__init__.py Module

This module is responsible for creating and configuring the Flask application instance.

create_app() Function

This function creates and returns an instance of a Flask application, configured with various settings, features, and endpoints.

Process

1. A Flask instance is created and stored in the **qflask** variable.
2. CSRF protection is established using **CSRFProtect()**. This helps protect against certain types of attacks, particularly cross-site request forgery.
3. The IBM QRadar app id is retrieved using **qpylib.get_app_id()**.
4. A unique session cookie name is generated using the QRadar app id and set in the Flask application configuration.
5. An attempt is made to decrypt the stored secret key. If the decryption fails or the key doesn't exist, a new secret key is generated and encrypted for storage. The secret key is then set in the Flask application configuration.
6. An **after_request** Flask decorator is used to modify the 'Server' response header to hide the actual server details, which is a security best practice.
7. The function **qpylib.q_url_for** is added to the global Jinja2 template context under the name 'q_url_for'.
8. Logging is initialized using **qpylib.create_log()**.
9. A **/debug** endpoint is registered, returning a simple **Pong!** message. This is required by the QRadar App Framework for app health checking.
10. Additional endpoints are imported and registered from the **views** and **dev** modules, using Flask's Blueprint functionality.

Returns

The configured Flask application instance.

The **views.py** module handles the app views and routes. It also contains the main logic for communicating with MISP and QRadar servers, fetching IOCs (Indicators of Compromise) from the MISP server, and posting them to the QRadar server.

Key Features and Classes

1. **Blueprint**: The module defines a Flask blueprint named **viewsbp**. This blueprint handles the app's main route ('/').
2. **Logging**: The module uses Python's built-in logging framework to create a logger. It logs debug and error messages to a file and the console.
3. **StoppableThread**: This is a custom subclass of **threading.Thread**. It adds an event that can be used to stop the thread from outside.
4. **MISP and QRadar communication**: The module includes logic for communicating with MISP and QRadar servers, fetching IOCs from MISP, and posting these IOCs to QRadar.

5. **set_polling_thread() function**: This function sets up a new polling thread, which fetches IOCs from the MISP server and posts them to the QRadar server every few minutes.
6. **index() view function**: This is the app's main view function. It handles GET and POST requests to the app's main route ('/'). On a GET request, it renders the index.html template with current configuration values. On a POST request, it updates the app's configuration and starts a new polling thread.
7. **poll_ioc_import() function**: This function is the main function run by the polling thread. It fetches IOCs from the MISP server, posts them to the QRadar server, and waits for a few minutes before repeating the process.
8. **read_logs() function**: This function reads the last ten lines from the log file and returns them as a list of strings.

Routes and Endpoints

1. / : This is the main endpoint, accessible via GET and POST methods. On a GET request, it displays the current app configuration and log messages. On a POST request, it updates the app configuration, starts a new polling thread, and redirects back to the main page.

Error Handling

Errors are logged using the Python logging module. The app logs debug messages to a file and error messages to both the file and the console. If an error occurs while fetching IOCs from the MISP server or posting them to the QRadar server, the error message and stack trace are logged, and the polling thread stops its current iteration and waits until the next one to try again.

Dependencies

This application is dependent on several Python libraries including **flask**, **urllib3**, **logging**, **threading**, **os**, and **traceback**. It also depends on **qpylib** which is a Python library developed by IBM for building QRadar applications.

misp.py module

This Python script file contains a set of helper functions used to interact with MISP (Malware Information Sharing Platform) and QRadar (a SIEM - Security Information and Event Management - product by IBM).

These functions make API requests to both platforms, performing operations like obtaining Indicators of Compromise (IOCs) from MISP, checking if a certain reference set exists in QRadar, creating reference sets in QRadar, and posting IOCs to QRadar.

Functions

1. get_misp_ips(misp_server, misp_auth_key, event_id, ioc_type, page=1, limit=100)

This function is used to fetch IOCs from a specified MISP server. It takes in the following parameters:

- **misp_server**: the address of the MISP server.
- **misp_auth_key**: the authorization key to access the MISP server.
- **event_id**: the ID of the MISP event from which IOCs will be fetched.
- **ioc_type**: the type of the IOCs.
- **page**: the page number to fetch from MISP (default is 1).
- **limit**: the number of entries per page to fetch from MISP (default is 100).

The function returns a list of IOCs.

2. check_ref_set(qradar_server, qradar_auth_key, qradar_ref_set)

This function checks if a certain reference set exists in the QRadar server. It takes in the following parameters:

- **qradar_server**: the address of the QRadar server.
- **qradar_auth_key**: the authorization key to access the QRadar server.
- **qradar_ref_set**: the name of the reference set to be checked.

The function returns **True** if the reference set exists and **False** otherwise.

3. create_ref_set(qradar_server, qradar_auth_key, qradar_ref_set)

This function creates a reference set in the QRadar server. It takes in the following parameters:

- **qradar_server**: the address of the QRadar server.
- **qradar_auth_key**: the authorization key to access the QRadar server.
- **qradar_ref_set**: the name of the reference set to be created.

