Дипломная работа по профессии «Специалист по информационной безопасности»

Дата выполнения: 21.01.2023

Выполнил: Новиков Виталий Сергеевич

Оглавление

[Введение 2](#_Toc156745941)

[1 Этап 1. CI/CD 3](#_Toc156745942)

[1.1 Настройка пайплайн по сборке и доставке программного обеспечения 3](#_Toc156745943)

[1.2 Настройка облачных сервисов 3](#_Toc156745944)

[1.3 Настройка gitlab runners 5](#_Toc156745945)

[2 Этап 2. SAST 6](#_Toc156745946)

[2.1 Snyk 6](#_Toc156745947)

[2.2 Trivy 7](#_Toc156745948)

[2.3 Semgrep 7](#_Toc156745949)

[2.4 Secret detection 8](#_Toc156745950)

[3 Этап 3. DAST 9](#_Toc156745951)

[3.1 OWASP Zed Attack Proxy (ZAP) 9](#_Toc156745952)

[3.2 Альтернативные варианты работы ZAP 10](#_Toc156745953)

[4 Пример триггера на остановку релиза при наличии уязвимостей 12](#_Toc156745954)

[5 Приложение 12](#_Toc156745955)

# 

# Введение

Взят проект сайта https://github.com/ixartz/Eleventy-Starter-Boilerplate и сделан форк https://github.com/vit81g/diplom01git для дальнейшей работы.

Необходимо провести анализ кода на уязвимости. Настроить pipline для проекта и встроить в него статические (SAST) и динамические (DAST) тесты. Описать процессы тестирования, провести аналитику результатов тестов.

# 

# Этап 1. CI/CD

В проекте задействованы общедоступные сервисы:

* GitHub – хранилище репозитория
* GitLab – построение процессов сбори, тестирования и доставки (CI/CD )
* <https://diplomvit81.ru/> – готовый сайт на основе Github репозитория.
* Snyk – проверка open-source кода и поиск уязвимостей
* Semgrep – инструмент статического анализа с открытым исходным кодом
* Trivy – сканер уязвимостей в контейнерах, Git репозиториях, поиск секретов
* ZAP (Zed Attack Proxy) - OWASP open-source решение, которое в том числе используется в GitLab DAST

## Настройка пайплайн по сборке и доставке программного обеспечения

На данном этапе мы используем сервис Gitlab.com. Используем пробную версию.

Для работы нам необходимо импортировать наш готовый проект, расположенный на Github (https://github.com/vit81g/diplom01git). В меню Code -> Repository мы увидим клонированный проект.

Следующим этапом идет настройка runners (раннеров или бегунков) для выполнения основных работ в меню Settings -> CI/CD -> Runners. Runner можно настроить на локальной машине, виртуальной или в облаке. В проекте будет использованы ресурсы Yandex.Cloud. Описание настроек будет приведено в пункте \_\_\_\_\_\_\_\_\_

После подключения runners необходимо подготовить gitlab.yaml файл, в котором будут описаны все этапы работ, включая этапы тестирования. Пример gitlab.yaml файла будет приведен в приложении

## Настройка облачных сервисов

На данном этапе мы используем сервис Yandex.Cloud. Для проекта будет использоваться грант предоставляемый компанией Яндекс для начального использования облачных сервисов.

Создание проекта. Выбор тестовый период с предоставленным грантом.

Переходим в Yandex Cloud Console. Создаем ресурс -> Виртуальная машина (ВМ).

