PHP Security Cheat Sheet - OWASP

Authors and Primary Editors

This page intends to provide basic PHP security tips for developers and administrators. Keep in mind that tips mentioned in this page may not be sufficient for securing your web application.

PHP overview

PHP is the most commonly used server-side programming language, with 81.8% of web servers deploying it, according to W3 Techs.

An open source technology, PHP is unusual in that it is both a language *and* a web framework, with typical web framework features built-in to the language. Like all web languages, there is also a large community of libraries etc. that contribute to the security (or otherwise) of programming in PHP. All three aspects (language, framework, and libraries) need to be taken into consideration when trying to secure a PHP site.

PHP is a 'grown' language rather than deliberately engineered, making writing insecure PHP applications far too easy and common. If you want to use PHP securely, then you should be aware of all its pitfalls.

Language issues

Weak typing

PHP is weakly typed, which means that it will automatically convert data of an incorrect type into the expected type. This feature very often masks errors by the developer or injections of unexpected data, leading to vulnerabilities (see “Input handling” below for an example).

Try to use functions and operators that do not do implicit type conversions (e.g. === and not ==). Not all operators have strict versions (for example greater than and less than), and many built-in functions (like `in_array`) use weakly typed comparison functions by default, making it difficult to write correct code.

Exceptions and error handling

Almost all PHP builtins, and many PHP libraries, do not use exceptions, but instead report errors in other ways (such as via notices) that allow the faulty code to carry on running. This has the effect of masking many bugs. In many other languages, and most high level languages that compete with PHP, error conditions that are caused by developer errors, or runtime errors that the developer has failed to anticipate, will cause the program to stop running, which is the safest thing to do.

Consider the following code which attempts to limit access to a certain function using a database query that checks to see if the username is on a black list:

```
   $db_link = mysqli_connect('localhost', 'dbuser', 'dbpassword', 'dbname');

   function can_access_feature($current_user) {
       global $db_link;
       $username = mysqli_real_escape_string($db_link, $current_user->username);
       $res = mysqli_query($db_link, "SELECT COUNT(id) FROM blacklisted_users
WHERE username = '$username';");
       $row = mysqli_fetch_array($res);
       if ((int)$row[0] > 0) {
           return false;
       } else {
           return true;
       }
   }

   if (!can_access_feature($current_user)) {
       exit();
   }

   // Code for feature here
```

There are various runtime errors that could occur in this - for example, the database connection could fail, due to a wrong password or the server being down etc., or the connection could be closed by the server after it was opened client side. In these cases, by default the `mysqli_` functions will issue warnings or notices, but will not throw exceptions or fatal errors. This means that the code simply carries on! The variable `$row` becomes `NULL`, and PHP will evaluate `$row[0]` also as `NULL`, and `(int)$row[0]` as `0`, due to weak typing. Eventually the `can_access_feature` function returns `true`, giving access to all users, whether they are on the blacklist or not.

If these native database APIs are used, error checking should be added at every point. However, since this requires additional work, and is easily missed, this is insecure by default. It also requires a lot of boilerplate. This is why accessing a database should always be done by using PHP Data Objects (PDO) specified with the ERRMODE_WARNING or ERRMODE_EXCEPTION flags unless there is a clearly compelling reason to use native drivers and careful error checking.

It is often best to turn up error reporting as high as possible using the error_reporting function, and never attempt to suppress error messages — always follow the warnings and write code that is more robust.

php.ini

The behaviour of PHP code often depends strongly on the values of many configuration settings, including fundamental changes to things like how errors are handled. This can make it very difficult to write code that works correctly in all circumstances. Different libraries can have different expectations or requirements about these settings, making it difficult to correctly use 3rd party code. Some are mentioned below under “Configuration.”

Unhelpful builtins

PHP comes with many built-in functions, such as `addslashes`, `mysql_escape_string` and `mysql_real_escape_string`, that appear to provide security, but are often buggy and, in fact, are unhelpful ways to deal with security problems. Some of these built-ins are being deprecated and removed, but due to backwards compatibility policies this takes a long time.

PHP also provides an 'array' data structure, which is used extensively in all PHP code and internally, that is a confusing mix between an array and a dictionary. This confusion can cause even experienced PHP developers to introduce critical security vulnerabilities such as Drupal SA-CORE-2014-005 (see the patch).

