Lab

Program 1: Setting Up a Virtual Machine and Executing C Programs on Ubuntu using VMware Workstation

Objective:

To install and configure a virtual machine using VMware Workstation on a Windows 11 system, set up Ubuntu 18.04, install a C compiler (GCC), and compile and execute a simple C program.

Requirements:

- Windows 11 Operating System
- VMware Workstation software
- Ubuntu 18.04 ISO file
- Basic knowledge of Linux terminal commands

Procedure:

Step 1: Install VMware Workstation

Install VMware Workstation on your Windows 11 system to create and manage virtual machines.

Step 2: Create a New Virtual Machine

- Open VMware Workstation → Click **“Create a New Virtual Machine”**.

Step 3: Select Configuration Type

- Select **“Typical (recommended)”** → Click **Next**.

Step 4: Select Installer Disk Image (ISO)

- Browse and select the **Ubuntu 18.04 ISO file** → Click **Next**.

Step 5: Enter Personal Details

Provide user details for the VM:

```
Full Name: (Any name)
Username: (VM username)
Password: (Your password)
Confirm Password: (Retype password)
```

Click **Next**.

Step 6: Set Virtual Machine Name and Location

- Give a name to your virtual machine.
- Keep the default storage location.
- Click **Next**.

Step 7: Configure Hardware Settings

Assign system resources:

```
Processor: 2 CPUs
RAM: 2 GB
Hard Disk: 20 GB
```

Click **Next**.

Step 8: Finish Setup

- Review your configuration and click **Finish** to create the virtual machine.

Installing GCC and Running a C Program in Ubuntu

Step 9: Open Terminal in Ubuntu VM

The terminal is used to install software and run commands in Linux.

The GCC (GNU Compiler Collection) is usually pre-installed on Ubuntu, but if it's not, you can install it with the following command:

Step 10: Update Package Lists

```
sudo apt update
```

- This updates the system's list of available packages.

Step 11: Install GCC Compiler

```
sudo apt install gcc
```

- Installs the C compiler (GCC) and other required development tools.

Step 12: Create a C Program File

```
nano my_program.c
```

- Opens the nano text editor to write the C program.

Step 13: Write a Simple C Program

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

- Save the file: Press `Ctrl + O`, then `Enter`.
- Exit nano: Press `Ctrl + X`.

Step 14: Compile the C Program

```
gcc my_program.c -o my_program
```

- `gcc`: Compiler name.

- `my_program.c`: C source file.
- `o my_program`: Output executable file name.
- If there are no errors in your code, this will create an executable file named my_program.

Step 15: Run the Program

```
./my_program
```

- This will display the output:

```
Hello, World!
```

Conclusion:

Successfully installed a virtual machine, configured Ubuntu, installed the GCC compiler, wrote a C program, compiled it, and executed it to display the output.

Program 2: Find a procedure to transfer the files from One Virtual Machine to another Virtual Machine.

Objective:

To transfer files securely from one virtual machine (VM1) to another virtual machine (VM2) using the SCP and Rsync commands over a network.

To understand inter-VM communication by transferring files from one virtual machine to another using common file transfer methods and network configuration.

Requirements:

- Two Virtual Machines with Ubuntu installed
- VMware Workstation
- Networking tools (net-tools package)

- OpenSSH Server installed on VM1
- IP addresses of both virtual machines
- Basic Linux terminal knowledge

Procedure:

Step 1: Create Virtual Machine 1 (VM1)

- Configure VM1 with the following details:

```
Full Name: Global10
User Name: gat1
```

(VM1 will act as the source machine.)

Step 2: Create Virtual Machine 2 (VM2)

- Configure VM2 with the following details:

```
Full Name: Global2
User Name: gat2
```

(VM2 will act as the destination machine.)

Step 3: Install Networking Tools on Both VMs

```
sudo apt install net-tools
```

- This installs `ifconfig` to check IP addresses.
- This command installs essential networking utilities like `ifconfig, netstat` and `route`.

Step 4: Find the IP Address of VM1

```
ifconfig
```

- Note the IP address of VM1.

  Example:

```
inet 192.168.90.128
```

- repeat the IP address step for VM2

Step 5: Install OpenSSH Server on VM2

```
sudo apt install openssh-server -y
```

- This enables secure file transfers via SSH.

Step 6: Ensure VM2 is Properly Set Up

- Both VMs must be running and connected to the same virtual network.

