# **File Upload Validator – How to avoid file upload attacks**

**Proper file type checks:**

Check for at least basic parameters like file size, mime-type etc and allow only a selected MIME type wherever possible. Make a white-list of file extensions to be allowed for upload. Try to keep away from executable files and scripts where possible. Set minimum and maximum file size for upload. This will prevent Dosing the webserver by uploading huge files and exhaust the storage space.

## File Type

Don’t just upload the files based on their extensions.  
Always verify the file headers before uploading the file.  
Meaning, a .doc file being upload should have .doc headers and not .jpeg, .mov extension.

**Mime Type validation**

Another common mistake web developers do when securing file upload forms, is to only check for mime type returned from PHP. When a file is uploaded to the server, PHP will set the variable $\_FILES[‘uploadedfile’][‘**type**’] to the mime-type provided by the web browser the client is using. However, a file upload form validation cannot depend on this value only. A malicious user can easily upload files using a script or some other automated application that allows sending of **HTTP POST** requests, which allow him to send a fake mime-type.

**Block dangerous extensions**

In other cases, we encountered file upload forms using a blacklist approach, as a security measure. A list of dangerous extensions is compiled from the developer, and the access is denied if the extension of the file being uploaded is on the compiled list.

One main disadvantage of using black listing of file extensions, is that it is almost impossible to compile a list that includes all possible extensions that an attacker can use. E.g. If the code is running in a hosted environment, usually such environments allow a large number of scripting languages, such as Perl, Python, Ruby etc, and the list can be endless.

**Double extensions**

This case’s security measures, as a concept are very similar to that one used in case 3. Though instead of simply checking the extension string present in the filename, the developer is extracting the file extension by looking for the ‘.’ character in the filename, and extracting the string after the dot character.

The method used to bypass this approach is a bit more complicated, but still realistic. First, let’s have a look at how Apache handles files with multiple extensions. A quote from the Apache manual states:

“Files can have more than one extension, and the order of the extensions is normally irrelevant. For example, if the file welcome.html.fr maps onto content type text/html and language French then the file welcome.fr.html will map onto exactly the same information. If more than one extension is given which maps onto the same type of meta-information, then the one to the right will be used, except for languages and content encodings. For example, if .gif maps to the MIME-type image/gif and .html maps to the MIME-type text/html, then the file welcome.gif.html will be associated with the MIME-type text/html.”

Therefore a file named ‘filename.php.123’, will be interpreted as a PHP file and will be executed. This only works if the last extension (in our case .123), is not specified in the list of mime-types known to the web server. Web developers, usually are not aware of such ‘feature’ in Apache, which can be very dangerous for a number of reasons. Knowing this, an attacker can upload a file named shell.php.123 and bypass the file upload form protection. The script will compute the last extension (.123), and concludes that this extension is not in the list of dangerous extension. Having said that, it is impossible to predict all the possible random extensions a malicious user will use to be able to upload a file on your web server.

## **Random filenames and folder**

**File name Validation:** While allowing users to upload the files, we allow them to specify the name the files should be referred to. Application should validate these file names for any XSS attacks.

Do not allow user input to specify the destination directory or file name of uploaded documents. Good practice is to rename the document to some random value and track them in your Database. In short guessing the name of the uploaded file should be made difficult for the attacker.

Another important security measure is to use system-generated file names instead of the names supplied by users when storing files on the file system. This will prevent local file inclusion attacks and also make any kind of file name manipulation by the user impossible.

## **Upload Directory Security:**

Upload all files outside your web directory. Possibly separate the upload directory from application and system directories/drive. This can help mitigate issues related to resource exhaustion & directory traversal. Set proper folder permissions (chroot() ). Do not allow user to choose the upload folder. Avoid giving writable permissions to users. Instead webserver like apache can be given writable permissions while preventing users from RWX access on the upload folder.

Prevent users from directly accessing the files in the upload directory. Files can directly be stored on the server or other alternative would be to store the files as blobs in database instead. However blobbing for very large files can affect DB performance and also the malicious data if uploaded will be directly saved in database without validating.

. when we talk about upload directory security, the upload path should not only have strict access permissions but also the path should not be revealed to users. A wrong pattern of revealing the path in URL is noticed.

The most important safeguard is to keep uploaded files where they cannot be directly accessed by the users via a direct URL. This can be done either by storing uploaded files outside of the web root or configuring the web server to deny access to the uploads directory.

## **Anti Virus Scan:**

Scan the upload files for any virus or malicious content. You can even try out ModSecurity which has a feature for inspecting files on upload, which you can combine with some antivirus. The advantage is that you get to block the HTTP request before the file even gets into your system. Alternatively files can also be scanned immediately just after it is uploaded. Both are affective in their own way and can be adapted accordingly depending on their implementation challenges. Other content filtering techniques include icap or CVP which are worth a thought.

## Denial of service

Users might be able to upload a lot of large files and consume all available disk space. The application designer might want to implement a limit on the size and number of files one user can upload in a given period (a day)