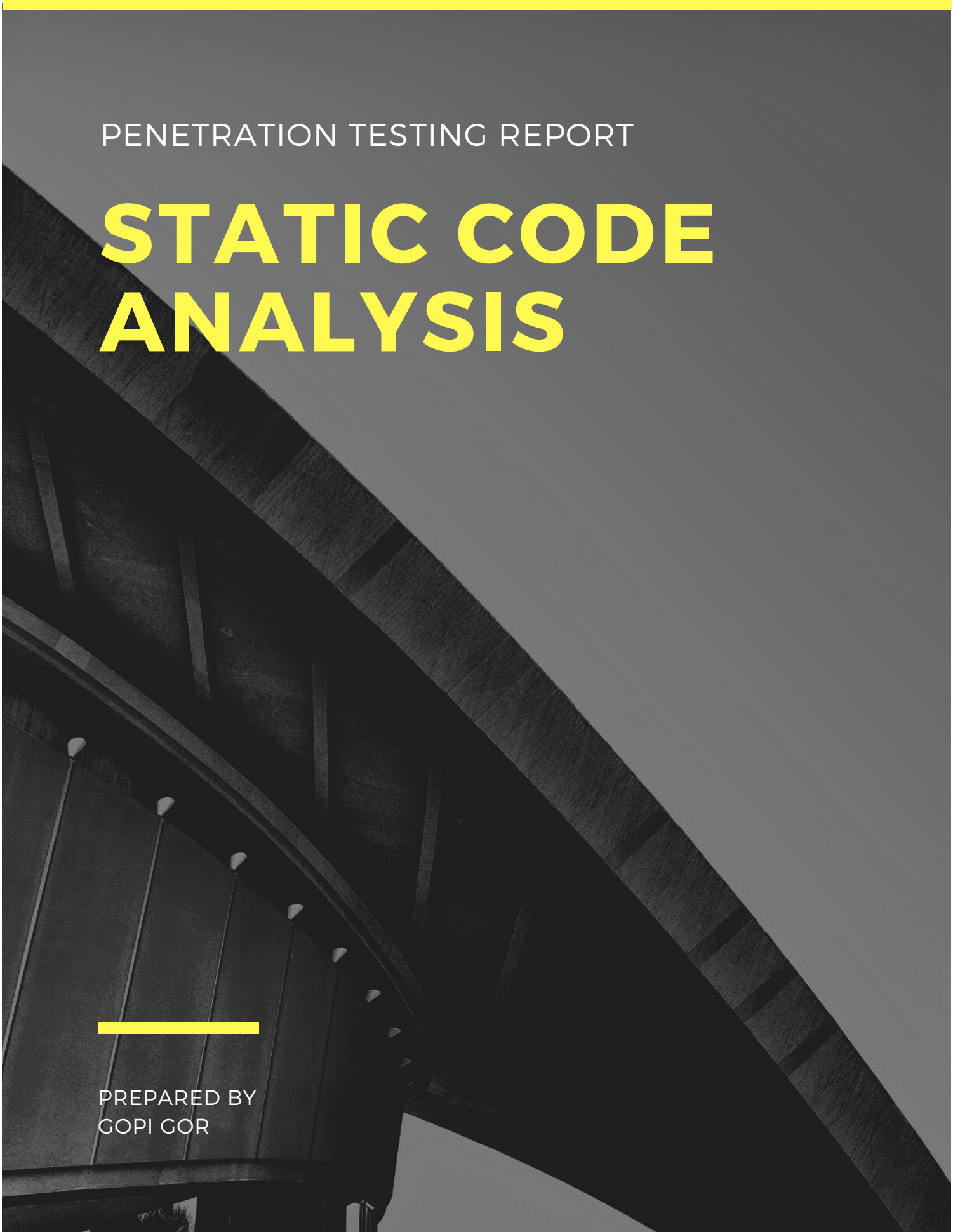
PENETRATION TESTING REPORT

STATIC CODE ANALYSIS



PREPARED BY
GOPI GOR

CONTENTS

EXECUTIVE SUMMARY

This penetration testing report identifies and presents different kinds of findings and recommendations for static code analysis of 2 applications in Python. The test aimed to identify the vulnerabilities in the source code through Bandit and provide recommendations to make the application more secure.

There were multiple vulnerabilities identified in both applications, mainly due to the open ports that allow these vulnerabilities to be exploited. Each vulnerability has been assigned a severity level which can lead to data breaches, security challenges, and any kind of unauthorized access.

The identified weaknesses in the source code of the application could lead to major financial losses, regulatory issues and loss of customer trust. Apart from technical recommendations provided in detail below, it would be advised to ensure secure coding practices and employee training for better development of applications.

SCOPE & OBJECTIVES

The scope of this penetration testing report is to conduct a static code analysis on two applications, for which the source code and repositories have been provided. The overall scope is to complete tests on the applications and provide recommendations to improve the application.

