Table of Contents

VulnScanAI Documentation

Welcome to the official documentation for VulnScanAI! This documentation provides comprehensive information on how to use the application, including:

- **App Directory:** An overview of the app's structure and features.
- **Scanners Directory:** Detailed information on the various security scanners integrated into VulnScanAI, including usage instructions, configuration options, and example outputs.
- **Views Directory:** Information on all the routes and views used in the application.

Key Features

VulnScanAI is designed to provide:

- **Comprehensive Security Scanning:** Integrates a wide range of security scanning tools to identify vulnerabilities.
- **Easy-to-Use Interface:** Simplifies the process of running complex security assessments.
- **Detailed Reporting:** Generates informative reports to help you understand and address security risks.
- **Customization:** Offers flexibility in configuring and customizing scans to meet your specific needs.

We hope this documentation is helpful. If you have any questions or feedback, please don't hesitate to contact us.

Good luck and happy scanning!

Directory: `app`

Purpose: This directory contains the core application logic for the VulnScanAI Flask website. It includes configuration, models, forms, views, and supporting modules.

Subdirectories:

- `reports` : Stores generated reports from various vulnerability scanners. These reports are often in PDF or HTML format.
- `scanners` : Contains the code for interacting with different vulnerability scanning tools. Each scanner has its own module for integration.
- `static` : Holds static files like CSS, JavaScript, images, and PDFs. These files are served directly to the client.
- `templates` : Contains the HTML templates used to render the website's user interface. Jinja2 templating engine is used.
- `uploads` : Stores uploaded PDF files for Generative AI Analysis.
- `views` : Contains the route handlers (view functions) that define the application's behavior. These functions handle user requests and interact with the models.

Files:

- `config.py` :
  - **Description:** Configuration settings for the Flask application. Loads settings from environment variables using `dotenv` , providing a centralized location for sensitive information like database URLs, API keys, and mail server details.
  - **Key Functionality:**
    - Defines configuration variables for different aspects of the application, including:
      - `SECRET_KEY` : Used for session management and CSRF protection.
      - `SQLALCHEMY_DATABASE_URI` : The database connection string.
      - `MAIL_*` : Settings for the Flask-Mail extension (server, port, credentials).
      - `RECAPTCHA_*` : Keys for Google reCAPTCHA integration.
      - `GEMINI_API_KEY` : API key for accessing the Google Gemini AI model.
      - `MOBSF_API_*` : API URL and Key for MobSF integration.
      - `PERMANENT_SESSION_LIFETIME` : Sets the session lifetime.

Directory: `app/scanners`

Purpose: This directory contains the code for integrating with different vulnerability scanning tools. Each file in this directory represents a scanner integration. The scanners are designed to be modular and easily extensible. They provide functions to run the scans and return reports (usually PDFs).

Files:

- `nmap_scanner.py`:
  - **Description:** Integrates with the Nmap port scanner to perform network reconnaissance and identify open ports and services. Generates reports in PDF format.
  - **Key Functionality:** Executes Nmap with various scan types, parses the output, and generates a PDF report with a custom header/logo.
  - **Dependencies:** `subprocess`, `os`, `reportlab`, `flask`
- `nikto_scanner.py`:
  - **Description:** Integrates with the Nikto web server scanner to identify potential vulnerabilities in web servers. Generates reports in PDF format.
  - **Key Functionality:** Executes Nikto, removes scan summary from HTML output, and converts to PDF with a custom header/logo. Supports authenticated scans.
  - **Dependencies:** `os`, `subprocess`, `pdfkit`, `bs4`, `fitz`, `flask`
- `zap_scanner.py`:
  - **Description:** Integrates with the OWASP ZAP proxy to perform dynamic application security testing (DAST) and generate reports in PDF format.
  - **Key Functionality:** Starts ZAP, configures it, runs a scan against a target URL (supports authenticated scans), and retrieves the report, converting it to PDF with a custom header/logo.
  - **Dependencies:** `os`, `subprocess`, `datetime`, `pdfkit`, `tempfile`, `time`, `fitz`, `flask`
- `mobsf_scanner.py`:
  - **Description:** Integrates with the Mobile Security Framework (MoBSF) to analyze mobile application binaries (APKs and IPAs) and generate reports.
  - **Key Functionality:** Uploads app binaries to the MobSF API, initiates scans, and fetches the PDF report, adding a custom logo/header.
  - **Dependencies:** `requests`, `os`, `fitz`, `flask`

