

DevSecOps Workflow for a Secure Web Application

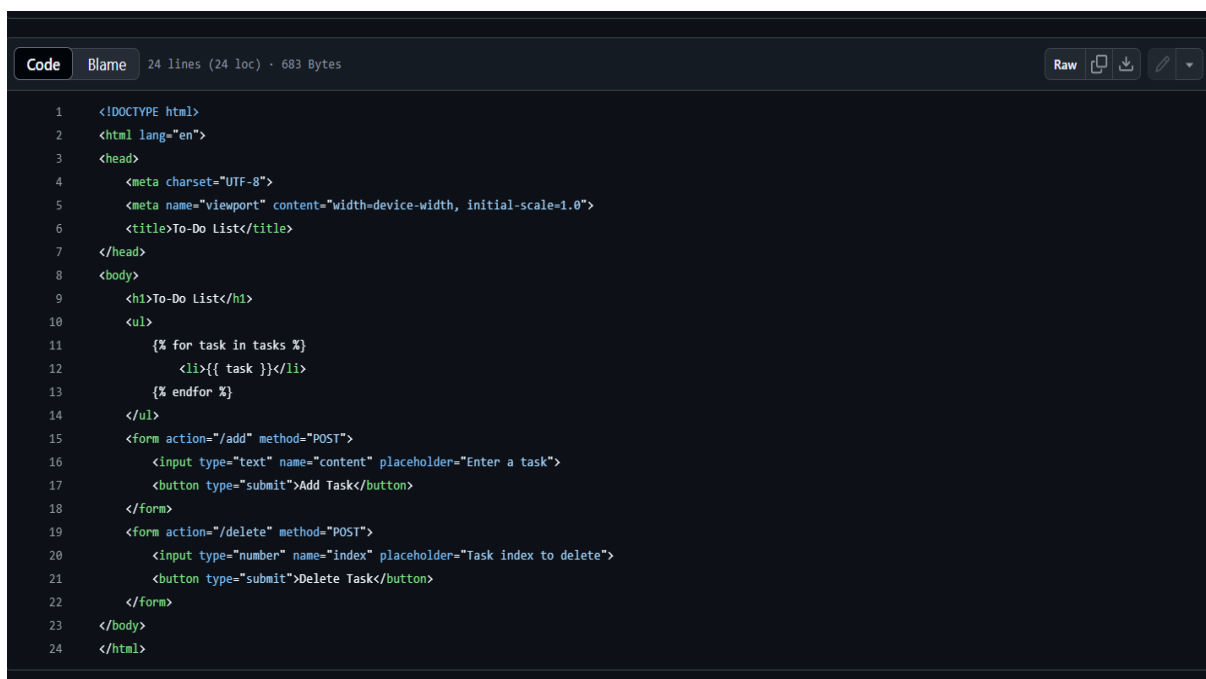
In this exercise, the task is deploying a basic to-do list web application securely. The end goal is to ensure the application is free of common vulnerabilities and follows secure DevSecOps practices. The solutions provided showcases practice of integrating security into the software development lifecycle using a simple CI/CD pipeline for a secure deployment.

Task Step 1: Secure the Application Code

For this task, I was provided with poorly written application code which required fixing the vulnerabilities to make the code more secure to ensure a secure application.

The images below show the application code which required securing, the first code is the index.html and the other is app.py

Index.html code with vulnerabilities



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>To-Do List</title>
7 </head>
8 <body>
9   <h1>To-Do List</h1>
10  <ul>
11    {% for task in tasks %}
12    <li>{{ task }}</li>
13    {% endfor %}
14  </ul>
15  <form action="/add" method="POST">
16    <input type="text" name="content" placeholder="Enter a task">
17    <button type="submit">Add Task</button>
18  </form>
19  <form action="/delete" method="POST">
20    <input type="number" name="index" placeholder="Task index to delete">
21    <button type="submit">Delete Task</button>
22  </form>
23 </body>
24 </html>
```

Issues and Vulnerabilities with the Index.html code

The HTML code is part of a to-do-list application which includes a task display and forms for adding and deleting tasks. Some aspects of the application can result in security concerns when combined with backend functionality.

The code is missing Cross-Site Request Forgery (CSRF) protection because the form uses the POST method to submit data i.e. for adding and deleting tasks with no CSRF protection implementation – this is evident in lines 15 and 19 of the code.

In line 20 the form uses `<input type="number" name="index" placeholder="Task index to delete">` and the form task deletion accepts an index but if the index is not properly validated and sanitized on the server side, the user can manipulate the index to delete tasks that they should not have authorisation to and leading to unauthorised deletions.

Line 16 `<input type="text" name="content" placeholder="Enter a task">` has unescaped user input on the form. This field takes user input but if the backend doesn't sanitize or escape the input, it will expose the application to Cross-Site Scripting (XSS) vulnerabilities and enable malicious contents to be stored.

There is lack of input validation for the index as the task deletion form accepts a number input for the task index. However, if the backend doesn't validate this input, the user can provide an invalid value like a negative number, a very large number or a non-numeric value.

The code has no evidence of authentication and authorization implemented. In the case of a multi-user scenario, it can lead to users modifying or deleting each other's tasks.

The form has improper error handling as the deleting tasks and adding tasks lacks error handling on the server side, and on the client-side i.e. submitting an empty or invalid task data, there is no way to handle the error.

Issues and Vulnerabilities for the app.py code

App.py code with vulnerabilities

```
Code Blame 33 lines (26 loc) · 964 Bytes
1 from flask import Flask, request, jsonify, render_template
2 import os
3
4 app = Flask(__name__)
5
6 # Hardcoded secret key (vulnerability)
7 SECRET_KEY = "mysecretkey123"
8
9 # Mock database
10 tasks = []
11
12 @app.route('/')
13 def home():
14     return render_template('index.html', tasks=tasks)
15
16 @app.route('/add', methods=['POST'])
17 def add_task():
18     task_content = request.form.get('content') # No input validation
19     if task_content:
20         tasks.append(task_content)
21     return jsonify({"message": "Task added successfully!"}), 200
22     return jsonify({"error": "Content cannot be empty!"}), 400
23
24 @app.route('/delete', methods=['POST'])
25 def delete_task():
26     task_index = int(request.form.get('index')) # No input validation
27     if 0 <= task_index < len(tasks):
28         tasks.pop(task_index)
29     return jsonify({"message": "Task deleted successfully!"}), 200
30     return jsonify({"error": "Invalid task index!"}), 400
31
32 if __name__ == '__main__':
33     app.run(debug=True)
```

This code is for a Flask application and has several potential security vulnerabilities and issues that can be exploited in different attack scenarios.

The code contains hard coded secret which is evident on line 7 `SECRET_KEY = "mysecretkey123"`. In the production environment, the Flask app uses this secret key to sign cookies, sessions and sensitive data. Hardcoding the key is poor security practice as any user can access the source code can retrieve it without efforts.

Lines 17 to 21 has no implementation of input validation for task content because the `add_task()` function doesn't validate the content of the task prior to adding it to the tasks. It can lead to malicious or unexpected input being added to the task such as code injection or malformed data.

Line 25 to 29 has no implementation of input validation also, as the `delete_task()` function accepts the index parameter from the `request.form.get('index')` and converts it to an integer without validating the value. It can result in `ValueError` or unintended behaviour if the value submitted is a non-integer value.

Line 33 `app.run(debug=True)`, has insecure debug mode implementation. In production, running the Flask app with the debugger enables is a critical security risk,

as it could expose sensitive data and enable remote code to be executed if an attacker finds a vulnerability.

There is no access controls implemented for adding and deleting task for the app which means that any user could modify the task list and leading to unauthorized access or manipulation of the data.

There is potential denial of service (DoS) with large task lists as there is no limit for the number of tasks that can be stored in the memory, which can lead to memory exhaustion if the list increases largely. Therefore, an attacker can exploit this vulnerability to cause a DoS attack by adding many tasks.

Proposed Solution for Index.html code

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>To-Do List</title>

</head>

<body>

  <h1>To-Do List</h1>

  <ul>

    {% for task in tasks %}

      <!-- Automatically escape variables to prevent XSS -->

      <li>{{ task | escape }}</li>

    {% endfor %}

  </ul>


  <!-- CSRF token implementation (assumed server-side generation) -->

  <form action="/add" method="POST">

    <input type="text" name="content" placeholder="Enter a task" required>

    <!-- CSRF token for protection -->

    <input type="hidden" name="csrf_token" value="{{ csrf_token }}">

    <button type="submit">Add Task</button>

  </form>
```

```

<!-- CSRF token for delete form -->

<form action="/delete" method="POST">

    <input type="number" name="index" placeholder="Task index to delete" required min="0">

    <!-- CSRF token for protection -->

    <input type="hidden" name="csrf_token" value="{{ csrf_token }}">

    <button type="submit">Delete Task</button>

</form>

</body>

</html>

```

Explanation of proposed Index.html Code

The code was adjusted to include XSS prevention, in the for loop where the tasks are displayed, the task content now escaped by using the escape filter. This implementation prevents any HTML or JavaScript code from executing and ensuring that any special characters in the task like <, > or & are rendered as plain text rather than being interpreted as HTML or JavaScript.

Implementation of CSRF protection by adding a hidden csrf_token field on the form. The server must then generate a CSRF token for each session and include it in the page's HTML. The token must be submitted with the form and then verified on the server side prior to processing the request.

Input validation was also implemented by adding the "required" attribute to the input fields to ensure that the fields cannot be submitted empty. The min="0" attribute on the delete form input ensures that the index is a valid non-negative number.

The code has been improved so that on the server side, CSRF token is generated and validated and when processing the submission form verify that the CSRF token submitted with the form matches the one stored on the session.

The forms were also enhanced so that input is validated and sanitized. The input values for adding tasks and index for deleting tasks are validated and sanitized on the server before performing any actions.

Proposed Solution for App.py code

```

"""Flask app for managing a simple task list with add and delete functionality."""

import os

import random

```

```

import string

from flask import Flask, request, jsonify, render_template

app = Flask(__name__)

# Use environment variable or random generation for secret key (avoids hardcoding)
app.config['SECRET_KEY'] = os.getenv('SECRET_KEY', ".join(
    random.choices(string.ascii_letters + string.digits, k=24)
))

# Mock database (tasks list)
tasks = []

@app.route('/')
def home():
    """Render the home page with the current list of tasks."""
    return render_template("index.html", tasks=tasks)

@app.route('/add', methods=['POST'])
def add_task():
    """Add a new task from the content."""

    task_content = request.form.get('content') # Get user input
    if task_content:
        # Input validation to prevent XSS
        task_content = task_content.strip() # Remove unnecessary whitespace
        if len(task_content) > 0:
            tasks.append(task_content)

            return jsonify({"message": "Task added successfully!"}), 200
        return jsonify({"error": "Content cannot be empty!"}), 400
    return jsonify({"error": "Content cannot be empty!"}), 400

@app.route('/delete', methods=['POST'])
def delete_task():
    """Delete a task."""

    try:

```

```

task_index = request.form.get('index')

# Validate the index to ensure it is an integer and within range
if task_index is None:

    return jsonify({"error": "Index is required!"}), 400

task_index = int(task_index)

if 0 <= task_index < len(tasks):

    tasks.pop(task_index)

    return jsonify({"message": "Task deleted successfully!"}), 200

return jsonify({"error": "Invalid task index!"}), 400
except ValueError:

    return jsonify({"error": "Invalid index format!"}), 400
except (TypeError, IndexError) as e:

    # Handle any unexpected exceptions gracefully
    app.logger.error(f"Unexpected error occurred: {e}")

    return jsonify({"error": f"An error occurred: {str(e)}"}), 500

if __name__ == '__main__':

    # In production, never use debug=True

    app.run(debug=False)

```

Explanation of Proposed solution for App.py

The SECRET_KEY is not hardcoded anymore, rather it is fetched from an environment variable using os.getenv and if not found, it is generated randomly with random.choices. This ensures that it is more secure and avoids exposure in the code repositories.

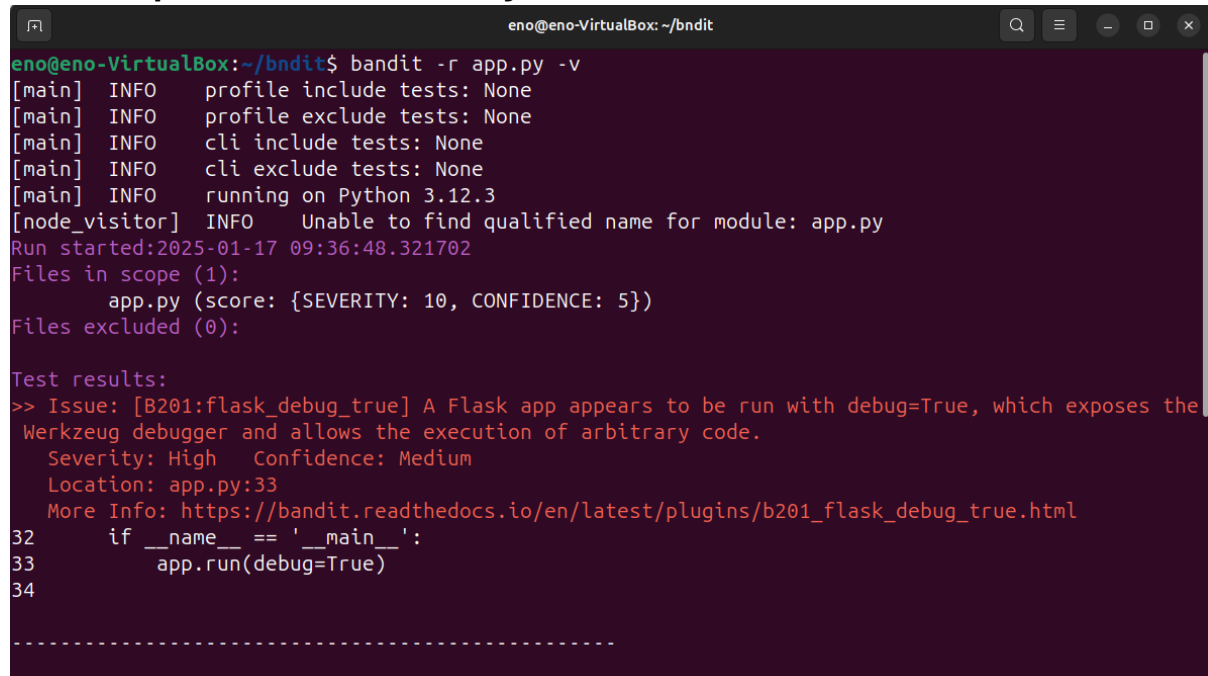
The input is sanitized – whitespace trimmed, and checks ensure its not just an empty string or only containing spaces. This prevents the possibility of storing empty or malformed data and provides a basic defence against XSS or injection attacks that can occur if the user data is rendered or used elsewhere.

Task index is now validated because before processing the deletion request, the task_index is validated to ensure it's a valid integer and is within the bounds of tasks list. Errors are caught if the conversion fails (VaueError). This prevents issues like invalid values from being passed that can cause crashes or unexpected behaviour.

Error handling was added to the code by adding try-except block around the deletion logic to handle unexpected exceptions, providing a meaningful error message to the

client. Proper error handling ensures that the application does not expose internal data or crash when unexpected situations occur.

Task Step 2: Static Code Analysis



```
eno@eno-VirtualBox: ~/bndit
eno@eno-VirtualBox:~/bndit$ bandit -r app.py -v
[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.12.3
[node_visitor] INFO Unable to find qualified name for module: app.py
Run started:2025-01-17 09:36:48.321702
Files in scope (1):
    app.py (score: {SEVERITY: 10, CONFIDENCE: 5})
Files excluded (0):

Test results:
>> Issue: [B201:flask_debug_true] A Flask app appears to be run with debug=True, which exposes the
Werkzeug debugger and allows the execution of arbitrary code.
Severity: High Confidence: Medium
Location: app.py:33
More Info: https://bandit.readthedocs.io/en/latest/plugins/b201\_flask\_debug\_true.html
32     if __name__ == '__main__':
33         app.run(debug=True)
34
-----
```

The Python script App.py was scanned using Bandit, a security analysis tool designed to identify potential vulnerabilities in Python code. One of the primary concerns highlighted by Bandit is the use of debug=True within the Flask application configuration. This configuration has significant security implications.

The Issue:

In a Flask application, when debug=True is set, it enables Flask's built-in debugger, which is powered by Werkzeug (a comprehensive WSGI utility library). This debugger provides an interactive Python shell that can be used for inspecting variables, evaluating expressions, and performing other tasks that are useful during development. While this can be helpful in a development environment, it becomes a critical security risk in a production environment.

By exposing the Werkzeug debugger in production, an attacker could potentially send a specially crafted request to the application and gain access to the interactive debugger. This debugger would allow them to execute arbitrary Python code on the server, which could lead to the following:

- **Remote Code Execution (RCE):** An attacker could exploit the debugger to run arbitrary code on the server. This could range from executing commands to compromise sensitive data, alter the application's functionality, or launch additional attacks.

- **Information Disclosure:** The interactive debugger may expose sensitive information about the application's internals, including the structure of the codebase, environment variables, and configurations.

Severity:

The severity of this issue is classified as "high" or even "critical" because:

1. **Remote Code Execution:** Allowing arbitrary code execution on the server can lead to a complete compromise of the application and potentially the underlying infrastructure. This is one of the most severe security risks in any application.
2. **Production Environment Vulnerability:** If `debug=True` is left in a production environment, it could easily be exploited by an attacker who discovers this vulnerability, potentially impacting the entire application and its users.
3. **Widespread Exposure:** Because Flask's default behaviour in development is to allow the debugger, developers may forget to disable it when deploying to production, making this a common but dangerous oversight.

Fixes:

```
app.config['DEBUG'] = False # Ensure debugging is off in production
```

Disable Debugging in Production: Ensure that the debug mode is explicitly set to `False` in production. This prevents the application from exposing the Werkzeug debugger. The debug mode should only be enabled during the development phase.

Environment Configuration: Use environment variables to differentiate between development and production environments. This ensures that debugging is enabled only in non-production environments. As seen in the code below:

```
import os
```

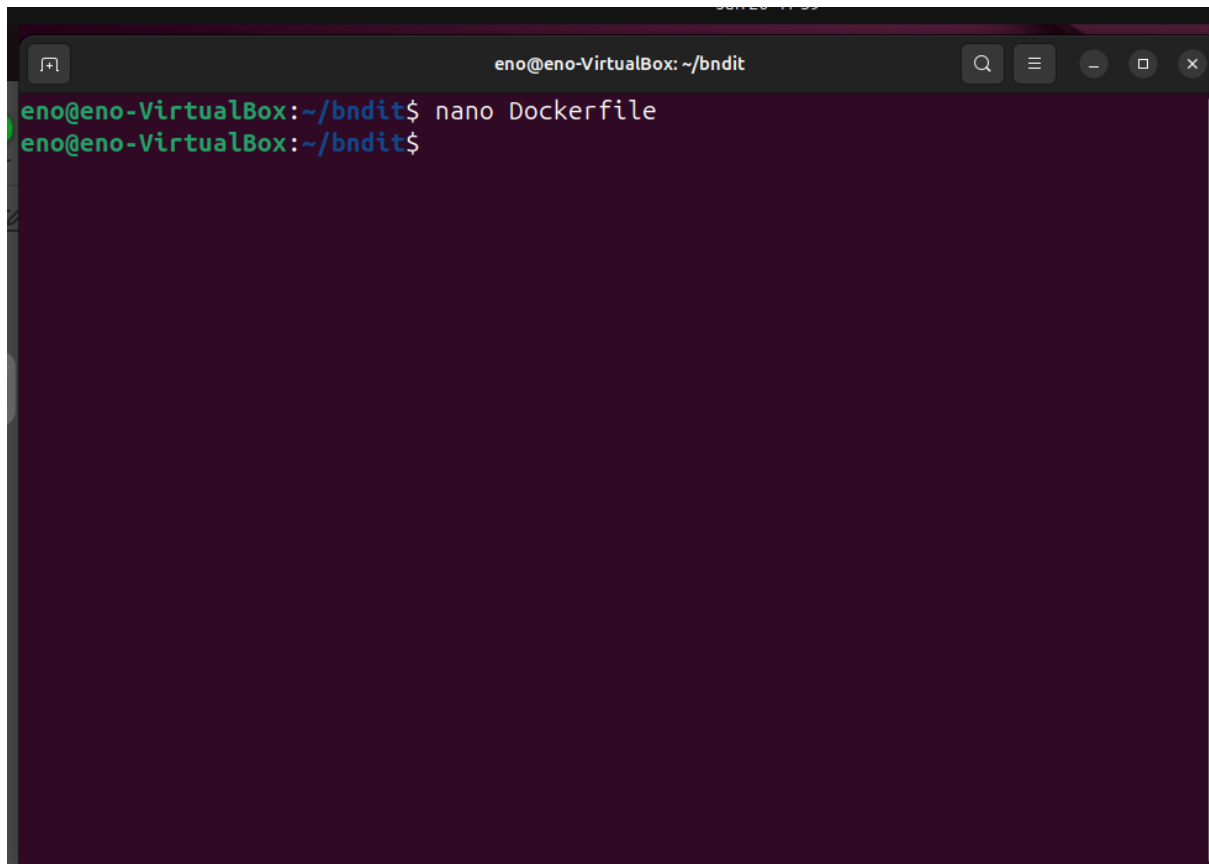
```
app.config['DEBUG'] = os.environ.get('FLASK_ENV') == 'development'
```

Security Review: A thorough review of the application's security settings was conducted in Step 1, ensuring other security best practices (such as input validation, proper authentication, etc.) are in place.

Task Step 3: Containerization

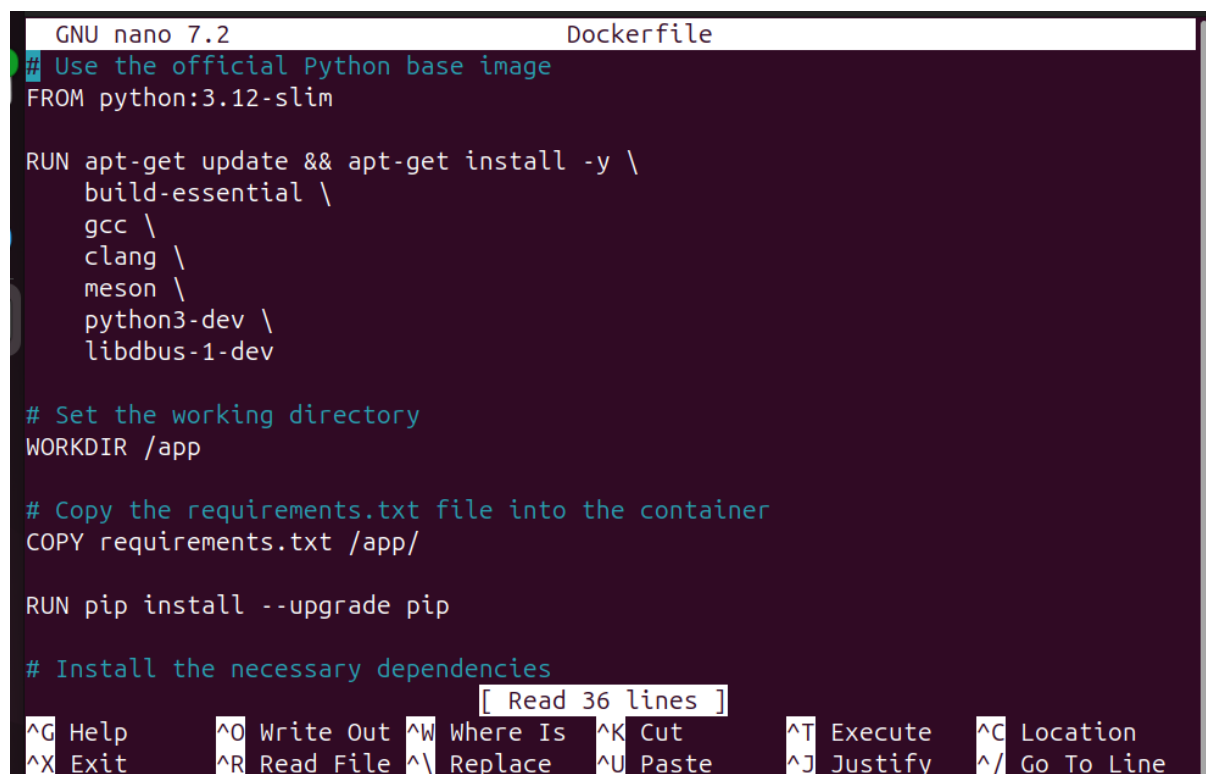
The `App.py` code was updated to avoid hardcoding sensitive information. The updated code is shown in step 1.

Docker was installed on the Ubuntu Desktop which is the environment being used. The Docker file was then created as shown in the images below:

A terminal window titled 'eno@eno-VirtualBox: ~/bndit' is shown. The prompt is 'eno@eno-VirtualBox:~/bndit\$'. The command 'nano Dockerfile' has been entered and executed, resulting in a new prompt 'eno@eno-VirtualBox:~/bndit\$'. The terminal background is dark purple.

```
eno@eno-VirtualBox:~/bndit$ nano Dockerfile
eno@eno-VirtualBox:~/bndit$
```

Creation of Dockerfile



```
GNU nano 7.2 Dockerfile
# Use the official Python base image
FROM python:3.12-slim

RUN apt-get update && apt-get install -y \
    build-essential \
    gcc \
    clang \
    meson \
    python3-dev \
    libdbus-1-dev

# Set the working directory
WORKDIR /app

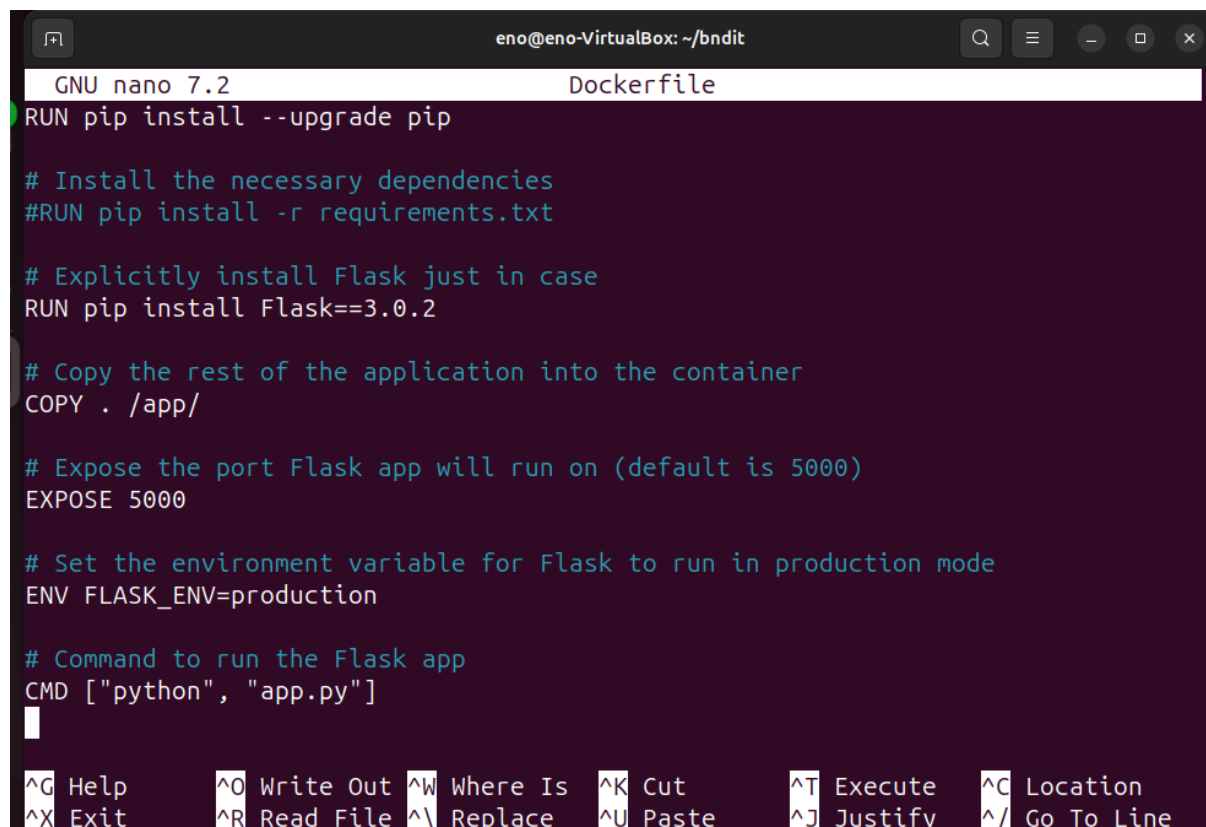
# Copy the requirements.txt file into the container
COPY requirements.txt /app/

RUN pip install --upgrade pip

# Install the necessary dependencies
```

[Read 36 lines]

^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line



```
GNU nano 7.2 Dockerfile
RUN pip install --upgrade pip

# Install the necessary dependencies
#RUN pip install -r requirements.txt

# Explicitly install Flask just in case
RUN pip install Flask==3.0.2

# Copy the rest of the application into the container
COPY . /app/

# Expose the port Flask app will run on (default is 5000)
EXPOSE 5000

# Set the environment variable for Flask to run in production mode
ENV FLASK_ENV=production

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

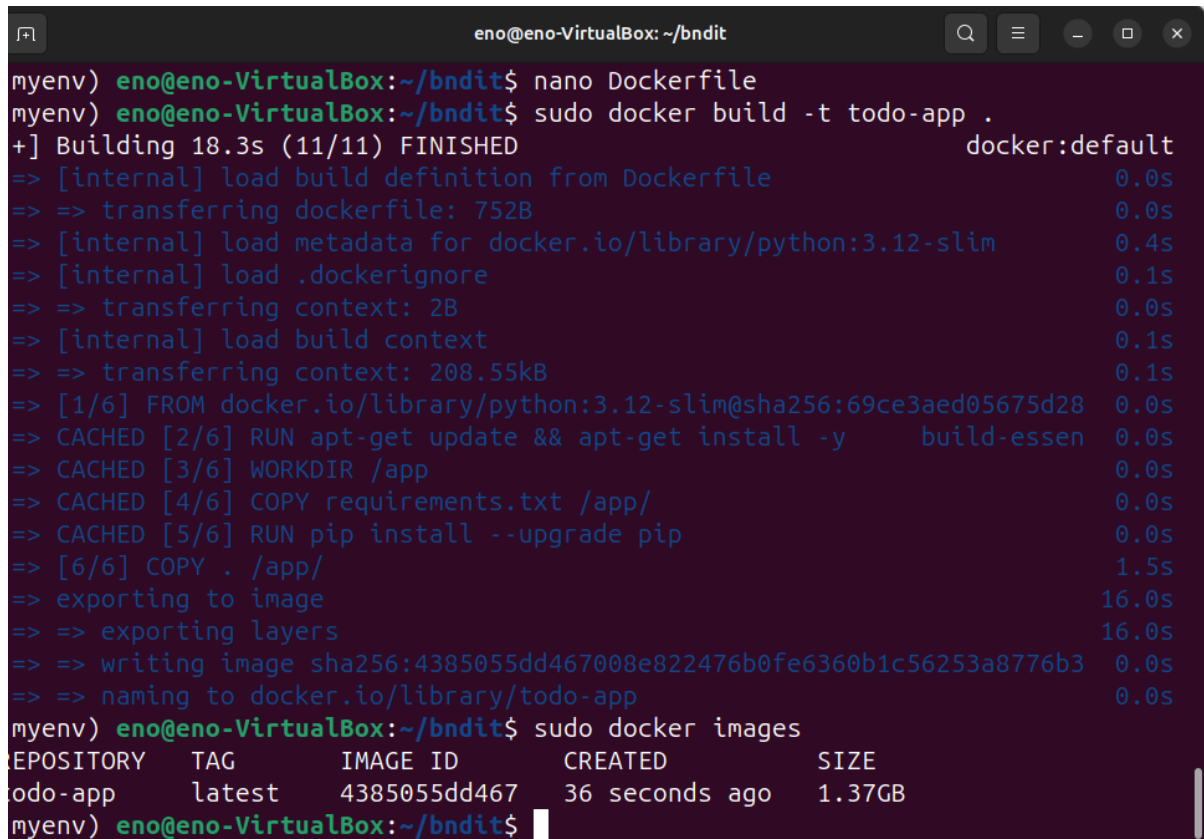
eno@eno-VirtualBox: ~/bndit

^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line

A directory “Bndit” was created and this was where all files were created in – the

App.py, Dockerfile, and requirements.txt files and it was also within this directory that the docker images and containers were executed.

The docker image for the Todo app was built successfully in Docker as seen in the screenshot below:

A terminal window titled 'eno@eno-VirtualBox: ~/bndit' showing the successful build of a Docker image. The user runs 'nano Dockerfile' and 'sudo docker build -t todo-app .'. The build process is detailed with steps like loading build definitions, transferring files, and installing dependencies. The final output shows the image 'todo-app:latest' with ID '4385055dd467' and size '1.37GB'.

```
myenv) eno@eno-VirtualBox:~/bndit$ nano Dockerfile
myenv) eno@eno-VirtualBox:~/bndit$ sudo docker build -t todo-app .
+ ] Building 18.3s (11/11) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 752B                                0.0s
=> [internal] load metadata for docker.io/library/python:3.12-slim 0.4s
=> [internal] load .dockerignore                                   0.1s
=> => transferring context: 2B                                       0.0s
=> [internal] load build context                                    0.1s
=> => transferring context: 208.55kB                                  0.1s
=> [1/6] FROM docker.io/library/python:3.12-slim@sha256:69ce3aed05675d28 0.0s
=> CACHED [2/6] RUN apt-get update && apt-get install -y build-essen 0.0s
=> CACHED [3/6] WORKDIR /app                                       0.0s
=> CACHED [4/6] COPY requirements.txt /app/                        0.0s
=> CACHED [5/6] RUN pip install --upgrade pip                      0.0s
=> [6/6] COPY . /app/                                              1.5s
=> exporting to image                                              16.0s
=> => exporting layers                                              16.0s
=> => writing image sha256:4385055dd467008e822476b0fe6360b1c56253a8776b3 0.0s
=> => naming to docker.io/library/todo-app                          0.0s
myenv) eno@eno-VirtualBox:~/bndit$ sudo docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
todo-app      latest   4385055dd467   36 seconds ago 1.37GB
myenv) eno@eno-VirtualBox:~/bndit$
```

Docker container was run successfully

```
eno@eno-VirtualBox: ~/bndit
=> [internal] load metadata for docker.io/library/python:3.12-slim 0.4s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> CACHED [1/7] FROM docker.io/library/python:3.12-slim@sha256:69ce3aed0 0.0s
=> [internal] load build context 0.2s
=> => transferring context: 239.44kB 0.2s
=> [2/7] RUN apt-get update && apt-get install -y build-essential 89.0s
=> [3/7] WORKDIR /app 0.1s
=> [4/7] COPY requirements.txt /app/ 0.1s
=> [5/7] RUN pip install --upgrade pip 4.3s
=> [6/7] RUN pip install Flask==3.0.2 2.3s
=> [7/7] COPY . /app/ 2.0s
=> exporting to image 13.6s
=> => exporting layers 13.6s
=> => writing image sha256:812e327dd5f31ac5c0065862c4953306d5a3d772c8b8c 0.0s
=> => naming to docker.io/library/todo-app 0.0s
eno@eno-VirtualBox:~/bndit$ sudo docker run -p 5000:5000 todo-app
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

I opened <http://127.0.0.1:5000> on web browser and the Todo app successfully loaded on the web browser.

Step 4: Basic Vulnerability Scanning with Trivy

Trivy was installed on the VM and used to scan todo-app docker image for vulnerabilities using the command “Sudo trivy image todo-app” which was successful, and it feedback a combination of critical, high, medium and low vulnerabilities.

Critical vulnerability

The only critical vulnerability was – “zlib: interger overflow and resultant heap-based buffer, overflow in zipOpenNewFileInZip4_6” and assign CVE: CVE-2023-45853. CVE-2023-45853 is a security vulnerability impacting the MiniZip in zlib versions up to and including 1.3. MiniZip, though not officially supported as part of the zlib product, is a component used for creating and modifying ZIP archives. The vulnerability arises from an integer overflow that leads to a heap-based buffer overflow in the zipOpenNewFileInZip4_64 function. This issue can be triggered by processing files with long filenames, comments, or extra fields.

The National Vulnerability Database (NVD) assigned it a CVSS v3.1 score of 9.8, indicating a critical severity level. This vulnerability also impacts pyminizip versions up to 0.2.6, as it bundles an affected version of zlib and exposes MiniZip functionality through its compression API.

To mitigate this vulnerability, it is recommended to upgrade to zlib version 1.3.1, which includes a fix addressing the integer overflow and buffer overflow issues. It is recommended to upgrade pyminizip a version later than 0.2.6 to prevent the vulnerability.

High vulnerability

A high vulnerability was discovered as – “perl: CPAN.pm does not verify TLS certificate when downloading distributions over HTTPS” with CVE of CVE-2023-31484. CVE-2023-31484 is a security vulnerability affecting CPAN.pm versions prior to 2.35. The issue arises because these versions do not verify TLS certificates when downloading distributions over HTTPS, potentially exposing systems to man-in-the-middle attacks.

The NVD assigns this vulnerability a CVSS v3.1 base score of 8.1, indicating a high severity level. To address this vulnerability, it is recommended to update CPAN.pm to version 2.35 or later and Perl to version 5.38.0 or later. It is recommended to configure systems to verify TLS certificates when downloading distributions over HTTPS.

Medium vulnerability

One of the medium vulnerabilities was – “python: cpython: URL parser allowed square brackets in domain names” and assigned CVE-2025-0938. CVE-2025-0938 is a vulnerability identified in the Python standard library's URL parsing functions, specifically urllib.parse.urlsplit and urlparse. These functions improperly accept domain names containing square brackets, which is inconsistent with RFC 3986 standards that reserve square brackets for delimiting IPv6 hosts in URLs. This discrepancy can lead to inconsistent parsing behaviours between Python's URL parser and other compliant parsers.

The Python Software Foundation has assigned a CVSS v4.0 base score of 6.3 (Medium severity) to this vulnerability. Exploiting this issue could allow attackers to manipulate data or potentially alter the control flow of applications, leading to unauthorized command execution. To mitigate this vulnerability, it is recommended to update Python to versions 3.12.9 or 3.13.2, where this issue has been addressed. It is recommended to validate and sanitize URLs to ensure compliance with RFC 3986, avoiding domain names with square brackets.

Low vulnerability

A low vulnerability was – “python: Mishandling of comma folding during folding and Unicode-encoding of email headers” and assigned with CVE-2025-1795. CVE-2025-1795 is a vulnerability identified in the CPython implementation of Python's email handling module. The issue arises during the folding of address lists in email headers, when a separating comma appears at the end of a folded line and that line is Unicode-encoded, the comma itself is also encoded. This behaviour deviates from the expected functionality, where the separating comma should remain a standard comma, potentially leading to misinterpretation of the address header by certain mail servers. The vulnerability has been assigned a CVSS v4.0 base score of 2.3, categorized as low severity.

The Python Software Foundation has acknowledged this vulnerability and addressed it in subsequent commits to the CPython repository. The primary mitigation for CVE-2025-1795 is to upgrade to the latest version of Python where the vulnerability has been addressed. The issue has been fixed in subsequent versions of CPython, so upgrading will resolve the problem. Ensuring that systems are patched to the version that contains the fix for the vulnerability. This includes patching any software or systems that rely on Python and are affected by the CVE.

Step 5: CI/CD Pipeline Setup

In this step, GitHub Actions was utilized to create a CI/CD pipeline that performs static code analysis, builds a Docker image, and scans it for vulnerabilities. CI/CD, which stands for Continuous Integration and Continuous Deployment (or Delivery), is a set of automated processes that enable developers to deliver code changes more frequently and reliably.

To set up the CI/CD pipeline, I created a GitHub Actions workflow as a YAML file within the repository for *Refonte Tasks*. I also uploaded the necessary project files — including Dockerfile, app.py, and requirements.txt — to the app directory of the repository. Once the setup was complete, I executed the workflow to initiate the build process.

The github-action.yml file for the CI/CD pipeline was designed to:

- Use the latest version of Ubuntu for the build environment
- Checkout the repository code
- Set up Python for static code analysis
- List directory files for debugging purposes
- Install Python dependencies
- Install additional tools required for static code analysis
- Perform static code analysis using pylint
- Build the Docker image
- Scan the Docker image for vulnerabilities using Trivy

Using Github-Actions to run the static code analysis for the project provides certain benefits. GitHub-Actions provides automation for consistent checks which ensures static analysis runs automatically on every push and pull request, which reduces human errors include providing immediate feedback regarding issues to reduce bugs and technical setbacks. GitHub-Actions detects vulnerabilities or insecure patterns especially with the use of python which was also used for the code generation for the project.

GitHub-Actions being used in building the Docker Image has some benefits as well. It provides CI/CD automation in automating the build, test, and push of the Docker images in every code commit, ensuring consistent and repeatable builds without manual intervention. GitHub-Actions seamlessly integrates with the existing GitHub repositories and triggers workflows based on events. It is easier to test the Docker image against multiple base images or configurations using matrix builds.