Al-Noor Journal for Information Technology and Cyber Security
December (2024); 1 (0): 113 - 138
DOI: https://doi.org/10.69513/jnfit.v1.i0.a7

Tailoring Static Code Analysis for Top 25 CWE in Python

Ali Shihab[1,a)] Mafaz Alanezi[2,b)]

[1] Dept. of Computer Science, College of Computer Science and Mathematics, University of Mosul, Iraq

[2] ICT Research Unit, Computer Center, University of Mosul, Iraq

[a)] *ali.22csp39@student.uomosul.edu.iq*

[b)] *mafazmhalanezi@uomosul.edu.iq*

Received: 15 / 10/ 2024 Accepted: 20 / 11/ 2024
Published: 17 / 12 / 2024

Abstract

The topic of security for computers is of significant importance. Over the past decade, countless cybercrimes have been executed by exploiting software flaws. This issue has led to considerable social stress, substantial losses, and higher interest in security. Vulnerabilities in applications developed in various programming languages can be identified using various methodologies and techniques. We can employ static or dynamic methods for analysis to detect vulnerabilities. Bandit is a tool for static analysis designed to identify security vulnerabilities in Python code, examining a defined range of issues. This study introduces an additional collection of vulnerabilities, specifically the top 25 CWE, to enhance the tool's detection capabilities. The approach involves analyzing Python code and constructing an Abstract Syntax Tree (AST) using the AST library in Python.

By traversing the nodes of the tree and gathering information regarding the code's characteristics, potential vulnerabilities are identified based on predefined checks for each scenario. The tool's capability for predicting all the incorporated scenarios was demonstrated after the completion of the tests added to it.

Keywords: python code analysis, software security, static analysis, vulnerability detection.

Introduction

Software development requires a critical practice of writing secure coding in order to ensure that the software is designed to be resistant to potential threats. This process requires that there be built-in security measures to prevent common problems in the coding life cycle such as unauthorized access, injection attacks, etc.(Nembhard et al., 2019).

Software defects that are targeted and exploited in security attacks are called security vulnerabilities(Kiran et al., 2021). Security vulnerabilities have significant impacts on millions of consumers and threaten computer systems to operate securely(G. Lin et al., 2020). Undetected vulnerabilities can be exploited by hackers and cause significant harm to users(Fan et al., 2020).

Software analysis includes dynamic analysis and static analysis. Possible programming errors in the code, as well as security weaknesses, are discovered through static analysis if the process is completed without the need to execute the code(da Costa et al., 2022) (Nachtigall & Bodden, 2019)

Programming languages such as C and C++ are pivotal in the static analysis process because their type systems provide additional details to the analyst because they are static type systems, unlike dynamic type systems such as Python, where there is an increasing demand for them(Gulabovska & Porkolab, 2019).

There are many analysis tools currently available that can statically analyze code written in Python(Kiska, 2021). The current analysis tools that are widely used in static analysis of the Python programming language are pyflakes, mypy, pylint and others(da Costa et al., 2022). Another important tool used to find vulnerabilities in Python code is the bandit tool(Guo et al., 2021). CWE, which is a classification and organization system for common vulnerabilities, helps developers identify software vulnerabilities by assigning a single CWE identifier to each vulnerability(C. Lin et al., 2023).

It has been observed that static analysis tools work to find vulnerabilities in software written in the Python programming language, as some of these tool's focus on code styling issues and do not focus on security issues, and others focus on security issues, but they lack coverage of many common vulnerabilities. For example, the bandit tool does not detect many of the top 25 CWEs. The goal of the research is to add the top 25 common vulnerabilities (CWEs) to the bandit tool. After adding custom checks, the tool now covers a wider range of common vulnerabilities and is able to detect the top 25 CWEs.

The remainder of this paper is structured as follows: Section 2 covers related works. Section 3 demonstrates top 25 CWE and python. Section 4 describes the methodology. Section 5 includes results and discussion. Section 6 provides the conclusion.

Related Works

Static code analysis is a method used when we want to evaluate the source code without having to execute the programs. The purpose of the analysis is to discover errors, issues related to code quality, and security vulnerabilities. It is of great benefit in evaluating the efficiency of programs and indicating the aspects in which programmers need to strengthen their skills to improve their programs(Souza, 2020).

(Ziems, 2021) improved the detection of security vulnerabilities in software by applying advanced deep learning models, particularly transformer-based models such as BERT. The research treats source code as text and models the detection process as a natural language processing (NLP) problem. By leveraging these NLP techniques, the researchers aim to detect vulnerabilities more effectively and efficiently than traditional methods like static and dynamic code analysis, which are often inaccurate and inefficient. The study also explores the use of transfer learning from written English to source code, highlighting its effectiveness in classifying security vulnerabilities in C/C++ code.