Step 7: Create a File on VM1

```
touch p1.txt
ls
```

```
nano p1.txt
```

- Add some text → Save with `Ctrl + O` → Exit with `Ctrl + X`.

File Transfer Using SCP

Step 8: Understand the SCP Command

- **SCP (Secure Copy):** Used to securely transfer files between machines using SSH [Secure Shell]
- The scp command in Linux is a command-line utility that uses the Secure Shell (SSH) protocol to securely copy files or directories between a local and a remote system, or between two remote systems.

Step 9: Transfer File from VM1 to VM2

```
scp p1.txt gat2@<VM2_IP_ADDRESS>:/home/gat2/
```

- Example:

```
scp p1.txt gat2@192.168.90.129:/home/gat2/
```

- Enter the VM2 user's password when prompted.

Step 10: Confirm File Transfer

- On VM2, check:

```
ls /home/gat2/
```

- You should see the file `p1.txt`.

Alternative: File Transfer Using Rsync

Step 11: Transfer File Using Rsync/remote synchronization

```
rsync -avz p1.txt gat2@192.168.90.129:/home/gat2/
```

- **Rsync:** Efficiently syncs files between machines and **minimizes** data transfer.

Conclusion:

Successfully transferred files from one virtual machine to another using the `scp` command and also learned how to perform efficient file transfers using `rsync`.

Program 3: Deploy a Flask Web Application to Google Cloud using Google Cloud CLI (PaaS)

Objective:

To deploy a Flask web application to Google Cloud Platform (GCP) using Google Cloud CLI as a Platform as a Service (PaaS) solution.

Requirements:

- Google Cloud CLI installed
- Google Cloud account
- Python 3.9 or 3.10 installed
- Flask installed
- Basic files: `main.py`, `app.yaml`, `requirements.txt`
- Internet connection

Procedure:

Step 1: Download and Install Google Cloud CLI

- URL: https://cloud.google.com/sdk/docs/install

Installation Steps:

1. Click **Next** → Click **I Agree** → Click **Next** → **Select User and Click Next**
2. Select default installation folder → Click **Next**.
3. Check all options → Click **Install**.
4. After installation → Click **Finish**.

Sign-In to Google Cloud:

```
gcloud init
```

- Press `Y` to proceed with authentication.
- Sign in with your Gmail account.
- After successful login, select the project from the list or create a new one.

Step 2: Create or Select Google Cloud Project

If no project exists, create a new one:

- Visit: https://console.cloud.google.com/terms
- Click **Select Project** → Click **New Project** → Enter project name [eg. Gloabl123] → Click **Create**.

Re-initialize Google Cloud CLI:

```
gcloud init
```

- Select option `1` to reinitialize.
- Choose the authenticated account.
- Select the project ID.

Step 3: Install Python and App Engine Component

3.1 Check Python Version:

```
python --version
```

3.2 Install App Engine Component:

```
gcloud components install app-engine-python
```

- The following components will be installed
  - Cloud Datastore Emulator
  - gRPC Python Library
  - gCloud App Python Extensions
- Press `Y` to install required components.
- Press any key to exit

Step 4: Install Flask

```
pip install flask
```

Verify Flask Installation:

```
flask --version
```

Step 5: Prepare Flask Application Files

5.1 Create Project Folder:

- Go to **C Drive** → Create a folder named:

```
flask-app
```

- Change directory:

```
cd C:\flask-app
```

5.2 Create `app.yaml` File:

To determine the Python version you're using, open a terminal or command prompt and type `python --version` or `python -V`. This command will display the installed Python version.
For example, you might see "Python 3.9.0"

```
runtime: python39
entrypoint: main.py

handlers:
- url : /.*
   script: auto
```

(Specifies Python version.)

5.3 Create `main.py` File:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
   return 'Hello, World! This is my Flask app on Google Cloud.'

if __name__ == '__main__':
   app.run(host='0.0.0.0', port=8080)
```

5.4 Create `requirements.txt` File:

```
flask
```

Step 6: Run Flask App Locally

Run the app using:

```
python "C:\Users\Admin\AppData\Local\Google\Cloud SDK\google-cloud-sdk\bin\dev_appserver.py" app.yaml
```

Open browser and type:

```
http://192.168.2.65:8080
```

(Check if the app is running locally.)

Step 7: Deploy Flask App to Google Cloud

7.1 Deploy the App:

