Preventing File Upload Vulnerabilities

Throughout this module, we have discussed various methods of exploiting different file upload vulnerabilities. In any penetration test or bug bounty exercise we take part in, we must be able to report action points to be taken to rectify the identified vulnerabilities.

This section will discuss what we can do to ensure that our file upload functions are securely coded and safe against exploitation and what action points we can recommend for each type of file upload vulnerability.

Extension Validation

The first and most common type of upload vulnerabilities we discussed in this module was file extension validation. File extensions play an important role in how files and scripts are executed, as most web servers and web applications tend to use file extensions to set their execution properties. This is why we should make sure that our file upload functions can securely handle extension validation.

While whitelisting extensions is always more secure, as we have seen previously, it is recommended to use both by whitelisting the allowed extensions and blacklisting dangerous extensions. This way, the blacklist list will prevent uploading malicious scripts if the whitelist is ever bypassed (e.g. `shell.php.jpg`). The following example shows how this can be done with a PHP web application, but the same concept can be applied to other frameworks:

Code: php

```php
$fileName = basename($_FILES["uploadFile"]["name"]);

// blacklist test
if (preg_match('/^.*\.ph(p|ps|ar|tml)/', $fileName)) {
    echo "Only images are allowed";
    die();
}

// whitelist test
if (!preg_match('/^.*\.(jpg|jpeg|png|gif)$/', $fileName)) {
    echo "Only images are allowed";
    die();
}
```

We see that with blacklisted extension, the web application checks `if the extension exists anywhere within the file name`, while with whitelists, the web application checks `if the file name ends with the extension`. Furthermore, we should also apply both back-end and front-end file validation. Even if front-end validation can be easily bypassed, it reduces the chances of users uploading unintended files, thus potentially triggering a defense mechanism and sending us a false alert.

Content Validation

As we have also learned in this module, extension validation is not enough, as we should also validate the file content. We cannot validate one without the other and must always validate both the file extension and its content. Furthermore, we should always make sure that the file extension matches the file's content.

The following example shows us how we can validate the file extension through whitelisting, and validate both the File Signature and the HTTP Content-Type header, while ensuring both of them match our expected file type:

Code: php

```php
$fileName = basename($_FILES["uploadFile"]["name"]);
$contentType = $_FILES['uploadFile']['type'];
$MIMEtype = mime_content_type($_FILES['uploadFile']['tmp_name']);

// whitelist test
if (!preg_match('/^.*\.png$/', $fileName)) {
    echo "Only PNG images are allowed";
    die();
}

// content test
foreach (array($contentType, $MIMEtype) as $type) {
    if (!in_array($type, array('image/png'))) {
        echo "Only SVG images are allowed";
        die();
    }
}
```

Upload Disclosure

Another thing we should avoid doing is disclosing the uploads directory or providing direct access to the uploaded file. It is always recommended to hide the uploads directory from the end-users and only allow them to download the uploaded files through a download page.

We may write a `download.php` script to fetch the requested file from the uploads directory and then download the file for the end-user. This way, the web application hides the uploads directory and prevents the user from directly accessing the uploaded file. This can significantly reduce the chances of accessing a maliciously uploaded script to execute code.

If we utilize a download page, we should make sure that the `download.php` script only grants access to files owned by the users (i.e., avoid `IDOR/LFI` vulnerabilities) and that the users do not have direct access to the uploads directory (i.e., `403 error`). This can be achieved by utilizing the `Content-Disposition` and `nosniff` headers and using an accurate `Content-Type` header.

In addition to restricting the uploads directory, we should also randomize the names of the uploaded files in storage and store their "sanitized" original names in a database. When the `download.php` script needs to download a file, it fetches its original name from the database and provides it at download time for the user. This way, users will neither know the uploads directory nor the uploaded file name. We can also avoid vulnerabilities caused by injections in the file names, as we saw in the previous section.

Another thing we can do is store the uploaded files in a separate server or container. If an attacker can gain remote code execution, they would only compromise the uploads server, not the entire back-end server. Furthermore, web servers can be configured to prevent web applications from accessing files outside their restricted directories by using configurations like (`open_basedir`) in PHP.

Further Security

The above tips should significantly reduce the chances of uploading and accessing a malicious file. We can take a few other measures to ensure that the back-end server is not compromised if any of the above measures are bypassed.

A critical configuration we can add is disabling specific functions that may be used to execute system commands through the web application. For example, to do so in PHP, we can use the `disable_functions` configuration in `php.ini` and add such dangerous functions, like `exec`, `shell_exec`, `system`, `passthru`, and a few others.

Another thing we should do is to disable showing any system or server errors, to avoid sensitive information disclosure. We should always handle errors at the web application level and print out simple errors that explain the error without disclosing any sensitive or specific details, like the file name, uploads directory, or the raw errors.

Finally, the following are a few other tips we should consider for our web applications:

- Limit file size
- Update any used libraries
- Scan uploaded files for malware or malicious strings
- Utilize a Web Application Firewall (WAF) as a secondary layer of protection

Once we perform all of the security measures discussed in this section, the web application should be relatively secure and not vulnerable to common file upload threats. When performing a web penetration test, we can use these points as a checklist and provide any missing ones to the developers to fill any remaining gaps.

Previous Next

Mark Complete & Next