PHP & Security: 3 Example Exploits

By Dennis Pallett

Introduction

One of the most common aspects of programming and scripting is security. Every program or script has to think about security, and each language has certain weak points. Actually, it isn't the language that has weak points - it's the programmer who creates weak points. PHP is no exception to this, and security should be at the top when you're creating a new script, no matter how simple or small the script may be.

I'm sure you've all read many basic PHP security articles, which include things like filtering user input, beware of XSS attacks, etc. That's why I'm not going to discuss them in this article. If you haven't heard about these things, have a look on Google. Heck, maybe I'll do a basic PHP security article in the near future.

In this article we're going to look a three different security cases. In each case I will try to explain the exploit, provide a working example, and then suggest possible fixes to prevent you from making the same mistake. Let's get cracking, shall we?

Security Case #1 - E-mail Injection

This is one of the neatest PHP exploits I've ever seen, and really requires some knowledge of how e-mail works. This exploit is often used by spammers to use contact forms on your websites to send out massive amounts of spam, without you even knowing. I used to be a victim of this as well, because I didn't even have a clue this existed. There was one thing I noticed though: really weird messages coming through my contact forms, for example:

```
aciov@yourdomain.com
 Content-Type: multipart/mixed; boundary=\"===============2145621685==\"
 MIME-Version: 1.0
 Subject: 9afb7555
 To: aciov@yourdomain.com
 bcc: mhkoch321@aol.com
 From: aciov@yourdomain.com

This is a multi-part message in MIME format.

--===============2145621685==
 Content-Type: text/plain; charset=\"us-ascii\"
 MIME-Version: 1.0
 Content-Transfer-Encoding: 7bit
```

wpbtizuum
--==========2145621685==--

As you can see this doesn't make much sense at all. But what it does mean is that spammers are trying to see if your contact form is open to e-mail injection.

This exploit works quite like SQL injection - untrusted users are able to inject data, because of poor input validation. When you use the mail() function you might think each argument is a separate thing, and they can't influence each other. But that's not the case, and the e-mail is actually one big long text message. For example, mail('me@example.com', 'My Subject', 'My Message', "From: contact@example.com\r\n";) is actually changed into:

To: me@example.com
Subject: My Subject
From: contact@example.com

My Message

The exploit happens when spammers are able to insert data into the e-mail, therefore being able to change the complete e-mail. For example, if your mail() function looks like this:

```
<?php
$email = $_POST['email'];
mail ('me@example.com', 'My Subject', 'My Message', "From: $email\\r\\n");
?>
```

As you can see an attacker can inject raw data into the e-mail. This means it's now possible to send a complete different e-mail with a new subject, message, and to header. Your contact form is used as an open relay!

To test this for yourself, have a look at the interactive demo. This doesn't actually send any e-mail, but does demonstrate how it's possible to change e-mails and use this exploit. Also have a look at http://securephp.damonkohler.com/index.php/Email_Injection for more information about E-mail Injection.

How do you protect against this exploit?
Easy - validate ALL input, and insert as little as possible into the e-mail. If you make sure you only get valid data from the user, the chance of e-mail injection has already been reduced hugely, and it probably won't be possibly any longer. Also see the above link for different solutions.

This is one of the sneakiest exploits around, but it is also very simple (and easy to forget about it as a developer!). The View Source exploit was the one that caused PHPit to be "hacked" recently.

It only works when you use a script to highlight or show the source of other PHP files, using the highlight_file() function or the fopen() function. These functions allow your visitors to view the source of other files, which is the whole point of this script. But you must also implement some security measures to prevent users from viewing files that they shouldn't see (such as database files, password files, etc). The best way of doing this is using a white-list array of filenames, and only allowing these files to be opened. This method is bullet-proof and will never fail.

But perhaps you want to restrict the script to only allow files from a certain directory, e.g. the 'demos' directory. To enforce this restriction, you'd probably use code like this:

```
if (strpos($file, 'demos') === false) {
```

```
        echo 'Security alert. Not a demo!';
} else {
        highlight_file ($file);
}
```

This checks whether the filename contains 'demos', and if so, displays the source. Seems fairly fail-proof, but that's where the exploit strikes. An attacker could include '..' in the filename, which means "go a directory up". So C:\program files\demos\..\ means the same as C:\program files\. This basically means that an attacker has unlimited access to everything on the server. Big whoops!

Just test it yourself, with the interactive demo:

Case 1: when trying an invalid filename, error is displayed
Case 2: when trying a demo filename, it shows the correct filename
Case 3: when using the exploit, no error is displayed, and it shows a forbidden filename

How do you protect against this exploit?
The easiest way is to check if the filename contains the two dots and display an error. So the above code becomes:

```
if (strpos($file, 'demos') === false OR strpos($file, '..') !== false) {
        echo 'Security alert. Not a demo!';
} else {
        highlight_file ($file);
}
```

UPDATE:
The above way isn't completely secury, and any file that contains 'demos' in the filename can still be viewed. If you really want to secure your view source script, don't allow a full path, and instead only allow a name. For example, instead of using '/home/phpit/public_html/demo/demo.php' simply use 'demo.php', and in your script add the full path. The only thing you have to worry about then is to filter out the two dots, and that's easy.

```
$file = '/home/phpit/public_html/demo/' . $_GET['filename'];

if (strpos($file, '..') !== false) {
        echo 'Security alert. Not a demo!';
} else {
        highlight_file ($file);
}
```

This would probably be enough to protect your view source script, and stop this exploit from working. You can also use dedicated source-viewing scripts, like phpViewSource.

A CSRF (Cross-Site Request Forgery) attack is really unique and interesting attack, but very uncommon, probably because it's so hard to pull off as a hacker. A malicious website, run by the hacker, causes the user ("you") to load a URL (of a different website) in the background, which causes a change on the server. Because the URL is loaded in your browser, it has all the credentials you have. So if you were logged into the admin control panel of that website, and the URL pointed to a deleteall.php file, everything could be deleted.

Pretty much anything is possible using CSRF, and attackers could make your browser do anything using clever JavaScript. But it's extremely hard to get all the circumstances right, and it hardly ever happens. However, to see it in action, check demo 3. For this to work, you must use Google Search History and be

logged into your account (see - even now it's already hard to pull off).

How do you protect against this exploit?
Protecting your scripts against CSRF attacks is extremely hard to do, and a really dedicated attacker will likely succeed anyway. But there are still steps you can take to make it harder.

First of all, make sure that any forms that change data (add/edit/delete) are POST only. GET requests should not be able to change anything.

Secondly, include a secret token with your forms, which also expire after a certain period of time. This will prevent almost any kind of CSRF attack, and is extremely different to counter.

Finally, don't worry too much about CSRF attacks. Although they are out there, it's likely you won't ever have to deal with it, and even so, it's an easy fix. If you're interested have a look on http://www.squarefree.com/securitytips/web-developers.html#CSRF for more information about CSRF attacks.

Conclusion

In this article I've given you three examples of exploits that can be used to abuse your PHP scripts. These are a lot more exploits, and I'm willing to bet there are more than 100 ways to attack a PHP script. Security isn't easy, but it is a necessity. That's why it should be your top priority.

I hope I've shown you something new about security in PHP, and that your contact forms and view source scripts are now secure!

About this PDF