```
gcloud app deploy
```

- Select the region when prompted.

7.2 View the Live App:

```
gcloud app browse
```

(Opens the deployed app in the browser.)

> Note: Billing must be enabled to complete the deployment.

Conclusion:

Successfully deployed a Flask web application to Google Cloud using Google Cloud CLI as PaaS. The application can now be accessed via the provided public URL.

Program 4: Build a Docker Image for a Flask Web Application and Push it to Docker Hub

Objective:

To create a Docker image for a simple Flask web application, run it in a Docker container, and push the image to a Docker Hub repository.

Requirements:

- Docker Desktop installed
- Docker Hub account
- Windows PowerShell with WSL installed
- Python installed
- Flask library
- Internet connection

Procedure:

Part 1: Docker Installation and Setup

Step 1: Download Docker Desktop

- URL: https://www.docker.com/products/docker-desktop

Step 2: Install Docker Desktop

- On successful installation → Click **OK**.

Step 3: Install WSL (Windows Subsystem for Linux)

```
wsl --install
```

- Enables Linux-based Docker containers on Windows.

Step 4: Confirm Docker Installation

- Docker will now be visible on your desktop.

Step 5: Start Docker

- Click the Docker icon → Accept license agreement.

Step 6: Skip Optional Setup

- Click **Skip** on the welcome/setup screen.

Step 7: Verify Docker is Running

- Check Docker in **Show Hidden Icons** on the taskbar.

Part 2: Docker Hub Setup

Step 8: Open Docker Dashboard

Step 9: Visit Docker Hub

- URL: https://hub.docker.com/

Step 10: Create a Docker Hub Account

- Sign up using email or Google account.

Step 11: Sign In on Docker Desktop

- Use Docker Hub credentials to sign in.

Step 12: Open Docker Dashboard

- After login, click **Open Docker Desktop**.

Part 3: Flask Application Setup

Step 13: Create Flask App Directory

```
C:\flask-app
```

Step 14: Create Python Flask File (`app.py`)

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Docker Flask App!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Step 15: Create `requirements.txt` File

```
flask
```

Step 16: Create `Dockerfile` in `flask-app` Directory

```
# Use official Python image
FROM python:3.9

# Set working directory
WORKDIR /app

# Copy all files to the container
COPY . /app

# Install dependencies
RUN pip install -r requirements.txt

# Expose the port
EXPOSE 5000

# Command to run the app
CMD ["python", "app.py"]
```

Part 4: Build and Run Docker Image

Step 17: Build Docker Image

```
cd flask-app
docker build -t flask-app .
```

(The dot '.' indicates the current directory.)

Step 18: List Docker Images

```
docker images
```

(Verify that `flask-app` image is created and visible in the dashboard)

Step 19: Run Docker Container

```
docker run -p 5000:5000 -d flask-app
```

(Maps container port 5000 to localhost port 5000.)

```
docker ps-a
```

- To check image is running in container

Step 20: Verify Flask App

- Open browser:

```
http://localhost:5000
```

(You should see: "Hello, Docker Flask App!")

Part 5: Push Docker Image to Docker Hub

Step 21a: Tag Docker Image

```
docker tag flask-app your-dockerhub-username/flask-app:latest
```

Step 21b: Push Image to Docker Hub

```
docker push your-dockerhub-username/flask-app:latest
```

(Replace `your-dockerhub-username` *with your actual Docker Hub username.)*

Step 21c: Verify Image on Docker Hub

- Visit your Docker Hub account and check if the image is uploaded.

Pull Docker Image from Docker Hub

Step 22: Pull Image from Docker Hub

```
docker pull your-dockerhub-username/flask-app:latest
```

(This downloads the image to any system with Docker installed.)

Conclusion:

Successfully built a Docker image for a Flask web application, ran it in a container, and pushed it to Docker Hub for sharing and deployment.

✅ Program 5: Pull a Docker Image from Docker Hub and Deploy your Flask App using Docker Swarm

---

🎯 Objective:

Deploy a Flask web application by pulling a Docker image from Docker Hub and managing it using Docker Swarm.

🛠️ Requirements:

- Docker installed and running
- Docker Hub account (optional for custom images)
- Basic knowledge of Docker commands

- Docker Swarm initialized

Requirements:
• Docker installed on your system
• Docker Hub account (with a Flask app image already pushed)
• Internet connection
• Basic Flask Docker image available on Docker Hub (from Program 4)
Pre-requisites:
Ensure your Flask app has already been containerized and pushed to Docker Hub
(E.g., yourusername/flask-docker-app:latest)

🚀 Steps:

🔹 Step 1: Install and Start Docker

Check Docker installation:

