PROJECT ON:

Dependency Security Auditor: Vulnerability Detection in Software Supply Chains

By:
Rathod Vishal

Table of content

ABSTRACT

The increasing reliance on open-source libraries in modern software development has introduced significant risks related to third-party dependency vulnerabilities. This project, Dependency Security Auditor, aims to provide a Python-based tool that scans software package dependencies for known security vulnerabilities by leveraging the Open Source Vulnerabilities (OSV) database. The tool analyzes a given project's requirements.txt file to identify and audit packages, then fetches security alerts for vulnerable versions using an API-based approach. It generates a comprehensive report that highlights affected packages, associated CVEs, and their severity. The main objective is to proactively detect and report vulnerabilities before they compromise the integrity of the software system. By integrating automated auditing during development, this tool empowers developers to make informed decisions about package updates and risk management. The tool features a simple yet effective GUI built with Tkinter, making it user-friendly even for those with minimal technical expertise. This project emphasizes functional requirements like dependency parsing, API integration, and vulnerability mapping, as well as non-functional aspects such as usability, performance, and accuracy. Testing procedures including unit, integration, and functional testing ensure system reliability and performance. The implementation of this tool supports a proactive security posture in the software supply chain and aligns with modern DevSecOps practices. The system is technically feasible, economically viable, and operationally efficient for small to medium-scale development teams. In conclusion, this mini project demonstrates a practical approach to software supply chain security by helping developers detect dependency vulnerabilities early, thus enhancing overall software resilience.

INTRODUCTION

In modern software development, developers often use third-party packages and open-source libraries to make their work easier and faster. These packages save a lot of time and help developers focus on important parts of the project without writing everything from scratch. However, using these external packages comes with a serious risk. Some packages may have security problems or bugs that attackers can use to harm the system. These problems are called vulnerabilities. If a developer is not aware of them, these issues may cause data loss, hacking, or failure of the entire application. Many times, developers forget to check the security of these packages or don't have enough tools to do it manually. This becomes a big problem, especially in small teams or academic projects, where there may be no dedicated security expert. So, it is important to have an easy way to check if the packages used in a project are safe or not. This is where our mini project, titled "Dependency Security Auditor: Vulnerability Detection in Software Supply Chains", comes into action.

This tool is built using Python and provides a simple and easy-to-use solution. The main job of the tool is to check whether any of the Python packages used in a project have known security issues. It works by reading the list of packages and their versions from a basic text file. Then it connects to the OSV (Open Source Vulnerability) database using the internet and checks each package for vulnerabilities. If it finds any, it shows clear details like the name of the package, how dangerous the problem is, and what it is about. The tool also gives the option to save the results in a text file so that users can share or keep the report. It uses Tkinter, a built-in Python GUI library, to make the interface user-friendly. Even people with little technical knowledge can use the tool easily. By using this tool, developers can fix security problems early in the development stage, which saves time and protects the application in the long run. This project supports better and safer coding practices and helps spread awareness about software supply chain security. It is especially useful for students, hobby developers, and small teams who want a simple but effective way to secure their projects.

LITERATURE SURVEY

S.No	Title of the Paper	Authors	Techniques Used	Key Findings
1	Automated Vulnerability Detection in Software Dependencies	Ankit Kumar, Ramesh, Kumar, Priyanka Singh	Dependency Parsing Vulnerability Scanning	Automated scanning of dependencies for vulnerabilities
2	How Vulnerable Dependencies Affect Open-Source Projects	Gede Aryan, Anushman Sharma,Lav Khush Shar	Dependency File parsing,Vulneranility Lookup	Vulnerable dependencies impact project security.
3	Investigating the Reproducibility of Packages	Pranab Goswami, Sourav Gupta	Dependency File Parsing, Vulnerability Verification	Validates dependency parsing and security checks.
4	Security Vulnerability Detection in Software Dependencies Using Public Databases	Priya Sharma, Anil K. Singh	API-based Vulnerability Querying	Uses public databases to audit dependencies.
5	Enhancing Software Security through Dependency Vulnerability Auditing	Richa Gupta, Anil Kumar, Divya Sharma	Dependency File Parsing, API Vulnerability Checking	Integrates vulnerability queries for better detection.
6	An Integrated Framework for Software Vulnerability Detection, Analysis and Mitigation: An Autonomic System	M. Kumar, A. Sharma	Static & Dynamic Analysis, Autonomic Framework	Provides a complete system for vulnerability handling.
7	Discovering Vulnerable Functions: A Code Similarity Based Approach	A. Chandran, L. Jain, S. Rawat, K. Srinathan	Code Similarity Matching	Detects vulnerabilities by comparing code similarity.

EXISTING SYSTEM

In the existing system, developers often manually install and use third-party packages in their Python projects without checking whether those packages have any known security issues. Some teams rely on basic commands like pip list or pip freeze to track dependencies, but these commands don't tell whether a package has vulnerabilities.

There are tools available that can check for vulnerabilities, but most of them are complicated to set up or use. Many people don't use them regularly, and this leads to security gaps. If a developer is not aware of a vulnerable package in their project, it might allow hackers to break into the software, steal information, or damage systems.

Limitations of the Existing System

• **No automatic checking:** Developers have to check for vulnerabilities manually, which is time-consuming and not always accurate.

• **Outdated dependencies:** Packages used in the project may not be updated regularly, and no alerts are generated if they become vulnerable.

• **No clear reports:** Manual checking doesn't give proper results or reports, making it hard to understand which packages are dangerous.

• **Lack of Integration with Development Tools:** Existing systems often do not integrate with IDEs or CI/CD pipelines, making vulnerability detection disconnected from the development workflow.

