

Vulnerability Scanning and Applicable Patch Management

FINAL REPORT

Kapil Rajesh Kavitha - 2021101028

P Harshavardhan - 2021111003

INTRODUCTION

In today's world, cybersecurity breaches often stem from a fundamental yet persistent issue: unpatched software vulnerabilities. Many users fail to update their systems promptly despite the availability of security patches, leaving them exposed to potential attacks.

The Vulnerability Scanning & Applicable Patch Management System addresses this critical need by providing an automated solution for detecting vulnerabilities in a user's system and directing them to the patches and updates for the vulnerabilities they wish to fix in their system. This enables the users to proactively identify vulnerabilities in their installed software and facilitates them in the application of the necessary updates and patches.

NVD TERMINOLOGY (will be used later on)

1. Common Vulnerabilities and Exposures (CVE)

- A standardized identifier for known cybersecurity vulnerabilities
- Format: CVE-YEAR-NUMBER (e.g., CVE-2021-44228)
- Each CVE includes:
 - Unique identifier
 - Description of the vulnerability
 - References to related vulnerability reports
 - Status of the vulnerability

2. Common Platform Enumeration (CPE)

- A standardized method for identifying and describing software packages and systems
- Format: `cpe:2.3:part:vendor:product:version:update:edition:language`
- Used to match vulnerable software configurations with installed applications
- Helps in precise identification of affected systems

3. Common Vulnerability Scoring System (CVSS)

- Industry standard for assessing the severity of security vulnerabilities
- Scores range from 0.0 to 10.0, with higher scores indicating greater severity
- Categories:
 - None (0.0)
 - Low (0.1-3.9)
 - Medium (4.0-6.9)
 - High (7.0-8.9)
 - Critical (9.0-10.0)

FUNCTIONAL REQUIREMENTS

1.1 Asset Scanning

- FR1.1: The system shall automatically detect installed applications and packages on the user's system
- FR1.2: The system shall use system-specific commands (e.g., PowerShell commands for Windows) to gather software information
- FR1.3: The system shall parse and normalize software information into a standard format containing:
 - Application name
 - Version
 - Provider name
- FR1.4: The system shall maintain a database of detected software packages

1.2 Vulnerability Identification and Assessment

- FR2.1: The system shall query and identify matching Common Platform Enumeration (CPE) entries for installed packages
- FR2.2: The system shall identify Common Vulnerabilities and Exposures (CVEs) associated with each CPE
- FR2.3: The system shall calculate and display vulnerability severity scores using the Common Vulnerability Scoring System (CVSS)
- FR2.4: The system shall select and highlight the CVE with the highest CVSS score for each application
- FR2.5: The system shall display a comprehensive list of identified vulnerabilities to the user

1.3 Patch Management

- FR3.1: The system shall allow users to select multiple assets for patch implementation
- FR3.2: The system shall implement a hierarchical update search system across multiple repositories:
 - Primary: UpdateStar repository
 - Secondary: SourceForge repository
 - Fallback: Google Search redirection
- FR3.3: The system shall provide direct links to available patches when found
- FR3.4: The system shall cache CPE-CVE extraction results to improve future performance

1.4 Operating System Support

- FR4.1: The system shall detect the user's operating system automatically
- FR4.2: The system shall initialize appropriate package managers based on the detected OS
- FR4.3: The system shall support modular expansion for different operating systems

NON FUNCTIONAL REQUIREMENTS

2.1 Performance

- NFR1.1: The system shall implement caching mechanisms for CPE-CVE extractions to reduce response time
- NFR1.2: The system shall process and display vulnerability scan results within acceptable time limits
- NFR1.3: The system shall handle multiple concurrent package scans efficiently

2.2 Reliability

- NFR2.1: The system shall implement comprehensive error handling for all scanning operations
- NFR2.2: The system shall provide fallback options when primary patch sources are unavailable
- NFR2.3: The system shall maintain data consistency during scanning and update operations

