PHP Security – The Big Picture

by Andrew J. Bennieston

Web Security

Whether your site is the web presence for a large multinational, a gallery showing your product range and inviting potential customers to come into the shop, or a personal site exhibiting your holiday photos, web security matters. After the hard work put in to make your site look good and respond to your users, the last thing you want is for a malicious hacker to come along, perform a PHP hack and break it somehow.

There are a number of problems in web security, and unfortunately not all of them have definite solutions, but here we'll look at some of the problems that should be considered every time you set out to write a PHP script to avoid a PHP hack attack. These are the problems which, with well-designed code, can be eliminated entirely. Before looking in detail at the solutions, though, lets take a moment to define the problems themselves.

SQL Injection

In this attack, a user is able to execute SQL queries in your website's database. This attack is usually performed by entering text into a form field which causes a subsequent SQL query, generated from the PHP form processing code, to execute part of the content of the form field as though it were SQL. The effects of this attack range from the harmless (simply using SELECT to pull another data set) to the devastating (DELETE, for instance). In more subtle attacks, data could be changed, or new data added.

Directory Traversal

This attack can occur anywhere user-supplied data (from a form field or uploaded filename, for example) is used in a filesystem operation. If a user specifies "../../../../../../etc/passwd" as form data, and your script appends that to a directory name to obtain user-specific files, this string could lead to the inclusion of the password file contents, instead of the intended file. More severe cases involve file operations such as moving and deleting, which allow an attacker to make arbitrary changes to your filesystem structure.

Authentication Issues

Authentication issues involve users gaining access to something they shouldn't, but to which other users should. An example would be a user who was able to steal (or construct) a cookie allowing them to login to your site under an Administrator session, and therefore be able to change anything they liked.

Remote Scripts (XSS)

XSS, or Cross-Site Scripting (http://www.acunetix.com/websitesecurity/cross-site-scripting/) (also sometimes referred to as CSS, but this can be confused with Cascading Style Sheets, something entirely different!) is the process of exploiting a security hole in one site to run arbitrary code on that site's server. The code is usually included into a running PHP script from a remote location.

This is a serious attack which could allow any code the attacker chooses to be run on the vulnerable server, with all of the permissions of the user hosting the script, including database and filesystem access.

Processing User Data – Form Input Verification & HTML Display

Validating Input And Stripping Tags

When a user enters information into a form which is to be later processed on your site, they have the power to enter anything they want. Code which processes form input should be carefully written to ensure that the input is as requested; password fields have the required level of complexity, e-mail fields have at least some characters, an @ sign, some more characters, a period, and two or more characters at the end, zip or postal codes are of the required format, and so on.

Each of these may be verified using regular expressions, which scan the input for certain patterns. An example for e-mail address verification is the PHP code shown below. This evaluates to true if an e-mail address was entered in the field named 'email'.

```
preg_match('/^.+@.+\..{2,3}$/',$_POST['email']);
```

This code just constructs a regular expression based on the format described above for an e-mail address. Note that this will return true for anything with an @ sign and a dot followed by 2 or 3 characters. That is the general format for an e-mail address, but it doesn't mean that address necessarily exists; you'd have to send mail to it to be sure of that.

Interesting as this is, how does it relate to security? Well, consider a guestbook as an example. Here, users are invited to enter a message into a form, which then gets displayed on the HTML page along with everyone else's messages. For now, we won't go into database security issues, the problems dealt with below can occur whether the data is stored in a database, a file, or some other construct.

If a user enters data which contains HTML, or even JavaScript, then when the data is included into your HTML for display later, their HTML or JavaScript will also get included.

If your guestbook page displayed whatever was entered into the form field, and a user entered the following,

```
Hi, I <b>love</b> your site.
```

Then the effect is minimal, when displayed later, this would appear as,

Hi, I **love** your site.

Of course, when the user enters JavaScript, things can get a lot worse. For example, the data below, when entered into a form which does not prevent JavaScript ending up in the final displayed page, will cause the page to redirect to a different website. Obviously, this only works if the client has JavaScript enabled in their browser, but the vast majority of users do.

```
Hi, I love your site. Its great!<script
language=”JavaScript”>document.location=”http://www.acunetix.com/”;</script>
```

For a split second when this is displayed, the user will see,

```
Hi, I love your site. Its great!
```

The browser will then kick in and the page will be refreshed from www.acunetix.com. In this case, a fairly harmless alternative page, although it does result in a denial of service attack; users can no longer get to your guestbook.