• **No Severity Assessment:** Manual methods do not provide severity ratings or prioritize vulnerabilities, making it difficult for developers to know which issues need urgent attention.

PROPOSED SYSTEM

This system automatically scans all Python packages listed in a requirements.txt file to detect known security vulnerabilities. By connecting to a reliable public database like the OSV (Open Source Vulnerability) API, it ensures the scan results are always up to date with the latest threat intelligence. The tool is designed to be user-friendly, featuring a simple graphical user interface (GUI) built with Python's Tkinter. Users do not need to have advanced technical knowledge or use command-line tools to operate it. It provides clear, readable reports that specify which packages are unsafe, what the problems are, and which versions are affected—saving developers valuable time and reducing the chances of overlooking security risks.

Additionally, the system supports scanning different projects by allowing the selection of various requirements.txt files and checks the exact versions used in those projects. This is crucial, as some versions of a package may be secure while others are not. It also categorizes vulnerabilities based on severity, helping teams prioritize critical fixes. The generated reports can be saved for documentation or shared with team members to aid in collaborative debugging and security planning. By providing early warnings and automated analysis, this tool makes it easier for developers to maintain secure codebases, avoid supply chain attacks, and comply with cybersecurity best practices—all in a lightweight and accessible format ideal for students, independent developers, and small teams.

Benefits of the Proposed System

- **Easy to use:** No need to run complicated commands — just select the file and click a button.
- **Automatic scanning:** It checks all packages automatically, saving time and reducing manual work.
- **Live vulnerability data:** The tool connects to a public database to always show the latest known risks.
- **Clear reports:** It tells which packages are unsafe and why, in simple language that anyone can understand.

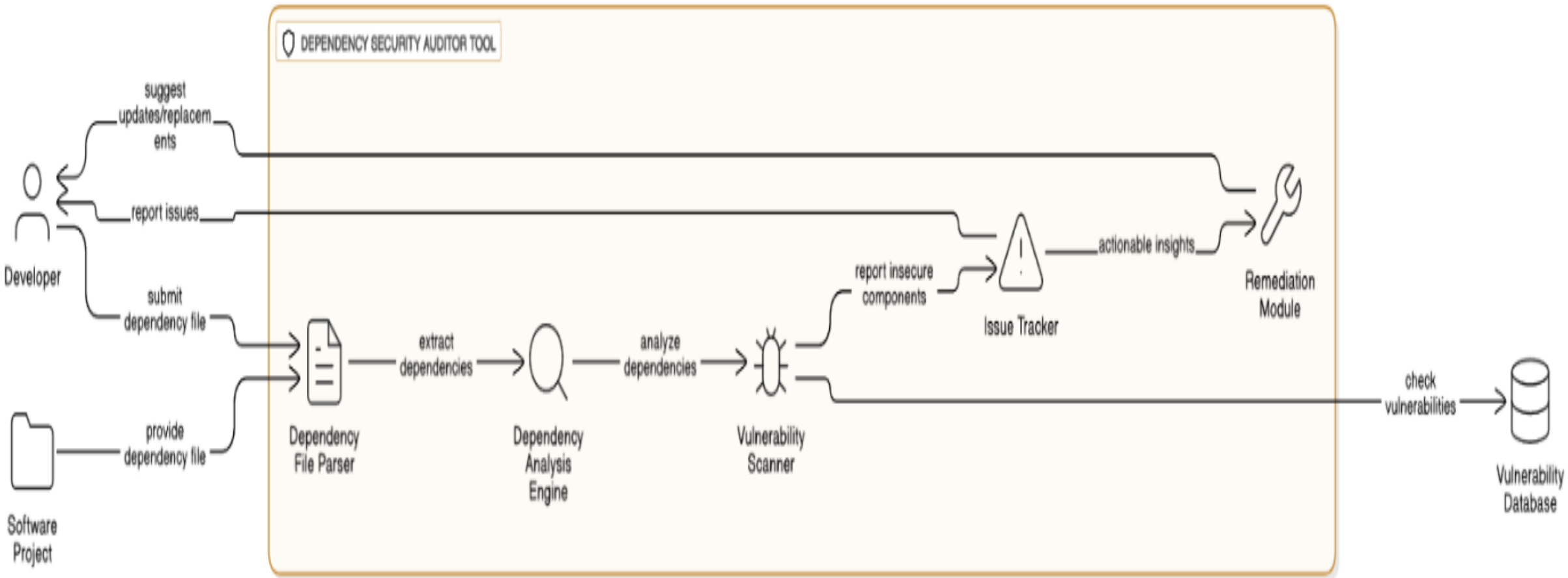
SOFTWARE REQUIREMENTS

1. **Software Environment:** Python 3.10 or higher
2. **Primary Language: Python**
3. **Frontend Framework:** Tkinter (used for the graphical user interface)
4. **Vulnerability Database:** OSV (Open Source Vulnerability) API
5. **Libraries Used:**
    - requests (for communicating with the OSV API)
    - tkinter (for building the GUI interface)
    - json (for parsing the API responses)
6. **Internet Requirement:** Required for connecting to the OSV API.
7. **Dependency File:** Text file listing Python packages with their versions.

