Testing for Local File Inclusion

Summary

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “dynamic file inclusion” mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation.

This can lead to something as outputting the contents of the file, but depending on the severity, it can also lead to:

- Code execution on the web server
- Code execution on the client-side such as JavaScript which can lead to other attacks such as cross site scripting (XSS)
- Denial of Service (DoS)
- Sensitive Information Disclosure

Local file inclusion (also known as LFI) is the process of including files, that are already locally present on the server, through the exploiting of vulnerable inclusion procedures implemented in the application. This vulnerability occurs, for example, when a page receives, as input, the path to the file that has to be included and this input is not properly sanitized, allowing directory traversal characters (such as dot-dot-slash) to be injected. Although most examples point to vulnerable PHP scripts, we should keep in mind that it is also common in other technologies such as JSP, ASP and others.

How to Test

Since LFI occurs when paths passed to `include` statements are not properly sanitized, in a blackbox testing approach, we should look for scripts which take filenames as parameters.

Consider the following example:

```
http://vulnerable_host/preview.php?file=example.html
```

This looks as a perfect place to try for LFI. If an attacker is lucky enough, and instead of selecting the appropriate page from the array by its name, the script directly includes the input parameter, it is possible to include arbitrary files on the server.

Typical proof-of-concept would be to load passwd file:

```
http://vulnerable_host/preview.php?file=../../../../etc/passwd
```

If the above mentioned conditions are met, an attacker would see something like the following:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
alex:x:500:500:alex:/home/alex:/bin/bash
```

```
margo:x:501:501::/home/margo:/bin/bash
...
```

Even when such a vulnerability exists, its exploitation could be more complex in real life scenarios. Consider the following piece of code:

```
<?php include($_GET['file'].".php"); ?>
```

Simple substitution with a random filename would not work as the postfix `.php` is appended to the provided input. In order to bypass it, a tester can use several techniques to get the expected exploitation.

Null Byte Injection

The `null character` (also known as `null terminator` or `null byte`) is a control character with the value zero present in many character sets that is being used as a reserved character to mark the end of a string. Once used, any character after this special byte will be ignored. Commonly the way to inject this character would be with the URL encoded string `%00` by appending it to the requested path. In our previous sample, performing a request to `http://vulnerable_host/preview.php?file=../../../../etc/passwd%00` would ignore the `.php` extension being added to the input filename, returning to an attacker a list of basic users as a result of a successful exploitation.

Path and Dot Truncation

Most PHP installations have a filename limit of 4096 bytes. If any given filename is longer than that length, PHP simply truncates it, discarding any additional characters. Abusing this behavior makes it possible to make the PHP engine ignore the `.php` extension by moving it out of the 4096 bytes limit. When this happens, no error is triggered; the additional characters are simply dropped and PHP continues its execution normally.

This bypass would commonly be combined with other logic bypass strategies such as encoding part of the file path with Unicode encoding, the introduction of double encoding, or any other input that would still represent the valid desired filename.

PHP Wrappers

Local File Inclusion vulnerabilities are commonly seen as read only vulnerabilities that an attacker can use to read sensitive data from the server hosting the vulnerable application. However, in some specific implementations this vulnerability can be used to upgrade the attack from LFI to Remote Code Execution vulnerabilities that could potentially fully compromise the host.

This enhancement is common when an attacker could be able to combine the LFI vulnerability with certain PHP wrappers.

A wrapper is a code that surrounds other code to perform some added functionality. PHP implements many built-in wrappers to be used with file system functions. Once their usage is detected during the testing process of an application, it's a good practice to try to abuse it to identify the real risk of the detected weakness(es). Below you can get a list with the most commonly used wrappers, even though you should consider that it is not exhaustive and at the same time it is possible to register custom wrappers that if employed by the target, would require a deeper ad hoc analysis.

PHP Filter

Used to access the local file system; this is a case insensitive wrapper that provides the capability to apply filters to a stream at the time of opening a file. This wrapper can be used to get content of a file preventing the server from executing it. For example, allowing an attacker to read the content of PHP files to get source code to identify sensitive information such as credentials or other exploitable vulnerabilities.

The wrapper can be used like `php://filter/convert.base64-encode/resource=FILE` where `FILE` is the file to retrieve. As a result of the usage of this execution, the content of the target file would be read, encoded to base64 (this is the step that prevents the execution server-side), and returned to the User-Agent.

PHP ZIP

On PHP 7.2.0, the `zip://` wrapper was introduced to manipulate `zip` compressed files. This wrapper expects the following parameter structure: `zip:///filename_path#internal_filename` where `filename_path` is the path to the malicious file and `internal_filename` is the path where the malicious file is place inside the processed ZIP file. During the exploitation, it's common that the `#` would be encoded with it's URL Encoded value `%23`.

Abuse of this wrapper could allow an attacker to design a malicious ZIP file that could be uploaded to the server, for example as an avatar image or using any file upload system available on the target website (the `php:zip://` wrapper does not require the zip file to have any specific extension) to be executed by the LFI vulnerability.

In order to test this vulnerability, the following procedure could be followed to attack the previous code example provided.

1. Create the PHP file to be executed, for example with the content `<?php phpinfo(); ?>` and save it as `code.php`
2. Compress it as a new ZIP file called `target.zip`
3. Rename the `target.zip` file to `target.jpg` to bypass the extension validation and upload it to the target website as your avatar image.
4. Supposing that the `target.jpg` file is stored locally on the server to the `../avatar/target.jpg` path, exploit the vulnerability with the PHP ZIP wrapper by injecting the following payload to the vulnerable URL: `zip://../avatar/target.jpg%23code` (remember that `%23` corresponds to `#`).

Since on our sample the `.php` extension is concatenated to our payload, the request to `http://vulnerable_host/preview.php?file=zip://../avatar/target.jpg%23code` will result in the execution of the `code.php` file existing in the malicious ZIP file.

PHP Data

Available since PHP 5.2.0, this wrapper expects the following usage: `data://text/plain;base64,BASE64_STR` where `BASE64_STR` is expected to be the Base64 encoded content of the file to be processed. It's important to consider that this wrapper would only be avaliable if the option `allow_url_include` would be enabled.

In order to test the LFI using this wrapper, the code to be executed should be Base64 encoded, for example, the `<?php phpinfo(); ?>` code would be encoded as: `PD9waHAgcGhwaW5mbygpOyA/Pg==` so the payload would result as: `data://text/plain;base64,PD9waHAgcGhwaW5mbygpOyA/Pg==`.

PHP Expect

This wrapper, which is not enabled by default, provides access to processes `stdio`, `stdout` and `stderr`. Expecting to be used as `expect://command` the server would execute the provided command on `BASH` and return it's result.

Remediation

The most effective solution to eliminate file inclusion vulnerabilities is to avoid passing user-submitted input to any filesystem/framework API. If this is not possible the application can maintain an allow list of files, that may be included by the page, and then use an identifier (for example the index number) to access to the selected file. Any request containing an invalid identifier has to be rejected, in this way there is no attack surface for malicious users to manipulate the path.