The objectives for the test are listed below:

- Run a port scan script on a server to check open ports
- Install Bandit and clone repositories to conduct the test
- Do a complete static code analysis on 2 applications
- Identify vulnerabilties from results obtained
- Showcase findings for the report
- Provide recommendations for the remediation

METHODOLOGY

The methodology was used to identify potential vulnerabilities in the applications was done using Kali Linux, Metasploitable 2, Python, and Bandit analysis tools. The overall approach used to conduct the test was cloning, running repos and then analyzing each vulnerability for recommendations. The stepwise methodology has been explained below:

- First, the bandit repository was cloned and then the tool was installed in Kali.
- Next, we ran the portscan on Metasploit to understand the automation process in the Python scripts.
- Application 1 was cloned via the GitHub repository by using the command **git clone <repo-link>** and the command **bandit -r lets-be-bad-guys > app1.txt** was run. The > saved the output into a text file app1.txt for further analysis.
- Application 2 was similarly cloned and the and **bandit -r vulpy > app2.txt** was run.

The automated script for bandit -r as a recursive script was a use case that allowed a quick return to the bandit tool for further analysis.

FINDINGS

1. Application 1

When the application **lets-be-bad-guys** was cloned, we first found out that it had **780 objects** and **434 deltas.** After this, we ran the bandit tool to evaluate all the vulnerabilities. The total count was **6 issues.** These issues also show additional information like issue details, CVEs, and severity. Furthermore, at the end of the scan, the run metrics are sorted in severity and confidence levels. A few of the vulnerabilities discovered are listed below.

- B105 - possible hard-coded password
- B110 - try, except, pass detected
- B102 - use of exec detected

These vulnerabilities have repeated in certain conditions and the proof can be found in the appendix.

The severity and confidence levels depend on each of the cases, and while the confidence level may be low, the severity being high could be a sign to be more careful during development processes.

FINDINGS

2. Application 2

When the application **vulpy** was cloned, we first found out that it had **437 objects** and **221 deltas**. After this, we ran the bandit tool to evaluate all the vulnerabilities. The total count was **49 issues**. These issues also show additional information like issue details, CVEs, and severity. Furthermore, at the end of the scan, the run metrics are sorted in severity and confidence levels. A few of the vulnerabilities discovered are listed below.

- B113 - requests call without timeout
- B108 - insecure usage of temp file/directory
- B404 - security implications, subprocess module
- B603 - subprocess, untrusted input
- B608 - hardcoded sql expression, can be SQLi
- B110 - try, except, pass detected
- B105 - possible hard-coded password
- B201 - flask debugger allows arbitrary code to run

It was also noticed that the occurrence of B108 was very frequently discovered in the results. Severity was: **4 (High), 9 (Low), 36 (Medium).**

RECOMMENDATIONS

As we have seen in the findings, there are vulnerabilities that can be harmful for the business if the applications are launched at this stage. Here are some recommendations that can be done to avoid that:

- The password storing mechanisms can be used securely and using of environment variables to avoid password issues.
- For the try, except, pass detections, the exceptions can be applied clearly in the code so that it is handled appropriately in each case.
- The exec functions allows arbitrary code to be run, so it better to avoid that and use functions instead and restrict file permissions.
- For the requests call timeouts, there can be a specific timeout defined based on the application.
- For the hardcoded tmp error, secure modules can be used for temporary files such as tempfile module.
- The subprocess module takes inputs, but these should be verified that they are taken from a secure and trusted so source. They can also be cleaned as a first step.
- The query can be developed as per developer intent to avoid SQLi. This means it should not be in string format.
- Finally, the flash debugger can be disabled to avoid any arbitrary code running.

