Exploiting File Uploads for Fun and Profit

File uploading is a scary thing for web developers. You're allowing complete strangers to put whatever they want onto your precious web server. By uploading malicious code, an attacker can compromise the web server or even serve malware to its users. This article explores the risk of remote code execution associated with insecure file uploads, its exploitation and its mitigations.

Web based file upload provides a simple way of accepting content from the users. It is a common requirement to allow users to upload files as it increases business efficiency. Whether it is a social networking site such as Facebook or Twitter, or a job portal, or web forums or blogs, websites often require users to upload their own content such as images, videos, documents and various other file types. File uploads, however, represent a significant risk to the web applications. Any attacker wants to find a way to get a code onto a victim system, and then looks for a way to execute that code. Using an uploaded file upload accomplishes this first step.

A Simple File Upload

Let us take a simple example of an application that does not impose any restrictions on the uploaded content. A file upload form usually consists of a HTML form and a upload script. The example below shows a code snipped for a HTML form and a PHP script (Listing 1 and Listing 2).

The HTML form provides the interface for the user to select and submit a file to upload, while the PHP script receives the file from the HTML form and places in the specified directory. When the HTML form is submitted, the PHP script receives a POST request with encoding multipart/form-data, it creates a file in a temporary directory. PHP also populates the global array `$_FILES` with the information about the uploaded file. Figure 1 shows an upload request sent to a web server.

Unrestricted File Uploads: A simple case

Let us take an example of an upload application where there are no restrictions on the uploaded

Listing 1. *HTML Form*

```
HTML Form
<form action="upload.php" method="POST"
          enctype="multipart/form-
          data">
Username: <input type="text" value="username" />
Select a file to upload: <input name="fileID"
          type="file" />
<br />
<input type="submit" value="Upload File" />
</form>
```

Listing 2. *PHP Script*

```
PHP upload script
<?php
$target_path  =  "uploads/";
$target_path  =  $target_path  . basename($_
          FILES['fileID']['name']);
if (move_uploaded_file($_FILES['fileID']['tmp_
          name'], $target_path)) {
   echo "The file " . basename($_FILES['fileID']
          ['name']) . " has been
          uploaded";
} else {
   echo "Error uploading file!";
}
?>
```

file. A malicious user can upload malicious code (for e.g. in the form of a malicious PHP script) and then execute it by opening the uploaded page in browser. A common approach is to upload a web shell that enables an attacker to execute arbitrary commands on the server. A web shell is nothing but a script that accepts commands through GET or POST requests. Commands executed using a web-shell will execute with the privileges of the web server.

Common Protections

Upload application often enforce some restrictions on the file being uploaded. Some of the common restrictions enforced by upload applications are enumerated below:

- Blacklist / whitelist certain file extensions
- Content-type / Mime-type check
- File header validation
- Content format
- Image compression

Defeating common restrictions

Let us have a look at some of the methods to bypass common restrictions used by the file upload applications.

Using double extensions

Some web developers try to filter uploads by extracting the file extension of the uploaded file by looking for the "." character in the filename, and then checking whether the extension is present in a whitelist. Such a naive check can easily be bypassed by using filenames with double extensions. The content interpreted by the web server often depends on the ordering of two extensions and server configuration.

For Apache web server, when handling files with multiple extensions, the ordering of extensions is irrelevant if only one of the extensions is in the list of mime-types known to the server, or if both the extensions map to the same mime-type. If both the extensions are known to the server, the one the right takes precedence. For e.g., if .gif maps to mime-type image/gif and .php maps to mime-type text/php, then a file named, test.php.gif' will be associated with mime-type image/gif and a file named, test.gif.php' will be associated with mime-type text/php. Also a file named, test.php.abc' will be associated with mime-type text/php if extension 'abc' is not specified in the list of mime-type known to the server. That means, an attacker can upload a file with a double extension such as file.php.xyz and it will be interpreted as php code by Apache, 'xyz' being one of the extensions not specified in the list of known mime-types.

On IIS 6, it is possible to execute ASP code by uploading a file with an extension such as '.asp;.jpg'. Note the semicolon between the two extensions. The upload application may treat the file as an image due to .jpg extension and the IIS server would stop parsing the referred URL at the first semicolon, treating the file as an ASP script. This approach works with other extensions such as .cer and .asa as well.

Using NULL byte

An attacker can also try to bypass blacklisting by using a NULL byte. A NULL byte can be inserted after the forbidden extension in the filename which ends with an extension that is permitted. This can be done using a web proxy by using a filename such as test.php%00.jpg if the upload application uses URLDecode or the filename is in the URL itself. Otherwise, the request can be edited to make a character between the two extensions as NULL (Figure 2).