```
docker --version
```

Start Docker Desktop if not already running.

🔹 Step 2: Initialize Docker Swarm

```
docker swarm init
```

This sets up your machine as a Swarm manager.

🔹 Step 3: Create a Simple Flask App

📂 Project Structure:

```
project/
│
├── app.py
├── requirements.txt
└── Dockerfile
```

📜 Flask App: `app.py`

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return "Hello from Flask app deployed with Docker Swarm!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

📦 Requirements: `requirements.txt`

```
flask
```

🐳 Dockerfile

```
FROM python:3.9

# Set working directory
WORKDIR /app

# Copy all files to the container
COPY . /app

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

EXPOSE 5000

CMD ["python", "app.py"]
```

🔹 Step 4: Build Docker Image

```
docker build -t my-flask-app .
```

🔹 Step 5: Push Docker Image to Docker Hub

Tag the image:

```
docker tag my-flask-app <your-dockerhub-username>/my-flask-app
```

Push to Docker Hub:

```
docker push <your-dockerhub-username>/my-flask-app
```

Pull from Docker Hub:

```
docker pull <your-dockerhub-username>/my-flask-app
```

🔹 Step 6: Deploy Flask App Using Docker Swarm

```
docker service create
--name flask-app
--publish 5000:5000
<your-dockerhub-username>/my-flask-app
```

- Creates a new service named flask-app
- Publishes it on port 5000 of your local machine
- Runs the image as a container in Swarm mode

🔹 Step 7: Verify Service

Check the service status:

```
docker service ls
```

Check running containers:

```
docker ps
```

🔹 Step 8: Access Flask App

Visit:

```
http://localhost:5000
```

You should see:

```
Hello from Flask app deployed with Docker Swarm!
```

🔹 Step 9: Scale the Service (Optional)

```
docker service scale flask-app=3
```

Check updated replicas:

```
docker service ps flask-app
```

🔹 Step 10: Remove the Service

```
docker service rm flask-app
```

🔹 Step 11: Leave Docker Swarm

```
docker swarm leave --force
```

✅ Conclusion:

You successfully pulled a Docker image of a Flask app from Docker Hub and deployed it using Docker Swarm, allowing easy scaling and container management.

Program 6: Vulnerability, Misconfiguration, and Secret Scanning using Trivy (with HTML

Report Generation)

Objective:

To scan files, Docker images, Docker Hub images, and GitHub repositories for **secrets, misconfigurations, and vulnerabilities** using **Trivy**, and to generate detailed summary reports in **HTML format**.

Requirements:

- Windows 10/11 Operating System
- Docker Desktop (for Docker image scanning)
- Internet connectivity
- Trivy installed via Scoop
- Access to GitHub (for remote repository scanning)

Procedure:

🔹 Step 1: Install Trivy on Windows using Scoop

Open **PowerShell as Administrator** and run:

```
Set-ExecutionPolicy RemoteSigned -scope CurrentUser
iwr -useb get.scoop.sh | iex
```

(Scoop is a Windows package manager for CLI tools.)

🔹 Step 2: Install Trivy

```
scoop install trivy
```

🔹 Step 3: Verify Installation

```
trivy --version
```

💻 Scenario 1: Secrets Scanning Example

Vulnerable Python Code (app.py)

```
from flask import Flask
app = Flask(__name__)

# API Key stored directly in code (SECRET)
API_KEY = "sk_test_ABC123SECRETKEY"

@app.route('/')
def home():
    return f"Hello, World!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

(Direct secret key storage is a major security risk.)

💻 Scenario 2: Misconfiguration Example

Insecure Dockerfile

```
FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD ["python", "app.py"]
```

> 🚩 Issues:

- Runs as root user
- Missing `HEALTHCHECK` instruction

Secure Dockerfile

```
FROM python:3.9
RUN useradd -m appuser
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt

HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3
\
 CMD curl --fail http://localhost:5000/health || exit 1

USER appuser
EXPOSE 5000
CMD ["python", "app.py"]
```

> ✅ Best Practices:

- Run as non-root user
- Add health check for the container

💻 Scenario 3: Vulnerability Example

Vulnerable Requirements File

```
flask==2.3.0
```

> 🚩 Version 2.3.0 contains known vulnerabilities (Example: CVE-2023-30861).

