

Web Application Security with PHP

By: Robert Lerner

April 11, 2009

With PHP? What do I mean “With PHP?” Well, simple: I'm addressing ways to aid in securing scripts written in PHP. Why do I ask “What do I mean”, (is this becoming recursive or what?), well some security experts may argue that PHP is an element in the entire spectrum of security related agendas, such as server software security, network security, and other important layers to this onion of lock down.

Well, let's cut the crap, probably the most important (and most heard of) is turning off the REGISTER_GLOBSALS ability in PHP. I have seen several security blogs, sites, and books say that “after version x, this is off by default”). While it may be, it is important to check. Some sites may release their own PHP compilations, of which all may not be up-to-par with PHP's latest security configurations. Register globals is a user-friendly setting that allows you the luxury of simply pulling variables from different arrays (such as \$_GET, \$_POST, and \$_COOKIE). For example, the script below shows a laughable way to get past a script:

```
<?PHP
IF ($_POST['password']=="mybigsecret")
    {
        $uservalid=1;
    }
IF ($userisvalid==1)
    {
        //Execute Code
    }
else
    {
        //Present Error Message and end
        die;
    }
?>
```

As you can see, the variable \$userisvalid controls whether or not the user can execute code, based on the POST array's “password” variable. Now this script is secure (assuming there's no other attack vectors) with register globals set to OFF.

But, with REGISTER_GLOBSALS on, you now have a rather dangerous (and exploitable) security concern.

A user visiting your script as follows can simply circumvent the entire validation by adding a well-written URI to the end of it:

```
http://www.mysite.com/password.php?userisvalid=1
```

The \$_GET array will contain the variable “userisvalid” which will be TRUE (or 1). REGISTER_GLOBSALS will simply take this variable, and throw it into your script as \$userisvalid.

Of course some additional caution must be taken when turning this off, especially if your scripts weren't developed by you. Some scripts must be changed to read the global array (GET, POST, etc), otherwise the system will just see this variable as null.

Browser Passed Variables

Are they reliable? Not at all. As a matter of fact, they can't be relied on at all. Most people wouldn't understand how to modify the HTTP_USER_AGENT (browser ID), the REMOTE_ADDR (user's IP), HTTP_REFERER (where they came from), or anything else that is meant to be "hidden". But, with those that possess the technical ability to get past these rather simple variable checks, they're bound to also know what to do with these changed variables. As a matter of fact, it is so simple to accomplish that you don't even need a browser to forge them. Simply use TELNET and request the page, all headers can be spoofed – browser type, whether or not the client accepts gzip'ed content or not, what page they came from, etc. The IP can be forged (to an extent) by simply using a proxy server or a proxy site. One thing to note as well, since I'm talking about IP addresses, is that by using the \$_SERVER['REMOTE_ADDR'] variable, you probably don't have the actual user's IP address. You'll find that if the user is on a local access network (LAN) that the IP simply reflects the IP of the modem, not of the actual computer. This presents an extreme issue when you're dealing with a large LAN that's part of a corporate network, schooling institution, or even a Denny's with Wi-Fi. AOL branded Internet Explorer presents the issue that the IP will never actually be the user's IP, but instead the IP of one of their servers – This actually can be an IP from a different part of the country, or even the world.

Okay, so why does the IP, REFERER, and browser matter so much? Well, it is usually misconstrued that they are concrete. You can use them on the scripts that post data to your site to see if off-site forms are actually posting to your site. While this may curtail any sort of attack, it will never eliminate it. Validation of these variables is encouraged (to an extent) as a complement to your other verification routines.

A common attack on Web Forums, Blogs, and other sites that do (and some that don't) require user login is an attack known as Cross-Site Request Forgery. XSRF or "Sea-Surf" are the industry-approved obfuscations to the name. This attack is extremely easy to implement.

How it works:

Simply put, pretend that you are banking with ABC Banking, and the script is so poorly written that this URL would actually do something:

```
http://www.abcbanking.com/?  
myaccount=123456&ttransferamt=10000&currency=USD&toaccount=654321
```

Well, that would transfer \$10,000 from your account to Mr.HappySlacks account simply by clicking the link.

