THE

# USUAL SUSPECTS

SECURING PHP

# Securing PHP: The Usual Suspects

Chris Cornutt

This book is for sale at http://leanpub.com/securingphp-usualsuspects

This version was published on 2014-07-08


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Tweet This Book!

Please help Chris Cornutt by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just picked up my copy of ”Securing PHP: The Usual Suspects”, check it out!
http://bit.ly/sphp-usualsuspects

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#

## Also By Chris Cornutt

Securing PHP: Core Concepts

# Contents

**Injection** . . . . . **1**
SQL Injection . . . . . 1
Path injection . . . . . 4
Code injection . . . . . 7
Command injection . . . . . 7
XML injection . . . . . 9
HTML5 Injection . . . . . 10

# Injection

**OWASP T10 A1: Injection**

At the very top of the OWASP list is one of the most common problems for any kind of web application, *injection*.

## Input Filtering & Validation

While the different types of injection can have different impacts depending on what context their executed in (ex. SQL versus XML) there is one thing - well, two - that can help prevent the problem: proper input filtering and validation. Much like many of the other things on the Top Ten list, even a little bit of filtering can go a long way when dealing with incoming data, regardless of the source.

It's surprising how many PHP applications and libraries don't take the time to implement even a basic level of filtering into their applications. Granted, being new to the language causes some of this, but it's also due to PHP's lack of a built-in filtering or validation mechanism. Some other language actually has objects you can pull the data from where filtering and data validation have already been applied. PHP makes the superglobals directly available with no filtering what so ever. Developers, especially those just starting out, see how easy it is to pull data from `$_GET` or `$_POST` and don't ever think about the malicious data they could be working with.

I'm not going to get into all of the methods of filtering and validation and best practices around it here as it does relate to many of the other in the Top Ten besides this one. Instead I've added an [an entire appendix]{#appendixa} around these ideas and some tools and methods you can use. It's a whole topic in and of itself and I don't want to keep these chapters focused.

So, let's move on and talk some about different kinds of injections including the all too common SQL, XML and command injection.

# SQL Injection

When most people (well, development people) hear the words "security" and "injection" in the same sentence, their mind usually jumps to *SQL injection*. It's become such a hot topic in development security and is becoming more and more well known. This is a definite plus but, unfortunately, it hasn't reduced how often it occurs.

So, what is *SQL injection* specifically? Well, put plainly is the insertion of either an entire or partial SQL statement into a piece of software due to unfiltered input. The most common example of this in PHP applications usually happens with a combination of two factors: direct use of unfiltered variables and concatenation. Here's a quick example:

```
<?php

$sql = "SELECT id, username FROM user where username = '".$_GET['username']."'";

?>
```

Can you spot the problem here? Let me make it a bit clearer with an example value:

```
<?php

$_GET['username'] = "' or 1=1; #";
$sql = "SELECT id, username FROM user where username = '".$_GET['username']."'";

echo 'Generated SQL: '.$sql;

?>
```

In this example, the generated SQL string ends up being `SELECT id, username FROM user where username = '' or 1=1; #` which, in all cases, will make the return `0` (a non-false value). The reason for this is that the injected string causes the where to use the `or 1=1` along with the `username` evaluation. The `; #` tells the SQL interpreter that you've finished the statement (the semi-colon) and the rest of the line should be considered a comment (the hash) so anything after that might have been in the where is effectively removed.

Here's an even more fleshed out example showing why this could be a *very* bad thing:

```
<?php

/**
 * We're going to check if a user exists by selecting them from the database
 */
$pdo = new PDO('mysql:dbname=test;host=127.0.0.1', 'test', 'foobarbaz');
$found = $pdo->exec("SELECT id, username FROM user where username = '".$_GET['username'\
]."'");

if ($found !== false) {
        echo 'Username "'.$_GET['username'].'" found!';
}

?>
```

If the attacker was to figure out that you're searching by username (maybe you have a `username` parameter in the URL) and was able to get the string `' or 1=1; #` past your filtering, the results of the `if` check would always be true. So that's slightly dangerous, but imagine what would happen if it was querying permissions:

```
<?php

/**
 * We're going to check if a user has the "admin" permission
 */
$pdo = new PDO('mysql:dbname=test;host=127.0.0.1', 'test', 'foobarbaz');

$found = $pdo->exec(
        "SELECT id, permission FROM permissions where permission = 'admin' and username = '".$\
_GET['username']."'"
);

if ($found !== false) {
        echo 'Username "'.$_GET['username'].'" has the admin permission!';
}

?>
```

See the problem? The script only checks for a `not false` value and with the SQL injection, the attacker can easily make that happen. If this `if` check was being done to see if a user could access the administration part of your application, you'd be in big trouble.

