Cross-Site Scripting (XSS) Prevention Testing

2 min

Cross-site scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts into webpages viewed by other users. XSS attacks can be used to steal information, hijack user sessions, deface webpages, or redirect users to malicious sites.

Creating XSS Prevention Testing

When creating XSS prevention tests, check that your web application properly escapes user input so that it is not executed as code in the browser.

What Will We Do?

In this lesson, we will use prompt engineering to aid in creating XSS prevention tests. We will be supplied with Python code that does not have any XSS prevention unit tests. We must prompt the AI to create some tests to ensure that our code is free from security vulnerabilities. (*Note: You can copy the code into the chat to help get better results.*)

Instructions

Below is a Python script with known XSS vulnerabilities.

```
from flask import Flask, request, render_template_string, session

app = Flask(__name__)
app.secret_key = 'super_secret_key'

# Simulate a simple in-memory 'database' to store user input
fake_db = {"user_input": ""}

@app.route('/store', methods=['POST'])
def store():
    user_input = request.form['input']
    fake_db['user_input'] = user_input  # Store the input in the 'database'
    return "Input stored successfully!"

@app.route('/get-stored-input')
def get_stored_input():
    # Return the stored input back to the user
    stored_input = fake_db['user_input']
    return render_template_string(f"<div>{stored_input}</div>")

if __name__ == '__main__':
    app.run(debug=True)
```

Your task is to prompt the AI chat to develop unit tests for the Python script. **You will know you have successfully completed the task when you obtain the following results from the machine:**

- Using the Python unittest framework
- Using the Python requests library
- Using the Python threading library

- A test that starts the Flask server in a new thread
- A test that stops the Flask server and waits for the thread to finish
- A test that checks for reflected XSS

Note: Depending on your prompt, you may receive different results than anticipated. That is okay. However, there should be at least four (4) tests.

If you would like some help creating prompts to send to the AI, open the hint dropdown below.

Prompt Hint

The AI remembers previous messages so you can chat back and forth like a conversation. You may need to use multiple prompts, but combining prompts saves computation time and energy. Here are some examples below:

- Create unit tests for the following Python code using the unittest, requests, and threading libraries: <paste in the code from above>
- Add a test which starts a Flask server in a new thread, a test which stops a Flask server and waits for the thread to finish, and a test which checks for reflected XSS.

If you need the AI to forget past prompts, click the trash can icon and the chat will reset.

Prompt

Create unit tests for the following Python code using the unittest, requests, and threading libraries:

```
from flask import Flask, request, render_template_string, session

app = Flask(__name__)
app.secret_key = 'super_secret_key'

# Simulate a simple in-memory 'database' to store user input
fake_db = {"user_input": ""}

@app.route('/store', methods=['POST'])
def store():
  user_input = request.form['input']
  fake_db['user_input'] = user_input  # Store the input in the 'database'
  return "Input stored successfully!"

@app.route('/get-stored-input')
def get_stored_input():
  # Return the stored input back to the user
  stored_input = fake_db['user_input']
  return render_template_string(f"<div>{stored_input}</div>")

if __name__ == '__main__':
  app.run(debug=True)
```

To create unit tests for the provided Python code, you can use the **unittest** library to test the Flask routes store and get_stored_input. Here is a basic outline of how you can structure the unit tests:

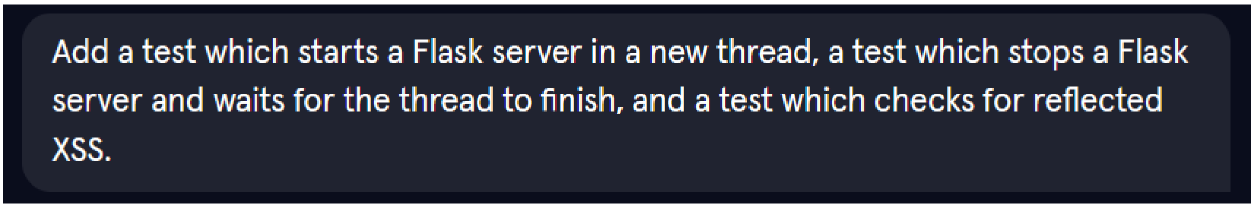
1. **Setup**:
    - Create a new Python file (e.g., test_app.py) for writing the unit tests.
    - Import the necessary modules: unittest, requests, and threading.
    - Import the Flask app and initialize the test client.
2. **Unit Test Cases**:
    - **Test case for /store route**:
        - Send a POST request to the /store route with user input.
        - Verify that the response status code is 200 and the response message is “Input stored successfully!”.
        - Check if the input is stored correctly in the fake database.
    - **Test case for /get-stored-input route**:
        - Check if the stored input can be retrieved correctly.
        - Verify that the rendered template contains the stored input.
3. **Clean up**:
    - Teardown any resources or clean up after each test case.