APPENDIX

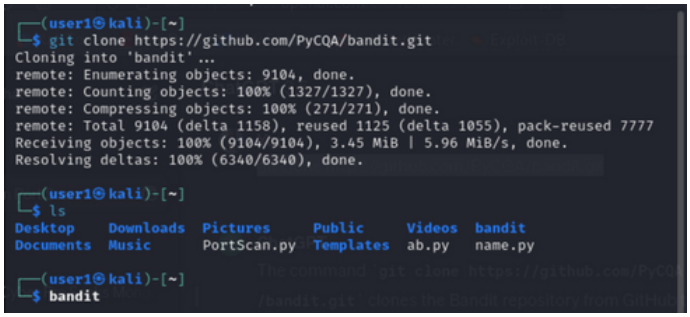


Fig: Bandit repo clone

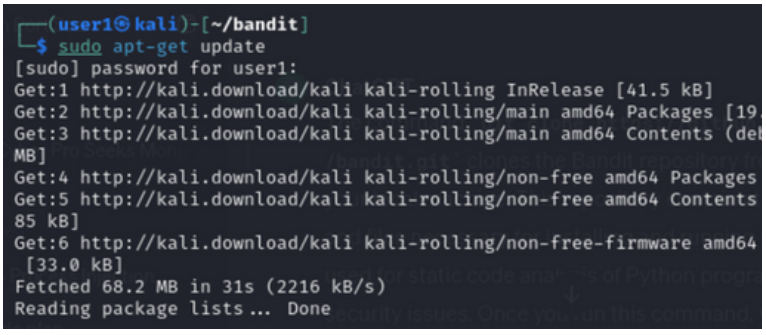


Fig: Bandit directory update

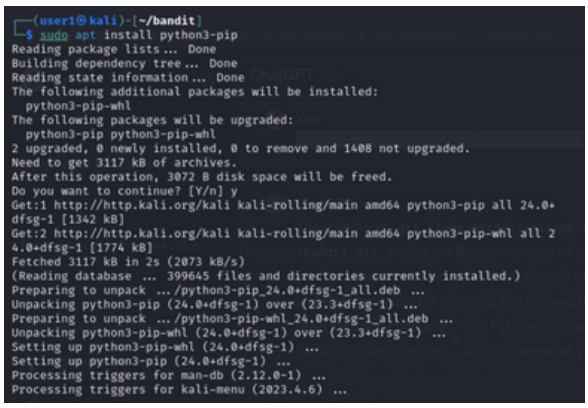


Fig: Bandit directory python install

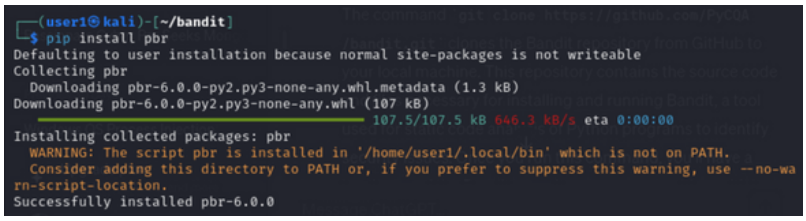


Fig: Bandit pbr to install for missing dependencies

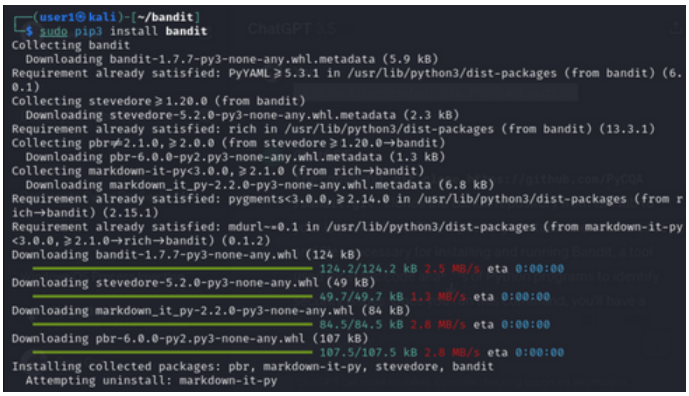


Fig: Installing bandit tool

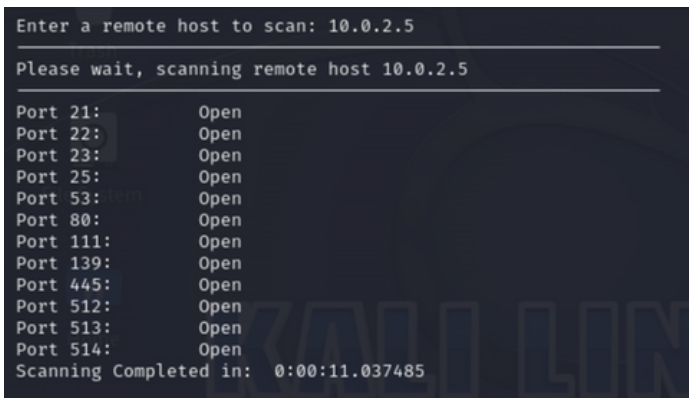


Fig: PortScan.py to check open ports

APPENDIX

Fig: Application 1 clone

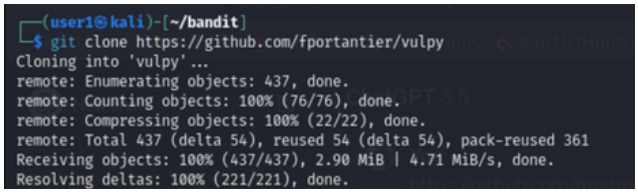


Fig: Application 2 clone

Fig: Application 1 results after bandit

Fig: Application 2 results after bandit

REFERENCES

1. https://realpython.com/prevent-python-sql-injection/
2. https://offensive360.com/how-to-prevent-hardcoded-passwords/
3. https://learn.snyk.io/lesson/insecure-temporary-file/
4. https://www.codiga.io/blog/python-subprocess-security/
5. https://github.com/ACandeias/PenTestingScripts.git
6. https://github.com/PyCQA/bandit.git
7. https://github.com/mpirnat/lets-be-bad-guys
8. https://github.com/fportantier/vulpy