HARDWARE REQUIREMENTS

1. **Operating System :** Windows 10/11 or Linux
2. **Processor :** Intel Core i3 or equivalent (minimum), Core i5 recommended
3. **RAM :** 4 GB minimum, 8 GB or higher recommended for better performance
4. **Hard Disk :** Minimum 500 MB of free space
5. **Display :** Basic graphics support for GUI execution with Tkinter

SYSTEM ARCHITECTURE



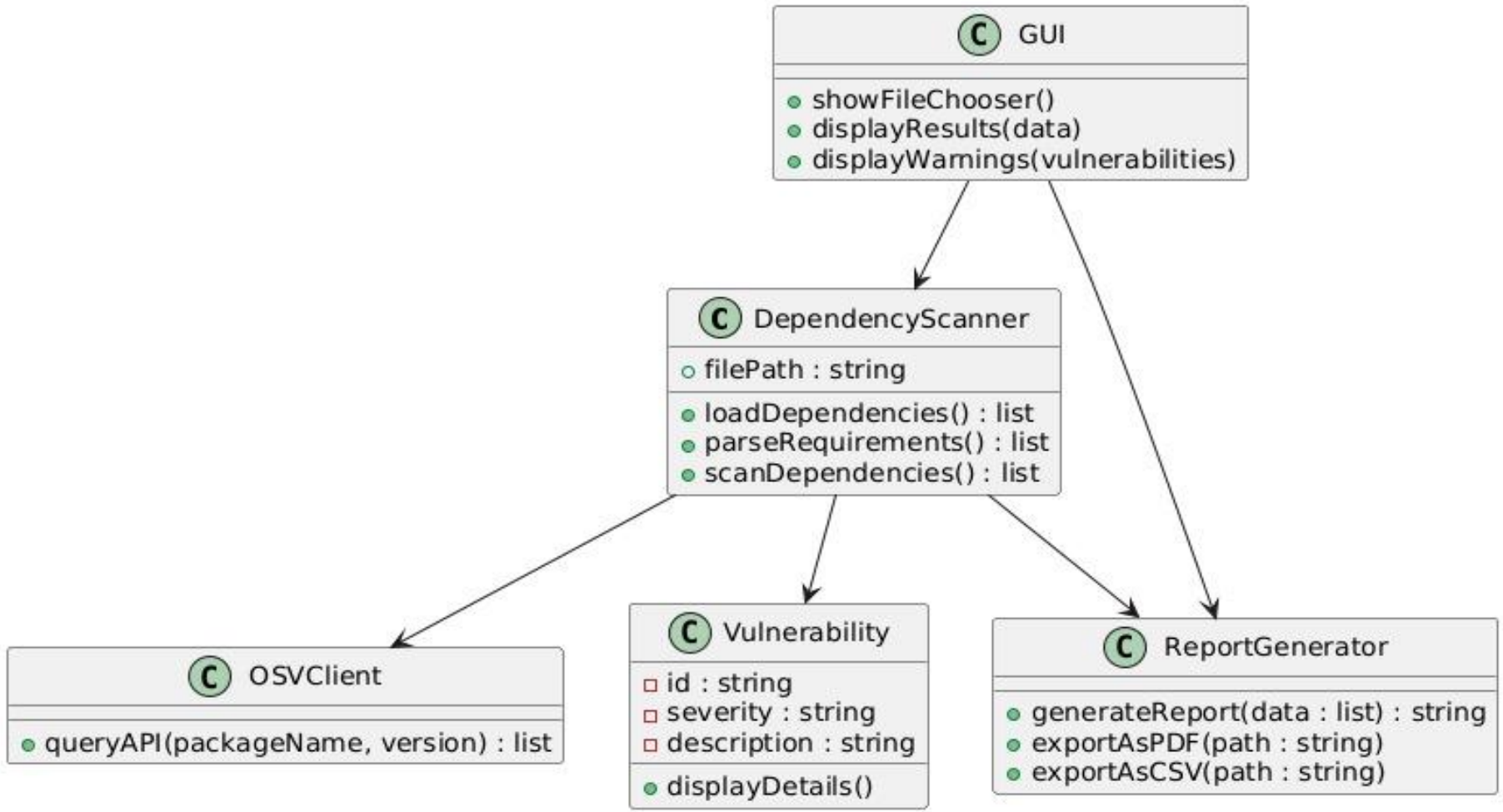
SYSTEM ARCHITECTURE

This diagram explains the working of the Dependency Security Auditor Tool, which helps developers find and fix security issues in software dependencies. In most software projects, developers use external libraries or packages to add features like database access, networking, or user interfaces. These packages, however, can sometimes have known security problems that attackers can exploit. To prevent this, the tool checks all the packages used in the project. The process begins when the developer provides a file—such as requirements.txt in Python—that contains a list of all the packages used. This file is given to the Dependency File Parser, which reads the contents and extracts the names and versions of each package. The parser converts this information into a format the tool can use. The extracted data is then passed to the Dependency Analysis Engine, which checks how the packages are used in the project and prepares them for scanning.

After the analysis is complete, the list is sent to the Vulnerability Scanner, which compares each package against a large vulnerability database. This database contains records of known security flaws in widely used software packages. If any package in the project is found to have a known issue, it is marked as risky. This information is sent to the Issue Tracker, which keeps a record of all the detected problems. The tracker then forwards the issue details to the Remediation Module, which provides suggestions to the developer for fixing the problems. For example, the tool might recommend updating to a safer version, removing the vulnerable package, or replacing it with a more secure one. Developers may also give feedback to improve the tool or report new issues. Overall, this tool helps ensure that software projects remain secure by identifying and fixing dependency-related vulnerabilities early in the development process. It saves time, reduces manual checking, and makes the software safer and more reliable for users.