2.3 Security

- NFR3.1: The system shall execute system commands securely using appropriate permissions
- NFR3.2: The system shall validate all external data sources before processing
- NFR3.3: The system shall maintain secure connections when accessing external repositories

2.4 Usability

- NFR4.1: The system shall present vulnerability information in a clear, understandable format
- NFR4.2: The system shall provide an intuitive interface for selecting and managing patches
- NFR4.3: The system shall display meaningful error messages when operations fail

2.5 Maintainability

- NFR5.1: The system shall follow a modular design pattern for easy extension
- NFR5.2: The system shall implement abstract classes for OS-specific functionality
- NFR5.3: The system shall maintain clear documentation of all major components

2.6 Scalability

- NFR6.1: The system shall handle varying numbers of installed applications efficiently
- NFR6.2: The database shall scale to accommodate growing cached data
- NFR6.3: The system shall support future addition of new vulnerability databases and patch sources

CORE FUNCTIONALITIES

1. Asset Scanning

- 1.1. The system detects installed applications and packages on the user's system
- 1.2. The information on applications and packages are obtained using system commands (for instance, Powershell commands in Windows)
- 1.3. The software information is parsed and normalized into a standard format - (*name, version, providerName*) and is stored in the *installed_pkgs* parameter of the OSInterface-based class (this class differs based on the operating system of the user: in Windows systems, the Windows sub-class is used)

2. Vulnerability Identification and Assessment

- 2.1. The list of installed packages is sent to a dedicated Express server. This server takes the information from the package and queries to find the **CPE** which is the closest match to it.
- 2.2. The CVEs (vulnerabilities) associated with a particular CPE are identified. Then, the CVE with the highest CVSS (severity score) is chosen. There might be cases when CVSS scores of multiple or all CVEs belonging to the same application is 0.
- 2.3. Information regarding the CPE identified and its corresponding CVSS severity score are returned for each asset.
- 2.4. The list of identified vulnerabilities and their severity scores are then displayed back to the user.

3. Patch Management (using update search)

- 3.1. Users can select 1 or more assets for which they would like to implement the vulnerability patch for.
- 3.2. A hierarchical update search system is implemented, in order to check multiple repositories.
 - 3.2.1. **UpdateStar** - as the initial development was focused towards Windows applications, the UpdateStar repository was chosen as it contains ~2.5 million applications.
 - 3.2.2. **SourceForge** - it is one of the largest repositories of open-source software, with over >1.5million results
 - 3.2.3. If neither of these repositories yielded the update requested, the user would be redirected to a Google Search page titled "Patch for <Application> <Version>". While this does not directly lead to patch application, it does provide some guidance to the user as to how he can patch this vulnerability and it serves as the best-possible solution in this situation.

IMPLEMENTATION DETAILS

- Currently, support is provided specifically for Windows systems. However, the design has been made modular so as to enable future expansion to other operating systems.
 - An OSInterface abstract class was created to serve as the superclass for all OS-related classes, which would be used for asset scanning purposes.
 - Similarly, the UpdateSearcher abstract class was created as a superclass, defining methods to be implemented by search classes for other operating systems.
- As recommended by the NVD, the CPE-CVE extractions which take place are cached to reduce fetching time in the future.

PROGRAM MODULES

High Level Functions

Operating System Detection

get_os() -> Tuple[str, object, object, Optional[str]]

Primary function for OS detection and initialization of appropriate package managers.