Framework issues

URL routing

PHP's built-in URL routing mechanism is to use files ending in ".php" in the directory structure. This opens up several vulnerabilities:

- Remote execution vulnerability for every file upload feature that does not sanitise the filename. (In other words, the web server executes something instead of serving it). Ensure that when saving uploaded files, the content and filename are appropriately sanitised.
- Source code, including config files, are stored in publicly accessible directories along with files that are meant to be downloaded (such as static assets). Misconfiguration (or lack of configuration) can mean that source code or config files that contain secret information can be downloaded by attackers. (In other words, the web server serves a resource which should have been private or executable only). You can use `.htaccess` to limit access. This is not ideal, because it is insecure by default, but there is no other alternative.
- The URL routing mechanism is the same as the module system. This means it is often possible for attackers to use files as entry points which were not designed as such. This can open up vulnerabilities where authentication mechanisms are bypassed entirely - a simple refactoring that pulls code out into a separate file can open a vulnerability. This is made particularly easy in PHP because it has globally accessible request data ($_GET etc), so file-level code can be imperative code that operates on the request, rather than needing request handling code to be within function definitions.
- The lack of a proper URL routing mechanism often leads to developers creating their own ad-hoc methods. These are often insecure and fail to apply appropriate authorization restrictions on different request handling functionality.

Input handling

Instead of treating HTTP input as simple strings, PHP will build arrays from HTTP input, at the control of the client. This can lead to confusion about data, and can easily lead to security bugs. For example, consider this simplified code from a "one time nonce" mechanism that might be used, for example in a password reset code:

```
$supplied_nonce = $_GET['nonce'];
$correct_nonce = get_correct_value_somehow();

if (strcmp($supplied_nonce, $correct_nonce) == 0) {
    // Go ahead and reset the password
} else {
    echo 'Sorry, incorrect link';
}
```

If an attacker uses a querystring like this:

```
http://example.com/?nonce[]=a
```

then we end up with `$supplied_nonce` being an array. The function `strcmp()` will then return `NULL` (instead of throwing an exception, which would be much more useful), and then, due to weak typing and the use of the == (equality) operator instead of the === (identity) operator, the comparison succeeds (since the expression `NULL == 0` is true according to PHP), and the attacker will be able to reset the password without providing a correct nonce.

Exactly the same issue, combined with the confusion of PHP's 'array' data structure, can be exploited in issues such as Drupal SA-CORE-2014-005 - see example exploit.

Template language

PHP is essentially a template language. However, it doesn't do HTML escaping by default, which makes it very problematic for use in a web application - see section on XSS below.

Other inadequacies

There are other important things that a web framework should supply, such as a CSRF protection mechanism that is on by default. Because PHP comes with a rudimentary web framework that is functional enough to allow people to create web sites, many people will do so without any knowledge that they need CSRF protection.

Third party PHP code

Libraries and projects written in PHP are often insecure due to the problems highlighted above, especially when proper web frameworks are not used. Do not trust PHP code that you find on the web, as many security vulnerabilities can hide in seemingly innocent code.

Poorly written PHP code often results in warnings being emitted, which can cause problems. A common solution is to turn off all notices, which is exactly the opposite of what ought to be done (see above), and leads to progressively worse code.

Update PHP regularly

Keep in mind that you should regularly upgrade your PHP distribution on an operational server. Every day new flaws are discovered and announced in PHP and attackers use these new flaws on random servers frequently.

The behaviour of PHP is strongly affected by configuration, which can be done through the "php.ini" file, Apache configuration directives and runtime mechanisms - see http://www.php.net/manual/en/configuration.php

There are many security related configuration options. Some are listed below:

SetHandler

PHP code should be configured to run using a 'SetHandler' directive. In many instances, it is wrongly configured using an 'AddHander' directive. This works, but also makes other files executable as PHP code - for example, a file name "foo.php.txt" will be handled as PHP code, which can be a very serious remote execution vulnerability if "foo.php.txt" was not intended to be executed (e.g. example code) or came from a malicious file upload.

All data that is a product, or subproduct, of user input is to NOT be trusted. They have to either be validated, using the correct methodology, or filtered, before considering them untainted.

Super globals which are not to be trusted are `$_SERVER`, `$_GET`, `$_POST`, `$_REQUEST`, `$_FILES` and `$_COOKIE`. Not all data in `$_SERVER` can be faked by the user, but a considerable amount in it can, particularly and specially everything that deals with HTTP headers (they start with `HTTP_`).