**Параметры ВМ:**

Операционная система - Ubuntu 22.04

Диски и файловые хранилища – HDD, 18ГБ

Вычислительные ресурсы – Intel IceLake, vCPU -2, 20%, RAM -2ГБ

Сетевые настройки – по умолчанию

Доступ – ввести логин и SSH-ключ (генерация ssh ключа – ssh-keygen –t ed25519; посмотреть ключ – cat ~/.ssh/id\_ed25519.pub )

Запущен процесс создания новой ВМ. После создания ВМ проведем дальнейшие настройки и установим необходимое программное обеспечение. Для этого произведем подключение к ВМ по протоколу ssh:

ssh@<публичный IP адрес>

*(посмотреть публичный IP можно в настройках ВМ в Yandex Cloud Console)*

**Установка Docker:**

* Обновите существующий список пакетов:

sudo apt-get update

* Установите необходимые пакеты, которые позволяют apt использовать пакеты по HTTPS:

sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

* Добавьте в систему ключ GPG официального репозитория Docker:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

* Добавьте репозиторий Docker в источники APT:

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

* Обновите список пакетов и установите Docker:

sudo apt-get update

sudo apt-get install docker-ce

* Проверьте статус Docker:

sudo systemctl status docker

## 1.3 Настройка gitlab runners

Подключаемся к ВМ с установленным gitlab runner:

ssh@<IP адрес ВМ>

**Установка gitlab runner:**

* Добавьте репозиторий GitLab Runner:

curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash

* Установите GitLab Runner:

sudo apt-get install gitlab-runner

* Запустите команду регистрации GitLab Runner:

sudo gitlab-runner register

Вам потребуется URL вашего GitLab сервера, токен регистрации (который вы можете найти в настройках вашего проекта на GitLab), описание для Runner, теги (если они нужны) и тип исполнителя (в вашем случае, это будет docker).

После регистрации GitLab Runner запустится автоматически. Вы можете проверить его статус с помощью следующей команды:

sudo gitlab-runner status

Проверить доступность runner можно на Gitlab в Settings -> CI/CD -> Runners. Активные и доступные runners будут подсвечены зеленым, неактивные или недоступные подсвечены серым.

# Этап 2. SAST

Static Application Security Testing (SAST) — проверка, при которой анализаторы не просто проверяют синтаксическую корректность, но и измеряют качество кода с помощью специальных метрик. Основная задача SAST-сканеров — тестирование на безопасность. В частности, SAST-анализаторы проверяют исходный код на наличие распространенных уязвимостей, например, некоторых из списка OWASP Top Ten. Важно сказать, что SAST-сканеры находят не только саму уязвимость, но и фрагмент кода, из-за которого она появилась.

SAST-анализ также называют проверкой методом «белого ящика» (White Box Testing), так как анализатор имеет доступ к внутренней структуре приложения. Важно отметить, что анализаторы проверяют исходный код, не запуская его, поэтому могут генерировать ложные срабатывания и не обнаружить некоторые типы уязвимостей. По этой причине не стоит ограничиваться только SAST-анализом. Лучше подойти к вопросу комплексно и использовать различные типы анализа: SCA, DAST, IAST, OAST.

Инструменты SAST

* Бесплатное решение:

GitLab SAST, Semgrep, Trivy

* Проприетарные решения:

SonarQube, Snyk, Checkmarx SAST, Solar appScreener

* Российские решения:

PT Application Inspector, PVS-Studio

## Snyk

Snyk - инструмент сканирования уязвимостей с открытым исходным кодом для постоянного поиска и исправления уязвимостей в зависимости, полученных из Rubygems, NPM, PYPI, Maven и других.

Для включения Snyk в pipeline в этап SAST нам необходимо перейти в pipeline editor для редактирования gitlab-cu.yml (сайт с проектом на Gitlab открываем Build -> Pipeline editor). Пример кода:

# Список этапов в pipeline

stages:

- test

# Название job или теста. Тестирование кода с помощью Snyk и выводом результатов отдельный файл

snyk:

stage: test

image: snyk/snyk:latest

script:

- snyk test --all-projects --json > snyk-report.json

- snyk-to-html -i snyk-report.json -o snyk-report.html

artifacts:

paths:

- snyk-report.html

allow\_failure: true

## Trivy

Trivy – инструмент для сканирования безопасности. Он позволяет определить потенциальные угрозы в программном коде, конфигурации, установленных пакетах операционной системы, контейнерах, выявить секреты.