(Duan et al., 2019) developed an advanced system for detecting fine-grained software vulnerabilities in code with very slight differences between vulnerable and non-vulnerable versions. The study aims to improve the accuracy of vulnerability detection by introducing VulSniper, a model that uses an attention neural network to focus on critical features in the code. This approach allows VulSniper to effectively capture subtle distinctions in code that may lead to vulnerabilities, such as minor changes in conditions that could cause buffer overflows or resource management errors. The ultimate objective is to surpass the limitations of traditional static analysis methods, which often struggle with false positives and false negatives, by utilizing attention mechanisms and deep learning to enhance vulnerability detection accuracy, particularly for fine-grained issues.

(Kronjee et al., 2018) combined data-flow analysis techniques with machine learning to create a static analysis method for detecting software vulnerabilities, particularly SQL injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities in PHP applications. The study aims to improve the accuracy and efficiency of detecting these vulnerabilities by using control-flow graphs (CFGs) to extract features from code samples, which are then used to train various probabilistic machine learning classifiers. By leveraging both data-flow analysis and machine learning, the research seeks to address the limitations of existing static analysis tools and enhance the detection of vulnerabilities in real-world and open-source software applications.

(Cao et al., 2020) developed a deep learning-based method for detecting software vulnerabilities more efficiently and accurately. The researchers introduce a hybrid model that combines a convolutional neural network (CNN) with a bidirectional long short-term memory (Bi-LSTM) network, and apply a discrete Fourier transform (DFT) to convert source code into the frequency domain. This approach helps in capturing significant patterns in the code to better detect vulnerabilities. The key objective is to improve the detection of common software vulnerabilities, such as buffer errors and resource management errors, by focusing on both local and global features of the code, while also addressing challenges related to feature extraction and the diverse nature of vulnerabilities in modern software systems.

(Mahyari, 2022) designed a deep learning-based method to detect software vulnerabilities in source code. Specifically, the research focuses on a hierarchical approach that first identifies whether a piece of source code is vulnerable and then pinpoints the exact lines of code responsible for the vulnerability. By using techniques inspired by natural language processing (NLP) and representing source code as binary vectors, the model leverages a bidirectional LSTM to capture dependencies between lines of code. The key objective is to improve accuracy and reduce false alarms in detecting vulnerable lines of code, which is crucial for mitigating potential cyberattacks.

(Piran, 2022) investigated the existence of security vulnerabilities in software that shares similar code, such as cloned or near-duplicate code fragments. The study aims to empirically analyze vulnerabilities in C/C++ projects to determine whether the same security issues occur across applications that reuse or share similar code or business logic. By examining a dataset of 315 open-source projects, the researchers aim to identify common security flaws and assess the prevalence of vulnerabilities in similar code structures. Ultimately, the research seeks to provide insights for improving automated vulnerability detection tools by tailoring them to specific classes of vulnerabilities frequently found in cloned or similar code fragments.

(Alsamel, 2023) proposed an automated tool that helps security engineers and developers classify and label software vulnerability reports with appropriate Common Weakness Enumeration (CWE) tags. The tool, called Vulnerability Report Tagger (VrT), leverages machine learning algorithms, specifically the FastText classifier, to automatically assign vulnerability types based on the descriptions found in the National Vulnerability Database (NVD). The purpose of VrT is to reduce the manual effort and potential errors involved in tagging vulnerabilities, streamline the vulnerability management process, and improve the prioritization of security fixes. This research aims to enhance the efficiency and accuracy of handling cybersecurity threats, making it easier for security professionals to manage and respond to new vulnerabilities.

(Sun & Wang, 2023) improved the process of analyzing and understanding vulnerabilities by using a knowledge graph-based system that links various vulnerability databases and infers hidden relationships between vulnerabilities. The research focuses on building a vulnerability knowledge graph from CVE, CWE, CAPEC, and other data sources to enhance vulnerability correlation and prediction. This knowledge graph, combined with chain reasoning, allows the researchers to detect complex, compound vulnerabilities and uncover hidden links between software products and their associated vulnerabilities. The main objective is to improve the accuracy and efficiency of vulnerability scanning and provide better insights for cybersecurity management.

