Done by : Ahmed Fouad

Hands-on Exercise: SAST with Bandit

Module: Introduction to DevSecOps

Introduction:

Welcome to this hands-on lab on Static Application Security Testing (SAST)! In this exercise, you will learn how to proactively identify and fix security vulnerabilities in source code without executing it. We'll use **Bandit**, a widely-used SAST tool for Python, to scan for common weaknesses. This practical lab is designed to give you direct experience with a core "shift left" security practice, ensuring you can find and fix issues early in the development cycle.

Duration: 45 minutes

Prerequisites:

- **Python:** Python 3.6 or newer must be installed on your system.
- **Text Editor:** An integrated development environment (IDE) or a simple text editor (e.g., Visual Studio Code, Notepad++) is necessary for writing code.
- **Terminal/Command Prompt:** You will need a terminal to execute Python commands and run Bandit.

Summary:

This hands-on lab will guide you through the process of using Bandit for SAST. In approximately 45 minutes, you will:

1. Create intentionally insecure Python code to demonstrate vulnerabilities.
2. Set up your environment by installing the Bandit tool.
3. Run a pre-SAST scan to identify and analyze security flaws.
4. Refactor and clean the code to address all identified vulnerabilities.
5. Run a post-SAST scan to validate that the security fixes were successful.

By the end of this lab, you will understand the importance of SAST and how to implement it as a crucial part of a secure development workflow.

Step 1 – Create the Original insecure_code.py

Launch your text editor and create a new file named insecure_code.py. Copy and paste the following intentionally insecure code into the file. This code contains two common security vulnerabilities that Bandit is designed to detect.

```
import os
import hashlib
import pickle

def run_dangerous_command():
    # Vulnerability: Using a function that executes shell commands unsafely
    user_input = "ls -la"
    os.system(user_input)

def create_hash_from_user_input(password):
    # Vulnerability: Using an insecure hashing algorithm
    return hashlib.md5(password.encode()).hexdigest()

def deserialize_data_from_file(filename):
    # Vulnerability: Deserializing untrusted data
    with open(filename, 'rb') as f:
        return pickle.load(f)

if __name__ == "__main__":
    run_dangerous_command()
    print(create_hash_from_user_input("mysecretpassword"))
    # In a real scenario, this would load untrusted data
    # from a file. We will just demonstrate the function here.
```

Step 2 – Set Up Your Environment

Open your terminal or command prompt. Use **pip**, Python's package manager, to install the Bandit tool.

```
pip install bandit
```

After the installation is complete, you are ready to run the security scan.

Step 3 – Run a Pre-SAST Scan on the Insecure Code

In your terminal, navigate to the directory where you saved insecure_code.py. Now, run the Bandit scan using the following command:

```
bandit insecure_code.py
```

Bandit will analyze your code and generate a report. Take note of the findings. You should see a report that highlights the vulnerabilities in the code, providing a description of each issue and its severity.

Step 4 – Create secure_code.py

Now, let's refactor the insecure code. Create a new file called secure_code.py and replace the old, vulnerable functions with the following secure alternatives. We'll replace the unsafe functions with safer, more modern methods.

```
import subprocess
import hashlib
import json

def run_safe_command():
    # Fix: Using subprocess.run for safer command execution
    subprocess.run(["ls", "-la"], check=True)

def create_strong_hash_from_user_input(password):
    # Fix: Using a strong, modern hashing algorithm
    return hashlib.sha256(password.encode()).hexdigest()

def deserialize_safe_data_from_file(filename):
    # Fix: Using a safer format like JSON for deserialization
    with open(filename, 'r') as f:
        return json.load(f)

if __name__ == "__main__":
    run_safe_command()
    print(create_strong_hash_from_user_input("mysecretpassword"))
    # The JSON-based deserialization is now safer.
```

Step 5 – Run a Post-SAST Scan on the Cleaned Code

Now, run Bandit on your new, secure file.

```
bandit secure_code.py
```

You should see a clean report from Bandit, indicating that all vulnerabilities have been successfully resolved. This confirms that your code is now more secure.

Conclusion

In this lab, you've gained practical experience with Static Application Security Testing. You learned how to use Bandit to scan for common vulnerabilities, and most importantly, you saw how simple code changes can significantly improve an application's security posture. By making SAST a regular part of your development process, you can find and fix security issues long before they become a problem.

Done by : Ahmed Fouad

Code Breakdown

Original Insecure Code Explanation:

The original code contains three common vulnerabilities:

- **os.system(user_input):** The os.system function is dangerous because it executes a command directly in the shell. If a user can control the user_input string, they could execute malicious commands, leading to a **Command Injection** vulnerability.
- **hashlib.md5(...):** The MD5 algorithm is a legacy hash function that is no longer considered secure. It is susceptible to collision attacks, making it a poor choice for handling passwords or other sensitive data.
- **pickle.load(f):** The pickle module can be extremely dangerous if used to deserialize data from an untrusted source. Maliciously crafted data can execute arbitrary code on the system, which is a severe security risk.

Cleaned Secure Code Explanation:

The cleaned code demonstrates how to fix these vulnerabilities:

- **subprocess.run(...):** Instead of os.system, we use subprocess.run. This function executes the command and its arguments as a list of strings, preventing command injection attacks by not interpreting the input as a single shell command.
- **hashlib.sha256(...):** We replace the weak MD5 algorithm with **SHA-256**, a modern, cryptographically secure hash function recommended for password hashing and data integrity.
- **json.load(...):** We replace the dangerous pickle with the **json** module. JSON is a safe data format that does not allow for arbitrary code execution, making it the industry standard for handling data from untrusted sources.

By implementing these best practices, you've transformed a vulnerable script into a secure and compliant one.