Experiment No. 9

Aim: To secure a file upload feature by validating file types and protecting against directory traversal attacks

Tools:

- Python 3
- Flask (pip install Flask)
- API Client (like Postman or curl

Learning Objective: To understand and implement multiple layers of security controls to create a robust file upload feature that prevents common vulnerabilities like arbitrary code execution and directory traversal.

Theory:

File upload features are a common entry point for attackers. If not properly secured, they can be exploited to upload malicious files (e.g., web shells, malware), leading to Remote Code Execution (RCE), or to overwrite critical system files using **Directory Traversal** attacks. A directory traversal attack uses ../ sequences to navigate outside the intended upload folder.
To build a secure file upload mechanism, a defense-in-depth strategy is required:

- **File Type Validation:** Restrict uploads to a whitelist of allowed file extensions (e.g., .png, .pdf). For greater security, the file's actual **MIME type** should also be validated, as an attacker can simply rename a malicious file (e.g., shell.php to image.jpg).
- **Directory Traversal Protection:** All user-supplied filenames must be sanitized to remove directory traversal sequences (../) and other malicious characters. Libraries like Werkzeug provide functions (secure_filename) for this purpose.
- **Randomized Filenames:** To prevent attackers from overwriting existing files or guessing filenames to execute them, uploaded files should be saved with a random, unpredictable name.
- **File Size Limits:** Enforcing a maximum file size prevents Denial of Service (DoS) attacks where an attacker attempts to fill up the server's disk space.

- **Isolated Upload Directory:** Files should always be saved to a dedicated, non-executable directory outside of the main application root.

Implementation:

Part 1: Application Development

- Install Flask: **pip install Flask**
- Create the Flask Application named secure_upload_app.py. This script includes all the necessary security checks.

```python
import os
from flask import Flask, request, jsonify
from werkzeug.utils import secure_filename

app = Flask(__name__)

UPLOAD_FOLDER = "uploads"
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif', 'pdf'}
MAX_FILE_SIZE = 5 * 1024 * 1024

app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
app.config['MAX_CONTENT_LENGTH'] = MAX_FILE_SIZE
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
        return jsonify({"error": "No file part in the request"}), 400

    file = request.files['file']
    if file.filename == '':
        return jsonify({"error": "No file selected"}), 400

    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file_ext = filename.rsplit('.', 1)[1].lower()
        random_filename = os.urandom(16).hex() + '.' + file_ext
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], random_filename)
        file.save(file_path)

        return jsonify({
            "message": "File uploaded successfully",
            "filename": random_filename
        }), 201
    else:
        return jsonify({"error": "File type not allowed"}), 400

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

Part 2: Testing the Feature

1. Run the application: **python secure_upload_app.py**.
2. **Test a successful upload:** Create a sample file (e.g., my_image.png) and use curl to upload it.

```
curl -X POST -F "file=@my_image.png" http://127.0.0.1:5000/upload
```

3. **Test an invalid file type:** Try to upload a file with a disallowed extension (e.g., test.txt).

```
curl -X POST -F "file=@test.txt" http://127.0.0.1:5000/upload
```

4. **Test a directory traversal attack:** Attempt to upload a file with a malicious name. The secure_filename function should neutralize this attack.

```
touch fake_file.png
curl -X POST -F "file=@fake_file.png;filename=../../../../../tmp/hacked.png"
```

Observe that the file is not saved in **/tmp/** but is safely handled and renamed within the uploads directory.

Output:

1. Successful File Upload

```
$ curl -X POST -F "file=@my_image.png" http://127.0.0.1:5000/upload

{
  "filename": "a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6.png",
  "message": "File uploaded successfully"
}
```

2. Attempted Upload of an Invalid File Type

```
$ curl -X POST -F "file=@document.txt" http://127.0.0.1:5000/upload

{
  "error": "File type not allowed"
}
```

3. Contents of the Uploads Directory

```
$ ls -l uploads/

-rw-r--r-- 1 user staff 51234 Jul 20 10:20 a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6.png
```

Learning Outcome: Performed an experiment to build a secure file upload endpoint in a Flask application. Successfully implemented multiple security controls, including whitelisting file extensions, sanitizing filenames to prevent directory traversal using secure_filename, and assigning random names to stored files to mitigate common web vulnerabilities associated with file uploads.

Conclusion: This experiment highlights that securing a file upload feature requires a multi-layered defense strategy. Relying on a single check is insufficient. By combining file type validation, strict filename sanitization, file size limits, and randomized storage names, we can create a robust system that effectively protects the server from malicious uploads, directory traversal, and other related attacks. These practices are essential for maintaining the integrity and security of any web application that accepts user-submitted files.

Name:
Class: BE-CSE
Roll No.:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				