(Kang & Son, 2022) developed an advanced static analysis framework, named Tracer, for detecting recurring software vulnerabilities by focusing on both syntactic and semantic similarities in the code. The research aims to address the limitations of existing vulnerability detection methods, which primarily focus on syntactic similarities and fail to detect vulnerabilities that recur with different syntactic structures but share the same underlying behavior. Tracer uses taint analysis and inter procedural data dependency to identify vulnerable code patterns and create vulnerability signatures, allowing it to detect both known and semantically similar, recurring vulnerabilities in new programs. The main objective is to improve the accuracy, robustness, and scalability of vulnerability detection in large-scale software projects.

(Id & Wang, 2024) suggested a method that improves the detection and prediction of software vulnerabilities by utilizing an enhanced information gain (IG) algorithm within a deep neural network (DNN) framework. The study aims to address the challenge of incomplete vulnerability data and improve both the accuracy and speed of vulnerability detection. By using techniques such as

Dropout to prevent overfitting, the model enhances its ability to extract and predict vulnerabilities effectively.

Below is a set of analysis tools for Python code, along with an explanation of some of the features and capabilities of these tools(Kiska, 2021):

1- Pylint: It detects errors, checks code smells, enforces coding standards, and provide refactoring. In addition, pylint calculates the complexity of the code and provides details of potential problems through a report.
2- Pyflakes: This tool focuses its work on finding coding errors and its performance is faster than other tools because it does not impose a style check, thus avoiding many false positives.
3- DeepSource: The tool provides code reviews and produces results with low false positive rates. It has an automatic repair feature, so it fixes problems automatically. The tool identifies issues such as unnecessary code, which leads to improved performance.
4- SonarQube: This tool focuses on code quality and detects code smells, duplicate copying, and security vulnerabilities. The tool provides detailed reports on issues such as duplicate blocks and function complexity because it has a large set of rules for analyzing Python code.
5- Bandit: It is a tool designed to find security vulnerabilities in code written in Python. It scans the code to find common vulnerabilities such as handling input insecurely, using weak encryption algorithms, etc. bandit has great adaptability because it accepts new checks to be added to it to meet the security requirements of developers.
6- DeepCode: This tool uses machine learning in the analysis process. It identifies issues related to deprecated modules, unprotected calls, and the use of unsafe parsing processes.

Table 1 shows comparisons between these tools:

TABLE 1. *Comparison for Static Code Analysis Tools*

Tool	Focus	False Positives	Performance	Style Checks	Unique Features
Bandit	Security vulnerabilities	Low	Moderate	No	Security-focused, plugin-based
Pylint	Code style, complexity, and errors	Moderate	Slow (high complexity)	Yes (PEP8 standards)	Customizable limits (complexity and standards)
Pyflakes	Coding errors	Low	Fast	No	Fast error detection
DeepSource	Code review, performance improvement, autofix	Very Low	Moderate	Yes (Autopep8, Black)	Autofix feature, integration by CI/CD
SonarQube	Code smells, technical debt, security vulnerabilities	Moderate (depending on rules enabled)	Slow (depending on rules enabled)	Yes (by all rules enabled)	Comprehensive code quality metrics
DeepCode	Real-time feedback by AI-driven analysis	Low	Fast	No	AI-based analysis, real-time feedback

Top 25 Cwe and Python

A list that is compiled annually, the Top 25 CWE vulnerabilities highlights the software security flaws that are the most prevalent and potentially most serious. As a result of these vulnerabilities, which are weaknesses in the design or implementation of software, attackers have the ability to take advantage of them in order to cause damage to systems or obtain improper access to data. For the purpose of assisting in the prioritization of security efforts and the development of methods for defense against attacks, the list is critically important for software developers and cybersecurity professionals(Wunder et al., 2024).

As a result of the fact that they are the most widespread and severe security flaws in software that may be exploited by attackers, the Top 25 CWE vulnerabilities are extremely important. These vulnerabilities frequently result in major implications, such as the shutting down of the system, the unauthorized access of data, or the complete control over the system. Because they are simple to locate and exploit, the vulnerabilities are serious. Furthermore, they affect a wide variety of systems, which makes them frequent targets for attackers. The address of these vulnerabilities is critical for the maintenance of software security and the prevention of exploits and breaches that are significant(Fu, 2022).

The CWEs particularly pertinent to Python comprise the following:

1. CWE-787: Buffer Overflow – An interaction with an external library written in C or another lower-level language may continue to present buffer overflow vulnerabilities, despite the fact that Python's dynamic memory management assists in minimizing the danger of buffer overflows. Memory corruption and the likelihood of unauthorized code execution can arise as a result of buffer overflows, which occur when the memory capacity of the buffer is exceeded(Shahab et al., 2020).
2. CWE-79: Cross-Site Scripting (XSS) – Web applications written in Python that do not effectively sanitize user inputs are susceptible to cross-site scripting (XSS) attacks. These attacks involve the injection of malicious scripts within web pages that are seen by other users. This could result in the theft of data or the hijacking of a session(Note, 2012).