File uploads

Files received from a user pose various security threats, especially if other users can download these files. In particular:

- Any file served as HTML can be used to do an XSS attack
- Any file treated as PHP can be used to do an extremely serious attack - a remote execution vulnerability.

Since PHP is designed to make it very easy to execute PHP code (just a file with the right extension), it is particularly important for PHP sites (any site with PHP installed and configured) to ensure that uploaded files are only saved with sanitised file names.

Common mistakes on the processing of $_FILES array

It is common to find code snippets online doing something similar to the following code:

```
if ($_FILES['some_name']['type'] == 'image/jpeg') {
    //Proceed to accept the file as a valid image
}
```

However, the type is not determined by using heuristics that validate it, but by simply reading the data sent by the HTTP request, which is created by a client. A better, yet not perfect, way of validating file types is to use finfo class.

```
$finfo = new finfo(FILEINFO_MIME_TYPE);
$fileContents = file_get_contents($_FILES['some_name']['tmp_name']);
$mimeType = $finfo->buffer($fileContents);
```

Where $mimeType is a better checked file type. This uses more resources on the server, but can prevent the user from sending a dangerous file and fooling the code into trusting it as an image, which would normally be regarded as a safe file type.

Use of $_REQUEST

Using $_REQUEST is strongly discouraged. This super global is not recommended since it includes not only POST and GET data, but also the cookies sent by the request. All of this data is combined into one array, making it almost impossible to determine the source of the data. This can lead to confusion and makes your code prone to mistakes, which could lead to security problems.

Since a single SQL Injection vulnerability permits the hacking of your website, and every hacker first tries SQL injection flaws, fixing SQL injections are the first step to securing your PHP powered application. Abide to the following rules:

Never concatenate or interpolate data in SQL

Never build up a string of SQL that includes user data, either by concatenation:

```
$sql = "SELECT * FROM users WHERE username = '" . $username . "';";
```

or interpolation, which is essentially the same:

```
$sql = "SELECT * FROM users WHERE username = '$username';";
```

If '$username' has come from an untrusted source (and you must assume it has, since you cannot easily see that in source code), it could contain characters such as ' that will allow an attacker to execute very different queries than the one intended, including deleting your entire database etc. Using prepared statements and bound parameters is a much better solution. PHP's [mysqli](http://php.net/mysqli) and [PDO](http://php.net/pdo) functionality includes this feature (see below).

Escaping is not safe

mysql_real_escape_string is not safe. Don't rely on it for your SQL injection prevention.

Why: When you use mysql_real_escape_string on every variable and then concat it to your query, *you are bound to forget that at least once*, and once is all it takes. You can't force yourself in any way to never forget. In addition, you have to ensure that you use quotes in the SQL as well, which is not a natural thing to do if you are assuming the data is numeric, for example. Instead use prepared statements, or equivalent APIs that always do the correct kind of SQL escaping for you. (Most ORMs will do this escaping, as well as creating the SQL for you).

Use Prepared Statements

Prepared statements are very secure. In a prepared statement, data is separated from the SQL command, so that everything user inputs is considered data and put into the table the way it was.

See the PHP docs on MySQLi prepared statements and PDO prepared statements

Where prepared statements do not work

The problem is, when you need to build dynamic queries, or need to set variables not supported as a prepared variable, or your database engine does not support prepared statements. For example, PDO MySQL does not support ? as LIMIT specifier. Additionally, they cannot be used for things like table names or columns in `SELECT` statements. In these cases, you should use query builder that is provided by a framework, if available. If not, several packages are available for use via Composer and Packagist Do not roll your own.

ORM

ORMs (Object Relational Mappers) are good security practice. If you're using an ORM (like Doctrine) in your PHP project, you're still prone to SQL attacks. Although injecting queries in ORM's is much harder, keep in mind that concatenating ORM queries makes for the same flaws that concatenating SQL queries, so **NEVER** concatenate strings sent to a database. ORM's support prepared statements as well.

Always be sure to evaluate the code of *any* ORM you use to validate how it handles the execution of the SQL it generates. Ensure it does not concatenate the values and instead uses prepared statements internally as well as following good security practices.