Nmap Scanner

This document details the functionality of the `nmap_scanner.py` script, which automates network scanning using Nmap. The script provides several functions to execute different types of Nmap scans, generate PDF reports from the scan results, and incorporates user identification within the reports.

Overview

The script performs the following steps:

1. **Defines Scan Types:** Provides functions for various Nmap scan types, such as port scans, TCP SYN scans, aggressive scans, OS detection, and fragmented packet scans.
2. **Executes Nmap:** Runs the Nmap command with specified options and target.
3. **Generates PDF Report:** Creates a formatted PDF report using `reportlab`, including the scan type, target, scan results, and a company logo. The reports are user-specific.
4. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_report_directory(scan_type)`

- **Purpose:** Determines and creates the directory where Nmap reports will be stored based on the scan type.
- **Functionality:**
  - Defines the base directory path as `reports/nmap` within the Flask application's root directory.
  - Creates a sub-directory based on the scan type. The scan type is converted to lowercase and spaces are replaced with underscores (e.g., "Port Scan" becomes `port_scan`).
  - Uses `os.makedirs(specific_report_dir, exist_ok=True)` to create the directory if it doesn't exist. `exist_ok=True` prevents errors if the directory already exists.
- **Code Snippet:**

Nikto Scanner

This document details the functionality of the `nikto_scanner.py` script, which automates web vulnerability scanning using Nikto. The script provides functions to execute both unauthenticated and authenticated Nikto scans, process the generated HTML reports, convert them to PDF format, and include a company logo within the reports.

Overview

The script performs the following steps:

1. **Defines Report Directories:** Establishes directories for storing both unauthenticated and authenticated Nikto reports.
2. **Executes Nikto Scans:** Runs Nikto scans against specified target URLs, with options for providing authentication credentials.
3. **Processes HTML Report:** Removes the "Scan Summary" section from the generated HTML report for cleaner output.
4. **Converts to PDF:** Converts the processed HTML report to a PDF format, adding a company logo and title page using `pdfkit` and `PyMuPDF`.
5. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_nikto_report_base_directory()`

- **Purpose:** Determines and creates the base directory where all Nikto reports will be stored.
- **Functionality:**
  - Defines the base directory path as `reports/nikto` within the Flask application's root directory.
  - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist. `exist_ok=True` prevents errors if the directory already exists.
- **Code Snippet:**

ZAP Scanner

This document details the functionality of the `zap_scanner.py` script, which automates web application security scanning using OWASP ZAP (Zed Attack Proxy). The script provides functions to execute both unauthenticated and authenticated ZAP scans, convert the generated HTML reports to PDF format, and include a company logo within the reports.

Overview

The script performs the following steps:

1. **Defines Report Directories:** Establishes directories for storing both unauthenticated and authenticated ZAP reports.
2. **Executes ZAP Scans:** Runs ZAP scans against specified target URLs, with options for providing authentication credentials.
3. **Converts to PDF:** Converts the generated HTML report to a PDF format, adding a company logo and title page using `pdfkit` and `PyMuPDF`. It also covers the original ZAP logo on the report.
4. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_zap_report_base_directory()`

- **Purpose:** Determines and creates the base directory where all ZAP reports will be stored.
- **Functionality:**
  - Defines the base directory path as `reports/zap` within the Flask application's root directory.
  - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist.
- **Code Snippet:**

MobSF Scanner

This document details the functionality of the `mobsf_scanner.py` script, which automates mobile application security assessments using MobSF (Mobile Security Framework). The script provides functions to upload and analyze an application, fetch the generated PDF report, and customize the first page with a company logo.