3. CWE-20: Improper Input Validation – Is a widespread vulnerability in Python applications which do not perform an adequate check on the inputs provided by users. This vulnerability can result in a variety of potential exploits, such as injection attacks(Bojanova et al., 2020).

For a number of different reasons, the dynamic typing, dependency on external libraries, and flexibility, the Python has substantial challenges when it comes to detecting vulnerabilities:

1. Dynamic Typing: Within the Python programming language, the types of variables are not clearly specified. This flexibility might result in runtime mistakes and security vulnerabilities which are not discovered until the program is executed. The shortage of static type checking causes it difficult to detect issues such as type mismatches or accidental type changes at an early stage. This, in turn, increases the likelihood that bugs and vulnerabilities will go unnoticed until they produce a failure or a security concern(Chen et al., 2020).
2. Flexibility: The permissive syntax of Python makes it possible to construct software quickly, but it also makes it simple to add security flaws. This is especially true when combined with complicated dynamic features such as dynamic attribute allocation or introspection. Due to this flexibility, automated vulnerability detection is made more difficult, and static analysis is rendered less effective(Monat & Mine, 2020).
3. Reliance on External Libraries: An additional layer of security risk is introduced by the extensive environment of third-party libraries that Python provides. It is difficult to detect vulnerabilities within these libraries because of the different codebases and the difficulty of managing dependencies.

Vulnerabilities from these libraries might spread to programs that are dependent on them(Shuanghe et al., 2019).

Methodology

Design Overview

In this work, a system was built, which is an expansion of the bandit tool, where checks were added to perform a test for common weaknesses (top 25 CWEs). The python code is received into the system and it analyzes the code, finds the weaknesses in it, and indicates the degree of severity and confidence for them.

Rule Set Tailoring for Top 25 CWE

For the purpose of ensuring that possible software vulnerabilities are recognized in an accurate and efficient manner, a set of processes was followed in order to build rules for evaluating code and finding software vulnerabilities. In order to develop a testing rule, the following fundamental procedures were carried out:

Understanding the Intended Security Vulnerability

To begin the process of developing a testing rule, the first thing you need to do is recognize and comprehend the vulnerability that you want to find. This was accomplished using the following studying:

1. The nature and type of the vulnerability (such as SQL Injection, path traversal, or Buffer Overflow).
2. How the vulnerability happens (for instance, the user entering data into the database without taking appropriate precautions).
3. Consequences for exploiting the vulnerability include (unauthorized command execution, information sharing, etc.).

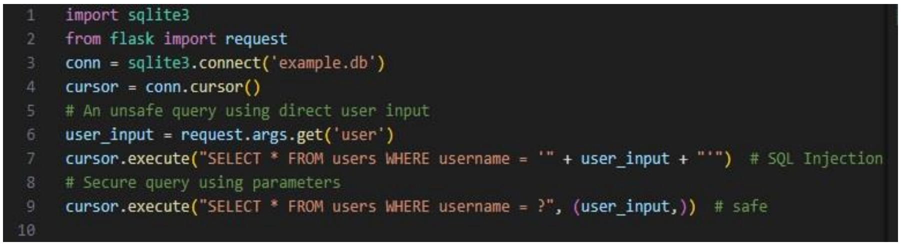
Identifying Code Patterns

Upon identifying the vulnerability, the code patterns that might cause it were determined. This stage involved identifying the structures or behaviors analyzed in the code. for instance:

- The ast library was utilized to perform an analysis of the source code and search for common patterns that cause SQL Injection vulnerabilities. This was done in order to identify instances of SQL Injection that occurred during the process of code analysis. Because untrusted input is injected inside SQL queries without adequate sanitization, these vulnerabilities occur. As a result, attackers are able to execute SQL queries that are not authorized. The following is a list of the discovery steps:

1. Identifying untrusted inputs: Look for places in the code that take data from the user, like request.GET, request.POST, along with additional sources that can't be trusted.
2. Search for SQL queries: It is important to check for SQL queries that were produced using string concatenation rather than by having parameterized queries.
3. Verify proper sanitization: In the process of building SQL queries, it is imperative to make certain that potentially dangerous functions like .format(), +, or % aren't used with user input.