Returns:

- **op_sys**: Operating system name (e.g., 'Windows')
- **pkg_mgr**: Package manager instance (Windows class)
- **update_searcher**: Update detection instance (WindowsUpdateSearcher class)
- **error**: Error message if any, None otherwise

Features:

- Uses platform.system() for OS detection
- Currently supports Windows OS only
- Initializes both package management and update detection capabilities
- Implements comprehensive error handling

Asset Detection

1. **get_assets(pkg_mgr) -> Tuple[Optional[List[Package]], Optional[str]]**

- Wrapper function for scanning and retrieving installed packages.
- **Parameters:**
 - **pkg_mgr**: Instance of the package manager (Windows class)
- **Returns:**
 - **installed_pkgs**: List of detected packages if successful, None if failed
 - **error**: Error message if any, None otherwise
- **Features:**
 - Provides a high-level interface to package scanning functionality
 - Implements error handling and propagation
 - Returns standardized output format for consistent handling

1. **scan_pkgs(self) -> Tuple[Optional[List[Package]], Optional[Exception]]**

- a. Primary method that orchestrates the package scanning process
- b. Implements a dual-scanning approach:
 - i. PowerShell's **Get-Package** for modern package installations
 - ii. WMI query using **Win32_Product** for traditional Windows installations
- c. Returns either a list of discovered packages or an exception if the scan fails

2. **exec_command(self, command: str) -> dict**

- a. Utility method to execute PowerShell commands safely
- b. Wraps subprocess execution with error handling
- c. Returns a dictionary containing:
 - i. **stdout**: Command output
 - ii. **stderr**: Error output

- iii. success: Boolean indicating command success

3. parse_get_package(self, stdout: str) ->

Tuple[List[str], Optional[Exception]]

- a. Parses output from **Get-Package** PowerShell cmdlet
- b. Handles complex name formats including embedded versions
- c. Features:
 - i. Extracts package names, versions, and provider names
 - ii. Handles version information embedded in package names
 - iii. Adds parsed packages to the installed_pkgs set

4. parse_get_package_wmi(self, stdout: str) ->

Tuple[List[str], Optional[Exception]]

- a. Parses output from WMI queries
- b. Simpler parsing logic focused on name and version extraction
- c. Maintains state between lines to properly pair names with versions

Search Utilities

- **scrape_updatestar_results(query)**
 - Constructs a search URL for the UpdateStar website using the query and sends an HTTP GET request.
 - Parses the webpage with **BeautifulSoup** to find and extract links from result elements if present.
 - Returns a list of URLs, a "No results found" message, or an error message if fetching or parsing fails.
- **extract_sourceforge_links(query)**
 - Sends an HTTP GET request to the SourceForge directory search URL for the given query.
 - Parses the webpage to find project links, extracting project names and URLs.
 - Returns a list of dictionaries with project names and URLs or an error message if the request fails.
- **search_for_updates(packages: List[VulnerablePackage]) -> List[str]**
 - Iterates over the list of packages.

- Calls `hierarchical_update_search()` with each package's name to find a relevant update URL.
- Handles exceptions during the search process, printing errors if encountered.
- **`open_links(links: List[str])`**
 - Iterates through the provided list of URLs.
 - Opens each URL in a new browser tab using `webbrowser.open_new_tab()`.
 - Adds a short delay (`time.sleep(1)`) between opening links to avoid overwhelming the browser.

Hierarchical Updates Search

1. `hierarchical_update_search(query: str)`

- Fetch UpdateStar results and use fuzzy matching to find the best match.
- If no match, fetch SourceForge links and use fuzzy matching to identify the best match.
- If no match on either, return a Google Search URL as a fallback.

Express Backend

1. API Endpoint

- Express js backend with a single endpoint to communicate with the frontend application.
- Receives the list of applications along with corresponding versions.
- Responds with applications identified in the NVD database along with corresponding CVEs and severity summary.

2. MongoDB Database

- Local database with the backend server, meant to act as a cache for recently queried applications.
- Built as per the guidelines of NVD, to reduce latency of service APIs, as well as serve a wider community.
- Two collections in CPE Cache and CVE Cache serve as lookup before requesting the NVD database.

