_faraday_zap.xml"
                }
            }
        }
    }
}
```

Again, we've used stored secrets in J**enkins: FARADAY_URL, FARADAY_ USERNAME and FARADAY_PASSWORD.**

As you can see, the first two steps are to **unstash** the previous generated files, so now they are accessible in the stage workspace.

The following step is the creation of the variables **CURRENT_DATE** and J**OB_NAME** that will be used to create the **Faraday** workspace name later.

Lastly, we execute the same commands we did in the previous post using the **faraday-cli** library. We need to pay attention to the stage configuration since here we're running all steps inside a docker image with Python 3. Also, we are linking the Jenkins workspace to this image in the directory /reports.

As mentioned before, we are going to show you an alternative to the previous configuration but now using a custom docker image created by us.

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                docker {
                    image 'python:3.9.1'
                    args '-u root:root'
                }
            }
            steps {
                sh 'pip install -r requirements.txt'
                sh 'bandit -r . -f xml -o flaskapp_faraday_bandit.xml || true'
                stash name:
                'flaskapp_faraday_bandit.xml', includes: 'flaskapp_faraday_bandit.xml'
            }
        }
        stage('Deploy') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'HEROKU_API_KEY',
                    variable: 'HEROKU_API_KEY'),
                    string(credentialsId: 'HEROKU_APP_NAME',
                    variable: 'HEROKU_APP_NAME'),
                ]) {
                    script {
                        try {
                            sh 'git remote add heroku
                  https://heroku:$HEROKU_API_KEY@git.heroku.com/$HEROKU_APP_NAME.git'
                        }
                        catch(Exception e) {
                            echo 'Remote heroku already exists'
                        }
                    }
                    sh 'git push heroku HEAD:master -f'
                }
            }

        }
        stage('Scan') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'ZAP_SCAN_URL', variable: 'ZAP_SCAN_URL')
                ]) {
                    sh 'docker run -v $WORKSPACE:/zap/wrk/:
rw --network=host -t owasp/zap2docker-stable zap-baseline.
py -t $ZAP_SCAN_URL -x flaskapp_faraday_zap.xml || echo 0'
                    stash name: 'flaskapp_faraday_zap.xml',
 includes: 'flaskapp_faraday_zap.xml'
                }
            }
        }
        stage('Upload') {
            agent any
            steps {
                withCredentials([
                    string(credentialsId: 'FARADAY_URL', variable: 'FARADAY_URL'),
                    string(credentialsId:
                   'FARADAY_USERNAME', variable: 'FARADAY_USERNAME'),
                    string(credentialsId:
                   'FARADAY_PASSWORD', variable: 'FARADAY_PASSWORD')
                ]) {
                    unstash 'flaskapp_faraday_bandit.xml'
                    unstash 'flaskapp_faraday_zap.xml'
                    script {
                        CURRENT_DATE = (
                            script: "echo \${date + '%Y-%m-%d'}"
                            returnStdout: true
                        ).trim()
                        JOB_NAME = (
                            script: "echo $JOB_NAME| cut -d'/' -f1"
                            returnStdout: true
                        ).trim()
                    }

                    sh "docker build https://github.com/infobyte/
                    docker-faraday-report-uplodaer.git#master -t faraday-uploader"

                   sh "docker run --rm -v $WORKSPACE:$WORKSPACE -e
HOST=$FARADAY_URL -e USERNAME=$FARADAY_USERNAME -e
PASSWORD=$FARADAY_PASSWORD -e WORKSPACE=$JOB_NAME-$CURRENT_DATE-$BUILD_NUMBER -e
FILES=$WORKSPACE/flaskapp_faraday_bandit.xml faraday-uploader"
                   sh "docker run --rm -v $WORKSPACE:$WORKSPACE -e
HOST=$FARADAY_URL -e USERNAME=$FARADAY_USERNAME -e PASSWORD=$FARADAY_PASSWORD -e
WORKSPACE=$JOB_NAME-$CURRENT_DATE-$BUILD_NUMBER -e FILES=$WORKSPACE/
flaskapp_faraday_zap.xml faraday-uploader"
                }
            }
        }
    }
}
```

This approach is pretty similar to the previous one but, instead of running inside a docker image, it runs on a Jenkins node.