So, how can we fix this? A SQL injection vulnerability could cause irreparable damage if left uncorrected. Thankfully, there's an easy way to help fix the issue caused by the dangerous combination of unfiltered data and concatenation - prepared statements. First, an example:

```
<?php

/**
 * We're going to check if a user has the "admin" permission
 */
$pdo = new PDO('mysql:dbname=test;host=127.0.0.1', 'test', 'foobarbaz');
$stmt = $pdo->prepare("SELECT id, permission FROM permissions where permission = ? and \
username = ?");

$stmt->execute(array('admin', $_GET['username']));
$found = $stmt->fetch();

if ($found !== false) {
        echo 'Username "'.$_GET['username'].'" has the admin permission!';
}

?>
```

Unlike in our previous example that just appended the `$_GET['username']` value to the SQL, we're making use of the *prepared statement* handling in PHP's PDO functionality.

*Prepared statements* take in a SQL statement with placeholders and "prepare" it for use, hence the name. When the SQL is executed the values given are substituted in for the placeholders, not concatenated.

Using prepared statements means that strings like `' or 1=1; #` will have no effect, or at least not the effect it had before when we were just appending things together. Instead, that literal value will be used in the evaluation in the `where` statement and not match anything. There are some caveats with using prepared statements, however. One limitation is that you can only replace one placeholder at a time. This makes it a little extra work to use arrays and things like `IN()` but it's well worth the trouble.

Many popular frameworks have created their database handling with SQL injection prevention as a focus. They've all opted to use the PDO handling to reduce the risk for you and your application. If you're investigating a new library or framework, be sure you check to be sure they're using prepared statements and not trying to escape things with any too-clever rules.

Prepared statements can prevent issues around SQL injections, but they don't prevent bad data from getting into your application. You still need to *filter and validate* your incoming data before storing it in the database.

## Path injection

Another context when thinking about possible injection methods in your application relates to the local file system. Where SQL injection is the push of a partial SQL statement into your data, *path injection* (or *directory traversal*) relates to directory paths and files. With path injection vulnerabilities, the user is able to inject a crafted path into the input and is either able to overwrite or create a file on your local system. One popular use of this is to upload a "webshell" script that looks something like:

```
1  <?php exec($_GET['cmd']); ?>
```

If the user is able to upload that anywhere into your application's web-accessible directories, it basically gives them free reign on your system. Writing files is only half of the equation, though. Depending on what the script is doing, the malicious user might be able to read local system files if they're not correctly locked down.

Let's look at a (very) simple example of a *path injection* vulnerability. This is a pretty common thing to see in a script and is usually used to load separate "sub-sections" of a site. For example, it might use a URL like `foo.com/index.php?path=test.php` to pull in the contents of the `test.php` page.

```
1  <?php
2
3  $path = $_POST['path'];
4  echo file_get_contents('../pages/'.$path);
5
6  ?>
```

Can you spot the problem? While this might seem like a quick and easy way to be able to include whatever files you might need from your `pages/` directory, there's a hidden problem here. This example assumes that what they're looking for is in the `pages/` directory and tries to load it based on a concatenated path. So, what happens if the attacker uses something like this for the `path` value: `../../../../../../../etc/passwd`? Obviously it depends on where your application is as to if this exact example would work or not, but it gives you the idea. Because the attacker could potentially define their

own path to concatenate, they can use it to read anything the web server user has access to. In this case, it would be dumping the `/etc/passwd` file with plenty of user information.

