SECURE UPLOAD METHODS IN PHP

Here's how to deal with file upload attacks against sites developed in PHP, and how to write more secure code to prevent these attacks.

In most Web applications, developers provide upload file functionality — images, for example. This functionality could be exploited by attackers to upload malicious "Web shell" code, which might give them command-prompt access to the server. In this article, we look at how such upload forms (such as the HTML form code below) can be exploited, and how to prevent such attacks.

```
<form name=upload action=upload.php method=post>
   upload a file: <input type=file name=fileName >
   <input type=submit name=upload>
</form>
```

The simplest implementation

In most implementations, no verification of the uploaded file is done. The form above invokes anupload.php PHP script to process the upload; the code in upload.php might move the file to a common/well-known folder, without verifying its content — like what's given below:

```
<?php
$uploaddir = 'uploads/'; // Relative path under webroot
$uploadfile = $uploaddir . basename($_FILES['userfile']['name']);
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $uploadfile)) {
   echo "File is valid, and was successfully uploaded.\n";
}
else {
   echo "File uploading failed.\n";
}
?>
```

Let's suppose the attacker were to upload a file containing code as follows:

```
<?php
   System("ls");
?>
```

In such a case, upload.php will move the attackers' file (named shell.php, for example) to theuploads subdirectory. If, for instance, the attackers then enter http://site/uploads/shell.phpas the URL in their browser/client, they will get a listing of the current working directory.

Similarly, they can upload other PHP scripts with different shell commands, or even include code to download and run a back-door program, giving themselves a direct command shell on the attacked system.

Content type verification

If a form upload with a non-image file (say, the shell.php script seen above) is received, the HTTP request could be something like what's shown below (note the Content-Type header):

```
POST /upload.php HTTP/1.1
[...]
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 156
--xYzZY
Content-Disposition: form-data; name="userfile"; filename="shell.php"
Content-Type: text/plain
```

You could check the content type of the file that is being uploaded by adding the following validation code:

```
if($_FILES['userfile']['type'] != "image/gif") {
    echo "Sorry, we only allow uploading GIF images";
    exit;
}
```

This new code will check the uploaded file's content/MIME type — that is, GIF — and block any others. However, this content-type header is sent by the client doing the upload, and attackers could easily forge a false content-type header while actually uploading a PHP script. The presentupload.php will still accept the file; the vulnerability is not fixed.

Image file content verification

The next precaution a developer can take is to check the actual content of the uploaded file, to verify whether it actually is an image or not, using the PHP function getImageSize():

```
$imageinfo = getimagesize($_FILES['userfile']['tmp_name']); //check image
size
if($imageinfo['mime'] != 'image/gif' && $imageinfo['mime'] != 'image/jpeg') {
    echo "Sorry, we only accept GIF and JPEG images\n";
    exit;
}
```

The function getImageSize() returns the image's size, its image type and MIME type, if the file is a valid image file (else it will generate an error). This measure will defeat Content-Type header forgery, but it is still not safe; the attackers can embed malicious PHP code inside a GIF image file, rename it to shell.php, and upload it.

When getImageSize() looks at that file, it extracts data about the first portion, which is a valid image. However, the PHP interpreter, when invoked on the file as before, executes the PHP code in it, along with the binary data! Hence, if you don't do filename verification, you have not made this upload any safer!

Filename verification

The final check is on the extension of the uploaded file's name, as in the following code:

```
$blacklist = array(".php", ".phtml", ".php3", ".php4");
foreach ($blacklist as $item) {
    if(preg_match("/$item\$/i", $_FILES['userfile']['name'])) {
        echo "We do not allow uploading PHP files\n";
        exit;
    }
}
```

Here, $blacklist contains a list of file extensions, and the preg_match() function applies them as a regular expression check against the uploaded file's name. These are all the extensions that the Web server is configured to accept as PHP executable files. Once you

block files with these extensions, even if the attacker uploads PHP code in files with other extensions, the PHP interpreter won't execute them — so the attacker is blocked.

Other issues concerning a file upload

Other than the above safeguards, developers should also consider the following:

- There has to be a limit on the size of the file that is being uploaded, lest it consume all the available HDD space on the server, causing a failure/denial of service.
- Developers have to take care that uploaded files are not easily or directly viewable by users or attackers (i.e., they should not have a well-known URL for uploaded files, such as the uploads folder mentioned earlier). It is best if uploaded files are stored in a folder that is not below the Web root. Also, developers could store the original filename (as uploaded) in a database table, and rename the file in the storage folder with a randomly generated name, storing that alongside the original filename in the database. Then, the application can present the user with the original name from the database, but stream the contents from the renamed file.

Source : http://www.opensourceforu.com/2010/12/secure-upload-methods-in-php/