Encoding Issues

Use UTF-8 unless necessary

Many new attack vectors rely on encoding bypassing. Use UTF-8 as your database and application charset unless you have a mandatory requirement to use another encoding.

```
$DB = new mysqli($Host, $Username, $Password, $DatabaseName);
if (mysqli_connect_errno())
    trigger_error("Unable to connect to MySQLi database.");
$DB->set_charset('utf8');
```

SQL aside, there are a few more injections possible *and common* in PHP:

Shell Injection

A few PHP functions namely

- shell_exec
- exec
- passthru
- system
- backtick operator (`)

run a string as shell scripts and commands. Input provided to these functions (specially backtick operator that is not like a function). Depending on your configuration, shell script injection can cause your application settings and configuration to leak, or your whole server to be hijacked. This is a very dangerous injection and is somehow considered the haven of an attacker.

Never pass tainted input to these functions - that is input somehow manipulated by the user - unless you're absolutely sure there's no way for it to be dangerous (which you never are without whitelisting). Escaping and any other countermeasures are ineffective, there are plenty of vectors for bypassing each and every one of them; don't believe what novice developers tell you.

Code Injection

All interpreted languages such as PHP, have some function that accepts a string and runs that in that language. In PHP this function is named eval(). Using eval is a very bad practice, not just for security. If you're absolutely sure you have no other way but eval, use it without any tainted input. Eval is usually also slower.

Function preg_replace() should not be used with unsanitised user input, because the payload will be eval()'ed.

```
preg_replace("/.*/e","system('echo /etc/passwd')");
```

Reflection also could have code injection flaws. Refer to the appropriate reflection documentations, since it is an advanced topic.

Other Injections

LDAP, XPath and any other third party application that runs a string, is vulnerable to injection. Always keep in mind that some strings are not data, but commands and thus should be secure before passing to third party libraries.

There are two scenarios when it comes to XSS, each one to be mitigated accordingly:

No Tags

Most of the time, there is no need for user supplied data to contain unescaped HTML tags when output. For example when you're about to dump a textbox value, or output user data in a cell.

If you are using standard PHP for templating, or `echo` etc., then you can mitigate XSS in this case by applying 'htmlspecialchars' to the data, or the following function (which is essentially a more convenient wrapper around 'htmlspecialchars'). **However, this is not recommended**. The problem is that you have to remember to apply it every time, and if you forget once, you have an XSS vulnerability. Methodologies that are insecure by default must be treated as insecure.

Instead of this, you should use a template engine that applies HTML escaping **by default** - see below. All HTML should be passed out through the template engine.

If you cannot switch to a secure template engine, you can use the function below on all untrusted data.

Keep in mind that this scenario won't mitigate XSS when you use user input in dangerous elements (style, script, image's src, a, etc.), but mostly you don't. Also keep in mind that every output that is not intended to contain HTML tags should be sent to the browser filtered with the following function.

```
//xss mitigation functions
function xssafe($data,$encoding='UTF-8')
{
   return htmlspecialchars($data,ENT_QUOTES | ENT_HTML401,$encoding);
}
function xecho($data)
{
   echo xssafe($data);
}

//usage example
<input type='text' name='test' value='<?php
xecho ("' onclick='alert(1)");
?>' />
```

Untrusted Tags

When you need to allow users to supply HTML tags that are used in your output, such as rich blog comments, forum posts, blog posts and etc., but cannot trust the user, you have to use a **Secure Encoding** library. This is usually hard and slow, and that's why most applications have XSS vulnerabilities in them. OWASP ESAPI has a bunch of codecs for encoding different sections of data. There's also OWASP AntiSammy and HTMLPurifier for PHP. Each of these require lots of configuration and learning to perform well, but you need them when you want that good of an application.

Templating engines

There are several templating engines that can help the programmer (and designer) to output data and protect from most XSS vulnerabilities. While their primary goal isn't security, but improving the designing experience, most important templating engines automatically escape the variables on output and force the developer to explicitly indicate if there is a variable that shouldn't be escaped. This makes output of variables have a white-list behavior. There exist several of these engines. A good example is twig[1]. Other popular template engines are Smarty, Haanga and Rain TPL.

Templating engines that follow a white-list approach to escaping are essential for properly dealing with XSS, because if you are manually applying escaping, it is too easy to forget, and developers should always use systems that are secure by default if they take security seriously.