3. Fetch APIs to NVD

- Two functions meant to query the NVD API with keyword searching as well as direct CPE23URI search.
- **fetchCPEsFromNVD(product: str, version: str)** uses keyword searching method along with version to get the respective CPEs of the applications.
- **fetchCVEsFromNVD(cpe23uri: str)** uses direct cpe23uri matching to get the respective CVEs of the applications.

4. Summarize Vulnerabilities

- The retrieved list of vulnerabilities can be very long and difficult or overwhelming to users.
- **summarizeVulnerabilities(cves: object)** uses goes through all the CVEs of each application and get the metrics of total number of vulnerabilities, number of them in each severity, and highest severity score.
- This gives a faster and more interpretable way of understanding security vulnerabilities than reading through multiple CVE descriptions.

TEST SUITE

1. API Vulnerability Analysis (**test_server_api.py** and **api_search_integration_test.py**)

- Tests the vulnerability analysis API endpoint
- Verifies API response status and result processing
- Supports analyzing single and multiple applications
- Validates that the API correctly identifies vulnerable software packages
- Both unit testing and integration testing done

2. Update Search Functionality (**patch_search_test.py**)

- Tests the Windows Update searcher mechanism
- Ensures search returns valid results and handles errors gracefully

3. Web Scraping Utilities (`search_utils_unittests.py`)

- Tests web scraping functions for different sources
- Includes tests for:
 - UpdateStar results scraping
 - SourceForge link extraction
- Handles various scenarios including:
 - Successful searches
 - No results found
 - Invalid search terms
 - Error handling for web scraping

4. Windows Asset Management (`assets_unittest.py`)

- Tests Windows-specific command translation
- Verifies package parsing from command outputs
- Checks handling of different package information formats

Test Strategies

- Pytest used for test framework
- Primarily unit testing is done, with some integration tests
- Includes error handling and edge case testing

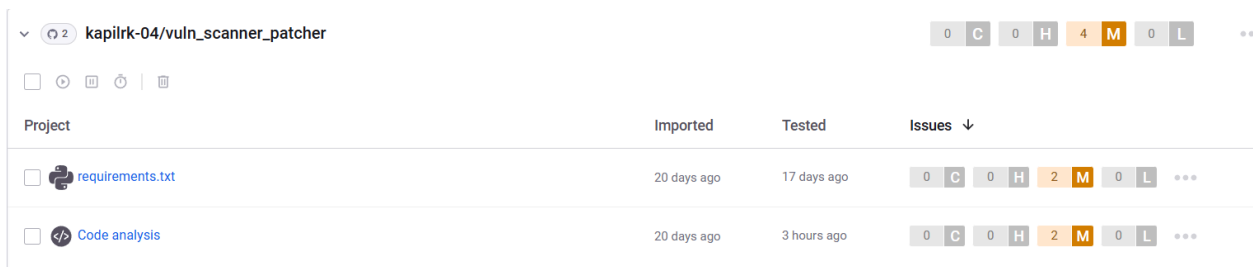
Key Test Scenarios

- Vulnerability detection for multiple software applications
- Update link retrieval
- Web scraping from different sources
- Command parsing and package information extraction
- API endpoint validation

INSTALLATION AND SETUP INSTRUCTIONS

- For Windows systems
- From the main directory, run **setup.sh** which will install backend server dependencies and Python dependencies for the CLI.
- To start the server, enter **backend** directory and run **npm start**
- To start the CLI, run **python src/main.py**

SAST TESTING WITH SNYK



kapiirk-04/vuln_scanner_patcher			0 C 0 H 4 M 0 L	..
<div>Project</div>				
Imported	Tested	Issues ↓		
requirements.txt	20 days ago	17 days ago	0 C 0 H 2 M 0 L	...
Code analysis	20 days ago	3 hours ago	0 C 0 H 2 M 0 L	...

In total, 4 medium vulnerabilities were discovered. The code-based vulnerabilities are due to the lack of CSRF protection in the backend server (not implemented due to time constraints)