🔍 Running Trivy Scans (File System)

📁 Preparation:

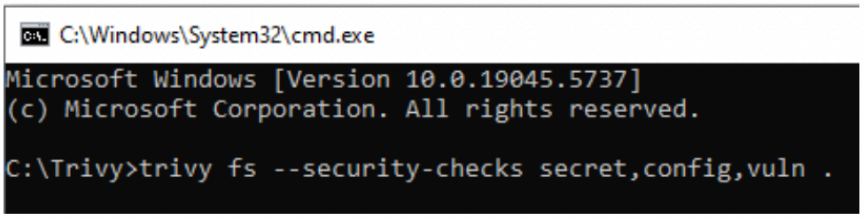
- Create a folder `C:\Trivy`
- Place `app.py`, `Dockerfile`, and `requirements.txt` inside it.

Full Scan for Secrets, Configurations, and Vulnerabilities

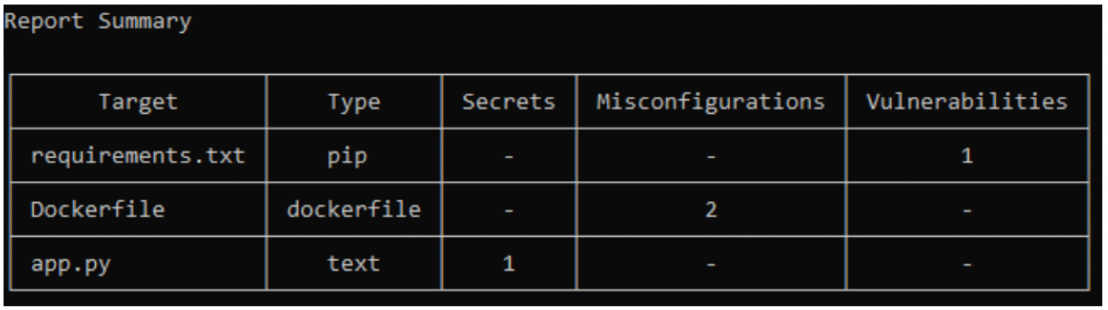
```
trivy fs --security-checks secret,config,vuln .
```

EX:

```
trivy fs --security-checks secret,config,vuln .
```



Report Summary in CLI Mode



Report Summary

Target	Type	Secrets	Misconfigurations	Vulnerabilities
requirements.txt	pip	-	-	1
Dockerfile	dockerfile	-	2	-
app.py	text	1	-	-