Uploading .htaccess file

Web applications often blacklist specific file extensions when uploading files. Blacklisting involves creating a list of extensions considered dangerous and refusing to upload the file if the file has an extension that is on the list. Blacklisting is often easy to bypass as it is almost impossible to create a list

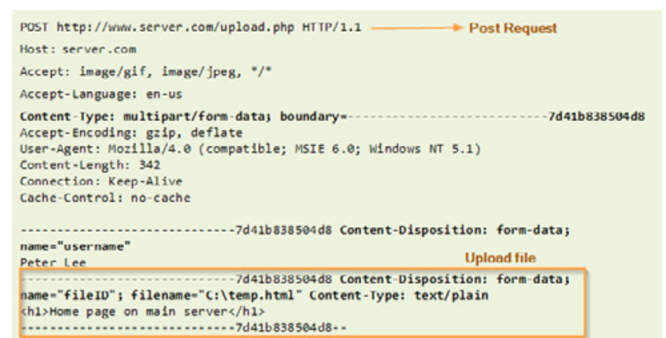

Figure 1. *An Upload Request*

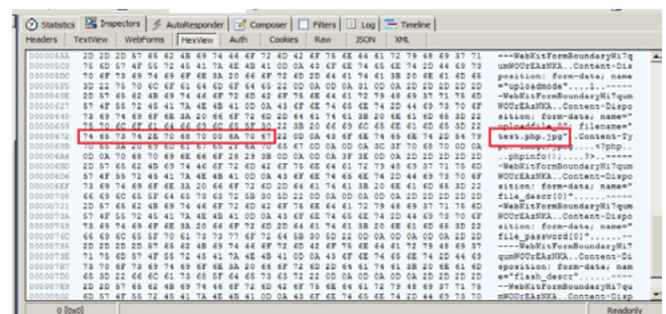

Figure 2. *Editing the Filename in a Web Proxy to Insert a NULL Byte. Notice the Second '.' Changed to a NULL Byte*

that includes all the possible extensions that an attacker can use. Often these extensions depend on the hosting environment and its configuration. An Apache web server hosted on a Linux platform may support a number of scripting languages such as Python, PHP, Perl etc. Failure to include any one of these may leave the server vulnerable.

A malicious user can also bypass file extension check by uploading a .htaccess file. A .htaccess file is a directory specific configuration file. It can override server configuration for the directory it is placed in, and for all the sub-directories. An attacker can upload a .htaccess file with the following line of code to bypass the file extension check.

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

The above line instructs Apache web server to execute .jpg files as PHP code. A malicious user can now upload a .jpg file containing PHP code.

Overwriting existing files

File upload applications often use .htaccess file inside the upload directory to restrict the execution of scripts from that directory. Typically, .htaccess files contain the following code to prevent execution of scripts:

```
AddHandler cgi-script .php .pl .py .jsp .asp .htm
                   .shtml .sh .cgi
Options -ExecCGI
```

Upload application often make use of `move_uploaded_file( )` to move the uploaded file from the temporary directory to the destination directory. This function overwrites the destination file if it exists. So a malicious user can name the file to be uploaded as .htaccess to replace the existing .htaccess file. This will enable him to execute scripts from the upload folder and compromise the server. Other sensitive files that can be overwritten include web.config, crossdomain.xml, global.asa, global.asax, clientaccesspolicy.xml etc.

On a server hosted on Windows platform, an attacker can make use of 8.3 filename support to overwrite sensitive files. An 8.3 filename is a filename convention used by old versions of DOS and Windows to name files, although it is even supported by the newer versions of Windows for backward compatibility. Under 8.3 filename convention, filenames consist of at most eight characters followed by a period '.' and an extension of at most three characters. Since under 8.3 filename support, web.config can be written as WEB~1.CON, an attacker can overwrite an existing web.config file by uploading a file named WEB~1.CON. This is particularly useful when the upload application blacklists certain filenames for upload files.

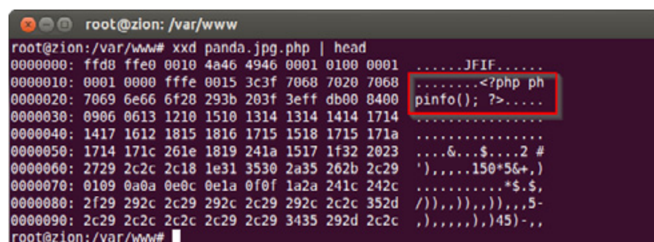

Figure 3. *PHP Code in a JPEG File*

Bypassing image header validation

File upload applications often try to validate image header to verify that the file uploaded is indeed an image, or where image files are required to be uploaded and then modified, for e.g., to be displayed as profile picture etc. This is typically done using the functions such as `getimagesize()` in PHP. If the header is valid, it returns the size of the image, otherwise it returns false. So if a malicious user tries to disguise a PHP file as a .png file by simply changing the file extension, this function will return false and he won't be able to upload the file.