For instance, if the next code snippet is inputted into the scanning tool:



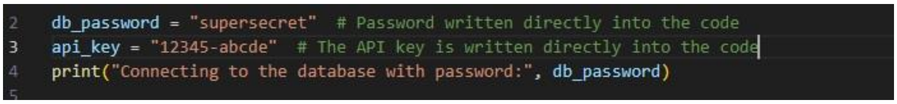
```
import sqlite3
from flask import request
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
# An unsafe query using direct user input
user_input = request.args.get('user')
cursor.execute("SELECT * FROM users WHERE username = '" + user_input + "'")  # SQL Injection
# Secure query using parameters
cursor.execute("SELECT * FROM users WHERE username = ?", (user_input,))  # safe

```

The result will indicate the potential presence of a SQL Injection vulnerability in the code.

- Source code analysis was employed to identify instances of hard-coded credentials by searching for patterns indicative of the direct inclusion of credentials, such as passwords, user names, and secret keys, within the code. This vulnerability is risky since it leads the system to exploitation if the attacker acquires access to the code. The steps for discovery are as follows:

1. Determine potential locations for credential utilization: Identify variables that include terms like password, api_key, etc.
2. Identify definitions of variables that explicitly include written credentials: Verify if these variables have been explicitly set in the code.
3. Guarantee the lack of sterilization or suitable protection: Investigate the lack of any mechanism for data sterilization or secure downloading.

For instance, if the next code snippet is inputted into the scanning tool:



```
db_password = "supersecret"  # Password written directly into the code
api_key = "12345-abcde"  # The API key is written directly into the code
print("Connecting to the database with password:", db_password)
```

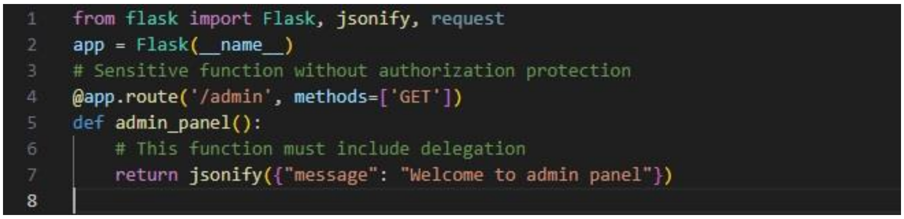
This will be demonstrated by the result, which will show that credentials are inserted right into the code.

- Source code analysis was employed to identify instances of Missing Authorization by looking for patterns that signify the lack of requisite authorization. This vulnerability arises when the system permits users to execute activities or obtain information without verifying their requisite permissions. The steps for discovery are as follows:

1. Determine points of protection that based upon authorization: Identify areas in the code that require protecting, such as accessing confidential information or executing critical operations.

2. Search for authorization checks: Examine for the existence of authorization functions, including check_permissions, is_authorized, as well as other tailored authorization decorators.
3. Recognizing the lack of authorization: Identify instances where functions are executed or sensitive data is obtained without authorization verification.

For example, if the following code snippet was entered into the tool for testing:



```
from flask import Flask, jsonify, request
app = Flask(__name__)
# Sensitive function without authorization protection
@app.route('/admin', methods=['GET'])
def admin_panel():
    # This function must include delegation
    return jsonify({"message": "Welcome to admin panel"})

```

On the basis of the result, it would be clear that authorization was absent.

The dynamic typing of Python and its dependence on external libraries present numerous challenges that complicate vulnerability detection:

- Dynamic Typing: Python does not require type declarations, allowing variables to alter their types during runtime. This complicates static analysis, as type-related defects or vulnerabilities, like type mismatches, might only become apparent during execution. The absence of static type checking may result in unnoticed faults until execution, hence elevating the likelihood of runtime mistakes or security vulnerabilities.
- Reliance on External Libraries: There is a major risk involved because the Python ecosystem is primarily dependent on libraries that are provided by third parties. The management of dependencies is difficult, and libraries have the potential to expose vulnerabilities. As a result of the enormous number of dependencies, it is difficult to verify that all of the libraries that

are being used are safe and free from bugs. This is because potential security flaws can be concealed within external packages.

Integration with Bandit

The architecture needs to take into consideration a number of different things in order to successfully incorporate CWE rules with the Bandit tool for identifying security vulnerabilities in Python. The following is an in-depth explanation to the architecture, with particular focus on the alterations that are required:

1. Modifications to AST Traversal

   Adding Hooks: In order for Bandit to be able to activate CWE-specific rules, its AST traversal scheme will need to be modified. Hooks are used to map CWE patterns with particular AST nodes. For example, hooks can be used to identify vulnerable functions such as eval() or exec(), which could be potential indicators of a vulnerability.