Consider a case where this was entered into an online order form. Your order dispatchers would not be able to view the data because every time they tried, their browser would redirect to another site. Worse still, if the redirection occurred on a critical page for a large business, or the redirection was to a site containing objectionable material, custom may be lost as a result of the attack.

Fortunately, PHP provides a way to prevent this style of PHP hack attack. The functions strip_tags(), nl2br() and htmlspecialchars() are your friends, here.

strip_tags() removes any PHP or HTML tags from a string. This prevents the HTML display problems, the JavaScript execution (the <script> tag will no longer be present) and a variety of problems where there is a chance that PHP code could be executed.

nl2br() converts newline characters in the input to
 HTML tags. This allows you to format multi-line input correctly, and is mentioned here only because it is important to run strip_tags() prior to running nl2br() on your data, otherwise the newly inserted
 tags will be stripped out when strip_tags() is run!

Finally, htmlspecialchars() will entity-quote characters such as <, > and & remaining in the input after strip_tags() has run. This prevents them being misinterpreted as HTML and makes sure they are displayed properly in any output.

Having presented those three functions, there are a few points to make about their usage. Clearly, nl2br() and htmlspecialchars() are suited for output formatting, called on data just before it is output, allowing the database or file-stored data to retain normal formatting such as newlines and characters such as &. These functions are designed mainly to ensure that output of data into an HTML page is presented neatly, even after running strip_tags() on any input.

strip_tags(), on the other hand, should be run immediately on input of data, before any other processing occurs. The code below is a function to clean user input of any PHP or HTML tags, and works for both GET and POST request methods.

```
function _INPUT($name)
{
if ($_SERVER['REQUEST_METHOD'] == 'GET')
return strip_tags($_GET[$name]);
if ($_SERVER['REQUEST_METHOD'] == 'POST')
return strip_tags($_POST[$name]);
}
```

This function could easily be expanded to include cookies in the search for a variable name. I called it _INPUT because it directly parallels the $_ arrays which store user input. Note also that when using this function, it does not matter whether the page was requested with a GET or a POST method, the code can use _INPUT() and expect the correct value regardless of request method. To use this function, consider the following two lines of code, which both have the same effect, but the second strips the PHP and HTML tags first, thus increasing the security of the script.

```
$name = $_GET['name');
$name = _INPUT('name');
```

If data is to be entered into a database, more processing is needed to prevent SQL injection, which will be discussed later.

Executing Code Containing User Input

Another concern when dealing with user data is the possibility that it may be executed in PHP code or on the system shell. PHP provides the eval() function, which allows arbitrary PHP code within a string to be evaluated (run). There are also the system(), passthru() and exec() functions, and the backtick operator, all of which allow a string to be run as a command on the operating system shell.

Where possible, the use of all such functions should be avoided, especially where user input is entered into the command or code. An example of a situation where this can lead to attack is the following command, which would display the results of the command on the web page.

```
echo 'Your usage log:<br />';

$username = $_GET['username'];

passthru(“cat /logs/usage/$username”);
```

passthru() runs a command and displays the output as output from the PHP script, which is included into the final page the user sees. Here, the intent is obvious, a user can pass their username in a GET request such as usage.php?username=andrew and their usage log would be displayed in the browser window.

But what if the user passed the following URL?

```
usage.php?username=andrew;cat%20/etc/passwd
```

Here, the username value now contains a semicolon, which is a shell command terminator, and a new command afterwards. The %20 is a URL-Encoded space character, and is converted to a space automatically by PHP. Now, the command which gets run by passthru() is,

```
cat /logs/usage/andrew;cat /etc/passwd
```

Clearly this kind of command abuse cannot be allowed. An attacker could use this vulnerability to read, delete or modify any file the web server has access to. Luckily, once again, PHP steps in to provide a solution, in the form of the escapeshellarg() function. escapeshellarg() escapes any characters which could cause an argument or command to be terminated. As an example, any single or double quotes in the string are replaced with \' or \", and semicolons are replaced with \;. These replacements, and any others performed by escapeshellarg(), ensure that code such as that presented below is safe to run.

```
$username = escapeshellarg($_GET['username']);
passthru(“cat /logs/usage/$username”);
```

Now, if the attacker attempts to read the password file using the request string above, the shell will attempt to access a file called “/logs/usage/andrew;cat /etc/passwd”, and will fail, since this file will almost certainly not exist.