Here is a rough example of how you can implement these unit tests using unittest:

```
import unittest
from flask import Flask, request, render_template_string
from threading import Thread

app = Flask(__name__)
app.secret_key = 'super_secret_key'
fake_db = {"user_input": ""}

@app.route('/store', methods=['POST'])
def store():
    user_input = request.form['input']
    fake_db['user_input'] = user_input
    return "Input stored successfully!"
```

```
@app.route('/get-stored-input')
def get_stored_input():
    stored_input = fake_db['user_input']
    return render_template_string(f"<div>{stored_input}</div>")

class TestApp(unittest.TestCase):
    def setUp(self):
        thread = Thread(target=lambda: app.run(debug=False))
        thread.start()

    def tearDown(self):
        thread.join()

    def test_store_route(self):
        with app.test_client() as client:
            response = client.post('/store', data={'input': 'test input'})
            self.assertEqual(response.status_code, 200)
            self.assertEqual(response.data.decode('utf-8'), "Input stored successfully!")
            self.assertEqual(fake_db['user_input'], 'test input')

    def test_get_stored_input_route(self):
        with app.test_client() as client:
            fake_db['user_input'] = 'stored input'
            response = client.get('/get-stored-input')
            self.assertIn(b"<div>stored input</div>", response.data)

if __name__ == '__main__':
    unittest.main()
```

Please note that this is a simplified example to get you started, and you may need to further refine the test cases based on your specific requirements and the complexity of your Flask application.



Add a test which starts a Flask server in a new thread, a test which stops a Flask server and waits for the thread to finish, and a test which checks for reflected XSS.

To add the additional test cases for starting a Flask server in a new thread, stopping the Flask server and waiting for the thread to finish, and checking for reflected XSS, you can update the existing unit test class with the following methods:

1. **Start Flask Server in a new thread**:
    - Create a method that starts the Flask server in a new thread.
    - Ensure that the Flask server starts successfully before proceeding with the tests.
2. **Stop Flask Server and Wait for Thread to Finish**:
    - Create a method that stops the Flask server and waits for the thread to finish.
    - Clean up the resources after each test.
3. **Check for Reflected XSS**:
    - Create a method that sends a request to the Flask route with a script injection payload (such as <script>alert('XSS')</script>) to test for reflected XSS.
    - Verify that the input is sanitized or escaped to prevent XSS attacks.

Here is the updated example code with the additional test cases:

```
import unittest
from flask import Flask, request, render_template_string
from threading import Thread
import requests

app = Flask(__name__)
app.secret_key = 'super_secret_key'
fake_db = {"user_input": ""}

class TestApp(unittest.TestCase):
    def setUp(self):
        self.server_thread = Thread(target=lambda: app.run(debug=False))
        self.server_thread.start()

    def tearDown(self):
        requests.post('http://127.0.0.1:5000/shutdown')  # Custom route to shut down the server
        self.server_thread.join()

    def start_flask_server(self):
        self.server_thread.start()

    def stop_flask_server(self):
        requests.post('http://127.0.0.1:5000/shutdown')  # Custom route to shut down the server
        self.server_thread.join()

    def test_store_route(self):
        with app.test_client() as client:
            response = client.post('/store', data={'input': 'test input'})
            self.assertEqual(response.status_code, 200)
            self.assertEqual(response.data.decode('utf-8'), "Input stored successfully!")
            self.assertEqual(fake_db['user_input'], 'test input')

    def test_get_stored_input_route(self):
        with app.test_client() as client:
            fake_db['user_input'] = 'stored input'
            response = client.get('/get-stored-input')
            self.assertIn(b"<div>stored input</div>", response.data)

    def test_reflected_xss(self):
        with app.test_client() as client:
            payload = '<script>alert("XSS")</script>'
            response = client.get(f'/get-stored-input?input={payload}')
            self.assertNotIn(payload, response.data.decode('utf-8'))

if __name__ == '__main__':
    unittest.main()
```

In the updated test class, we added the start_flask_server, stop_flask_server, and test_reflected_xss methods to handle starting the Flask server in a separate thread, stopping the server, and checking for reflected XSS, respectively. Always ensure proper input validation and escaping to protect against XSS attacks in your application.