The most important part of this approach appears in the last 3 lines of the stage:

1. First we perform an image build of our custom docker image.

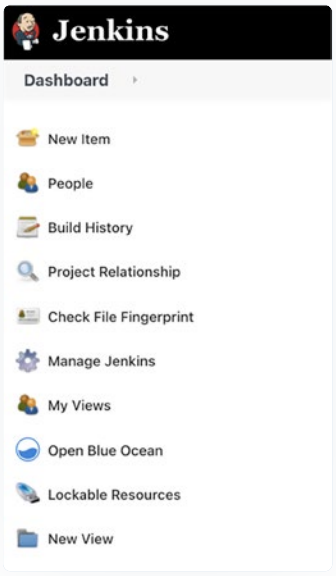
2. Then we run the container twice using the built image previously (one step per report file). It's important to mention that, in order to let the docker image access to unstashed report files, we've run the containers using the param **-v $WORKSPACE:$WORKSPACE** which allows docker containers to see the Jenkins workspace in an absolute path.

As you can see, the Jenkinsfile setup is pretty straightforward and it doesn't have a lot of differences compared to **Github Actions**. Now we are going to explain how to run our new Jenkinsfile in **Jenkins** using the Blue **Ocean plugin**.

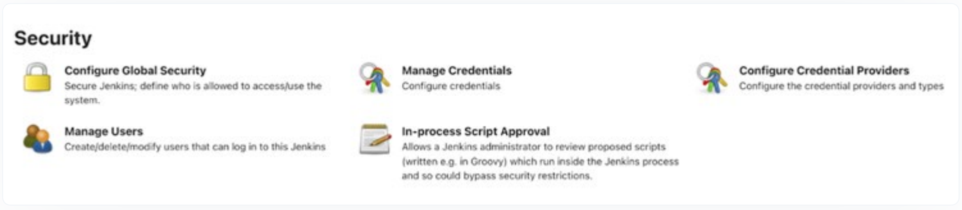
6. Storing secrets in Jenkins

As mentioned, we used secrets for the most important variables during the Jenkinsfile creation. We did this because storing credentials or keys in a public repository is not a good practice.

To add new secrets, we need to go to our **Jenkins** instance (in our case it is http://localhost:8080). Then, we need to click on **Manage Jenkins** in the left menu as you can see in the next image.



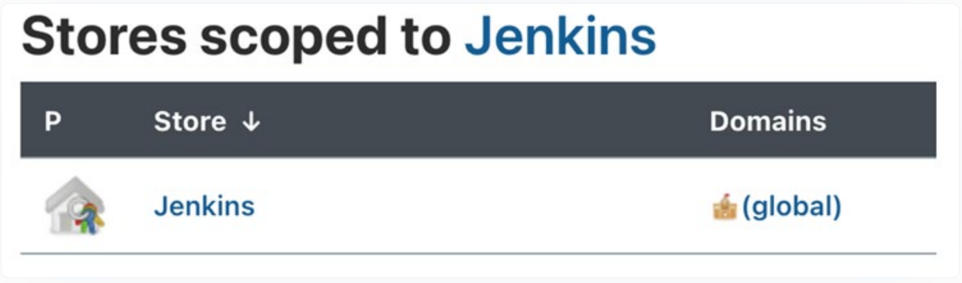
Now we need to find the **Manage Credentials** option in the **Security** section:



Security

Configure Global Security	Secure Jenkins; define who is allowed to access/use the system.
Manage Credentials	Configure credentials
Configure Credential Providers	Configure the credential providers and types
Manage Users	Create/delete/modify users that can log in to this Jenkins
In-process Script Approval	Allows a Jenkins administrator to review proposed scripts (written e.g. in Groovy) which run inside the Jenkins process and so could bypass security restrictions.