UML DIAGRAMS

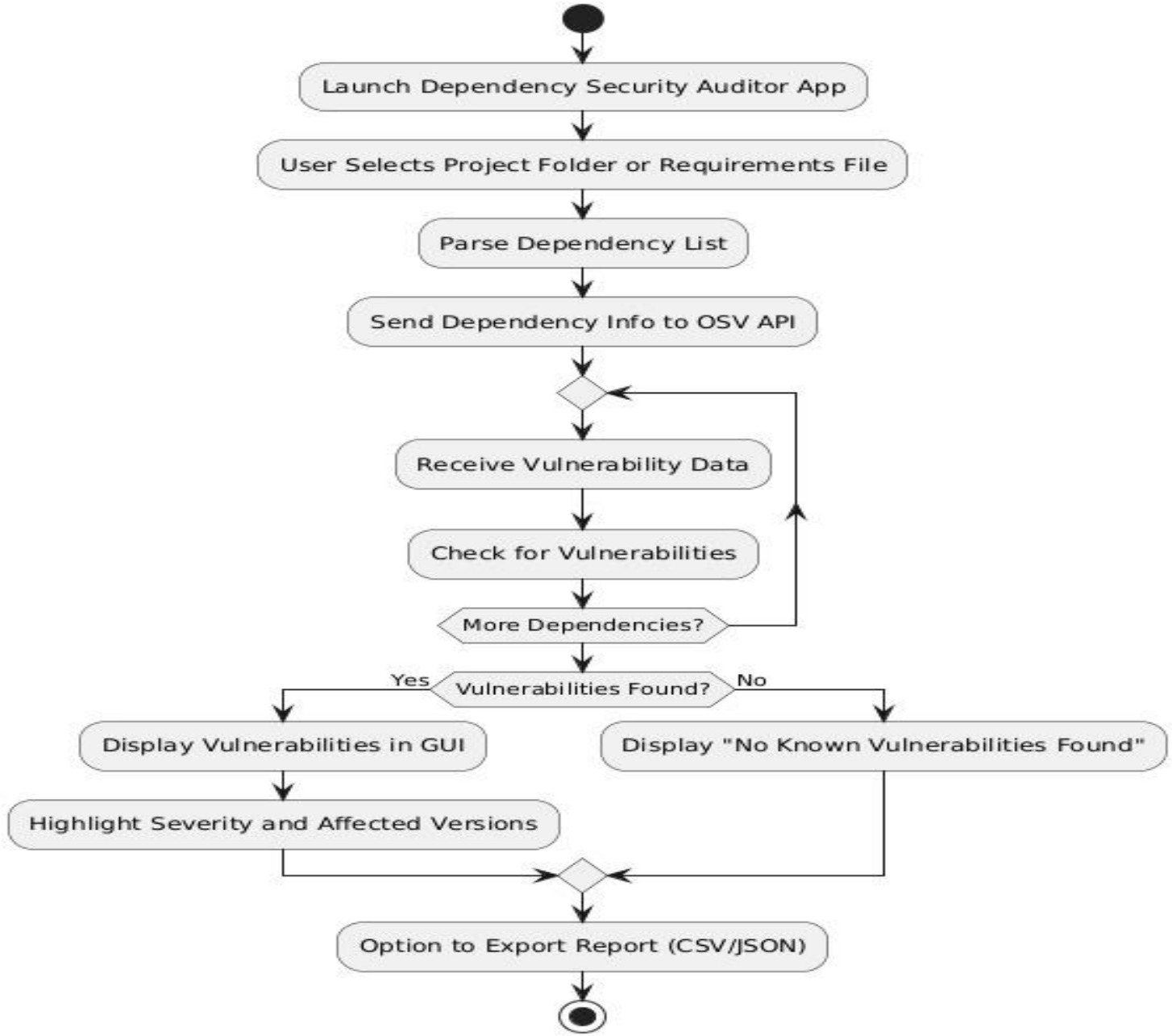
CLASS DIAGRAM



This class diagram shows the structure of the Dependency Security Auditor Tool, which is built to detect security issues in software dependencies. The process begins with the GUI class, which acts as the user interface. It allows the user to select a file through showFileChooser(), display results using displayResults(data), and show warnings if any vulnerabilities are found using displayWarnings(vulnerabilities). Once a file is selected, the DependencyScanner class handles the main processing. It stores the file path and has functions like loadDependencies() to read the file, parseRequirements() to understand the contents, and scanDependencies() to check for vulnerabilities. To perform the scan, it uses the OSVClient class, which connects to the Open Source Vulnerability database. The queryAPI(packageName, version) function sends package details and receives a list of known security issues.

If any vulnerabilities are found, they are stored in the Vulnerability class, which contains details like the ID, severity level, and description of each issue. It also includes the displayDetails() function to present this information clearly. After the scan is complete, the results can be processed using the ReportGenerator class. This class includes the generateReport(data) method to create a detailed report and functions like exportAsPDF(path) and exportAsCSV(path) to save the report in PDF or CSV formats. Together, these classes help in scanning, identifying, and reporting vulnerabilities in a clear and efficient way, making the tool useful for improving software security.

ACTIVITY DIAGRAM



The activity diagram explains the step-by-step process of how the Dependency Security Auditor Tool works to detect security issues in software projects. The process begins when the user launches the application. Once the tool is open, the user is asked to select a project folder or a requirements file. This file (like requirements.txt or package.json) contains a list of all the external libraries or dependencies used in the software project. After the file is selected, the tool begins to parse or read the dependency list from the file. It extracts the names and versions of all the packages that the project depends on. Then, the tool sends this information to the OSV API (Open Source Vulnerabilities Application Programming Interface). This API helps the tool to check whether any of the listed packages are known to have security vulnerabilities.