Before I get into the fix for this kind of vulnerability, it's also worth mentioning that there's another commonly used piece of PHP functionality that could be used in an attack like this. The `include` and `require` family of methods make it simple to include other files inside your application. A common example is the autoloading functionality in frameworks that pull in class files are they're needed. Usually, these class files and paths are handled internal to the application, but what if you were getting part of the file name from the user input?

```
<html>
        <head><title>I'm vulnerable</title></head>
        <body>
        <?php include_once $_GET['file']; ?>
        </body>
</html>
```

You can see there's a similar issue as there was with the `path` example above. With a well-crafted input, the end user could include just about any content into the page. Obviously, this example is a little contrived since you'd *never* want to blindly include a path from the URL into our page.

I've outlined the problem in two different places, now lets see how we can fix it. Thankfully the fix(es) are relatively easy. First off, the same fix that can help with all injection vulnerabilities, validation of incoming data:

```
<?php

// Ensure they're not trying anything sneaky with directories
if (strstr($_GET['path'], '../') !== false) {
        throw new \Exception("Path traversal attempt!");
}

// There's also remote file inclusions to check for
if (strstr($_GET['path'], 'file://') !== false) {
        throw new \Exception("Remote file inclusion attempt!");
}

?>
```

This is all well and good, but there's another stronger method you can use to prevent the user from reaching outside of the document root and getting at files they shouldn't. This method, *directory restriction* can be achieved in two ways. One way is through the open_basedir setting in the `php.ini` configuration. This setting defines the root of the file system where PHP is allowed to access. For example, it could be set like `open_basedir = /var/www/securingphp/` to restrict the application just to that directory. The `open_basedir` setting also allows for multiple directories as well like `open_basedir = /var/www/securingphp/:/tmp/securingphp-data`.

There's two problems with this method, unfortunately. First, you'll need access to change the `php.ini` file to set this configuration (not always possible, especially on shared hosting) and second, the value is hard-coded and can't be adjusted on the fly. Consider this situation: your users can upload files to your

application, but you make a custom directory for them based on their username (outside of the document root, of course). You want to restrict the current user from trying anything funny and accessing another user's directory. To be able to handle this, you need something a bit more "user land" and code based like:

```
<?php

$tmpPath = $_FILES['file']['tmp_name'];
$fileName = $_FILES['file']['name'];
$username = 'ccornutt';

$moveToPath = '/var/www/user-uploads/'.$username.'/'.$fileName;
move_uploaded_file($_FILES['file']['tmp_path'], $moveToPath);

?>
```

In our above example, we're pulling in the information from a form-based file upload. The code uses the `tmp_name` and `name` values along with the move_uploaded_file function to put the uploaded file in the right place. Unfortunately, there's no validation happening here, so what might happen if, say, a malicious user somehow got something like `../../../../../tmp/test/foo.txt` into the file `name` value. If you're not checking things correctly, you could leave a hole that would allow users to write files to any web-writable directory on the system (maybe even other user directories). The move_uploaded_file function does nothing to prevent it being written to the wrong directory.

This needs to be fixed, but how? Here's one example that catches when the input contains the directory traversal attempt:

```
<?php

$tmpPath = $_FILES['file']['tmp_name'];
$fileName = $_FILES['file']['name'];
$username = 'ccornutt';

// Check for the directory traversal
if (preg_match('#\.\./#', $_FILES['file']['name']) !== false) {
        throw new \Exception('Directory traversal attempt!');
}

$moveToPath = '/var/www/user-uploads/'.$username.'/'.$fileName;
move_uploaded_file($_FILES['file']['tmp_path'], $moveToPath);

?>
```

This simple validation prevents the `../` value from being in the file name, preventing this form of directory traversal issue.

One note before we leave the path/directory injection section about an actual PHP bug. There was an issue in PHP versions <= 5.3.6 with file uploads where the path could be injected. It was corrected quickly and new versions were released with the fix, but it's something to stay aware of. See this article for more details.

## Code injection

*Code injection* has one main cause - eval. The `eval` function allows you to input a string and have it executed as PHP. Pair this with unfiltered user input, and you have a recipe for disaster. Take the following example:

```
1  <?php
2
3  eval('include '.$_GET['path']);
4
5  ?>
```

This is a pretty basic example (and hopefully never done) but it shows the most basic vulnerability that could happen. Most usages of PHP's `eval` are more complex than this, but I wanted to be sure you knew the fundamentals first. In this code, we're passing in a `path` from the URL to an `include` inside the `eval`. With the user able to control this `path` value completely, they could include any file in the system they might want to.