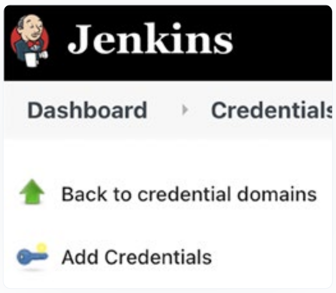
Once inside, we will click on the **(globals)** option:



Stores scoped to Jenkins

P	Store ↓	Domains
	Jenkins	(global)

If we've never added a secret, there will be a message saying that our credentials vault is empty with a link for us to create a new secret. In addition, we can click on the left menu option called **Add Credentials**:



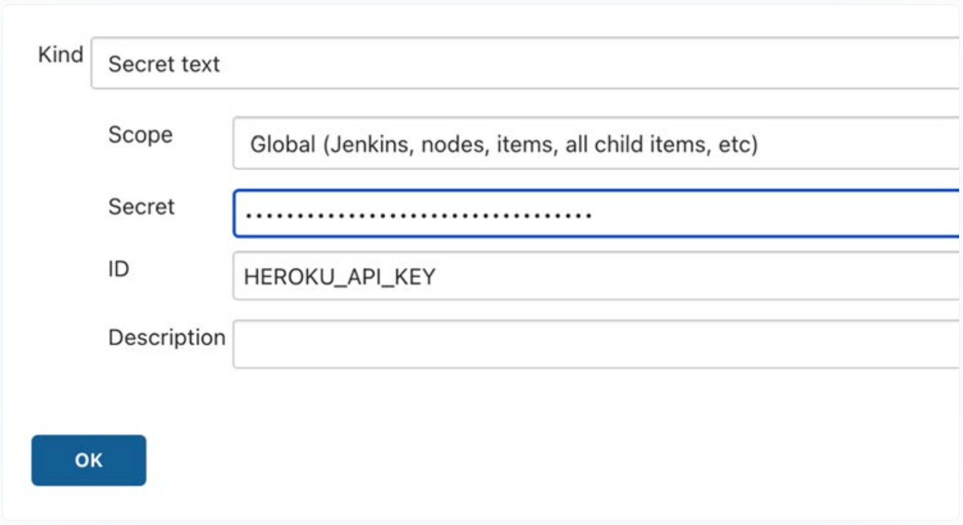
Jenkins

Dashboard › Credentials

Back to credential domains

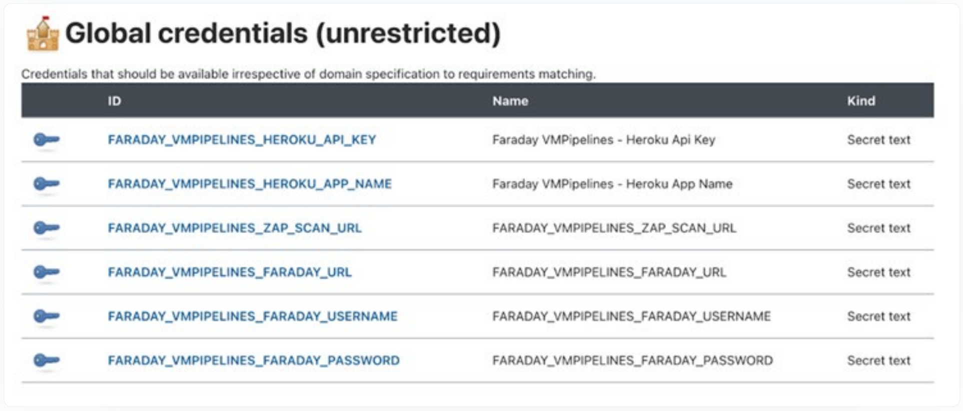
Add Credentials

Once we are located in the new credential page, we need to change the credential type. For example purposes, we are going to use Secret Text of credential type (for Faraday username and password too). After selecting the kind of credential, we need to write the ID and the Secret. The ID should be the same that we've used in the Jenkinsfile and in the secret is the value:



Kind	Secret text
Scope	Global (Jenkins, nodes, items, all child items, etc)
Secret	••••••••••••••••••••••••••••••••••••
ID	HEROKU_API_KEY
Description	

All credentials are available in **Jenkins** for all the jobs, so our recommendation is to use a prefix per job to avoid future problems. Please remember you need to update the Jenkinsfile with the new credential ids. In our case, we added the prefix **FARADAY_VMPIPELINES_**.



Global credentials (unrestricted)

Credentials that should be available irrespective of domain specification to requirements matching.