Well, you reading this essay may say "Well, that's not going to happen – I ALWAYS hover over my links!"

Simply put, that won't do jack when it is put inside of image tags:

```
<img src='http://www.abcbanking.com/?  
myaccount=123456&transferamt=10000&currency=USD&toaccount=654321 '  
width='0px' height='0px' />
```

Or even Bulletin Board Code tags:

```
[img]http://www.abcbanking.com/?  
myaccount=123456&transferamt=10000&currency=USD&toaccount=654321[/img  
]
```

Sure the latter may show up as a broken image, but the former won't show anything. Simply by viewing the page you now executed this page through an image. Of course, being realistic, if your bank used this sort of validation then the FDIC would be bankrupt.

But what about this simple URL:

```
http://www.mysite.com/logoff.php
```

Well simply putting this URL in image tags and posting it in a forum, a message, etc will log the user off. This can present a huge issue in an inbox where the user cannot delete the message, because they're no longer logged in. This is in essence locking out the persons account, or at least their inbox.

How can I prevent this? Well, using the HTTP_REFERER variable WILL work here, because the chances are slim that the user will forge headers to force-logout themselves. Another way you can prevent this is adding a session ID to the end of the URL:

```
http://www.mysite.com/logoff.php?session=d41abcd1234efgh6789
```

What is worse is that this sort of attack can be performed *from another site!*

I can load a page on <http://www.example.com> that contains an image link back to that page, and it will get executed (note however that [example.com](http://www.example.com) will show up in the REFERER variable).

Combining both session IDs and REFERER checking will add additional security to the link.

Now, to leave the XSRF tangent that I was on, and going back to browser passed variables. Do not ever, and I mean NEVER use Javascript for any sort of "variable cleansing". Everything should always be cleaned as much as possible on the server side. As stated earlier, you can use TELENET to completely circumvent the Javascript validation, or even a browser that doesn't support Javascript.

The following are a few examples of variables that people typically don't think need to be checked: **Passwords** – Actually, this may be the worst variable to leave unprotected. You may say "Well I use MD5 to encrypt it in the database." But that *doesn't* protect you from an SQL injection attack, which can drop the entire table of users, or even list all of the users to the attacker – with the password.

Fixed-Length Inputs – The “maxlength” attribute of the INPUT tag is great for aesthetic control, preventing users from adding more than the expected character length in a field. The fact is that these limits can be removed extremely easy. There's a great extension for Firefox called the “Web Developer Toolbar”...One of the available options is “Remove MaxLength.” The server should validate the input as follows:

Say your input field is as follows:

```
<input type='text' name='data' maxlength='30' />
```

Your form handling script should have a provision that will enforce this restriction:

```
$data = substr($_POST['data'], 0, 30);
```

Cookies – Simply put, this one was overlooked by myself at one point. I would recommend only storing what characters you absolutely need to store in a cookie only. The best option is only numbers, only letters, or a combination. Cookies are easy to forge, and with the aforementioned add-on it is even easier. Using a REGEX expression with a function such as preg_replace, or ereg_replace, you can avoid it entirely.

If you use a MD5 string for sessions, for example, a great way to enforce this would be:

```
$cookie = substr(ereg_replace("[^A-Za-z0-9]", "",  
$_COOKIE['session']), 0, 32);
```

This simple (yet complicated looking) line of code will remove anything that is not a number or letter, and chop anything that is longer than 32 characters off the end of the variable. Because an MD5 checksum is composed of only alphanumeric characters, and is never shorter or longer than 32 characters, this guarantees that the input doesn't contain any sort of attack code (such as an SQL injection into your login sessions database).

Regular expressions are among the best options you have for cleaning the input from users.

The Rest – All input should be filtered, and a good option is the strip_tags() function. This removes HTML, Javascript, and other tags that can potentially break the layout or harm your users. Of course your individual requirements will vary depending on the content you are handling.