2. Construction of Plugins for the CWE Rules

- The CWE Rules as Individual Plugins: A different plugin will be used to implement each CWE rule. These plugins will check the source code for security flaws associated with that particular CWE.
- Configurable Rules: Users must have the capability to either enable or disable particular CWE rules via configuration files to tailor the scanning procedure.
- Severity Levels: Every CWE rule plugin has to contain a severity rating (high, medium, low) determined by the potential effect of the vulnerability.

3. API Modifications

   CWE Rule Configuration: The Bandit API requires an upgrade to facilitate the activation and configuration of CWE rules.

4. Custom Hooks

   Support for Custom Plugins: It should be possible for developers to construct and register their own custom CWE rules using the API. This is something that can be accomplished by extending the design of Bandit's plugins, which will allow users to produce their own security tests.

5. Improved Reports

- CWE-Specific Reports: It is important for reports to include CWE identifiers and descriptions for each vulnerability that has been discovered. Additionally, reports should include links to CWE documentation in order to offer developers with additional context.
- Incorporation with Vulnerability Databases: Bandit should make it possible to link vulnerabilities that have been discovered to external databases, such as the CWE database maintained by MITRE, in order to provide more detailed remedial assistance.

A custom plugin is created in order to work on discovering a specific case of CWEs that was not originally present in bandit. This is done by understanding the vulnerability and the reasons that lead to it. The custom plugin is saved in a Python file with a specific name that indicates the scan it is performing, and it is called through the main file (main_code.py) when there is an analysis process to discover vulnerabilities in a specific code.

Dataset for Testing and Verification

For the purposes of testing and Verification, the datasets were selected using open-source Python projects and known vulnerable codebases (for example, vulnerable web applications and GitHub repositories).

Evaluation Criteria

- After custom plugins were added to the analysis tool to detect cases of the top 25 CWEs, the detection range of the tool expanded and it is able to cover all cases of the top 25 CWEs.
- The tailored tool has good capabilities and outperforms other analysis tools such as Pylint, Flake8 and others in discovering vulnerabilities, especially security issues.

Results and Discussion

The findings in table 2 demonstrate a tool's efficacy for determining the Top 25 CWE vulnerabilities:

TABLE 2. *The results.*

CWE ID	**CWE Name**	**TP**	**FP**	**TN**	**FN**	**Accuracy**	**Precision**	**Recall**	**F1 Score**
CWE-79	Cross-Site Scripting (XSS)	5	1	4	0	0.90	0.83	1	0.90
CWE-89	SQL Injection	5	2	3	0	0.80	0.71	1	0.83
CWE-22	Path Traversal	4	1	4	1	0.80	0.8	0.8	0.80
CWE-352	Cross-Site Request Forgery	5	1	4	0	0.90	0.83	1	0.90
CWE-20	Improper Input Validation	4	0	5	1	0.90	1	0.8	0.88

where:

1. CWE ID: The distinct identifier given to every vulnerability within the Common Weakness Enumeration.
2. CWE Name: The name or brief description of the vulnerability.
3. True Positives (TP): The number of accurately recognized vulnerabilities.
4. False Positives (FP): The number of vulnerabilities that were incorrectly identified.
5. True Negatives (TN): the number of samples with no vulnerabilities undetected.
6. False Negatives (FN): The number of vulnerabilities which were not found by the tool but are present in the system.
7. Accuracy: It is the ratio between the number of correct predictions over the total number of predictions.
8. Precision: The precision with which the tool finds each vulnerability.
9. Recall: The recall or sensitivity of the tool for each vulnerability.
10. F1 Score: Taking the harmonic mean for recall and precision, we can see how well the tool handled each vulnerability as a whole.

Here is how the metrics were computed(C. Lin et al., 2023):

- Precision is computed as: Precision = TP / (TP + FP)
- Recall is computed as: Recall = TP / (TP + FN)
- Accuracy is computed as: (TP + TN) / (TP + FP + TN + FN)
- F1 Score is computed as: F1 Score = 2 * (Precision * Recall) / (Precision + Recall)

Bandit's strengths for discovering vulnerabilities involve the following:

1. Effective automated analysis: It discovers Python vulnerabilities, including input injections and insecure functions and others.
2. Python-focused: It finds security flaws that are unique to this language in an accurate manner.

3. Comprehensive scanning: A large variety of common security flaws are addressed.
4. Customizable checks: Customizing checks to match the unique requirements of each project is made possible.
5. CI/CD integration: The integration of this tool into development pipelines is simple, and it guarantees continuous security.