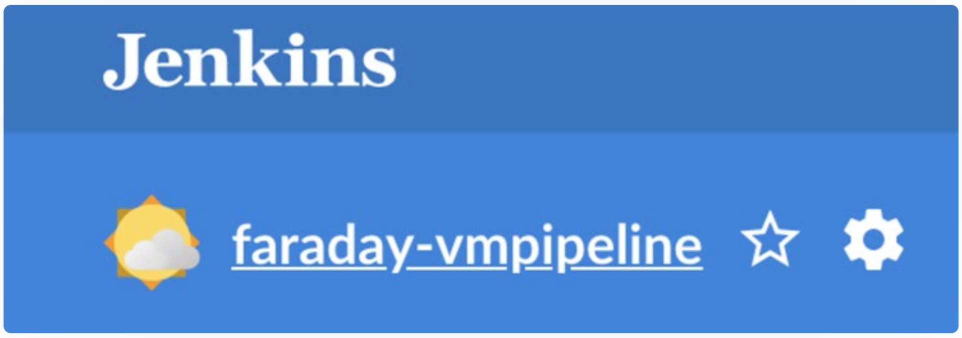
ID	Name	Kind
FARADAY_VMPIPELINES_HEROKU_API_KEY	Faraday VMPipelines - Heroku Api Key	Secret text
FARADAY_VMPIPELINES_HEROKU_APP_NAME	Faraday VMPipelines - Heroku App Name	Secret text
FARADAY_VMPIPELINES_ZAP_SCAN_URL	FARADAY_VMPIPELINES_ZAP_SCAN_URL	Secret text
FARADAY_VMPIPELINES_FARADAY_URL	FARADAY_VMPIPELINES_FARADAY_URL	Secret text
FARADAY_VMPIPELINES_FARADAY_USERNAME	FARADAY_VMPIPELINES_FARADAY_USERNAME	Secret text
FARADAY_VMPIPELINES_FARADAY_PASSWORD	FARADAY_VMPIPELINES_FARADAY_PASSWORD	Secret text

Once we've added all the secrets, we need to set up our pipeline entering to /blue url under our **Jenkins** instance (in our case it is http://localhost:8080/blue).

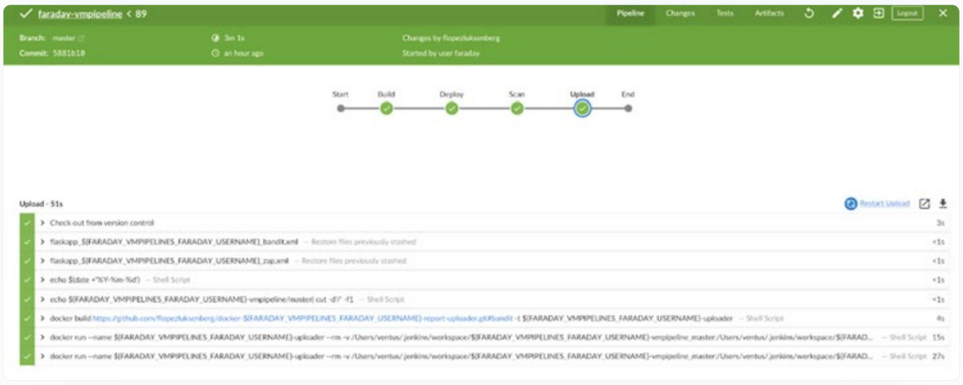
If this is your first time using this plugin, there will be only one option and it will be Add New Pipeline. If it is not the first time, you can click on the **New Pipeline** button.

The new pipeline assistant is pretty straightforward so we are not going to detail how to add the repository here. Please remember that if you have a GitHub repository you need to create a personal token and you need to add it in **Jenkins**. This will allow Jenkins to see all your repositories.

Once you've selected the repository that has the Jenkinsfile, Jenkins will run it for the first time. As happened with **GitHub Actions**, it's possible to set up the conditions to trigger the jobs running by clicking on the setup icon.



