**Path Traversal:**

AKA directory traversal

Possible vulnerable technologies include Python, PHP, Apache, ColdFusion, Perl and more

**What is it?**

These vulnerabilities enable an attacker to read arbitrary files on the server that is running an application. This might include:

* Application code and data.
* Credentials for back-end systems.
* Sensitive operating system files.

In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

**Reading Arbitrary Files via Path Traversal**

Imagine a shopping application that displays images of items for sale. This might load an image using the following HTML:

<img src="/loadImage?filename=218.png">

The loadImage URL takes a filename parameter and returns the contents of the specified file. The image files are stored on disk in the location /var/www/images/. To return an image, the application appends the requested filename to this base directory and uses a filesystem API to read the contents of the file. In other words, the application reads from the following file path:

/var/www/images/218.png

This application implements no defenses against path traversal attacks. As a result, an attacker can request the following URL to retrieve the /etc/passwd file from the server's filesystem:

https://insecure-website.com/loadImage?filename=../../../etc/passwd

This causes the application to read from the following file path:

/var/www/images/../../../etc/passwd

**\*\*The sequence ../ is valid within a file path, and means to step up one level in the directory structure. The three consecutive ../ sequences step up from /var/www/images/ to the filesystem root, and so the file that is actually read is:**

/etc/passwd

On Unix-based operating systems, this is a standard file containing details of the users that are registered on the server, but an attacker could retrieve other arbitrary files using the same technique.

On Windows, both ../ and ..\ are valid directory traversal sequences. The following is an example of an equivalent attack against a Windows-based server:

<https://insecure-website.com/loadImage?filename=..\..\..\windows\win.ini>

\*\* look for any parameter referencing a server-side file or user input that is put in a file path or requests that reference a file path this is a good spot to try path traversal \*\*

**\*\* sometimes burp will be weird about picking up these file references in the site map, we may have to use intercept to see the file being referenced if it does not do so explicitly in the URL or if its not shown in the site map. When you believe a file is being retrieved from the backend its best to intercept the requests manually and check by hand for file references.. pass this interesting request to repeater and begin experimenting. Really anytime an image is loaded we should check by manually intercepting \*\***

**Common Obstacles for exploiting Path traversal:**

Many applications that place user input into file paths implement defenses against path traversal attacks. These can often be bypassed.

If an application strips or blocks directory traversal sequences from the user-supplied filename, it might be possible to bypass the defense using a variety of techniques.

You might be able to use an absolute path from the filesystem root, such as filename=/etc/passwd, to directly reference a file without using any traversal sequences.

{You might be able to use nested traversal sequences, such as ....// or ....\/. These revert to simple traversal sequences when the inner sequence is stripped.

The above method is useful it traversal sequences are not stripped recursively so they only get stripped once resulting in the standard traversal sequence }

In some contexts, such as in a URL path or the filename parameter of a multipart/form-data request, web servers may strip any directory traversal sequences before passing your input to the application. You can sometimes bypass this kind of sanitization by URL encoding, or even double URL encoding, the ../ characters. This results in %2e%2e%2f and %252e%252e%252f respectively. Various non-standard encodings, such as ..%c0%af or ..%ef%bc%8f, may also work.

../ URL encodes to %2e%2e%2f

../ double URL encoded is %252e%252e%252f

*\*\*fuzzing filename parameters can also be very handy for detecting path traversal if you find a file reference, we can alter definitely try fuzzing it (with zap) different payloads such as the ones shown and various encodings (\*\*URL encoding is one of the most common\*\*) can be hard to produce manually for would take long so find some good payload lists and start fuzzing?? \*\**

{{An application may require the user-supplied filename to start with the expected base folder, such as /var/www/images. In this case, it might be possible to include the required base folder followed by suitable traversal sequences. For example:

filename=/var/www/images/../../../etc/passwd

so in the above its basically just saying sometimes a path is hard coded into the request or URL and webpages may be designed to validate that path is unchanged. Which in the case of path traversal can still be easily bypassed by simply leaving the hardcoded pat hand adding in the current number of step up sequences (../) to get to root directory enabling us to leak /etc/passwd }}

\*\* cool bypass: An application may require the user-supplied filename to end with an expected file extension, such as .png. In this case, it might be possible to use a null byte to effectively terminate the file path before the required extension. For example:

filename=../../../etc/passwd%00.png

so if you think a site is requiring a certain file extensions add a null byte followed by that file extensions to possibly bypass.