Other Tips

- Don't have a **trusted section** in any web application. Many developers tend to leave admin areas out of XSS mitigation, but most intruders are interested in admin cookies and XSS. Every output should be cleared by the functions provided above, if it has a variable in it. Remove every instance of echo, print, and printf from your application and replace them with a secure template engine.
- HTTP-Only cookies are a very good practice, for a near future when every browser is compatible. Start using them now. (See PHP.ini configuration for best practice)
- The function declared above, only works for valid HTML syntax. If you put your Element Attributes without quotation, you're doomed. Go for valid HTML.
- Reflected XSS is as dangerous as normal XSS, and usually comes at the most dusty corners of an application. Seek it and mitigate it.
- Not every PHP installation has a working **mhash** extension, so if you need to do hashing, check it before using it. Otherwise you can't do SHA-256
- Not every PHP installation has a working **mcrypt** extension, and without it you can't do AES. Do check if you need it.

CSRF mitigation is easy in theory, but hard to implement correctly. First, a few tips about CSRF:

- Every request that does something noteworthy, should be CSRF mitigated. Noteworthy things are changes to the system, and reads that take a long time.
- CSRF mostly happens on GET, but is easy to happen on POST. Don't ever think that post is secure.

The OWASP PHP CSRFGuard is a code snippet that shows how to mitigate CSRF. Only copy pasting it is not enough. In the near future, a copy-pasteable version would be available (hopefully). For now, mix that with the following tips:

- Use re-authentication for critical operations (change password, recovery email, etc.)
- If you're not sure whether your operation is CSRF proof, consider adding CAPTCHAs (however CAPTCHAs are inconvenience for users)
- If you're performing operations based on other parts of a request (neither GET nor POST) e.g Cookies or HTTP Headers, you might need to add CSRF tokens there as well.
- AJAX powered forms need to re-create their CSRF tokens. Use the function provided above (in code snippet) for that and never rely on Javascript.
- CSRF on GET or Cookies will lead to inconvenience, consider your design and architecture for best practices.

PHP doesn't ship with a readily available authentication module, you need to implement your own or use a PHP framework, unfortunately most PHP frameworks are far from perfect in this manner, due to the fact that they are developed by open source developer community rather than security experts. A few instructive and useful tips are listed below:

Session Management

PHP's default session facilities are considered safe, the generated PHPSessionID is random enough, but the storage is not necessarily safe:

- Session files are stored in temp (/tmp) folder and are world writable unless suPHP installed, so any LFI or other leak might end-up manipulating them.
- Sessions are stored in files in default configuration, which is terribly slow for highly visited websites. You can store them on a memory folder (if UNIX).
- You can implement your own session mechanism, without ever relying on PHP for it. If you did that, store session data in a database. You could use all, some or none of the PHP functionality for session handling if you go with that.

Session Hijacking Prevention

It is good practice to bind sessions to IP addresses, that would prevent most session hijacking scenarios (but not all), however some users might use anonymity tools (such as TOR) and they would have problems with your service.

To implement this, simply store the client IP in the session first time it is created, and enforce it to be the same afterwards. The code snippet below returns client IP address:

```
$IP = getenv ( "REMOTE_ADDR" );
```

Keep in mind that in local environments, a valid IP is not returned, and usually the string **:::1** or **:::127** might pop up, thus adapt your IP checking logic. Also beware of versions of this code which make use of the HTTP_X_FORWARDED_FOR variable as this data is effectively user input and therefore susceptible to spoofing (more information here and here)

Invalidate Session ID

You should invalidate (unset cookie, unset session storage, remove traces) of a session whenever a violation occurs (e.g 2 IP addresses are observed). A log event would prove useful. Many applications also notify the logged in user (e.g GMail).

Rolling of Session ID

You should roll session ID whenever elevation occurs, e.g when a user logs in, the session ID of the session should be changed, since it's importance is changed.

Exposed Session ID

Session IDs are considered confidential, your application should not expose them anywhere (specially when bound to a logged in user). Try not to use URLs as session ID medium.

Transfer session ID over TLS whenever session holds confidential information, otherwise a passive attacker would be able to perform session hijacking.

Session Fixation

Invalidate the Session id after user login (or even after each request) with session_regenerate_id().

Session Expiration

A session should expire after a certain amount of inactivity, and after a certain time of activity as well. The expiration process means invalidating and removing a session, and creating a new one when another request is met.

