

# Directory Traversal

Directory traversal, also known as path traversal, is **a type of security vulnerability where an attacker manipulates file paths to access files and directories outside the intended scope of the application**. If exploited, it can lead to unauthorized access to sensitive files, source code, or system configurations.

## 1. How Directory Traversal Works

- **Path Manipulation**
  - Applications that take user input to construct file paths without proper validation are vulnerable to directory traversal.
  - Attackers use special characters like `../` or `..\` to navigate up the directory structure.
- **Example**
  - Vulnerable code:
    - An attacker supplies `../../etc/passwd` as the filename, accessing sensitive system files.

```
def read_file(filename):  
    with open(f"/app/data/{filename}", "r") as f:  
        return f.read()
```

- **Impact**
  - Exposure of sensitive files such as configuration files, password hashes (`/etc/passwd`), and source code.
  - Gaining further access to execute or modify files.

## 2. Common Attack Patterns

- **Relative Path**
  - Using `../` to navigate to parent directories:
    - Example: `GET /file?name=../../etc/passwd`
- **Encoded Path**
  - Encoding traversal characters to bypass basic filtering:
    - Example: `GET /file?name=%2e%2e%2f%2e%2e%2fetc/passwd`
- **Mixed Path**
  - Combining traversal with valid paths:
    - Example: `GET /file?name=/var/www/html/../../etc/shadow`

## 3. Preventing Directory Traversal

### a. Input Validation

- **Validate User Input**
  - Allow only specific, expected patterns in file path input.
  - Example: Use regular expressions to match valid filenames.
- **Reject Special Characters**
  - Block characters like `../`, `/`, and `\` in user input.

- **Whitelist Valid Paths**

- Use a whitelist of acceptable filenames or directories.

## b. Path Normalization

- Definition
  - **Normalize file paths to remove traversal characters** and resolve the final path.
- How
  - **Use system APIs to normalize paths** and ensure they fall within allowed directories.
  - Example in Python:

```
import os

def secure_read_file(filename):
    base_dir = "/app/data"
    filepath = os.path.normpath(os.path.join(base_dir, filename))
    if not filepath.startswith(base_dir):
        raise ValueError("Invalid file path")
    with open(filepath, "r") as f:
        return f.read()
```

## c. Restrict File Access

- **Use Safe Directories**
  - Store files in a directory with restricted access (e.g., /app/data/).
- **Chroot or Jail**
  - Use a chroot jail to isolate file access and restrict the application's access to only specific directories.

## d. Implement Access Controls

- **File Permissions**
  - Set appropriate file permissions to prevent unauthorized access.
  - Example: Ensure sensitive files like /etc/passwd are not accessible by the application's user.
- User Isolation:
  - Run the application as a low-privilege user to limit the impact of traversal attacks.

## e. Use Frameworks or Libraries

- **Built-In File APIs**
  - Use framework-provided functions for file handling that inherently prevent traversal.
  - Example:
    - PHP's `basename()` to extract the filename.
    - Python's `os.path.join()` with validation.

## f. Avoid Dynamic Path Construction

- **Hardcoded Paths**

- Avoid constructing file paths dynamically based on user input.
- **Predefined Mapping**
  - Use a mapping of user inputs to predefined file paths:

```
valid_files = {
    "file1": "/app/data/file1.txt",
    "file2": "/app/data/file2.txt"
}

def read_file(file_key):
    if file_key not in valid_files:
        raise ValueError("Invalid file key")
    with open(valid_files[file_key], "r") as f:
        return f.read()
```

#### g. URL Decoding and Canonicalization

- Decode and canonicalize the input to handle encoded traversal sequences
  - Example: Decode %2e%2e%2f into ../.

## 4. Tools for Detection and Prevention

| Tool                             | Purpose  |
|----------------------------------|--|
| Static Analysis Tools            | Identify traversal vulnerabilities in code.      |
| Web Application Firewalls (WAFs) | Block directory traversal patterns in requests.  |
| Fuzzers                          | Test applications for traversal vulnerabilities. |
| Dynamic Analysis Tools           | Detect runtime vulnerabilities.                  |

## 5. Best Practices

- Input Validation: Strictly validate all user inputs.
- Path Normalization: Normalize paths to resolve traversal attempts.
- Access Controls: Limit application file access to specific directories.
- Use Security Frameworks: Rely on libraries and frameworks for secure file handling.
- Regular Testing: Test for directory traversal vulnerabilities using security tools.

## 6. Summary

| Aspect                | Details  |
|-----------------------|--|
| What It Is            | Exploiting file paths to access unauthorized directories or files.         |
| Impact                | Exposes sensitive files, source code, or configuration data.               |
| Prevention Techniques | Input validation, path normalization, access controls, and file API usage. |
| Tools                 | Static analysis tools, WAFs, fuzzers, dynamic analysis tools.              |

Directory traversal vulnerabilities are **preventable with robust input validation, proper path handling, and secure configurations**. By adhering to best practices such as restricting file access, normalizing paths, and leveraging security frameworks, developers can protect applications from exploitation while ensuring a secure user experience.