After sending the dependency information, the tool waits for a response from the OSV API. Once it receives the vulnerability data, it starts to check each dependency one by one to see if there are any security problems associated with them. If there are more dependencies to be scanned, the tool continues repeating the same process until all the dependencies have been checked. After scanning all packages, the tool checks whether any vulnerabilities were found. If no issues are detected, it shows a message on the screen that says "No Known Vulnerabilities Found". However, if there are any vulnerabilities, the tool displays them in the graphical user interface (GUI), making it easy for the user to see what’s wrong. It also highlights important details like the severity level of each vulnerability and which versions of the packages are affected. This helps developers quickly understand how serious the problems are and where exactly they exist. Finally, the user is given an option to export the report. The report, which contains the scan results, can be downloaded in formats like CSV or JSON. This report can be saved for records or shared with other team members for fixing the issues. Overall, the tool provides a smooth and simple way to find and report security problems in software dependencies, helping developers build more secure and reliable applications.

SOURCE CODE

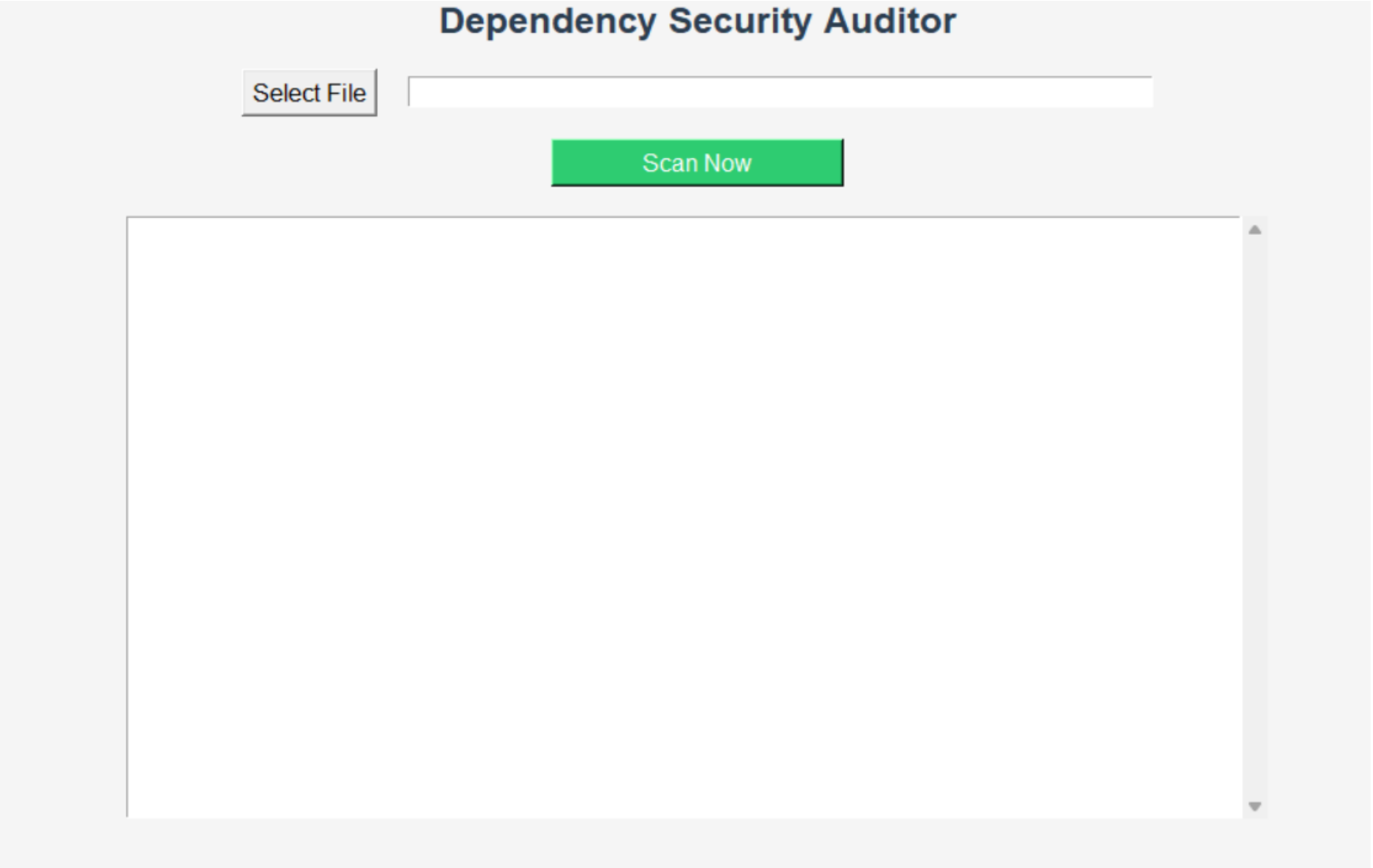
```
import requests
import tkinter as tk
from tkinter import filedialog, messagebox, scrolledtext

def read_requirements(file_path):
    dependencies = []
    try:
        with open(file_path, "r") as file:
            for line in file:
                if "==" in line:
                    name, version = line.strip().split("==")
                    dependencies.append((name, version))
    except FileNotFoundError:
        messagebox.showerror("File Not Found", f"Could not open: {file_path}")
    return dependencies

def check_vulnerability(package_name, package_version):
    url = "https://api.osv.dev/v1/query"
    payload = {
        "package": {"name": package_name, "ecosystem": "PyPI"},
        "version": package_version
    }
    try:
        response = requests.post(url, json=payload)
        if response.status_code == 200:
            return response.json().get("vulns", [])
        else:
            return []
    except requests.exceptions.RequestException as e:
        return [f"Error: {str(e)}"]
```

```
def select_file():
    file_path = filedialog.askopenfilename(filetypes=[("Text files", "*.txt")])
    if file_path:
        file_path_var.set(file_path)
        status_var.set("File selected. Ready to scan.")

def run_audit():
    file_path = file_path_var.get()