Why go through all of this work for security?

While it is a lot of work to implement good security practices, I'm sure you'll understand the efforts as fruitful as compared to losing your entire site, damaging the reputation of your employer (when you expose your companies information), etc

Introducing Security Logically

Face it, the best programmers make mistakes. The best way to test the security routines are to use a development server that has no connection to the priceless data that the routines are protecting. Using techniques discussed, such as validation, should be bombarded with exploits and attacks by *you first*. Once you're sure that the script is performing to-par with what is needed, and still functional for

the requirement of the rest of the script, then introduction will be a breeze. Of course not everybody has the luxury of using a full-blown development server with all of the data for a true testing experience. May I introduce my own personal trick, the "Test Container". The philosophy of a test container is simple. Use a script that is built cleanly (as in not in your current production scripts, however on your production server) that will allow you to test the security routines. For example, you can do the following:

```
<?PHP
$test = $_POST['test'];
//Your Variable cleansing routines here ($test = strip_tags($test);
for example)
if ($test!="")
    {
    echo "-" . $test . "-";
    }
?>
<form action="thisscript.php" method="post">
<input type='text' name='test' /><input type='submit' value='Go' />
</form>
```

You can now test the routines by using the page, and the content will appear above the input box after it is posted. I tend to add dashes in the programming so that I can also see if there are beginning or trailing spaces. Note that these aren't part of the \$test variable.

Cleansing Recursion and Non-Printable Characters

Luckily for us programmers, PHP's strip_tags() function allows us an easy way to remove tags from input. In addition to that, it also removes tags recursively.

A new programmer may try to develop their own script to remove data from input, however if it isn't recursive, it may be prone to a simple (and well-known) attack vector:

```
<img<br />src="http://www.nasty.com?image=shocking" />
```

What will occur here? Well, the trusty script will see the
 tag in the input, and as expected, it will remove it. But, since it only made this one pass over the input; guess what we'll have left?

```

```

Oops! Didn't plan for that, did we? Okay, we should probably make that a recursive function then, huh? So let's loop it and check for conformity. If the last pass went through validation, and is equal to the loop beforehand, then it must be good – right?

Wrong.

Adding Non-Printable Characters to this can potentially do the same thing. Strip_tags() is well provisioned for these attacks and will protect against it. If you want to go even farther, simply use regular expressions *in addition to* strip_tags(). This will eliminate the chance that your forum of travel

spots will be littered with shocking porn.

Independently Run Sites – Non Critical

A clause that I always add to my site's legal usage agreement is one of my own writing. I call it the "Undesirable Executable Situation and Disclosure Agreement." With this I invite my members, and guests to perform attacks against my sites (due to their non-critical nature). I'd never allow this sort of clause on a hospital, commercial, or banking site for example. What it generally states is that "If a member finds a loop-hole, exploit, or undesirable executable situation on the site, you agree to disclose the issue with the site owner (or administrator) and keep confidential the information until the exploit is repaired and the issue is sealed. You also agree that upon finding such a situation, that you will discontinue any additional exercises exploiting such situation(s)."

This does invite some issues, but call it a cheap education – it not only allows my guests a rare opportunity, but it also alerts me to new attack methods that were never used before, or problems I unintentionally introduced into the scripts. In addition it protects the information from getting published to other users – something that can have an outstanding impact when compounded by a barrage of users with ill intent.

Rainbow Tables and Salting

Many organizations use MD5 for encrypting their passwords. It is an industry-standard on most forums and other user-content sites. While it is true that you cannot take an MD5-encrypted string and convert it back to the original text, it can still be cracked. Enter "Rainbow Tables", the reverse-reference dictionary of MD5. With a Rainbow Table, you can simply get a MD5 hash, and enter it into a rainbow table, which will then search through potentially millions of hashes for an equal hash. It will then return the text that produced the hash (should it be available).

Try it yourself, go to Google and search for "Online MD5 Encryption". Note that searching "Rainbow tables" will simply return an endless charade of commercially available tables. While you're on these sites (all of which may not offer the table lookup), enter a hash into the table and see if you get a result...If you encrypted a standard English word (like "password" that some people use as their password) then there's a great chance you'll be getting a result. I strongly recommend that you do not use the site to perform an MD5 calculation though. Many of these sites will store the text that you just entered along with the hash into their own rainbow table, making your text easily accessible.

How can you prevent this? Well with an ingredient we call "salt". No, I don't recommend you open your server and pour salt all over the inside. Adding a salt can be as easy as this:

```
<?PHP
$password = "Hello";
$password = md5($password . "secretsalt");
?>
```

As you can see, the password is now going into the database as "Hellosecretsalt", which before was really easy to convert back to text, is now increasingly difficult. The text "secretsalt", or whatever you

prefer is called the salt. You will have to add this to your validation script when validating the user's password as well. Note also that changing the salt will in essence be changing the password that is entered, thus making it a different checksum. You will have to make a "change password" script that validates a password with an old salt (if there was one), and adds it to the new database with the new salt, should you ever need to change it.

MD5 x2 – Enhancing Security?

I've seen it time and time again, where programmers use double-md5'd strings thinking that it is more secure than doing it once. The fact is, it is actually detrimental to the security that is offered with MD5 (and a salt, of course).

Here's a newbie script with imagined security:

```
<?PHP
$password = md5(md5($password . "dontusethismethod"));
?>
```

The problem is, the first MD5 function (the inside one) will do as instructed and convert the password to the MD5 hash. This leaves you with 32 alphanumeric characters that the next MD5 will convert to another MD5 hash, based on the results of the first round.

Why is this bad, doesn't it prevent rainbow table attacks?

I wish it did. The fact is, MD5 actually uses the diversity of a long variable, special characters, and the length of the string all while calculating the hash. By hashing a string that is already hashed, you're removing a lot of the potential that MD5 offers. This same caution applies to many ciphers such as MD4, SHA1, etc.

If Session Identifiers are so Great, I'll use them all over!

This is a great idea – On paper. There are two leading reasons why you should only use session identifiers on certain pages:

- Search Engine Optimization techniques dictate that you should never have duplicate content on your site. It damages search engine rankings immensely. By having SIDs on the end of URLs that don't need them, every time GoogleBot comes to your site, it will see the page with the session ID on the end of it, and assume that this is a different page than the same page with a different session ID on it. This can also be detrimental to a user who sees said link in the SERP (search engine results page), and when they visit the page they find that the page won't load because their session ID isn't consistent with that presented in the URL from their search engine.
- URLs can be posted anywhere. If a user of your site decides that they like something that's posted, they may post the link to it on another site. If the session ID is attached to this link, somebody may pick up on that and assume control over your user's account.

SQL Injection

Because this topic is so extremely immense, I will only sprinkle a little information here about it. There are several good blogs and books relating to this subject, and if you're using a database system that uses the structured query language, they are must reads. Most of this goes back to the topic of input filtering.

For example, the following query may DROP an entire table, when you're looking to SELECT a table instead. Note that the `mysql_query()` function will not execute two queries as shown below.

Query:

```
$x = mysql_query("SELECT * FROM flowers.roses WHERE `roses.price` = '$price';)");
```

Without proper filtering, `$price` could be set as this:

```
1';DROP TABLE flowers.roses WHERE '1'='1
```

Simply put, without filtering this becomes:

```
$x = mysql_query("SELECT * FROM flowers.roses WHERE `roses.price` = '1';DROP TABLE flowers.roses WHERE '1'='1';)");
```

Well, ashes to ashes, dust to dust, you're entire flower store is now a bust.

The End

As always, I intend to add more to this as it occurs to me. I tend to leave out a few items unintentionally due to lack of sleep, or my general absent-mindedness. I hope that this page was written to inform a professional, while still being compatible with a new programmers mindset. A good saying is "Don't let security overcome you before let the lack thereof overcomes you." Yeah, I'm a bit proud of that.