Для включения Trivy в pipeline в этап SAST нам необходимо перейти в pipeline editor для редактирования gitlab-cu.yml (сайт с проектом на Gitlab открываем Build -> Pipeline editor). Пример кода:

# Список этапов в pipeline

stages:

- analyze

# Название job или теста. Анализ кода с помощью Trivy и выводом результатов отдельный файл

trivy\_scan:

stage: analyze

image: aquasec/trivy:latest

script:

- trivy image --severity HIGH,CRITICAL --format template --template "@contrib/html.tpl" -o trivy-report.html $CI\_REGISTRY\_IMAGE:$CI\_COMMIT\_REF\_SLUG

artifacts:

paths:

- trivy-report.html

allow\_failure: true

## Semgrep

Semgrep – бесплатный инструмент для статического анализа и поиска ошибок в коде.

Для включения Semgrep в pipeline в этап SAST нам необходимо перейти в pipeline editor для редактирования gitlab-cu.yml (сайт с проектом на Gitlab открываем Build -> Pipeline editor). Пример кода:

# Список этапов в pipeline

stages:

- analyze

# Название job или теста. Анализ кода с помощью Semgrep и выводом результатов отдельный файл

semgrep:

stage: analyze

image: returntocorp/semgrep:latest

script:

- semgrep --config p/security-audit . --json > semgrep-report.json

- json2html < semgrep-report.json > semgrep-report.html

artifacts:

paths:

- semgrep-report.html

allow\_failure: true

## Secret detection

Можно использовать встроенный инструмент Gitlab для статического анализа кода на предмет наличия в нем конфиденциальных данных, таких как пароли или ключи API, которые были включены в исходный код в явном виде без использования переменных или конфигурационных файлов.

Подключаем встроенный - template: Security/Secret-Detection.gitlab-ci.yml и настраиваем pipeline в этап SAST. Необходимо перейти в pipeline editor для редактирования gitlab-cu.yml (сайт с проектом на Gitlab открываем Build -> Pipeline editor). Пример кода:

stages:

- analyze

secret\_detection:

stage: analyze

include:

- template: Security/Secret-Detection.gitlab-ci.yml

variables:

SECRET\_DETECTION\_HISTORIC\_SCAN: "true"

В проекте будет использоваться инструмент статического анализа кода на Python 3.9. Пример кода:

secret\_detection:

stage: analyze

image: python:3.9

script:

- pip install detect-secrets

- detect-secrets scan > secrets.baseline

artifacts:

paths:

- secrets.baseline

allow\_failure: true

# Этап 3. DAST

DAST (Dynamic Application Security Testing) — динамическое тестирование безопасности приложений. DAST-сканеры работают автоматически и проверяют приложения, имитируя внешние атаки через различные уязвимости. Получается, что приложение — чёрный ящик для анализатора, о нём ничего не известно.

Для DAST-проверок необходимо иметь доступный для сканера запущенный экземпляр приложения. Причём, чем ближе параметры тестового экземпляра приложения к production-инсталляции, тем меньше будет ложных срабатываний. Например, если вы развернули тестовый экземпляр приложения, доступный только по протоколу http, а в production приложение доступно только по протоколу https, DAST-сканер выдаст ряд ошибок, связанных с отсутствием шифрования трафика между клиентом (анализатором) и сервером (экземпляром приложения).

Примеры DAST-инструментов: GitLab DAST (доступно только в Ultimate-версии), OWASP Zed Attack Proxy (опенсорсное решение, которое в том числе используется в GitLab DAST), Acunetix, Fortify WebInspect, HCL Security AppScan, Synopsys Managed DAST, Tenable. io (Web App Scanning), Veracode Dynamic Analysis.