    if not file_path:
        messagebox.showwarning("No File", "Please select a file first.")
        return

    dependencies = read_requirements(file_path)
    results = []
    vulnerable_count = 0

    results.append("Dependency Security Audit Report\n")

    for name, version in dependencies:
        vulns = check_vulnerability(name, version)
        if vulns:
            vulnerable_count += 1
            results.append(f"{name}=={version} is VULNERABLE!")
            for vuln in vulns:
                results.append(f"   {vuln.get('id')} - {vuln.get('summary')}")
        else:
            results.append(f"{name}=={version} is SAFE.")

    results.append(f"\nTotal dependencies scanned: {len(dependencies)}")
    results.append(f"Total vulnerable dependencies: {vulnerable_count}")
    results.append("\nAudit Complete.")

    result_textbox.delete(1.0, tk.END)
    result_textbox.insert(tk.END, "\n".join(results))
```

```
    save_path = filedialog.asksaveasfilename(
        defaultextension=".txt",
        title="Save Report As",
        filetypes=[("Text files", "*.txt")]
    )

    if save_path:
        try:
            with open(save_path, "w", encoding="utf-8") as f:
                output = result_textbox.get("1.0", tk.END).strip()
                f.write(output)
            status_var.set(f"Report saved to {save_path}")
            messagebox.showinfo("Success", f"Audit complete!\nReport saved to:\n{save_path}")
        except Exception as e:
            messagebox.showerror("Error", f"Failed to save file: {e}")
    else:
        status_var.set("Scan complete, report not saved.")




root = tk.Tk()
root.title("Dependency Security Auditor")
root.geometry("800x600")
root.configure(bg="#f5f5f5")

file_path_var = tk.StringVar()
status_var = tk.StringVar()
```

```
tk.Label(root, text="Dependency Security Auditor", font=("Helvetica", 18, "bold"), bg="#f5f5f5", fg="#2c3e50").pack(pady=10)


frame_top = tk.Frame(root, bg="#f5f5f5")
frame_top.pack(pady=5)
tk.Button(frame_top, text="Select File", command=select_file, font=("Arial", 12)).pack(side=tk.LEFT, padx=10)
tk.Entry(frame_top, textvariable=file_path_var, width=60, font=("Arial", 11)).pack(side=tk.LEFT, padx=10)


tk.Button(root, text="Scan Now", command=run_audit, font=("Arial", 12), bg="#2ecc71", fg="white", width=20).pack(pady=10)


result_textbox = scrolledtext.ScrolledText(root, wrap=tk.WORD, width=90, height=25, font=("Courier New", 10))
result_textbox.pack(padx=10, pady=10)


status_label = tk.Label(root, textvariable=status_var, relief=tk.SUNKEN, anchor="w", bg="#ecf0f1", font=("Arial", 10))
status_label.pack(fill=tk.X, side=tk.BOTTOM)


status_var.set("Waiting for file selection...")
root.mainloop()
```

