

## File Exposure Testing

File exposure testing involves checking whether a web application or script improperly exposes files and directories to users. This can happen due to misconfigurations, lack of access controls, or insecure file handling. The goal of file exposure testing is to ensure that sensitive files, such as configuration files, source [code](#), or user data, are not accessible to unauthorized users.

### Creating File Exposure Tests

When conducting file exposure testing, you typically:

- Check for directory listing vulnerabilities that may reveal the presence of sensitive files.
- Attempt to access files that are not publicly accessible using direct URL access or path traversal techniques.
- Evaluate file upload functionality to ensure that uploaded files cannot be executed or accessed if they contain sensitive information.

### What Will We Do?

In this lesson, we will use [prompt engineering](#) to aid in creating file exposure tests. We will be supplied with Python code that does not have any file exposure 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 file exposure vulnerabilities.

```
from flask import Flask, send_from_directory, request

app = Flask(__name__)

@app.route('/files/', methods=['GET'])
def files(filename):
    # Insecure file serving - vulnerable to path traversal attacks.
    return send_from_directory('files', filename)

@app.route('/upload', methods=['POST'])
def upload_file():
    # File upload without proper validation can lead to sensitive file exposure.
    file = request.files['file']
    file.save(f"./uploaded_files/{file.filename}")
    return "File uploaded successfully!"

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 for path traversal vulnerability and if the response contains part of the source code
- A test for direct file access vulnerability and if the response contains sensitive data
- A test for file upload vulnerability and check for successful upload response and if the uploaded file can be accessed
- A test for access to protected files and if the response contains protected file data

**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>
- There should be a test which starts a Flask server in a new thread and a test which stops a Flask server and waits for the thread to finish.
- Add a test which checks for a path traversal vulnerability and if the response contains part of the source code, a test which checks for a direct file access vulnerability and if the response contains sensitive data, a test which checks for a file upload vulnerability, a successful upload response, and if the uploaded file can be accessed, and a test which checks for access to protected files and if the response contains protected file data.

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, send_from_directory, request
```

```
app = Flask(__name__)
```

```
@app.route('/files/', methods=['GET'])
```

```
def files(filename):
```

```
    # Insecure file serving - vulnerable to path traversal attacks.
```

```
    return send_from_directory('files', filename)
```

```
@app.route('/upload', methods=['POST'])
```

```
def upload_file():
```

```
    # File upload without proper validation can lead to sensitive file exposure.
```

```
file = request.files['file']
file.save(f"./uploaded_files/{file.filename}")
return "File uploaded successfully!"
```

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

There should be a test which starts a Flask server in a new thread and a test which stops a Flask server and waits for the thread to finish. Add a test which checks for a path traversal vulnerability and if the response contains part of the source code, a test which checks for a direct file access vulnerability and if the response contains sensitive data, a test which checks for a file upload vulnerability, a successful upload response, and if the uploaded file can be accessed, and a test which checks for access to protected files and if the response contains protected file data.

### Response

To create unit tests for the given Python code, you can use the unittest library for defining test cases. Here is an outline of how you can create test cases for the described scenarios:

```
import unittest
```

```
import threading
```

```
import requests
```

```
from flask import Flask
```

```
from your_file import files, upload_file
```

```
class TestFlaskApp(unittest.TestCase):
```

```
    def setUp(self):
```

```
        self.app = Flask(__name__)
```

```
        self.app.add_url_rule('/files/<path:filename>', 'files', files)
```

```
        self.app.add_url_rule('/upload', 'upload_file', upload_file, methods=['POST'])
```

```
        self.server_thread = threading.Thread(target=self.app.run, kwargs={'debug': False})
```

```
        self.server_thread.start()
```

```
    def tearDown(self):
```

```
        self.app.do_teardown()
```

```
        self.server_thread.join()
```

```
def test_path_traversal_vulnerability(self):
    response = requests.get('http://127.0.0.1:5000/files/../run.py')
    # Check if the response contains part of the source code
    self.assertIn("from flask import Flask, send_from_directory, request", response.content)
```

```
def test_direct_file_access_vulnerability(self):
    response = requests.get('http://127.0.0.1:5000/files/sensitive_file.txt')
    # Check if the response contains sensitive data
    self.assertIn("Sensitive Data", response.content)
```

```
def test_file_upload_vulnerability(self):
    files = {'file': open('test_file.txt', 'rb')}
    response = requests.post('http://127.0.0.1:5000/upload', files=files)
    self.assertEqual(response.text, "File uploaded successfully!")

    # Check if the uploaded file can be accessed
    response = requests.get('http://127.0.0.1:5000/files/test_file.txt')
    self.assertEqual(response.status_code, 200)
```

```
def test_protected_files_access(self):
    response = requests.get('http://127.0.0.1:5000/files/protected_file.txt')
    # Check if the response contains protected file data
    self.assertIn("Protected Data", response.content)
```

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

Please adapt the code according to your project structure and requirements. Remember to replace 'your\_file' with the actual file/module name where files and upload\_file functions are defined. Also, make sure to have suitable test files for file upload vulnerability testing.

These test cases cover the scenarios you mentioned, such as path traversal vulnerability, direct file access vulnerability, file upload vulnerability, and access to protected files.