# PHP LFI to arbitratry code execution via rfc1867 file upload temporary files

by Gynvael Coldwind
18 March 2011

## Prologue

This article describes a **method** of taking advantage of a .php script Local File Inclusion vulnerability. It **does not** describe any vulnerability in the PHP engine itself, **nor** does it describe any new vulnerability class.

## LFI to code execution, common methods

One of the problems commonly encountered during security audits of PHP applications is proving that a Local File Inclusion indeed leads to arbitrary code execution, which may not be the case if the attacker cannot inject code to any file on the server.
Several methods are commonly used to prove that arbitrary code execution is possible:

- **including uploaded files** - straight forward method; this requires existence of an upload functionality in the tested website (e.g. photo upload, or document upload), access to upload functionality and storage of uploaded files in a place accessible by the PHP script

- **include data:// or php://input pseudo protocols** - these protocols must be enabled and accessible via include (allow_url_include set to on); also, php://filter pseudo protocol is usable in some cases

- **including logs** - this required PHP script to be able to access certain types of logs, e.g. httpd server error logs or access logs; also, size of these logs might make the attack harder (e.g. if error log has 2GB)

- **including /proc/self/environ** - this requires PHP to be run as CGI on a

system that has the /proc pseudo-filesystem and PHP script is required to have access to the aforementioned pseudo-file

- **include session files** - this requires the attacker to be able to influence the value of any string in a session (to inject code, e.g. <?php phpinfo(); ?>), the sessions must be stored in a serialized session file (as e.g. x|s:19:"<?php phpinfo(); ?>"; - this is the default setting for PHP) and the PHP script must be able to access the session file (usually names /tmp/sess_SESSIONID)

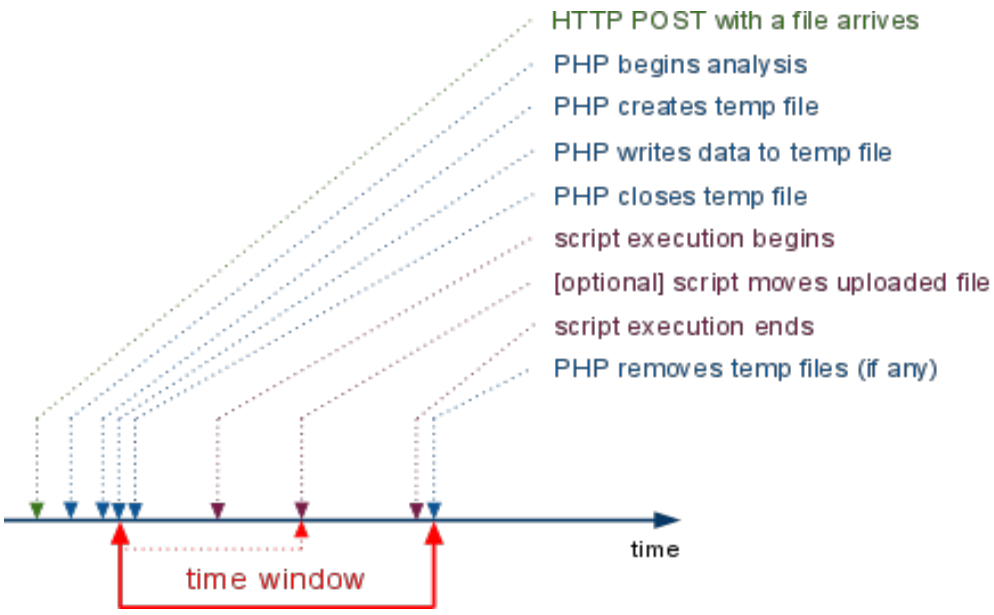
- **include other files created by PHP application** - this is very application and system specific, but it basically describes any other file that is created the websites functionality and the way it works, e.g. database files, cache files, application-level logs, etc

Additional tools included both the poison nul byte (addressed in PHP 5.3.4[1] released 2010-12-09) and excessive slash (/) suffix into path truncation bug[2] (patched in 2009).

# Temporary upload file inclusion

One other option is to take advantage of the way PHP handles file uploads via HTTP. This method is far less known and not really usable on platforms other than Windows (see Exploiting on Linux for details). Actually when I started writing this paper I believed that this method is not known at all, but after few days of searching and asking around I found a person (hi phunk ;>) that stumbled on this method before. Nevertheless I couldn't find any other proof of this being common knowledge, hence I decided to publish this article.

PHP engine, upon receiving a POST packet with RFC 1867 coded file(s), creates one or more temporary files which are used to store the uploaded files data. A PHP script handling file uploads is required to use the move_uploaded_file function to move the uploaded temporary file to a place of it's desire (if the script requires the file to exists after it terminates that is).
When the script ends PHP engine removes all temporary files for files that were uploaded (if any are left after the script ends that is).
The image below shows the timeline of this behaviour:




The important fact here is that PHP engine creates the temporary files even if the PHP script does not expect them (i.e. is not an upload handling script).
Hence, it is possible to **send a file** with arbitrary code to any PHP script, and **include the temporary file** that the PHP engine has created.