It's not just for local file inclusion (LFI) either. If the `allow_url_fopen` value is set to allow it, an `include` statement like this could pull in any file the attacker might want. Imagine if the `path` value was set to something like `http://badguy.ru/evilscript.php`. By including that file in the page, they would have access to all of the variables in the current scope. A quick call to get_defined_vars can tell them that right away.

How can this be prevented? Well, there's two main things:

1. Don't use eval...**ever**. If at all possible avoid the use of the `eval` function in your code. There's a lot of things you can do with *variable variables* that might prevent you from having to use it.
2. If you absolutely do have to use `eval`, validate everything going into it and ensure that there's no unfiltered user input anywhere near it. Even better, if you do have to use it, don't take *any* data from the user to use in the code string.

A lot of times, if there's a vulnerability related to *code injection* the attacker will take advantage of it by also using `eval` themselves to disguise the code coming in with a base64_encode call to an obfuscated string of code. If the request is made via a `$_GET` variable, you'll have all you need to decode it in your logs, but with a `$_POST` request you may not be so lucky.

There is also a known exploit that revolves around a bug in PHP itself (versions 5.3.12 and 5.4.x before 5.4.2) where the attacker could make the request with a string like `GET /index.php?-dsafe_mode%3dOff` and modify the `php.ini` settings for the currently running script and forcefully allow a remote file upload. You can find out more about this vulnerability on the CVE site.

## Command injection

Similar to its other injection cousins, *code injection* is usually a result poor input validation. What is *code injection* you ask? Well, its a form of injection where the input from the user is appended to a part of a command being run on the local system. Here's a quick example:

```php
<?php

exec('rm -rf '.$_GET['path']);

?>
```

It's a silly example as it's pretty easy to see where the problem is, but it gives you an idea of what kind of trouble it could cause in the wrong hands. It's not just limited to the `exec` method either. All of the PHP system execution methods do no filtering at all on the input they're being given. They assume that the command they've been handed has been through good validation and should be executed blindly. With several of them (like passthru) the result of the execution is even echoed directly back out and not stored in any way, allowing the attacker to inject the content of their choosing into the page.

Regardless of if you're using user input in your commands, it's always a good idea to do some filtering and validation. Command line strings can be quite different from case to case, so what's a good way to handle the filtering? PHP comes to the rescue with two functions:

- escapeshellcmd
- escapeshellarg

The first, `escapeshellcmd` is used to do filtering on the command part of the string, doing some escaping on characters that could be used to break out of the command and run anything the attacker chooses. The following characters are escaped: `#&;`|*?~<>^()[]{}$\, \x0A and \xFF `as well as single and double quotes (if they're not paired correctly). This prevents the attacker from being able to give the command something like` foo; rm -rf /‘ and execute a second command.

The second, `escapeshellarg`, is the second part of the equation. It's used to escape the parameters being set on the command and escape them similarly. Here's an example of both in action:

```php
<?php

$cmd = escapeshellcmd('mycmd.php');
$argList = '';

foreach ($args as $value) {
        $argList .= escapeshellarg($value).' ';
}

exec($cmd.' '.$argList);

?>
```

You can see both functions in use here, one to escape the command itself and the other to escape the options. These are really just a shortcut to using something like str_replace to do the replacement. It just handles it a bit more automatically rather than you having to remember all of those characters each time.

When thinking through the functionality for your app, be sure to evaluate if you'll even need the system execution functions. If you don't, consider using the `disable_functions` setting in the `php.ini` to prevent their use. This helps prevent any additional issues if a vulnerability allowed for a PHP file to be uploaded where it could be executed.

## XML injection

XML can be very useful in some circumstances. In others, it could be a bit of overkill, but there's no doubt that this markup type isn't going anywhere. There's been a bigger surge for JSON formatted data recently, but XML still reigns king for complex data structures. XML is also quite flexible, but with this flexibility can come issues. One of these issues is a special kind of vulnerability called *external entity injection*. The actual issue is really related to two things: the fact that XML allows the definition of external entities and that the XML interpreters allow expansion by default.

In more recent versions (like PHP >=5.3.28) the default for entity expansion is to be disabled, so you may not see this behavior on newer installations. It is still a problem on older systems, however, so it bears mentioning.