The function returns the JSON response from the QRadar server.

4. post_iocs_to_qradar(qradar_server, qradar_auth_key, qradar_ref_set, ioc_list)

This function posts a list of IOCs to a reference set in the QRadar server. It takes in the following parameters:

- **qradar_server**: the address of the QRadar server.
- **qradar_auth_key**: the authorization key to access the QRadar server.
- **qradar_ref_set**: the name of the reference set to which IOCs will be posted.
- **ioc_list**: a list of IOCs to be posted.

The function does not return any value.

views.py Module

The **views.py** module handles the app views and routes. It also contains the main logic for communicating with MISP and QRadar servers, fetching IOCs (Indicators of Compromise) from the MISP server, and posting them to the QRadar server.

Key Features and Classes

- **Blueprint**: The module defines a Flask blueprint named **viewsbp**. This blueprint handles the app's main route (/).
- **Logging**: The module uses Python's built-in logging framework to create a logger. It logs debug and error messages to a file and the console.
- **StoppableThread**: A custom subclass of **threading.Thread**. It adds an event that can be used to stop the thread from outside.
- MISP and QRadar communication: The module includes logic for communicating with MISP and QRadar servers, fetching IOCs from MISP, and posting these IOCs to QRadar.
- **set_polling_thread()** function: This function sets up a new polling thread, which fetches IOCs from the MISP server and posts them to the QRadar server every few minutes.
- **index()** view function: The app's main view function. It handles GET and POST requests to the app's main route (/). On a GET request, it renders the **index.html** template with current configuration values. On a POST request, it updates the app's configuration and starts a new polling thread.
- **poll_ioc_import()** function: The main function run by the polling thread. It fetches IOCs from the MISP server, posts them to the QRadar server, and waits for a few minutes before repeating the process.
- **read_logs()** function: Reads the last ten lines from the log file and returns them as a list of strings.

Routes and Endpoints

- /: The main endpoint, accessible via GET and POST methods. On a GET request, it displays the current app configuration and log messages. On a POST request, it updates the app configuration, starts a new polling thread, and redirects back to the main page.

Error Handling

Errors are logged using the Python logging module. The app logs debug messages to a file and error messages to both the file and the console. If an error occurs while fetching IOCs from the MISP server or posting them to the QRadar server, the error message and stack trace are logged, and the polling thread stops its current iteration and waits until the next one to try again.

Dependencies

This application is dependent on several Python libraries, including **flask**, **urllib3**, **logging**, **threading**, **os**, and **traceback**. It also depends on **qpylib**, a Python library developed by IBM for building QRadar applications.

index.html

index.html serves as web-based tool for importing Indicators of Compromise (IOCs) from a MISP (Malware Information Sharing Platform) server to a QRadar (IBM Security intelligence platform) server. The application is developed using a Python web framework (e.g., Flask) and uses the HTTP POST requests to get the IOCs from the MISP server and post them to the QRadar server.

Structure

The application comprises three main Python functions for importing IOCs:

- **get_misp_ips**: This function fetches the IOCs from the MISP server using a REST API. The IOCs are fetched based on the given event ID and IOC type. The function takes several parameters, including the MISP server address, authentication key, event ID, IOC type, and pagination details (page number and limit). It returns a list of IOCs.
- **check_ref_set** & **create_ref_set**: These functions work with the QRadar reference set. The first one checks if a given reference set exists on the QRadar server, while the second one creates a new reference set if it doesn't exist.
- **post_iocs_to_qradar**: This function takes a list of IOCs and posts them to the QRadar server using the QRadar REST API. It sends the IOCs to a given reference set.

User Interface

The user interface of the application is a single-page form where the user can enter the necessary information for importing the IOCs, including the addresses and API keys of the MISP and QRadar servers, the event ID and IOC type, and the reference set name.

Flow of Operation

1. The user provides necessary details on the form, including MISP and QRadar servers details, event ID, IOC type, reference set name, and the polling interval.
2. When the form is submitted, the application first gets the IOCs from the MISP server using the details provided.
3. The application then checks if the given reference set exists on the QRadar server. If the reference set does not exist, the application creates a new one.
4. After confirming the reference set, the application posts the IOCs to the QRadar server.
5. The application also offers a polling mechanism. Users can specify a polling interval in minutes, and the application will periodically (every minute) fetch the IOCs from the MISP server and post them to the QRadar server. The countdown until the next poll is displayed on the page.

Logging

The application keeps track of its operations and displays them on the page. If there are errors during the operation, they are displayed as well.

Important Notes

- The application doesn't validate the user input, and it's assumed that the user provides valid data. It's recommended to add data validation for better robustness.
- The application doesn't handle the situation where the MISP server doesn't have the specified event ID or IOC type, or the QRadar server doesn't accept the posted IOCs.
- The application uses a **verify=False** argument in the requests, which means it doesn't verify the SSL certificates. This is not recommended for a production environment because it leaves the application vulnerable to man-in-the-middle attacks.