Overview

The script performs the following steps:

1. **Defines Report Directory:** Establishes the directory for storing MobSF reports.
2. **Uploads and Analyzes App:** Uploads the provided application file to the MobSF API for analysis.
3. **Fetches and Customizes PDF Report:** Downloads the generated PDF report from MobSF, adds a company logo and title to the first page using `PyMuPDF`, and removes the original first page.
4. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_mobsf_report_directory()`

- **Purpose:** Determines and creates the directory where MobSF reports will be stored.
- **Functionality:**
  - Defines the base directory path as `reports/mobsf` within the Flask application's root directory.
  - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist.
- **Code Snippet:**

SQLMap Scanner

This document details the functionality of the `sqlmap_scanner.py` script, which automates SQL injection vulnerability scanning using SQLMap. The script provides a function to execute a SQLMap scan, process the generated raw text report, and generate a PDF report.

Overview

The script performs the following steps:

1. **Defines Report Directories:** Establishes directories for storing raw SQLMap reports and generated PDF reports.
2. **Executes SQLMap Scan:** Runs SQLMap against a specified target URL.
3. **Processes Raw Report:** Reads the raw text report (CSV, log, or target.txt) generated by SQLMap.
4. **Generates PDF Report:** Creates a formatted PDF report using `reportlab`, including user information, target URL, and the content of the raw SQLMap report.
5. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_sqlmap_base_report_directory()`

- **Purpose:** Determines and creates the base directory where all SQLMap reports will be stored.
- **Functionality:**
  - Defines the base directory path as `reports/sqlmap` within the Flask application's root directory.
  - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist.
- **Code Snippet:**

SSL Scanner

This document details the functionality of the `ssl_scanner.py` script, which automates SSL/TLS vulnerability scanning using `sslscan`. The script provides a function to execute `sslscan` against a specified hostname and generate a PDF report of the results.

Overview

The script performs the following steps:

1. **Defines Report Directory:** Establishes the directory for storing SSL scan reports.
2. **Executes `sslscan`:** Runs `sslscan` against the specified hostname, capturing its output.
3. **Generates PDF Report:** Creates a formatted PDF report using `reportlab`, including the hostname, scan results, and a company logo. The reports are user-specific.
4. **Returns Report Path:** Returns the full path to the generated PDF report.

Functions

1. `get_ssl_report_directory()`

- **Purpose:** Determines and creates the directory where SSL scan reports will be stored.
- **Functionality:**
  - Defines the base directory path as `reports/ssl` within the Flask application's root directory.
  - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist.
- **Code Example:**

```
import os
from flask import current_app

def get_ssl_report_directory():
    report_dir_base = os.path.join(current_app.root_path, 'reports', '
ssl')
    os.makedirs(report_dir_base, exist_ok=True)
    return report_dir_base
```

Prowler AWS Scanner

This document details the functionality of the `prowler_scanner_aws.py` script, which automates security assessments of Amazon Web Services (AWS) environments using Prowler. The script takes AWS access key ID, AWS secret access key, regions, and user ID as input, runs a Prowler scan, processes the generated HTML report (removing sensitive information and expanding "read more" sections), converts it to a PDF report with a company logo, and returns the filename of the resulting PDF report.

Overview

The script performs the following steps:

1. **Receives Input:** Takes AWS access key ID, AWS secret access key, a list of AWS regions, and user ID.
2. **Executes Prowler:** Runs the Prowler security assessment tool against the specified AWS account and regions, utilizing the provided credentials for authentication.
3. **Processes HTML Report:**
    - Expands the "read more" sections within the HTML report.
    - Removes sensitive data from the HTML report.
4. **Converts to PDF:** Converts the processed HTML report to a PDF format, adding a company logo and title page using `pdfkit` and `PyMuPDF`.
5. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_prowler_report_directory()`

- **Purpose:** Determines and creates the directory where Prowler AWS reports will be stored.
- **Functionality:**
    - Defines the base directory path as `reports/cloud_audit/aws_reports` within the Flask application's root directory.
    - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist. `exist_ok=True` prevents errors if the directory already exists.
- **Code Snippet:**

Prowler GCP Scanner

This document details the functionality of the `prowler_scanner_gcp.py` script, which automates security assessments of Google Cloud Platform (GCP) environments using Prowler. The script takes a GCP service account key, project ID, and user ID as input, runs a Prowler scan, processes the generated HTML report (removing sensitive information), converts it to a PDF report with a company logo, and returns the filename of the resulting PDF report.