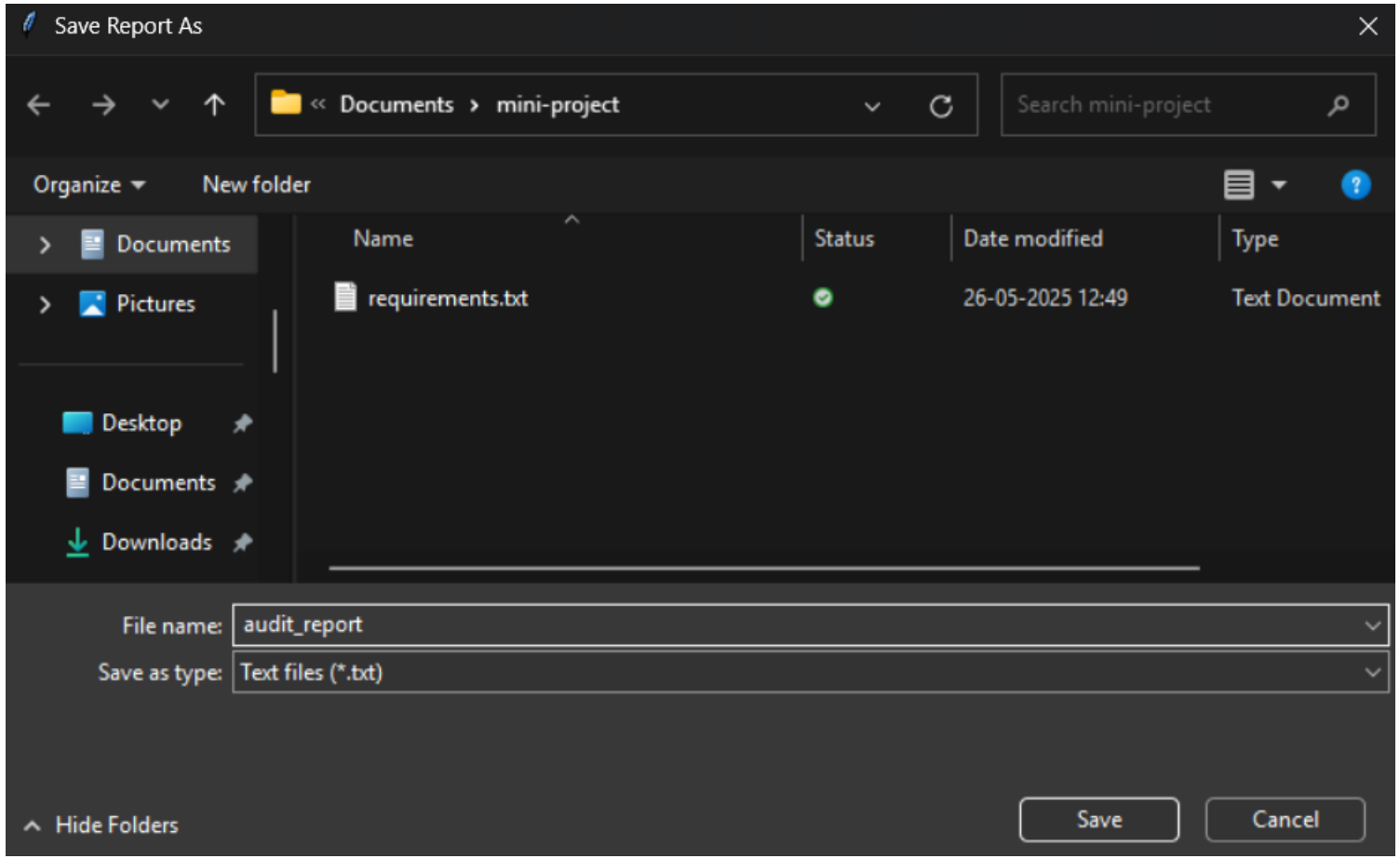
OUTPUT SCREENSHOTS

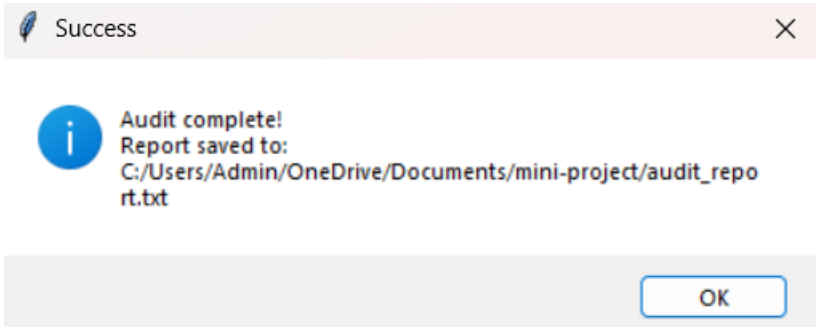


Dependency Security Auditor

Select File C:/Users/Admin/OneDrive/Documents/mini-project/requirements.txt

Scan Now


Save Report As
Documents › mini-project
Search mini-project
Organize
New folder
Documents
Pictures
Desktop
Documents
Downloads
Name
Status
Date modified
Type
requirements.txt
26-05-2025 12:49
Text Document
File name:
audit_report
Save as type:
Text files (*.txt)
Hide Folders
Save
Cancel

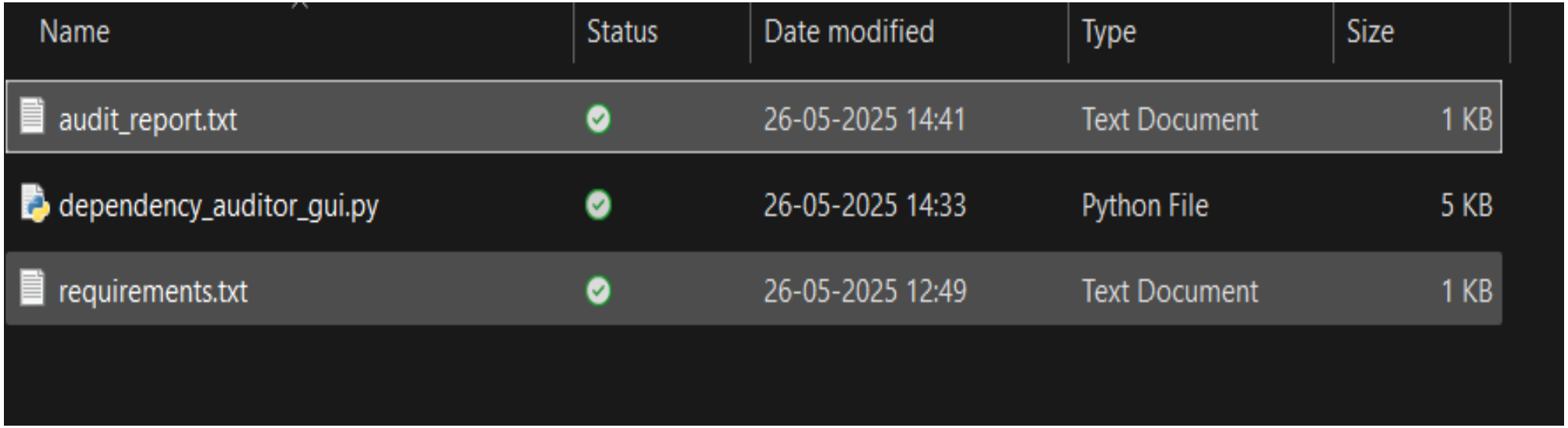

Success
Audit complete!
Report saved to:
C:/Users/Admin/OneDrive/Documents/mini-project/audit_report.txt
OK

Dependency Security Auditor

Select File C:/Users/Admin/OneDrive/Documents/mini-project/requirements.txt

Scan Now

```
Dependency Security Audit Report

requests==2.28.1 is VULNERABLE!
   GHSA-9wx4-h78v-vm56 - Requests `Session` object does not verify requests after making
first request with verify=False
   GHSA-j8r2-6x86-q33q - Unintended leak of Proxy-Authorization header in requests
   PYSEC-2023-74 - None
flask==2.2.2 is VULNERABLE!
   GHSA-m2qf-hxjv-5gpq - Flask vulnerable to possible disclosure of permanent session
cookie due to missing Vary: Cookie header
   PYSEC-2023-62 - None
numpy==1.24.0 is SAFE.

Total dependencies scanned: 3
Total vulnerable dependencies: 2

Audit Complete.
```