Once the execution is finished, you will see the results as in the following image:

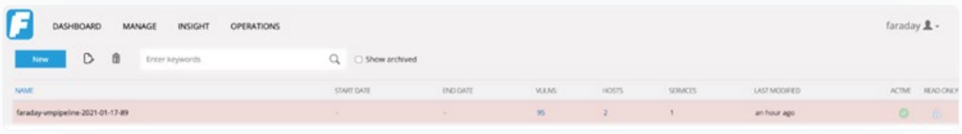


Branch: master
Commit: 5881b18
3m 1s
an hour ago
Changes by flopezluksenberg
Started by user faraday

Upload - 51s

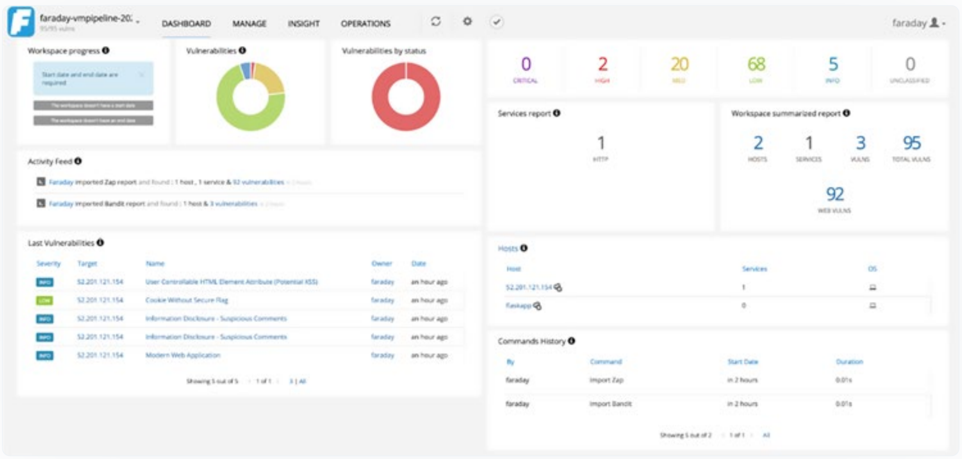
```
> Check out from version control                                                                    3s
> flaskapp_${FARADAY_VMPIPELINES_FARADAY_USERNAME}_bandit.xml — Restore files previously stashed    <1s
> flaskapp_${FARADAY_VMPIPELINES_FARADAY_USERNAME}_zap.xml — Restore files previously stashed       <1s
> echo $(date +'%Y-%m-%d') — Shell Script                                                           <1s
> echo ${FARADAY_VMPIPELINES_FARADAY_USERNAME}-vmpipeline/master| cut -d'/' -f1 — Shell Script       <1s
> docker build https://github.com/flopezluksenberg/docker-${FARADAY_VMPIPELINES_FARADAY_USERNAME}-report-uploader.git#bandit -t ${FARADAY_VMPIPELINES_FARADAY_USERNAME}-uploader — Shell Script   4s
> docker run --name ${FARADAY_VMPIPELINES_FARADAY_USERNAME}-uploader --rm -v /Users/ventus/jenkins/workspace/${FARADAY_VMPIPELINES_FARADAY_USERNAME}-vmpipeline_master:/Users/ventus/jenkins/workspace/${FARAD... — Shell Script   15s
> docker run --name ${FARADAY_VMPIPELINES_FARADAY_USERNAME}-uploader --rm -v /Users/ventus/jenkins/workspace/${FARADAY_VMPIPELINES_FARADAY_USERNAME}-vmpipeline_master:/Users/ventus/jenkins/workspace/${FARAD... — Shell Script   27s
```

As you've probably noticed, the execution results were successful. Now we can see our new workspace on our Faraday instance:



NAME	START DATE	END DATE	VULNS	HOSTS	SERVICES	LAST MODIFIED	ACTIVE	READ ONLY
faraday-vmpipeline-2021-01-17-89	-	-	95	2	1	an hour ago	●	

There you go. If you enter to the new workspace we will see the dashboard:



0	2	20	68	5	0
CRITICAL	HIGH	MED	LOW	INFO	UNCLASSIFIED

Services report

1 HTTP

Workspace summarized report

2	1	3	95
HOSTS	SERVICES	VULNS	TOTAL VULNS

92 WEB VULNS

Activity Feed

Faraday imported Zap report and found : 1 host , 1 service & 92 vulnerabilities

Faraday imported Bandit report and found : 1 host & 3 vulnerabilities

Last Vulnerabilities

Severity	Target	Name	Owner	Date
INFO	52.201.121.154	User Controllable HTML Element Attribute (Potential XSS)	faraday	an hour ago
LOW	52.201.121.154	Cookie Without Secure Flag	faraday	an hour ago
INFO	52.201.121.154	Information Disclosure - Suspicious Comments	faraday	an hour ago
INFO	52.201.121.154	Information Disclosure - Suspicious Comments	faraday	an hour ago
INFO	52.201.121.154	Modern Web Application	faraday	an hour ago

Showing 5 out of 5 · 1 of 1 · 5 | All

Hosts

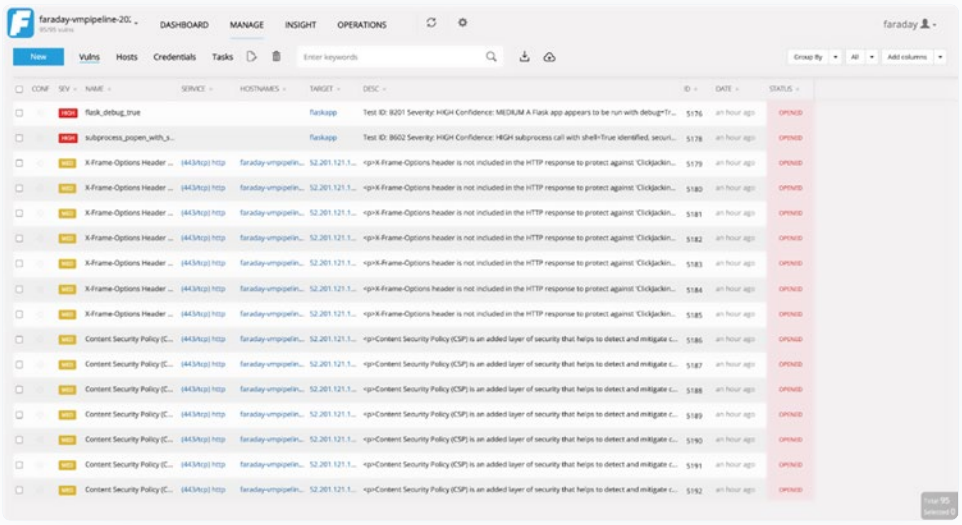
Host	Services	OS
52.201.121.154	1	🖥
flaskapp	0	🖥

Commands History

By	Command	Start Date	Duration
faraday	import Zap	in 2 hours	0.01s
faraday	import Bandit	in 2 hours	0.01s

Showing 5 out of 2 · 1 of 1 · All

As always, you can easily manage those vulns by using the Status Report:



CONF	SEV	NAME	SERVICE	HOSTNAMES	TARGET	DESC	ID	DATE	STATUS
	HIGH	flask_debug_true			flaskapp	Test ID: B201 Severity: HIGH Confidence: MEDIUM A Flask app appears to be run with debug=Tr...	5176	an hour ago	OPENED
	HIGH	subprocess_popen_with_s...			flaskapp	Test ID: B602 Severity: HIGH Confidence: HIGH subprocess call with shell=True identified, securi...	5178	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5179	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5180	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5181	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5182	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5183	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5184	an hour ago	OPENED
	MED	X-Frame-Options Header ...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>X-Frame-Options header is not included in the HTTP response to protect against 'ClickJackin...	5185	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5186	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5187	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5188	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5189	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5190	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5191	an hour ago	OPENED
	MED	Content Security Policy (C...	(443/tcp) http	faraday-vmpipelin...	52.201.121.1...	<p>Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate c...	5192	an hour ago	OPENED

Total 95
Selected 0

Conclusions

In the previous post we said that **Faraday** can be easily integrated in the software development cycle by using any CI/CD tool, which was demonstrated here.

You can use **GitHub Actions, Jenkins, Travis CI** or any other tool. The important factor here is to have earlier visibility of the vulns that are affecting our application, in order to decide how we are going to work with them.

Useful links

App Vuln Management: Integrating Faraday in the software development process

Docker Faraday Report Uploader
Example repository (branch: jenkins)
Faraday plugin list
OWASP Zap official site
Bandit official site
Vulnerable example app