It is generally considered that eval() called on code containing user input be avoided at all costs; there is almost always a better way to achieve the desired effect. However, if it must be done, ensure that strip_tags has been called, and that any quoting and character escapes have been performed.

Combining the above techniques to provide stripping of tags, escaping of special shell characters, entity-quoting of HTML and regular expression-based input validation, it is possible to construct secure web scripts with relatively little work over and above constructing one without the security considerations. In particular, using a function such as the _INPUT() presented above makes the secure version of input acquisition almost as painless as the insecure version PHP provides.

How to Check for PHP Vulnerabilities

The best way to check whether your web site & applications are vulnerable to PHP hack attacks is by using a Web Vulnerability Scanner. A Web Vulnerability Scanner (http://www.acunetix.com/vulnerability-scanner/) crawls your entire website and automatically checks for vulnerabilities to PHP attacks. It will indicate which scripts are vulnerable so that you can fix the vulnerability easily. Besides PHP security vulnerabilities, a web application scanner will also check for SQL injection (http://www.acunetix.com/websitesecurity/sql-injection/), Cross site scripting (http://www.acunetix.com/websitesecurity/cross-site-scripting/) & other web vulnerabilities.

Click here to read part two (http://www.acunetix.com/websitesecurity/sql-security).

Subscribe for Updates

Enter E-Mail Subscribe

Learn More

SQL Injection (http://www.acunetix.com/websitesecurity/sql-injection/)

Cross-site Scripting (http://www.acunetix.com/websitesecurity/cross-site-scripting/)

Web Site Security (http://www.acunetix.com/websitesecurity/web-site-security/)

Directory Traversal (http://www.acunetix.com/websitesecurity/directory-traversal/)

AJAX Security (http://www.acunetix.com/websitesecurity/ajax/)

Troubleshooting Apache (http://www.acunetix.com/websitesecurity/troubleshooting-tips-for-apache/)

WordPress Security (http://www.acunetix.com/websitesecurity/wordpress-security-top-tips-secure-wordpress-application/)

Find Us on Facebook

Acunetix

Like 9,742

Product Information

HTML5 Security (http://www.acunetix.com/vulnerability-scanner/html5-website-security/)

AcuSensor Technology (http://www.acunetix.com/vulnerability-scanner/acusensor-technology/)

DeepScan Technology (http://www.acunetix.com/vulnerability-scanner/crawling-html5-javascript-websites/)

Blind XSS Detection (http://www.acunetix.com/vulnerability-scanner/acumonitor-blind-xss-detection/)

Network Security Scanning (http://www.acunetix.com/vulnerability-scanner/network-security-scanner/)

Website Security

Cross-site Scripting (http://www.acunetix.com/websitesecurity/cross-site-scripting/)

SQL Injection (http://www.acunetix.com/websitesecurity/sql-injection/)

DOM-based XSS (http://www.acunetix.com/websitesecurity/improving-dom-xss-vulnerabilities-detection/)

CSRF Attacks (http://www.acunetix.com/websitesecurity/csrf-attacks/)

Directory Traversal (http://www.acunetix.com/websitesecurity/directory-traversal/)

Learn More

PHP Security (http://www.acunetix.com/websitesecurity/php-security-1/)

Web Service Security (http://www.acunetix.com/websitesecurity/web-services-wp/)

WordPress Security (http://www.acunetix.com/websitesecurity/wordpress-security-top-tips-secure-wordpress-application/)

AJAX Application Security (http://www.acunetix.com/websitesecurity/ajax/)

PCI-DSS Compliance (http://www.acunetix.com/websitesecurity/pci-compliance-wp/)

Documentation

Manual (http://www.acunetix.com/support/)

FAQs (http://www.acunetix.com/support/faq/)

Web Vulnerabilities (//acunetix.com/vulnerabilities/web/)

Network Vulnerabilities (//www.acunetix.com/vulnerabilities/network/)

Trojans and Backdoors (//www.acunetix.com/vulnerabilities/trojans/)

© Acunetix | 2015 (http://www.acunetix.com)

About (http://www.acunetix.com/company/) Acunetix Online Login (https://ovs.acunetix.com)

Pen-Testing Tools (http://www.acunetix.com/vulnerability-scanner/pen-testing-tools/)

Web Application Security (http://www.acunetix.com/vulnerability-scanner/web-application-security/)

JavaScript Security (http://www.acunetix.com/vulnerability-scanner/javascript-html5-security/)

HIPAA Compliance (http://www.acunetix.com/websitesecurity/hipaa-compliance-standards/)