# Pros and cons

Good news is that it's common for a PHP script to have access to the directory (see upload_tmp_dir in php.ini) where the temporary files are created (i.e. can include files from this directory). On the default PHP installation the upload_tmp_dir is not set - in this case either /tmp on Linux-based systems or C:\Windows\Temp on Windows are used.
The bad news is that the name of the temporary file is random, which renders this method unusable in most cases - it is fully exploitable on Windows and exploitable in some cases on other systems.

# Exploitation on Windows

To generate the random name on Windows PHP uses the GetTempFileName function. Looking into documentation we can find the following explanation:

The **GetTempFileName** function creates a temporary file name of the following form: *<path>**<pre>*<*uuuu*>.TMP

In case of PHP <path> is upload_tmp_dir (normally it's just C:\Windows\Temp) and <pre> is "php" (without the quotes). The last part is described as:

<uuuu> Hexadecimal value of uUnique

uUnique is one the arguments of GetTempFileName and in case of PHP, it's set to 0, which is a special value telling the function to use the current system time. The important part here can be found in the remarks section:

Only the lower 16 bits of the *uUnique* parameter are used. This limits **GetTempFileName** to a maximum of 65,535 unique file names if the *lpPathName* and *lpPrefixString*parameters remain the same.

Only 65k unique names makes a brute force possible, and using current system time (number of milliseconds) makes things even simpler.

However, brute force is not needed here, thanks to a certain FindFirstFile quirk [3] which allows using masks (<< as * and > as ?) in LFI paths on Windows. Thanks to this, one can form an include path like this:

```
http://site/vuln.php?inc=c:\windows\temp\php<<
```

Commonly there are other files with "php" prefix in this directory, so narrowing the mask might be required. In this case it's best to choose the widest possible mask that does not include any file (e.g. php1<< or phpA<<, in worse case php11<<, etc) and send upload-packets until the PHP engine will create a temporary file that will match the mask.

# Exploitation on GNU/Linux

For temporary name generation PHP engine on GNU/Linux uses mkstemp from GNU libc. This function, depending on the way glibc is compiled, uses either (in pseudocode; variables are uint64_t):

1. random_value = (seed += time() ^ PID)
2. random_value = (seed += (gettimeofday().sec << 32 | gettimeofday().usec) ^ PID)
3. random_value = (seed += rdtsc ^ PID)

The random_value is later written as 6 digits of k=62 (A-Za-z0-9 charset) numeric system, and appended to the "/tmp/php" prefix (unless another directory is set), e.g. /tmp/phpUs7MxA.

From the aforementioned random_value generation methods, the 3rd is most commonly used nowadays.

According to initial tests I made, the random value is unpredictable enough to be considered safe against remote attacker.

This leads to the assumption that using this exploitation path is possible only in these specific cases:

- When the tester is able to list the files in the upload (/tmp) directory (presumably by another PHP script) - this would require a race condition method where the tester uploads one file to a slow script (i.e. a script that runs for a relatively long time) and lists the files in the upload directory in another script, acquires the file temporary file name, and fires the LFI exploit.

- When the tester is able to list the $_FILES array in a script - this requires an even more tight race condition and is not always possible, since the attacker must obtain the $_FILES array content during upload and send another packet before the previous script ends and the temporary file is removed. This is not possible in case of buffering of PHP output (e.g. if mod_gzip is enabled in Apache).

Hence, this method is usable only in some specific cases and should not be considered a generic method.

See also: part 2 of "A note on research done" section.

# A note on research done

One of the ideas I had to widen the exploitation window (i.e. the time window during which the temporary file exist) was to create an HTTP packet with e.g. 20 files (this is the default file-per-HTTP-packet limit in PHP) and send the first 19 files normally, but the last file in a slow manner (byte, sleep, byte, sleep, repeat). In theory, the previous uploaded files from the same packet should still exist until the upload will finish and the time window closes. However, it turns out that the PHP engine's upload parsing function is not executed until the whole HTTP packet arrives, thus, these files won't be yet created.

Also, in some cases, when httpd has access to /proc/self/fd (commonly httpd works with privileges dropped to www-data, while /proc/self/fd has root as it's owner, hence requires root privileges to access), this method is easier to exploit by trying

to upload files (in one thread) and include /proc/self/fd/XYZ in the other (e.g. start with XYZ=10). However, the temporary file fd handle exists only between opening the temp file and closing it, so it's a very narrow time window. Uploading large files might widen this window a little though.

# Future work

There is some room for improvements and additional research related to this method:

- test mktemp and mkstemp on other systems (*BSD, Solaris, etc) / in other libc implementations

- do proper research on mkstemp on GNU/Linux and it's predictability for both a local and a remote attacker, that has both no knowledge and knowledge of previous generated temporary names

- the FindFirstFile << quirk works now, but it may be removed later, hence doing proper research on remotly predicting the temporary file name on Windows might also be worth doing

# References

[1] PHP 5.3 Changelog. http://php.net/ChangeLog-5.php
[2] Francesco "ascii" Ongaro, Giovanni "evilaliv3" Pellerano. *PHP filesystem attack vectors*. http://www.evilaliv3.org/articles/php-filesystem-attack-vectors/
[3] Vladimir Vorontsov, Arthur Gerkis. *Oddities of PHP file access in Windows®. Cheat-sheet*, 2011. http://onsec.ru/onsec.whitepaper-02.eng.pdf

# Thanks to

Felix Gröbert for the interesting discussion that led to this article :)

# Disclaimer

The views expressed here are mine alone and not those of my employer.