Individual Scans

- **Secrets Only:**

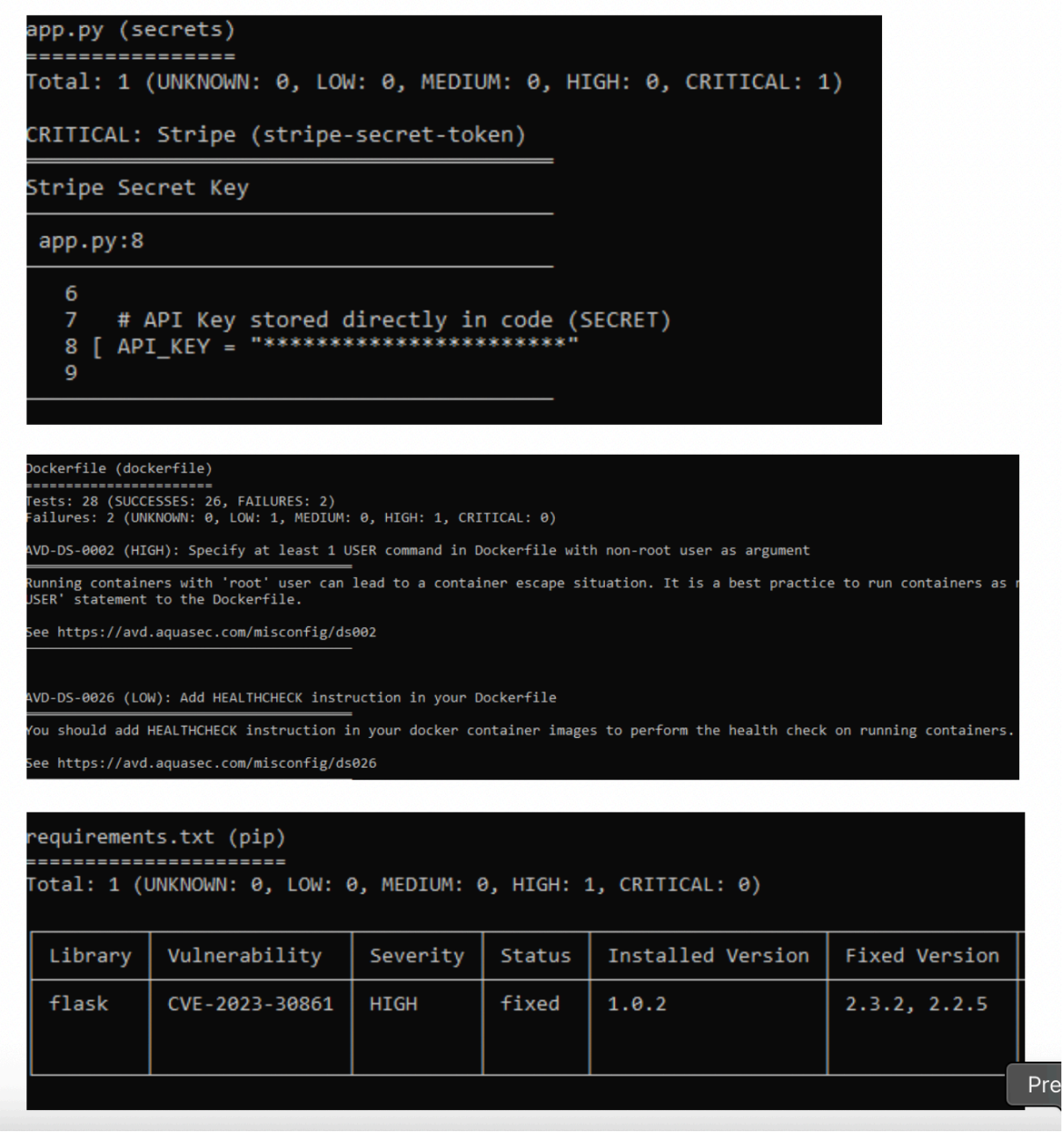
```
trivy fs --security-checks secret .
```

- **Misconfigurations Only:**

```
trivy fs --security-checks config .
```

- **Vulnerabilities Only:**

```
trivy fs --security-checks vuln .
```



📄 HTML Report Generation

Step 1: Download HTML Template

- URL: html.tpl Template
- Save as `html.tpl` inside `C:\Trivy` folder.

Step 2: Run Full Scan with HTML Output

```
trivy fs --format template --template "@html.tpl" -o report.html --security-checks secret,config,vuln .
```

Step 3: Generate Individual HTML Reports

- Secrets:

```
trivy fs --format template --template "@html.tpl" -o secret-report.html --security-checks secret .
```

- Misconfigurations:

```
trivy fs --format template --template "@html.tpl" -o misscon-report.html --security-checks config .
```

- Vulnerabilities:

```
trivy fs --format template --template "@html.tpl" -o vuln-report.html --security-checks vuln .
```

🐳 Scanning Docker Images

Full Docker Image Scan

```
trivy image --scanners vuln,secret,misconfig trivy:latest
```

Individual Docker Image Scans

- Secrets:

```
trivy image --scanners secret trivy:latest
```

- Misconfigurations:

```
trivy image --scanners misconfig trivy:latest
```

- Vulnerabilities:

```
trivy image --scanners vuln trivy:latest
```

HTML Report for Docker Image

```
trivy image --format template --template "@html.tpl" -o docfullscan.html --scanners vuln,secret,misconfig trivy:latest
```

🐳 Scanning Docker Hub Images

Example: `shwethapmurthy/trivy:latest`

Full Docker Hub Image Scan

```
trivy image --scanners vuln,secret,misconfig shwethapmurthy/trivy:latest
```

HTML Report for Docker Hub Image

```
trivy image --format template --template "@html.tpl" -o dhfullscan.html --scanners vuln,secret,misconfig shwethapmurthy/trivy:latest
```

🔗 Scanning a Remote GitHub Repository

Full GitHub Repo Scan

```
trivy repo --scanners secret,misconfig,vuln https://github.com/ParashivamurthyC/Trivy-Security
```

HTML Report for GitHub Repo

```
trivy repo --format template --template "@html.tpl" -o git.html --scanners vuln,secret,misconfig https://github.com/ParashivamurthyC/Trivy-Security
```

✅ Conclusion:

- Trivy is an efficient security tool to detect **secrets, misconfigurations, and vulnerabilities** in local files, Docker images, Docker Hub images, and Git repositories.
- Detailed reports can be generated in **HTML format** for easy analysis.