## OWASP Zed Attack Proxy (ZAP)

OWASP ZAP – сканер веб-приложений, основанный на методике DAST (метод тестирования «черного ящика»). Позволяет обнаружить проблемы безопасности в работающем приложении или веб-сайте при помощи их сканирования на известные уязвимости (SQL-инъекции, XSS - межсайтовый скриптинг, Clickjacking и другие).

ZAP поддерживается проектом OWASP (Open Web Application Security Project) – некоммерческой организацией, которая специализируется на создании статей, материалов, инструментов и технологий, позволяющих разрабатывать приложения безопаснее и обеспечивать должный уровень безопасности.

В проекте мы будем использовать встроенный инструмент Gitlab для динамического тестирования DAST/Dependency-Scanning и дополнительно запущенный ZAP на локальной машине.

Настройка сканирования в Gitlab – перейти в Security->Security configuration -> Enable DAST. В открывшемся окне DAST CI/CD configuration необходимо настроить профили.

**Scanner profile:**

* Profile name – DAST01
* Scan mode – active
* Target timeout - 60 seconds
* AJAX spyder – on
* Debug message – Hide debug messages

**Site profile:**

* Profile name – DAST02
* Target URL - https://diplomvit81.ru/
* Site type – Website
* Обязательно пройти валидацию о том, что это наш сайт и мы можем проводить тестирование.

Подключаем встроенный - template: template: DAST.gitlab-ci.yml и настраиваем pipeline в этап DAST. Необходимо перейти в pipeline editor для редактирования gitlab-cu.yml (сайт с проектом на Gitlab открываем Build -> Pipeline editor). Пример кода:

stages:

- dast

# Include the DAST template

include:

- template: DAST.gitlab-ci.yml

# Ваши профили сканирования:

dast:

stage: dast

dast\_configuration:

site\_profile: "DAST02"

scanner\_profile: "DAST01"

## Альтернативные варианты работы ZAP

Дополнительно проведем сканирование на уязвимости с помощью запущенного docker с ZAP с локальной машины.

Перед началом тестирования необходимо установить docker, как описано в пункте 1.2 (установка Docker).

Запуск тестирования осуществляется с помощью команды:

sudo docker run -v $(pwd):/zap/wrk/:rw -t ghcr.io/zaproxy/zaproxy:stable zap-full-scan.py -t https://<ваш сайт>/ -g gen.conf -r testreport.html

*В нашем случае https://<ваш сайт>/ это* [*https://diplomvit81.ru/*](https://diplomvit81.ru/)

testreport.html – итоговый файл протестированного сайта

OWASP ZAP можно включить в pipeline на Gitlab, если у вас истек пробный период использования. Пример кода:

stages:

- dast

# Название job или теста. Тестирование с помощью OWASP ZAP и выводом результатов отдельный файл

zap:

stage: DAST

image: owasp/zap2docker-stable:latest

script:

- zap-full-scan.py -t https://diplomvit81.ru/ -m 5 -z "-config spider.maxDepth=1 -config spider.postForm=true" -r zap-report.html

artifacts:

paths:

- zap-report.html

allow\_failure: true

# Пример триггера на остановку релиза при наличии уязвимостей

Пример кода:

trivy\_scan:

stage: analyze

image: aquasec/trivy:latest

script:

- trivy image --severity HIGH,CRITICAL --format json $CI\_REGISTRY\_IMAGE:$CI\_COMMIT\_REF\_SLUG > trivy\_report.json

- HIGH\_COUNT=$(jq '.[].Vulnerabilities[] | select(.Severity == "HIGH") | .VulnerabilityID' trivy\_report.json | wc -l)

- CRITICAL\_COUNT=$(jq '.[].Vulnerabilities[] | select(.Severity == "CRITICAL") | .VulnerabilityID' trivy\_report.json | wc -l)

- if [ $HIGH\_COUNT -gt 0 ] || [ $CRITICAL\_COUNT -gt 0 ]; then exit 1; fi

allow\_failure: true

В этом примере мы используем утилиту jq для анализа JSON-отчета, созданного Trivy. Если количество уязвимостей "HIGH" или "CRITICAL" больше 0, сценарий завершается с ошибкой, что приводит к остановке пайплайна. Обратите внимание, что у вас должен быть установлен jq для работы этого сценария.