Considering all of these advantages, Bandit is a highly efficient tool for improving code security.

The enhanced capability of Bandit to discover CWE vulnerabilities which are generally challenging to detect in Python consist of multiple aspects:

- Analysis of external libraries: Bandit can find dangerous uses of external libraries, and that is hard to do by hand.
- Identification of dealing with files vulnerabilities: It detects flaws in file management, including unsafe path handling, and this may result in vulnerabilities that allow Path Traversal attacks.
- Identification of input injection vulnerabilities: Bandit detects regions sensitive for inputting injection vulnerabilities, which might be challenging to identify without meticulous code examination.
- Identification of risky function usage: It identifies the utilization of risky functions or methods, such as eval, which may be vulnerable to exploitation when untrusted inputs are provided.
- Management of environmental variables and sensitive keys: Bandit is able to identify instances of sensitive variables or secret keys being leaked in the code, which is an issue that is frequently missed during the development process.

Because of these features, Bandit is a powerful tool for detecting vulnerabilities that are difficult to identify, especially on projects that are large or complex.

Utilizing the Bandit tool to identify vulnerabilities associated with the CWE Top 25 may present challenges in mitigating false negatives or false positives. These instances include:

1. Limited contextual analysis: Bandit employs static analysis, which may result in an incomplete contextual awareness of variable or function utilization during runtime. This may lead to false positives, like identifying a non-existent threat due to inadequate comprehension of dynamic data flow.
2. Logical vulnerabilities: Business logic issues, including improper permission or logical validation mistakes, are challenging to identify with tools like Bandit, as they necessitate an in-depth knowledge of an application's context and objectives. This could result in false negatives.
3. Highly strict vulnerability detection: Bandit may exhibit excessive sensitivity to particular vulnerabilities, such as the utilization of functions like eval or exec, which could lead to false positives whenever these functions are employed correctly and securely in specific scenarios.
4. Resource and memory management: Resource management vulnerabilities, such as memory or file leaks may be complicated and hard to identify using static analysis, resulting in false negatives.
5. Use of external libraries: Bandit may encounter difficulties in detecting vulnerabilities within external libraries or those arising from interactions between numerous libraries, particularly when the source code of these libraries is inaccessible for examination, resulting in false negatives.
6. Complex input validation: If Bandit uses bespoke techniques for input validation, it is possible that it will overlook vulnerabilities that need deep input validation. Some instances of these vulnerabilities include SQL injection and cross-site scripting. It is

possible that this could result in false negatives or false positives when the validation for the input is not made explicitly visible. Future enhancements will involve the incorporation of extra CWEs and the development of future Python versions.

Conclusion

After work was done on adding a set of tests that detect cases of the top 25 CWEs to the bandit tool, which is one of the analysis tools that performs static analysis on codes written in the Python language and which focuses when analyzing security issues in the code, the tool's ability to detect vulnerabilities has become larger and able to cover all common vulnerabilities top 25 CWEs. The tool will be developed in future work to also include dynamic analysis of the code in order to give us more accurate results when testing to detect vulnerabilities.

Acknowledgments

The authors are grateful to Computer Science Department at the University of Mosul/ Iraq for the collaborative efforts that make this work achieved and to the ICT Research unit at Computer Center of University of Mosul/ Iraq, for the support during the course of this work.

References

Alsamel, Y. A. (2023). *VrT : A CWE-Based Vulnerability Report Tagger*. *2*, 2–5.

Bojanova, I., Galhardo, C. E., & Moshtari, S. (2020). *Input / Output Check Bugs Taxonomy : Injection Errors in Spotlight*. 1–10.

Cao, D., Huang, J., Zhang, X., & Liu, X. (2020). FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection. *Proceedings - 2020 IEEE 19th International Conference on Trust, Security and Privacy in*

Computing and Communications, TrustCom 2020, December 2020, 539–546. https://doi.org/10.1109/TrustCom50675.2020.00078

Chen, Z., Chen, B., & Chen, L. (2020). *An Empirical Study on Dynamic Typing Related Practices in Python Systems*. 83–93.

da Costa, F. H., Medeiros, I., Menezes, T., da Silva, J. V., da Silva, I. L., Bonifácio, R., Narasimhan, K., & Ribeiro, M. (2022). Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification. *Journal of Systems and Software*, *183*. https://doi.org/10.1016/j.jss.2021.111092

Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., & Wu, Y. (2019). Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. *IJCAI International Joint Conference on Artificial Intelligence*, *2019-Augus*, 4665–4671. https://doi.org/10.24963/ijcai.2019/648

Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020). A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 508–512. https://doi.org/10.1145/3379597.3387501

Fu, M. (2022). LineVul : A Transformer-based Line-Level Vulnerability Prediction LineVul : A Transformer-based Line-Level Vulnerability Prediction. In *19th International Conference on Mining Software Repositories (MSR '22), May 23â•fi24, 2022, Pittsburgh, PA, USA* (Vol. 1, Issue 1). Association for Computing Machinery. https://doi.org/10.1145/3524842.3528452

Gulabovska, H., & Porkolab, Z. (2019). Survey on static analysis tools of python programs. *CEUR Workshop Proceedings*, *2508*(September), 22–25.

Guo, W., Huang, C., Niu, W., & Fang, Y. (2021). Intelligent mining vulnerabilities in python code snippets. *Journal of Intelligent and Fuzzy Systems*, *41*(2), 3615–3628. https://doi.org/10.3233/JIFS-211011

Id, P. Y., & Wang, X. (2024). *Vulnerability extraction and prediction method based on improved information gain algorithm*. 1–23. https://doi.org/10.1371/journal.pone.0309809

Kang, W., & Son, B. (2022). Tracer : Signature-based Static Analysis for Detecting Recurring Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7â•fi11, 2022, Los Angeles, CA, USA* (Vol. 1, Issue 1). Association for Computing Machinery. https://doi.org/10.1145/3548606.3560664

Kiran, S. R. A., Rajper, S., & Shaikh, R. A. (2021). Categorization of CVE Based on Vulnerability Software By Using Machine Learning Techniques. *International Journal of Advanced Trends in Computer Science and Engineering*, *10*(3), 2637–2644. https://doi.org/10.30534/ijatcse/2021/1581032021

Kiska, J. (2021). *Static analysis of Python code*. 1–48.

Kronjee, J., Hommersom, A., & Vranken, H. (2018). Discovering software vulnerabilities using data-flow analysis and machine learning. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3230833.3230856

Lin, C., Xu, Y., Fang, Y., & Liu, Z. (2023). *applied sciences VulEye : A Novel Graph Neural Network Vulnerability Detection Approach for PHP Application*.

Lin, G., Wen, S., Han, Q. L., Zhang, J., & Xiang, Y. (2020). Software Vulnerability Detection Using Deep Neural

Networks: A Survey. *Proceedings of the IEEE*, *108*(10), 1825–1848. https://doi.org/10.1109/JPROC.2020.2993293

Mahyari, A. (2022). *A Hierarchical Deep Neural Network for Detecting Lines of Codes with Vulnerabilities*. *Qrs*.

Monat, R., & Mine, A. (2020). *Static Type Analysis by Abstract Interpretation of Python Programs*.

Nachtigall, M., & Bodden, E. (2019). *Explaining Static Analysis for Software Security – A Perspective*.

Nembhard, F. D., Carvalho, M. M., & Eskridge, T. C. (2019). Towards the application of recommender systems to secure coding. *Eurasip Journal on Information Security*, *2019*(1). https://doi.org/10.1186/s13635-019-0092-4

Note, T. (2012). *Available Online at www.jgrcs.info DEFENDING AGAINST WEB VULNERABILITIES AND CROSS-SITE SCRIPTING*. *3*(5), 61–64.

Piran, A. (2022). *Vulnerability Analysis of Similar Code*. *December 2021*. https://doi.org/10.1109/QRS54544.2021.00076

Shahab, A., Engineers, E., Alenezi, M., Technology, T. S., & Nadeem, M. (2020). *An automated approach to fix buffer overflows*. *February*, 3778–3788. https://doi.org/10.11591/ijece.v10i4.pp3778-3788

Shuanghe, P., Peiyao, L. I. U., & Jing, H. A. N. (2019). *A Python Security Analysis Framework in Integrity Verification and Vulnerability Detection*. *24*(2), 141–148.

Souza, L. De. (2020). *A Proposal for Source Code Assessment Through Static Analysis*.

Sun, X., & Wang, Z. (2023). *Intelligent Association of CVE Vulnerabilities Based on Chain Reasoning*. 28–34. https://doi.org/10.3233/FAIA230788

Wunder, J., Kurtz, A., Eichenmüller, C., Gassmann, F., &

Benenson, Z. (2024). *Shedding Light on CVSS Scoring Inconsistencies: A User-Centric Study on Evaluating Widespread Security Vulnerabilities*.

Ziems, N. (2021). *Security Vulnerability Detection Using Deep Learning Natural Language Processing*.