Program 8: Simulating Amazon S3 and EC2 using LocalStack

Objective

Simulate and manage Amazon S3 buckets and EC2 instances locally using LocalStack Cloud Dashboard and AWS CLI.

Requirements

- LocalStack Cloud Account
- LocalStack (Installed via pip)
- Docker (Installed and running)
- AWS CLI (Installed)

Steps

Step 1: Setup LocalStack Cloud Account

- Sign up: https://app.localstack.cloud/sign-up
- Access Dashboard: https://app.localstack.cloud/dashboard
- Check if localstack is offline

Step 2: Install LocalStack

```
pip install localstack-core
```

Step 3: Ensure Docker is Running

Make sure Docker is installed and running. No sign-in is required.

Step 4: Start LocalStack

```
localstack start
```

> Note: Running LocalStack with Docker will pull the LocalStack image if not already downloaded. Do not close the terminal where LocalStack is running.

Step 5: Check LocalStack Cloud Dashboard

- Go to https://app.localstack.cloud/dashboard and refresh the browser.
- Check "Local Instance" status: **Online/Running**

Make sure LocalStack is running with HTTPS support:
https://localhost.localstack.cloud:4566

Simulating S3 Buckets

Using LocalStack Cloud Dashboard

Step 1: Open Dashboard

https://app.localstack.cloud/dashboard

Step 2: Select Local Instance

Step 3: Open Resource Browser

- Select **S3** from services menu.

Step 4: Create S3 Bucket

- Click **Create Bucket**
- Example: `dept-cse`

Step 5: Upload Files to S3 Bucket

- Select the bucket
- Click **Upload File**

Step 6: View Files and Delete Bucket

- You can view files inside the bucket.
- Buckets can be deleted only after all files and folders inside them are deleted.

Using AWS CLI with LocalStack

Prerequisites

```
pip3 install awscli
```

Configure Dummy AWS Credentials

```
aws configure
```

Provide:

- AWS Access Key ID: test
- AWS Secret Access Key: test
- Default region: us-east-1
- Output format: json (or leave blank)

Create a Bucket

```
aws --endpoint-url=http://localhost:4566 s3 mb s3://dept-cse
```

List Buckets

```
aws --endpoint-url=http://localhost:4566 s3 ls
```

Upload File to Bucket

```
echo "Hello LocalStack" > test.txt
aws --endpoint-url=http://localhost:4566 s3 cp test.txt s3://dept-cse
```

Delete Files and Bucket

```
aws s3 rm s3://dept-cse --recursive --endpoint-url=http://localhost:4566
```

Simulating EC2 Instances

Using AWS CLI with LocalStack

Step 1: Create a Key Pair

```
aws --endpoint-url=http://localhost:4566 ec2 create-key-pair --key-name test-key
```

Step 2: Launch EC2 Instance

```
aws --endpoint-url=http://localhost:4566 ec2 run-instances --image-id ami-12345678 --instance-type t2.micro --key-name test-key
```

> Note: Image ID is a dummy ID for simulation in LocalStack.

Step 3: List EC2 Instances

```
aws --endpoint-url=http://localhost:4566 ec2 describe-instances
```

Identify the Instance ID (e.g., i-283153368cc22fb2c)

Step 4: Stop EC2 Instance

```
aws --endpoint-url=http://localhost:4566 ec2 stop-instances --instance-ids i-283153368cc22fb2c
```

Step 5: Terminate EC2 Instance

```
aws --endpoint-url=http://localhost:4566 ec2 terminate-instances --instance-ids i-283153368cc22fb2c
```

Step 6: Delete Key Pair

```
aws --endpoint-url=http://localhost:4566 ec2 delete-key-pair --key-name test-key
```

Using LocalStack Cloud Dashboard

Step 1: Open Dashboard -> Local Instance -> Resource Browser

Step 2: Select EC2 -> Click Launch Instance

- Image ID: ami-12345678
- Instance Type: t2.micro
- Key Name: test-key

Step 3: Manage EC2 Instances

- Available Actions: Start, Stop, Terminate

✅ Program 8: Simulate and Manage Amazon DynamoDB Using AWS CLI with AWS LocalStack Cloud

---

🎯 Objective:

Simulate and manage Amazon DynamoDB using **AWS CLI** and **LocalStack Cloud**, without requiring actual AWS resources.