# Приложение

**Основные команды для настройки виртуальной машины**

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

sudo apt-get update

sudo apt-get install docker-ce

sudo systemctl status docker

curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash

sudo apt-get install gitlab-runner

sudo gitlab-runner register

**Код gitlab.yml файла в pipeline editor**

stages:

- build

- test

- analyze

- deploy

- dast

build:

stage: build

script:

- echo "Building the application..."

snyk:

stage: test

image: snyk/snyk:latest

script:

- snyk test --all-projects --json > snyk-report.json

- snyk-to-html -i snyk-report.json -o snyk-report.html

artifacts:

paths:

- snyk-report.html

allow\_failure: true

trivy\_scan:

stage: analyze

image: aquasec/trivy:latest

script:

- trivy image --severity HIGH,CRITICAL --format template --template "@contrib/html.tpl" -o trivy-report.html $CI\_REGISTRY\_IMAGE:$CI\_COMMIT\_REF\_SLUG

artifacts:

paths:

- trivy-report.html

allow\_failure: true

secret\_detection:

stage: analyze

image: python:3.9

script:

- pip install detect-secrets

- detect-secrets scan > secrets.baseline

artifacts:

paths:

- secrets.baseline

allow\_failure: true

semgrep:

stage: analyze

image: returntocorp/semgrep:latest

script:

- semgrep --config p/security-audit . --json > semgrep-report.json

- json2html < semgrep-report.json > semgrep-report.html

artifacts:

paths:

- semgrep-report.html

allow\_failure: true

zap:

stage: dast

image: owasp/zap2docker-stable:latest

script:

- zap-baseline.py -t $CI\_PROJECT\_URL -m 5 -z "-config spider.maxDepth=1 -config spider.postForm=true" -r zap-report.html

artifacts:

paths:

- zap-report.html

allow\_failure: true

deploy:

stage: deploy

image: mwienk/docker-lftp:latest

script:

- lftp -c "set ftp:ssl-allow no; open -u u2408844, 71BGlky8b8HipzAm; mirror -Rev . 31.31.196.161 --ignore-time --parallel=10 --exclude-glob .git\* --exclude .git/"

only:

- master

**Проверки работоспособности docker на виртуальной машине с runner**

Для проверки необходимо вставить следующий код в gitlab.yml в самом начале (он проверит запуск, статус службы и версию docker:

default:

tags:

- gitlab-runner

stages:

- build

image:

name: my-custom-image

entrypoint: [""]

variables:

# your variables

before\_script:

- docker --version || echo 'Docker CLI not found'

- which dockerd || echo 'Docker daemon not found'

- service docker status || echo 'Docker service status check failed'

- ls -lah /var/run/docker.sock || echo 'Cannot access Docker socket'

- dockerd --version || echo 'Docker daemon version command failed'

- dockerd &

- sleep 10 # Give time for the Docker daemon to start

build:

stage: build

only:

- tags

script:

# - other scripts

- docker build -t ${DOCKER\_IMAGE\_PATH}:${DOCKER\_TAG} .

- docker push ${DOCKER\_IMAGE\_PATH}:${DOCKER\_TAG}

В ходе написания дипломной работы возникли проблемы с настройкой docker на виртуальной машине. Runner не мог найти и запустить службу docker. В следствии чего, пришлось проводить тесты DAST встроенными средствами gitlab и дополнительно запускать ZAP на локальной машине.

Команда запуска ZAP для проверки web сайта на уязвимости: