Testing for Directory or Path Traversal Vulnerabilities

goswamiijaya Follow
Nov 29, 2020 · 10 min read

In this article, we'll be discussing, how to perform Directory Traversal or Path Traversal attacks, aka **"dot-dot-slash", "directory climbing"** and **"backtracking".**

What is Path Traversal Vulnerability?

> *In Simple Words:* Path traversal vulnerabilities arise when the application **uses user-controllable data to access files and directories** on the application server or another backend filesystem in an unsafe way.

By submitting crafted input, an attacker may be able to **cause arbitrary content to be read from, or written to, anywhere on the filesystem being accessed, read sensitive information from the server, or overwrite sensitive files**, ultimately leading to arbitrary command execution on the server.

> *In Technical Words:* A path traversal attack aims to access files and directories that are stored outside the webroot folder. **By manipulating variables that reference files with "dot-dot-slash (../)" sequences and its variations or by using absolute file paths**, it may be possible to access arbitrary files and directories stored on file system including application source code or configuration and critical system files.

Note: access to files is limited by system operational access control (such as in the case of locked or in-use files on the Microsoft Windows operating system).

Let's **Consider the following example:**

Here, an application **uses a dynamic page to return static images** to the client. The name of the requested image is specified in a query string parameter:

```
https://testsite/image/8/getfile.ashx?filename=profile1.jpg
```

When the server processes this request, it follows these steps:

1. Extracts the value of the filename parameter from the query string.
2. Appends this value to the prefix `C:\filedirectory\.`
3. Opens the file with this name.
4. Reads the file's contents and returns it to the client.

An attacker can place p**ath traversal sequences(../) into the filename to backtrack** up from the directory specified & therefore access files from anywhere on the server that the user context used by the application has privileges to access.

```
https://testsite/image/8/getfile.ashx?filename=..\windows\win.ini
```

When the application appends the value of the filename parameter to the name of the images directory, it obtains the following path:

```
C:\filedirectory\..\windows\win.ini
```

The two traversal sequences effectively step back up from the images directory to the root of the `C:` drive, so the preceding path is equivalent to this:

```
C:\windows\win.ini
```

Hence, instead of returning an image file, the **server actually returns a default Windows configuration file.**

How to Test for Path traversal vulnerabilities?

STEPS:

1. Look out for instances where **a request parameter appears to contain the name of a file or directory**, such as `include=main.inc` or `template=/en/sidebar` .

Any functions whose implementation is likely to **involve retrieval of data from a server filesystem** such as the displaying of office documents or images.

2. Look for error messages or other anomalous events that are of interest or instances where **user-supplied data is being passed to file APIs or as parameters to operating system commands**.

"*If you have local access to the application (white-box testing) monitor all filesystem interaction that the application performs.*"

3. Modify the parameter's value to **insert an arbitrary subdirectory** and a **single traversal sequence**.

For example:

If the application submits this parameter: `file=foo/file1.txt`

try submitting: `file=foo/bar/../file1.txt`

If the application's **behavior is identical** in the two cases, it **may be vulnerable**.

Attempt to access a different file by traversing above the start directory.

4. If the application's **behavior is different** in the two cases, it **may be blocking, stripping, or sanitizing traversal sequences, resulting in an invalid file path**.

Examine whether there are any ways to circumvent the application's validation filters.
5. If you find any instances where submitting traversal sequences without stepping above the starting directory does not affect the application's behavior, **attempt to traverse out of the starting directory and access files** from elsewhere on the server filesystem.
6. If the application function you are attacking provides **read access to a file, attempt to access a known world-readable file on the operating system** in question. Submit one of the following values as the filename parameter you control:

```
../../../../../../../../../../../../etc/passwd
```

```
../../../../../../../../../../../../windows/win.ini
```

If successful, your browser displays the contents of the file you have requested.
7. If the function you are attacking provides **write access to a file, attempt to write two files** one that should be **writable by any user, and one that should not be writable even by root or Administrator**.

For example, on **Windows platforms** try this:

```
../../../../../../../../../../../../writetest.txt
```

```
../../../../../../../../../../../../windows/system32/config/sam
```

On **UNIX-based platforms**, files that root may not write are version dependent, but attempting to overwrite a directory with a file should always fail, so you can try this:

```
../../../../../../../../../../../../tmp/writetest.txt
```

```
../../../../../../../../../../../../tmp
```

If the application's behavior is different in response to the first and second requests, the application probably is vulnerable.

8. Or to verify a traversal flaw with **write access** is to **try to write a new file within the webroot of the web server and then attempt to retrieve this with a browser**. However, this method will work if you know the location of the webroot directory or if the user context in which the file access occurs does not have permission to write there.

> ***TIPS:***
>
> *Submit a large number of traversal sequences when probing for a flaw. It is possible that the starting directory to which your data is appended lies deep within the filesystem, so using an excessive number of sequences helps avoid false negatives.*
>
> *Also, the Windows platform tolerates both forward slashes and backslashes as directory separators, whereas UNIX-based platforms tolerate only the forward slash.*
>
> *Furthermore, some web applications filter one version but not the other. Even if you are certain that the webserver is running a UNIX-based operating system, the application may still be calling out to a Windows-based back-end component. Because of this, it is always advisable to try both versions when probing for traversal flaws.*

If your initial attempts to perform a traversal attack are unsuccessful, this does not mean that the application is not vulnerable.

Many application developers are aware of path traversal vulnerabilities and implement various kinds of input validation checks in an attempt to prevent them. However, those defenses are often flawed and can be bypassed by a skilled attacker.

1. The first type of input filter commonly encountered involves **checking whether the filename parameter contains any path traversal sequences**. If it does, the filter either rejects the request or attempts to sanitize the input to remove the sequences. This type of filter is often vulnerable to various attacks that use alternative encodings and other

tricks to defeat the filter.

1.1. Try simple **URL-encoded representations of traversal sequences** using the following encodings. **Be sure to encode every single slash and dot within your input**:

`Dot – %2e Forward slash – %2f Backslash – %5c`

1.2. Try using **16-bit Unicode encoding**:

`Dot – %u002e Forward slash – %u2215 Backslash – %u2216`

1.3. Try **double URL encoding**:

`Dot – %252e Forward slash – %252f Backslash – %255c`

1.4. Try **overlong UTF-8 Unicode encoding**:

`Dot – %c0%2e, %e0%40%ae, %c0ae,` and so on

`Forward slash – %c0%af, %e0%80%af, %c0%2f,` and so on

`Backslash – %c0%5c, %c0%80%5c,` and so on.

You can use the **illegal Unicode payload** type within Burp Intruder to generate a huge number of alternate representations of any given character and submit this at the relevant place within your target parameter.

These representations strictly violate the rules for Unicode representation but nevertheless are accepted by many implementations of Unicode decoders, particularly on the Windows platform.

1.5. If the application is attempting to sanitize user input by removing traversal sequences and does not apply this filter recursively, it may be possible to bypass the filter by placing one sequence within another. For example:

`….//`

`….\/`

`….\/`

`….\\`

2. The second type of input filter commonly encountered in defenses against path traversal attacks involves **verifying whether the user-supplied filename contains a suffix (file type) or prefix (starting directory)** that the application expects. This type of defense may be used in tandem with the filters already described.

2.1. Some applications **check whether the user-supplied filename ends in**

a particular file type or set of file types and reject attempts to access anything else. Sometimes this check can be subverted by **placing a URLencoded null byte** at the end of your requested filename, followed by a file type that the application accepts. For example:

```
../../../../../boot.ini%00.jpg
```

> *The reason this attack sometimes succeeds is that the file type check is implemented using an API in a managed execution environment in which strings are permitted to contain null characters (such as String.endswith() in Java). However, when the file is actually retrieved, the application ultimately uses an API in an unmanaged environment in which strings are null-terminated. Therefore, your filename is effectively truncated to your desired value.*

2. Some applications attempt to control the file type being accessed by **appending their own file-type suffix to the filename supplied by the user**.
In this situation, either of the preceding exploits may be effective, for the same reasons.
3. Some applications check whether the user-supplied **filename starts with a particular subdirectory of the start directory or even a specific filename**. This check can, of course, be bypassed easily as follows:

```
filestore/../../../../../../../etc/passwd
```

4. If none of the preceding attacks against input filters is successful individually, the **application might be implementing multiple types of filters**. Therefore, you need to combine several of these attacks simultaneously (both against traversal sequence filters and file type or directory filters). Try to break the problem into separate stages.

For example, if the request for: `diagram1.jpg` is successful, but the request for: `foo/../diagram1.jpg` fails, try all the possible traversal sequence bypasses until a variation on the second request is successful.

If these successful traversal sequence bypasses don't enable you to access `/etc/passwd`, probe whether any file type filtering is implemented and can be bypassed by requesting:

```
diagram1.jpg%00.jpg
```

Working entirely within the start directory defined by the application, try to probe to understand all the filters being implemented, and see whether each can be bypassed individually with the techniques described.

5. You can exploit read access path traversal flaws to **retrieve interesting files from the server** that may contain directly useful information or that help you refine attacks against other vulnerabilities. For example:
a) Password files for the operating system and application
b) Server and application configuration files to discover other vulnerabilities or fine-tune a different attack
c) Include files that may contain database credentials
d) Data sources used by the application, such as MySQL database files or XML files
e) The source code to server-executable pages to perform a code review in a search of bugs (for example, `GetImage.aspx?file=GetImage.aspx`)
f) Application log files that may contain usernames and session tokens and the like

If you find a path traversal vulnerability that grants write access, exploit this to **achieve arbitrary execution of commands on the server**. Here are some ways to exploit this vulnerability:
1. Create scripts in users' startup folders.
2. Modify files such as in `.ftpd` to execute arbitrary commands when a user connects.
3. Write scripts to a web directory with execute permissions, and call them from your browser.

Path Traversal attacks against a **web server**

```
https://testsite/../../../../../etc/passwd
```

`https://testsite/..%255c..%255c..%255cboot.ini`

`https://testsite/..%u2216..%u2216someother/file`

Path Traversal attacks against a **web application**
Original: `https://testsite/foo.cgi?home=index.htm`
Attack: `https://testsite/foo.cgi?home=foo.cgi`

In the above example, the web application reveals the source code of the `foo.cgi` file.

Path Traversal attacks against a **web application using special-character sequences:**
Original: `https://testsite/scripts/foo.cgi?page=menu.txt`
Attack: `http://testsite/scripts/foo.cgi?page=../scripts/foo.cgi%00txt`

In the above example, the web application reveals the source code of the `foo.cgi` file by using special-characters sequences. The "`../`" sequence was used to traverse one directory above the current and enter the `/` scripts directory. The "`%00`" sequence was used both to bypass the file extension check and snip off the extension when the file was read in.

Absolute Path Traversal

The following URLs may be vulnerable to this attack:

```
https://testsite/get.php?f=list
https://testsite/get.cgi?f=5
https://testsite/get.asp?f=test
```

An attacker can execute this attack like this:

```
https://testsite/get.php?f=/var/www/html/get.php
https://testsite/get.cgi?f=/var/www/html/admin/get.inc
https://testsite/get.asp?f=/etc/passwd
```

When the webserver returns information about errors in a web application, it is much easier for the attacker to guess the correct locations (e.g. path to the file with a source

code, which then may be displayed).

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:

1. Angle brackets `<` and `>` at the end of the path
2. Double quotes (closed properly) at the end of the path
3. Extraneous current directory markers such as `./` or `.\\`
4. 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
```

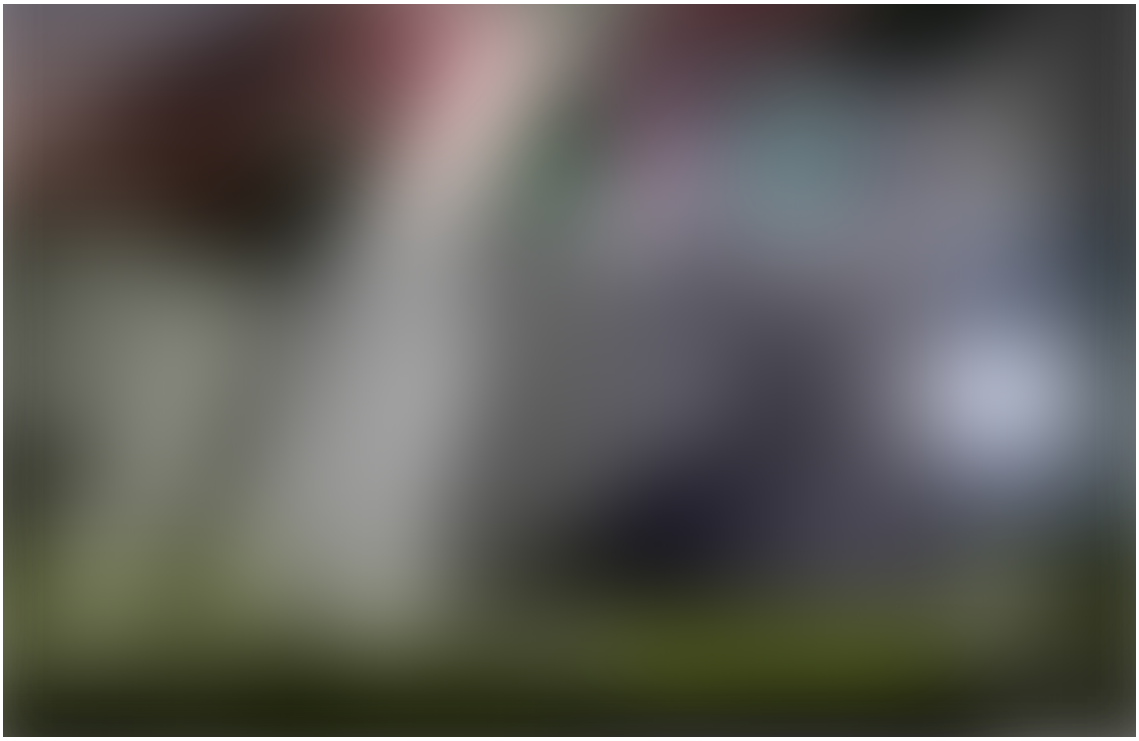
Windows API: The `periods or spaces` are discarded when used in any shell command or API call where a string is taken as a filename.

Windows UNC Filepaths: Used to reference files on SMB shares. Sometimes, an application can be made to refer to files on a remote UNC file path. 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 webserver.

- `\\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\`



References:

The Web Application: Hacker's Handbook- 2

OWASP/wstg

Many web applications use and manage files as part of their daily operation. Using input validation methods that have...

github.com

Infosec Bug Bounty Security Directory Traversal Path Traversal