Writing secure and portable PHP applications

This article was published by the Applied Thinking Group

You can read it online, here:
http://www.appliedthinking.org/articles/secure_and_portable_php/

Abstract:

This article outlines some simple rules of thumb to make your PHP applications more secure and more portable. These things are pretty important if you're going to let other people use your scripts.

An insecure script is bad enough, but if you're giving it out, then it's 100 times worse. If 50 people install your app, and someone notices the obvious hole, then those 50 people's servers may be at risk.

If you want 50 people to be able to run your app in the first place, you'll need to think about portability. Does your script require register_globals? autoquoting? short tags? Everybody's PHP installation differs, and you'll need to try and cater for as many setups as possible.

Use Long Tags

The first step to making your PHP applications portable to to use long PHP tags. That is, use '<?php' instead of '<?'. This means your previous short write tags:

```
<?=$foo?>
```

must now be:

```
<?php echo $foo; ?>
```

Note: The last semicolon in any PHP block is optional. I prefer to always include it, but this is a personal preference.

It's annoying, but it's sensible. Some people don't have short tags turned on (they can be annoying if you also want XML declarations in your files) but PHP always supports the long tag versions.

Use E_ALL On Your Server

PHP allows you to set different levels of error reporting in your scripts. You should always use E_ALL when building applications, to show all errors and notices. To enable E_ALL, edit your php.ini file (ask your sysadmin about this), or add the following line at the top of your script:

```
error_reporting(E_ALL);
```

With E_ALL turned on, you're likely to get a lot of errors. But take the time to fix them — new PHP installations come with E_ALL on by default. The main errors you'll come across are:

1. Initialising variables
2. Hash keys that don't exist
3. Bare strings

Initialising variables is easy. Consider the following piece of code:

```
if ($HTTP_GET_VARS['password'] == 'foo') {
 $admin_ok = 1;
}
```

Looks fine, but $admin_ok hasn't been initialised. If register_globals were turned on, someone could pass a value in for $admin_ok and your script would think they had gotten the password right. It's very easy to fix:

```
$admin_ok = 0;
if ($HTTP_GET_VARS['password'] == 'foo') {
 $admin_ok = 1;
}
```

Which then can be cleaned up into a single statement:

```
$admin_ok = ($HTTP_GET_VARS['password'] == 'foo')?1:0;
```

You have learned about the ternary operator, right?

With E_ALL turned on, you need to check hash keys exist before using them. Consider this code:

```
$foo = $HTTP_POST_VARS['foo'];
```

If 'foo' wasn't passed as a form field, then the key doesn't exist in the $HTTP_POST_VARS hash. What you need to do is check the key exists before referencing it:

```
$foo = 0;
if (array_key_exists('foo', $HTTP_POST_VARS)){
 $foo = $HTTP_POST_VARS['foo'];
}
```

Than can get quite tedious, quite quickly, especially for $HTTP_GET_VARS and $HTTP_POST_VARS, so I like to construct a little routine:

```
function get_http_var($name, $default=''){
 global $HTTP_GET_VARS, $HTTP_POST_VARS;
 if (array_key_exists($name, $HTTP_GET_VARS)){return $HTTP_GET_VARS[$name];}
 if (array_key_exists($name, $HTTP_POST_VARS)){return $HTTP_POST_VARS[$name];}
 return $default;
}
```

Note: I get an email every so often telling me that i should be using $_POST instead of $HTTP_POST_VARS (etc.) since it's a superglobal and the $HTTP_* form is deprecated. This article is entitled "Writing secure and **portable** PHP applications". The superglobals weren't introduced until PHP 4.1.0, so early versions of PHP (which alot of poeple have to live with) wont support this new syntax.

We'll revisit that function again a little later - you'll find it quite useful.

Thirdly, barewords. This is very simple. Instead of this:

```
$foo[bar]
```

Write this:

```
$foo['bar']
```

But don't worry about it when it's in an interpolated string:

```
$baz = "$foo[bar]";
```

Don't Use register_globals

register_globals is evil. Well, actually, when used with E_ALL, it's not really that evil if you are careful about variable initialisation. But they present a portability issue. Turn off register_globals in your php.ini or put a .htaccess file on your server with the following line in:

```
php_value register_globals off
```

If you build your application with register_globals off, using the get_http_var() function above, then it'll be nice and secure (E_ALL will pick up on any uninitialised variables) and will also run fine on servers with register_globals on. But BEWARE — you should always have it disabled while developing your application.

Don't Use Auto Slashing

The other evil options that are sometimes enabled are the annoying magic_quotes_gpc and the deadly magic_quotes_runtime. magic_quotes_gpc causes the GET, POST and COOKIE variables to have AddSlashes() run on

them before your script runs. magic_quotes_runtime is even more dangerous, doing it during the running of your script. We'll deal with them one by one.

magic_quotes_runtime can be battled easily enough. Since it has no effect until after your script is running, you can simply disable it at the top of your script:

```
ini_set("magic_quotes_runtime", 0);
```

magic_quotes_gpc is trickier, since it has already affected your variables by the time your script runs. To battle this, you'll have to detect for it and undo its actions. We can do this by adding a little magic to our get_http_var() function:

```
function get_http_var($name, $default=''){
 global $HTTP_GET_VARS, $HTTP_POST_VARS;
 if (array_key_exists($name, $HTTP_GET_VARS)){return
clean_var($HTTP_GET_VARS[$name]);}
 if (array_key_exists($name, $HTTP_POST_VARS)){return
clean_var($HTTP_POST_VARS[$name]);}
 return $default;
}

function clean_var($a){
 return (ini_get("magic_quotes_gpc") == 1)?recursive_strip($a):$a;
}

function recursive_strip($a){
 if (is_array($a)) {
  while (list($key, $val) = each($a)) {
   $a[$key] = recursive_strip($val);
  }
 }else{
  $a = StripSlashes($a);
 }
 return $a;
}
```

We use a recursive strip function since HTTP variables can contain arrays and the normal StripSlashes() function doesn't recurse structures.

Of course, now that you don't have auto-slashed variables, you MUST always call AddSlashes() on your variables before constructing an SQL statement. A good rule of thumb is to make sure ALL variables have been escaped, even if you're sure they wont contain quotes. All it takes is one unescaped variable and your whole server can be compromised.

If you're using cookies you'll also want to add a cookie-getting function:

```
function get_cookie_var($name, $default=''){
 global $HTTP_COOKIE_VARS;
 if (array_key_exists($name, $HTTP_COOKIE_VARS)){return
clean_var($HTTP_COOKIE_VARS[$name]);}
 return $default;
}
```

That just about wraps up our handy functions, but we're not quite done with the rules yet.

Write Variables Out With HtmlEntities()

Whenever you output something to the browser, consider its content. If you don't know exactly what's in a variable, you should probably be using HtmlEntities() to make sure it won't break. Consider this code, which is part of a form to edit a database record:

```
<input type="text" name="foo_bar" value="<?php echo $row['foo_bar']; ?>"><br>
```

This is fine, UNLESS $row['foo_bar'] contains the string "><img src="/images/woo.gif

To make sure that the variable is displayed as expected, wrap it with HtmlEntities():

```
<input type="text" name="foo_bar" value="<?php echo
htmlentities($row['foo_bar']); ?>"><br>
```

That's all for my main security and portability rules. There are lots of other things you should be doing to make sure you produce quality code, but these are a good start. If you have any good tips which should be included here, mail me.

Postscript

Thanks to Matt Jacob for pointing out all my spelling and formatting mistakes.