However, this approach can be bypassed by inserting the PHP code in a valid image file. Image files can contain metadata information such as author, title, copyright, comments etc. which may contain arbitrary text. An image can be edited using an image editor such as Gimp or the command line jhead tool, to insert PHP code in the metadata.

```
jhead -cl "<?php phpinfo(); ?>" panda.jpg.php
```

Figure 1 shows the above PHP code in the image file. Such a technique usually makes use of `__halt_compiler();` after the code, which stops the compiler from parsing image data and interpreting it as code. This is done because if a `<?` appears in the following image data, the execution will break. Figure 2 shows `phpinfo()` being run from within the comment field of an image. Notice the first few bytes of the JPEG header being displayed as gibberish on the page.

Using Alternate Data Streams

On servers hosted on Windows based platforms with NTFS file systems, it is possible to bypass upload restrictions imposed by a blacklist based approach using *Alternate Data Streams* (ADS). ADS are file system based forks for NTFS file systems. These are used to store additional information with a file such as file access time, modification time or other metadata. A stream associated with a file is

referred as filename:streamname. Every file in a NTFS volume has at least one data stream called :$DATA, which contains the contents of the file. The following two command lines result in hello being inserted in the file test.txt.

```
echo hello > test.txt
echo hello > test.txt::$DATA
```

Therefore, a file test.txt can be referred as text. txt::$DATA as well. For the fact that blacklist based approaches typically check the last extension of the filename, it is possible for an attacker to use a filename such as shell.php::$DATA for the upload file, which will result in the file contents being written in shell.php. A file upload vulnerability was exploited in a similar way in FCKEditor 2.x.

Uploading a folder

On IIS 6, it is possible to execute code by uploading an allowed file type containing code inside a folder which ends with an executive extension such as .asp, e.g. from file folder.asp\file.txt. Besides, it is possible to create a folder on NTFS based servers using ADS. If the filename ends with ::$Index_Allocation or :$I30:$Index_Allocation, a folder will be created instead of a file. For e.g., if an attacker uses a filename test.asp::$Index_Allocation, a folder named test. asp will be created in the upload folder. This method can be used to bypass blacklist based approaches and is particularly useful when an attacker can later place a file in the newly created folder.

Other Risks

Exploiting file upload vulnerabilities enables an attacker to run arbitrary code on the server. Apart from code execution, there are plenty of things an attacker can do, depending upon his motive and the extent of his access ranging from cross-site scripting (XSS) to denial of service (DoS). Some of these have been enumerated below.

- Phishing: Upload fake login page
- Cross site scripting: Upload HTML files containing script that steal cookies
- Serve malware
- DoS: Consume server's hard drive by uploading a large number of files
- Upload trojan or virus
- Exploit local vulnerabilities on server such as image library flaws

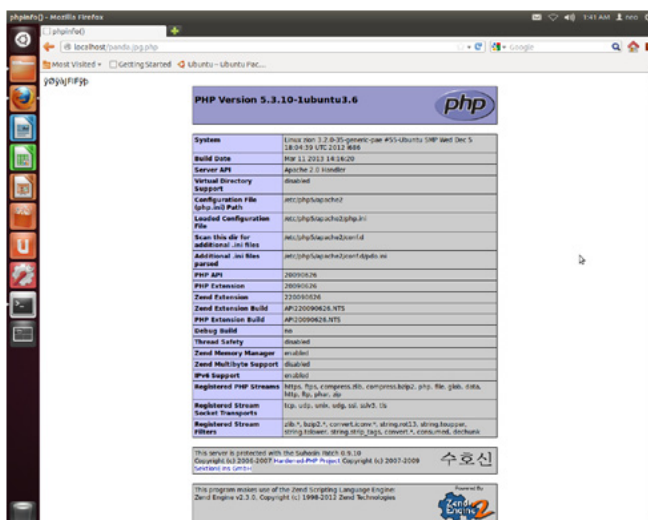

Figure 4. *phpinfo() in Action from within an Image File*

Mitigations

The following best practices can be enforced by the web applications to secure servers from exploitation of file upload vulnerabilities.

- Store the uploaded file outside the document root or in database.
- Do not rely on content-type request header or the file extension to identify file content.
- Compression can be used to store images.
- The names of the uploaded files can be randomized.
- Do not rely on client side validation.
- Place the .htaccess file in the parent directory and not in the upload directory.
- Restrict size of the upload files.
- Disable overwriting of existing files.

Conclusion

As seen above, there are several ways how a malicious user can bypass file upload form security. For this reason, when implementing a file upload form in a web application, one should make sure to follow correct security guidelines and test them properly. Unfortunately, to perform the number of tests required, can take a lot of time and require a good amount of web security expertise.

PANKAJ KOHLI

Pankaj works as a security consultant in one of the world's largest banks. In his free time, he likes to dig deep in programs trying to uncover vulnerabilities. He holds a Master of Science in Computer Science from IIIT, Hyderabad (India). Blog: http://www.codepwn.com.