Overview

The script performs the following steps:

1. **Receives Input:** Takes GCP service account key content, project ID, and user ID.
2. **Executes Prowler:** Runs the Prowler security assessment tool against the specified GCP project, utilizing the provided service account for authentication.
3. **Processes HTML Report:** Modifies the generated HTML report to remove sensitive data.
4. **Converts to PDF:** Converts the processed HTML report to a PDF format, adding a company logo and title page using `pdfkit` and `PyMuPDF`.
5. **Returns Report Filename:** Returns the filename of the generated PDF report.

CloudSploit Azure Scanner

This document details the functionality of the `cloudsploit_scanner_azure.py` script, which automates security assessments of Azure environments using CloudSploit. The script takes Azure subscription ID, tenant ID, client ID, client secret, and user ID as input, runs a CloudSploit scan, converts the generated CSV report to HTML and then PDF format, adds a company logo and title page, and returns the filename of the resulting PDF report.

Overview

The script performs the following steps:

1. **Receives Input:** Takes Azure subscription ID, tenant ID, client ID, client secret, and user ID.
2. **Executes CloudSploit:** Runs the CloudSploit security assessment tool against the specified Azure environment, utilizing the provided credentials for authentication.
3. **Converts CSV to HTML:** Converts the generated CSV report into a basic HTML table format.
4. **Converts HTML to PDF:** Converts the generated HTML to a PDF format, adding a company logo and title page using `pdfkit` and `PyMuPDF`.
5. **Returns Report Filename:** Returns the filename of the generated PDF report.

Functions

1. `get_cloudsploit_report_directory()`

- **Purpose:** Determines and creates the directory where CloudSploit Azure reports will be stored.
- **Functionality:**
  - Defines the base directory path as `reports/cloud_audit/azure_reports` within the Flask application's root directory.
  - Uses `os.makedirs(report_dir_base, exist_ok=True)` to create the directory if it doesn't exist. `exist_ok=True` prevents errors if the directory already exists.
- **Code Snippet:**

Directory: `app/views`

Purpose: This directory contains the view functions (route handlers) that define the application's behavior. These functions handle user requests, interact with models, and render HTML templates.

Files:

- `analyze_report_ai.py`:
  - **Description:** Analyzes scan reports using the Google Gemini model. Extracts text from PDFs, masks PII, and generates insights.
  - **Key Functionality:** Uses `GeminiReportAnalyzer` class to handle PDF processing, PII masking, and Gemini API interactions.
  - **Dependencies:** `google.generativeai`, `pypdf`, `flask`, `re`
- `auth.py`:
  - **Description:** Handles user authentication routes (login, registration, password reset, update password, email verification, etc.).
  - **Key Functionality:** Manages user registration, login, logout, password reset, and email verification using Flask-Login, `bcrypt`, and itsdangerous for secure token generation and email verification process.
  - **Dependencies:** `flask`, `flask_login`, `bcrypt`, `forms`, `models`, `flask_mail`, `itsdangerous`, `requests`, `os`
- `genai.py`:
  - **Description:** Contains routes and functions for interacting with the Gemini AI model for security report analysis.
  - **Key Functionality:** Handles uploading reports, analyzing them with Gemini, and managing chat sessions for follow-up questions.
  - **Dependencies:** `flask`, `flask_login`, `google.generativeai`, `werkzeug`, `os`, `logging`, `app.analyze_report_ai`
- `main.py`:
  - **Description:** Contains the main application routes (home page, profile page, about page, scan page, contact page, pricing page, download report page, etc.).
  - **Key Functionality:** Handles core web application routes, manages vulnerability scans, provides report downloads, and facilitates user communication via contact forms.