Also keep the **log out** button close, and unset all traces of the session on log out.

Inactivity Timeout

Expire a session if current request is X seconds later than the last request. For this you should update session data with time of the request each time a request is made. The common practice time is 30 minutes, but highly depends on application criteria.

This expiration helps when a user is logged in on a publicly accessible machine, but forgets to log out. It also helps with session hijacking.

General Timeout

Expire a session if current session has been active for a certain amount of time, even if active. This helps keeping track of things. The amount differs but something between a day and a week is usually good. To implement this you need to store start time of a session.

Cookies

Handling cookies in a PHP script has some tricks to it:

Never Serialize

Never serialize data stored in a cookie. It can easily be manipulated, resulting in adding variables to your scope.

Proper Deletion

To delete a cookie safely, use the following snippet:

```
setcookie ($name, "", 1);
setcookie ($name, false);
unset($_COOKIE[$name]);
```

The first line ensures that cookie expires in browser, the second line is the standard way of removing a cookie (thus you can't store false in a cookie). The third line removes the cookie from your script. Many guides tell developers to use time() - 3600 for expiry, but it might not work if browser time is not correct.

You can also use **session_name()** to retrieve the name default PHP session cookie.

HTTP Only

Most modern browsers support HTTP-only cookies. These cookies are only accessible via HTTP(s) requests and not JavaScript, so XSS snippets can not access them. They are very good practice, but are not satisfactory since there are many flaws discovered in major browsers that lead to exposure of HTTP only cookies to JavaScript.

To use HTTP-only cookies in PHP (5.2+), you should perform session cookie setting manually (not using **session_start**):

```
#prototype
bool setcookie ( string $name [, string $value [, int $expire = 0 [, string
$path [, string $domain [, bool $secure = false [, bool $httponly = false ]]]]]]
)

#usage
if (!setcookie("MySessionID", $secureRandomSessionID, $generalTimeout,
$applicationRootURLwithoutHost, NULL, NULL,true))
    echo ("could not set HTTP-only cookie");
```

The **path** parameter sets the path which cookie is valid for, e.g if you have your website at example.com/some/folder the path should be /some/folder or other applications residing at example.com could also see your cookie. If you're on a whole domain, don't mind it. **Domain** parameter enforces the domain, if you're accessible on multiple domains or IPs ignore this, otherwise set it accordingly. If **secure** parameter is set, cookie can only be transmitted over HTTPS. See the example below:

```
$r=setcookie("SECSESSID","1203j01j0s1209jw0s21jxd01h029y779g724jahsa9opk123973",
time()+60*60*24*7 /*a week*/,"/","owasp.org",true,true);
if (!$r) die("Could not set session cookie.");
```

Internet Explorer issues

Many version of Internet Explorer tend to have problems with cookies. Mostly setting Expire time to 0 fixes their issues.

Authentication

Remember Me

Many websites implement vulnerable "remember me" features. Often, these vulnerabilities are trivially exploitable (e.g. storing **userid=13** or **username=paul&password=abcdefg**).

- **Rule 1: Never store username/password or any relevant information in the cookie.**

Instead, you should use a random token (at least 16 characters from random_bytes(), encoded however you like), which should be stored in only two places: The user's cookie and a database record. However, even this strategy is fraught with peril: Your database lookups (like most search operations) can leak timing information, which is typically visible to an attacker over the network.

To avoid side-channels, consider adopting this strategy for "remember me" tokens:

```
1. Generate two tokens: A selector and a verifier, using a CSPRNG.
2. Store selector and a SHA-384 hash of verifier in the database.
3. Store selector and verifier together in a cookie (say: rememberme).
```

When a user visits your website and they aren't authenticated, but they do have a rememberme cookie, initiate this process:

```
1. Split the cookie into the selector and verifier compnents.
2. Do a database lookup with the selector.
3. If a row is found, grab that record.
4. Compare the SHA-384 of the user-provided verifier with the value stored in
the database, using hash_equals().
5. If they match, automatically authenticate the user.
```

Please see PHP Configuration Cheat Sheet.

Abbas Naderi Afooshteh (abbas.naderi@owasp.org)

Achim - Achim at owasp.org

Andrew van der Stock

Luke Plant

OWASP Cheat Sheets Project Homepage

- OWASP Cheat Sheet Series