Name	Status	Date modified	Type	Size
audit_report.txt	✓	26-05-2025 14:41	Text Document	1 KB
dependency_auditor_gui.py	✓	26-05-2025 14:33	Python File	5 KB
requirements.txt	✓	26-05-2025 12:49	Text Document	1 KB

```
Dependency Security Audit Report

requests==2.28.1 is VULNERABLE!
   GHSA-9wx4-h78v-vm56 - Requests `Session` object does not verify requests after making first request with verify=False
   GHSA-j8r2-6x86-q33q - Unintended leak of Proxy-Authorization header in requests
   PYSEC-2023-74 - None
flask==2.2.2 is VULNERABLE!
   GHSA-m2qf-hxjv-5gpq - Flask vulnerable to possible disclosure of permanent session cookie due to missing Vary: Cookie header
   PYSEC-2023-62 - None
numpy==1.24.0 is SAFE.

Total dependencies scanned: 3
Total vulnerable dependencies: 2

Audit Complete.
```

CONCLUSION

In today's fast-paced software development environment, the use of third-party libraries and open-source packages has become standard practice to accelerate development and enhance functionality. However, this convenience brings a significant risk—many of these external packages may contain known vulnerabilities that, if not addressed, can be exploited by attackers. The Dependency Security Auditor project aims to tackle this issue by providing a lightweight, easy-to-use solution for identifying security flaws in Python project dependencies. This tool is especially beneficial for students, individual developers, and small teams who may lack access to sophisticated enterprise-level security tools. With a clean and simple GUI built using Python's Tkinter library, the application allows users to easily select their requirements.txt file and initiate a scan. Each package and its version are automatically parsed, and the system queries the Open Source Vulnerability (OSV) database using its API to detect any known vulnerabilities. This process eliminates the need for manual searching and reduces the chances of oversight.

The results of the scan are presented in a clear and readable format, indicating whether each package is safe or vulnerable, along with details such as vulnerability IDs, severity levels, and short descriptions. Users can also save the report as a text file for future reference, documentation, or sharing with teammates. Beyond convenience, this system promotes proactive security practices by enabling early detection of threats, thus allowing developers to take corrective action before deployment. The tool is capable of checking exact package versions, which is crucial since some vulnerabilities are version-specific. It also helps prioritize fixes by highlighting the severity of each vulnerability. In addition, the tool is designed to be extensible for future improvements such as multi-language support, CI/CD integration, or automated updates. Overall, the Dependency Security Auditor serves as a practical and efficient solution to an increasingly common problem in modern software development. By integrating this tool into their workflow, developers can ensure greater software security, reduce risks associated with supply chain attacks, and maintain a more robust and reliable codebase. This project not only demonstrates technical implementation skills but also showcases an understanding of real-world software security challenges and how automation can simplify critical development tasks.

FUTURE ENHANCEMENTS

1. Multi-language Dependency Support

Currently, the tool supports only Python dependencies (via requirements.txt). In the future, it can be extended to handle other languages like JavaScript (package.json), Java (pom.xml), and Ruby (Gemfile). This would make the tool more versatile and suitable for multi-language teams. It will require integrating language-specific parsers and vulnerability databases.

2. Email Notification System

An email alert system can notify developers when vulnerabilities are found, even if they're not actively using the tool. The email can include a summary and suggested actions. This ensures faster response to threats, especially in team environments or critical systems.

3. Severity-Based Suggestions and Auto-Fix

The tool can offer suggestions based on severity—for example, upgrading low-risk packages or alerting for urgent fixes. It can even auto-update certain packages to secure versions. This reduces manual work and adds intelligent support for developers.

4. Vulnerability Trend Analysis and Visualization

Graphs and dashboards can be added to show vulnerability trends over time. Developers can identify frequently affected packages and track improvements. Visual reports, made using Python libraries like matplotlib or seaborn, help in quick understanding and better decision-making.

REFERENCES

[1] A. Kumar, R. Kumar, and P. Singh, "Automated Vulnerability Detection in Software Dependencies," Unpublished manuscript, 2025.

[2] G. Aryan, A. Sharma, and L. K. Shar, "How Vulnerable Dependencies Affect Open-Source Projects," International Journal of Open Source Research, vol. 8, no. 2, pp. 34–40, 2024.

[3] P. Goswami and S. Gupta, "Investigating the Reproducibility of Packages," Proceedings of the National Conference on Software Engineering, pp. 55–60, 2023.

[4] P. Sharma and A. K. Singh, "Security Vulnerability Detection in Software Dependencies Using Public Databases," Journal of Cybersecurity Innovations, vol. 6, no. 1, pp. 25–32, 2024.

[5] R. Gupta, A. Kumar, and D. Sharma, "Enhancing Software Security through Dependency Vulnerability Auditing," International Conference on Information Security and Privacy, pp. 101–107, 2023.[6] P. S., C. C. B., and L. K. Raju, "Developer's Roadmap to Design Software Vulnerability Detection Model Using Different AI Approaches," IEEE Access, vol. 10, pp. 89437–220, 2022.

[6] M. Kumar and A. Sharma, "An Integrated Framework for Software Vulnerability Detection, Analysis and Mitigation: An Autonomic System," Sādhanā, vol. 42, pp. 1481–1493, 2017.

[7] A. Chandran, L. Jain, S. Rawat, and K. Srinathan, "Discovering Vulnerable Functions: A Code Similarity Based Approach," in Security in Computing and Communications, Singapore: Springer, 2016, pp. 417–429.