So, what is *external entity expansion* (XXE)? Well, it's easier to show with an example:

```
<!DOCTYPE root
[
        <!ENTITY foo SYSTEM "http://test.localhost:8080/contents.txt">
]>
<test><testing>&foo;</testing></test>
```

In this XML example, the `foo` entity is defined in the `root DOCTYPE` block to point to an external entity. In this case, it's just a test script on a local system, but it could point to anything web accessible or locally accessible. If used to pull in a remote file, the attacker could potentially any data they wanted. The local file operates almost as an include and can be used to pull in any file data the web server user has access to.

As mentioned, the definition of the entity is only half of the problem. The other half, at least in PHP, is in the way it expands entities automatically when the XML is loaded. Down in the actual XML markup you'll see the `&foo;` entity. When the XML is loaded, that entity will be expanded out and the contents from the `ENTITY` target will be pulled directly into the XML. You can see how this could be dangerous if abused. If the attacker was able to get in entity definition into the XML you're parsing.

If you're using something like the SimpleXML functionality of PHP, you could be open to this kind of attack. Here's an example of some bad XML being parsed:

```
<?php
$badXml = '<!DOCTYPE root
    [
    <!ENTITY foo SYSTEM "http://test.localhost:8080/contents.txt">
    ]>
    <test><testing>&foo;</testing></test>';

$doc = simplexml_load_string($goodXml);
echo $doc->testing;
?>
```

When we echo out (or use) the `$doc->testing` element, the entity `&foo;` is expanded and we get the contents of `http://test.localhost:8080/contents.txt` injected into the XML document. To prevent

these external entities, URL- or file-based, from being loaded automatically, we use the libxml_disable_entity_loader function to disable the expansion. A call to `libxml_disable_entity_loader` with a parameter of `true` will disable the entity expansion completely.This tells the underlying `libxml` library to leave the entity references intact without substitution.

This method has the unfortunate side effect of also not expending any legitimate entities you might have defined in the document as well. There's two other XML parsing methods that give you the best of both worlds though. The XMLReader and DOM functionality of PHP (usually already built in). Here's an example of both:

```
<?php
// with the XMLReader functionality:
$doc = XMLReader::xml($badXml,'UTF-8',LIBXML_NONET);

// with the DOM functionality:
$dom = new DOMDocument();
$dom->loadXML($badXml,LIBXML_DTDLOAD|LIBXML_DTDATTR);
?>
```

## HTML5 Injection

Finally, I wanted to talk a bit about an up and coming new form of injection you'll need to keep an eye out for - *HTML5 injection*. While it's not strictly a type in itself, there's a whole host of injection types that come along with the new technologies introduced in these browser-based enhancements. As always, with new technology comes new kinds of risk and this is no different. With the HTML5 enhancements, browsers are becoming more and more powerful, allowing for features like the direct loading of media files and even in-browser data stores.

One example is the introduction of a feature called *localStorage*. For those not familiar with it, *localStorage* is an HTML5-enabled data storage method that lets you persist data either during the session or longer. The data in *localStorage* is stored in key/value pairs, much in the way you'd work with caching like with Redis or memcache. This data is stored on the local machine ina special kind of file accessible to the browser.

With this technology specifically, there's two exploits that could happen allowing the attacker to get at your data. The first is [cross-site scripting]{#xss}. If your application has a XSS vulnerability in it, the attacker has access to execute Javascript in the page's local scope. This local scope includes access to everything in your *localStorage* data set. They could potentially extract this information and send it back to their own systems. This isn't really an issue with *localStorage* itself, but it does illustrate what could be the start of an "attack chain" leading to even more compromise.

The second method an attacker could get at the *localStorage* data is through the file the data resides in. This is a much more difficult kind of attack as it requires access to the user's machine, but malware that can provide this is not unheard of. The *localStore* data isn't encrypted in any way and is always stored in the same place for each browser. Armed with this knowledge it's also possible the attacker could modify the data and you'd never be the wiser.

It's easy to think that something like *localStorage* is safer than other data your application might be using - it's on the user's machine, right? That has to account for something. As you've seen, though, there are still threats that could allow unauthorized access to the data stored there. You should always treat *any* data the same way and not only assume that it's tainted but also never store sensitive data in any place where it cannot be protected by encryption or strong access protection.