🛠️ Prerequisites:

- LocalStack Cloud account
- Python 3.6.9 or higher
- LocalStack installed
- Docker installed and running

- AWS CLI installed and configured

🚀 Setup and Configuration:

🔹 Step 1: Sign Up for LocalStack Cloud

Register for a free account at:

👉 https://app.localstack.cloud/sign-up

🔹 Step 2: Sign In to LocalStack Cloud

👉 https://app.localstack.cloud/sign-in

🔹 Step 3: Install LocalStack

Use pip to install LocalStack:

```
pip install localstack
```

🔹 Step 4: Start Docker

Make sure Docker is installed and running.

No sign-in is required for Docker.

🔹 Step 5: Start LocalStack

```
localstack start
```

⚡ **Note:**

- LocalStack will start inside a Docker container.
- Do not close the terminal while LocalStack is running.

🔹 Step 6: Verify LocalStack is Running

Access the LocalStack Web UI:

👉 https://localhost.localstack.cloud:4566

Ensure that LocalStack is running with **HTTPS support**.

🔹 Step 7: Install AWS CLI

```
pip3 install awscli
```

🔹 Step 8: Configure AWS CLI with Dummy Credentials

```
aws configure
```

Provide the following:

- **AWS Access Key ID:** `test`
- **AWS Secret Access Key:** `test`
- **Default region name:** `us-east-1`
- **Default output format:** `json`

🔧 DynamoDB Simulation Using AWS CLI

#	Action	AWS CLI Command	Description
1	Create Table	create-table	Create a new DynamoDB table
2	List Tables	list-tables	List all DynamoDB tables
3	Describe Table	describe-table	Show metadata (status, schema) of a table
4	Put Item	put-item	Insert a new item (record)
5	Get Item	get-item	Fetch an item by primary key
6	Update Item	update-item	Update specific attribute(s) of an item
7	Delete Item	delete-item	Delete a specific item
8	Delete Attribute	update-item with REMOVE	Remove a specific attribute from an item
9	Scan	scan	Read all items (like SELECT * in SQL)

✅ Detailed AWS CLI Commands

📄 Create Table

```
aws --endpoint-url=http://localhost:4566 dynamodb create-table \
    --table-name Users \
    --attribute-definitions AttributeName=UserID,AttributeType=S \
    --key-schema AttributeName=UserID,KeyType=HASH \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

📄 List Tables

```
aws --endpoint-url=http://localhost:4566 dynamodb list-tables
```

📄 Describe Table

```
aws --endpoint-url=http://localhost:4566 dynamodb describe-table --table
-name Users
```

📄 Put Item

```
aws --endpoint-url=http://localhost:4566 dynamodb put-item \
    --table-name Users \
    --item '{"UserID": {"S": "101"}, "Name": {"S": "John Doe"}, "Age": {"N":
"30"}}'
```

📄 Get Item

```
aws --endpoint-url=http://localhost:4566 dynamodb get-item \
    --table-name Users \
    --key '{"UserID": {"S": "101"}}'
```

📄 Update Item

```
aws --endpoint-url=http://localhost:4566 dynamodb update-item \
    --table-name Users \
    --key '{"UserID": {"S": "101"}}' \
```

```
    --update-expression "SET Age = :newAge" \
    --expression-attribute-values '{":newAge":{"N":"35"}}'
```

📄 Delete Attribute

```
aws --endpoint-url=http://localhost:4566 dynamodb update-item \
    --table-name Users \
    --key '{"UserID": {"S": "101"}}' \
    --update-expression "REMOVE Age"
```

📄 Delete Item

```
aws --endpoint-url=http://localhost:4566 dynamodb delete-item \
    --table-name Users \
    --key '{"UserID": {"S": "101"}}'
```

📄 Scan Table

```
aws --endpoint-url=http://localhost:4566 dynamodb scan --table-name Us
ers
```

✅ Conclusion:

Successfully simulated Amazon DynamoDB using **LocalStack Cloud** and **AWS CLI**, allowing CRUD operations, schema management, and testing in a local development environment without incurring AWS costs.