Lab Sheet: Pre-Commit Stage
Secrets Management & Security Linting

Dr. Roshan N Rajapakse

Session Details

- **Module:** Fundamentals of Secure Software Development
- **Tools Required:** Git, Python, Bandit, Gitleaks
- **Objectives:**
  1. Detect and manage secrets in code during the pre-commit stage.
  2. Use Bandit to detect security issues in Python code.
  3. Implement best practices for secure coding during the pre-commit stage.

Secrets Management

Step-by-Step Instructions

1. Setup:

1. Open your terminal or command prompt.
2. Navigate to or create a new project directory:

```
mkdir pre_commit_lab
cd pre_commit_lab
```

2. Install Gitleaks:

1. Install Gitleaks based on your operating system:

```
# For Linux / macOS
curl -s https://api.github.com/repos/gitleaks/gitleaks/releases
   /latest | \
grep "browser_download_url.*gitleaks_.*_linux_amd64.tar.gz" | \
cut -d '"' -f 4 | wget -qi -
tar -xvf gitleaks_*.tar.gz
sudo mv gitleaks /usr/local/bin/
```

2. Verify installation:

```
gitleaks --version
```

3. Create a Repository with Secrets:

1. Initialize a new Git repository:

```
git init secrets-demo
cd secrets-demo
echo "AWS_SECRET_KEY=abcd1234efgh5678" > secrets.txt
git add secrets.txt
git commit -m "Add secrets for testing"
```

2. Create a Python script named ‘app.py‘ with the following content:

```
# This Python script contains hardcoded secrets, which is a bad practice.
def connect_to_service():
    api_key = "abcd1234efgh5678"  # Hardcoded API key (secret)
    print(f"Connecting to the service with API key: {api_key}")

connect_to_service()
```

3. Add the file to the repository:

```
git add app.py
git commit -m "Add Python script with hardcoded secrets"
```

4. Detect Secrets with Gitleaks:

1. Run Gitleaks in the repository:

```
gitleaks detect --source .
```

2. Report the output.

Using Environment Variables

Step-by-Step Instructions

1. Create a Configuration File:

1. Create a ‘.env‘ file in the project directory:

```
touch .env
```

2. Add your secret to the ‘.env‘ file:

```
AWS_SECRET_KEY=abcd1234efgh5678
```

3. Add ‘.env‘ to the ‘.gitignore‘ file to prevent it from being committed:

```
echo ".env" >> .gitignore
git add .gitignore
git commit -m "Update .gitignore to exclude .env files"
```

2. Load Environment Variables in Code:

1. Use a package like ‘python-dotenv‘ to load the environment variables in Python:

```
pip install python-dotenv
```

2. Update your Python code to load secrets from the ‘.env‘ file:

```
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv()

# Access the secret key
aws_secret_key = os.getenv("AWS_SECRET_KEY")
print(f"The secret key is: {aws_secret_key}")
```

Follow-Up Questions

1. Why is storing secrets directly in code insecure?
2. Does using environment variables improve security? If yes, how? Was it implemented correctly in the above example?
3. What are the limitations of using ‘.env‘ files for secrets management?
4. What are other approaches for secrets management in a Git-based environment?

Security Linting with Bandit

Step-by-Step Instructions

1. Install Bandit:

```
    pip install bandit
    bandit --version
```

2. Create a Python Script with Vulnerabilities:

```
import os
import subprocess

def insecure_function():
    user_input = input("Enter a shell command: ")
    subprocess.call(user_input, shell=True)

def weak_crypto():
    from cryptography.hazmat.primitives.ciphers import Cipher,
        algorithms, modes
    cipher = Cipher(algorithms.AES(b"weakkey12345678"), modes.
       CFB(b"iv12345678901234"))

insecure_function()
weak_crypto()
```

3. Run Bandit:

```
    bandit -r .
```

Review the output for identified vulnerabilities.

4. Rectify Security Issues:

```
import os

def secure_function():
    user_input = input("Enter a command: ")
    print("Executing:", user_input)  # Safe handling without
       shell execution

def strong_crypto():
    from cryptography.hazmat.primitives.ciphers import Cipher,
        algorithms, modes
    cipher = Cipher(algorithms.AES(os.urandom(32)), modes.CFB(
       os.urandom(16)))

secure_function()
strong_crypto()
```

Follow-Up Questions

1. What types of vulnerabilities did Bandit detect in the initial script?
2. How did the code example provided in 4. rectify the security issues? Any other improvements that you can propose/alternative approaches?
3. Why is it important to automate security checks like Bandit in pre-commit workflows?

Submission Requirements

1. Screenshots of Gitleaks detecting secrets and the updated implementation using environment variables.
2. Screenshots of Bandit outputs before and after fixing vulnerabilities.
3. Answers to the follow-up questions.
4. Create a document with the above details and submit a pdf.