Security Researcher - SCA Task

SCA Task: Detect Vulnerable npm Packages

Build a Dockerized tool to scan repositories that use `npm`, detect vulnerable dependencies using open-source tools, and transform the results with Python into actionable JSON output.

Submission: Please share a Git repository containing your Dockerfile, Python code, and all relevant files.

SCA Introduction

- SCA is a static analysis for dependency vulnerabilities (libraries installed through package managers such as pip, npm, yarn, poetry, pnpm, gradle, maven, etc.)
- Vulnerabilities in SCA happen when there's usage of a certain library and version, and there's a published CVE for this particular library and version. Example: axios, a popular npm library for HTTP requests, is vulnerable to SSRF (CVE-2025-27152) in versions below 1.8.2 (https://nvd.nist.gov/vuln/detail/CVE-2025-27152)
- Vulnerabilities are identified statically by examining the lockfile to identify all used libraries, and communicating with a reliable database that contains CVE data, to check for any CVEs on the used packages.
- In npm, a package can be used in multiple paths, for example - we might install axios and lodash, but axios might transitively use lodash. So there are two "paths" from which we introduce lodash to our node_modules.
- If there's a vulnerability on lodash, we have to fix it both directly and by upgrading axios, because it is used in two different versions.

🎯 Objective

Create a Docker image that:

- Scans `package-lock.json` files for vulnerabilities.

- Parses and transforms the scan results using Python.
- Outputs findings in a structured, actionable JSON format.

🛠️ Requirements

- **Base Image**: Use any image you prefer (Node.js, Python, Alpine, etc.).
- **Vulnerability Detection**: Install and use **one** of:
  - `npm audit`
  - `osv-scanner`
- **Processing**:
  - Write a Python script to parse and structure results.
  - Include full dependency paths for each vulnerable package (e.g. `A → B → Z`).
  - Set the Python script as the Docker image's entrypoint.
- **Assumption**: The target always includes a `package-lock.json` file.
- **Output Format**: A `.json` file in the following structure:

```
{
 "results": [
  {
   "CVE": "CVE-x",
   "name": "Z",
   "version": "4.2.1",
   "dependency_graph": "A → B → Z"
  },
  {
   "CVE": "CVE-y",
   "name": "Z",
   "version": "3.5.5",
   "dependency_graph": "X → Y → Z"
  },
  {
   "CVE": "CVE-y",
```

```
      "name": "Z",
      "version": "3.12.2",
      "dependency_graph": "Z"  // direct use in package.json
    }
  ]
}
```

- **Testing**: Use `pytest` to cover relevant cases:
  - Direct dependencies
  - Transitive dependencies
  - Multiple introduction paths

💡 Bonus: Suggesting Fix Versions

Explain (no need to implement) how you'd identify a **safe upgrade path**.

> Example: If Z is vulnerable and introduced through A → B → Z, you may need to upgrade A to a version that pulls in a patched version of Z (since npm doesn't support pinning transitive deps like pip does).

📌 Using your selected tool (`npm audit` or `osv-scanner`):

- How would you discover which version of **A** removes the vulnerable version of **Z**?
- Favor approaches that avoid **major version upgrades** to minimize the risk of breaking changes.