Writing secure code

Whether you are writing a PHP snippet or an entire module, it is important to keep your code secure.

**Use check functions on output to prevent cross site scripting attacks**

No piece of user-submitted content should ever be placed as-is into HTML.

* Use [check\_plain](https://api.drupal.org/api/function/check_plain/7.x) for plain text.
* Use [filter\_xss](https://api.drupal.org/api/function/filter_xss/7.x) for markup-containing text, or if the context is entered by an admin and should allow most markup, use [filter\_xss\_admin](https://api.drupal.org/api/function/filter_xss_admin/7.x). While it can also sanitize text, it's almost never correct to use [check\_markup](https://api.drupal.org/api/function/check_markup/7.x) in a theme or module except in the context of something like a text area with an associated text format.
* Use the [t()](https://api.drupal.org/api/function/t/7.x) function with @ or % placeholders to construct safe, translatable strings.

### Use the database abstraction layer to avoid SQL injection attacks

[Use the database layer correctly.](https://www.drupal.org/node/101496) For example, never concatenate data directly into SQL queries, like this:

db\_query('SELECT foo FROM {table} t WHERE t.name = '. $\_GET['user']);

Instead, use proper argument substitution with [db\_query](http://api.drupal.org/api/function/db_query/7.x):

The DB layer works on top of PHP PDO and uses an array of named placholders:

db\_query("SELECT foo FROM {table} t WHERE t.name = :name", array(':name' => $\_GET['user']));

For a variable number of argument, use an array of arguments or use db\_select():

db\_query("SELECT t.s FROM {table} t WHERE t.field IN (:users)", array(':users' => $from\_user));

OR

$result = db\_select('table', 't')

->fields('t', array('s'))

->condition('t.field', $from\_user, 'IN')

->execute();

# Handle user input with care

Input, whether it comes from visitors or servers, should be handled with care.

/\*\* Example 1 - Insecure

\* SQL injection via $keyword

\*/

$keyword = $\_REQUEST['keyword'];

$query = "SELECT cust\_id, cust\_name, cust\_email FROM customers WHERE category = '$keyword'";

$result = db\_query($query);

This enables the user to inject SQL statements into the query. What would happen if a visitor used the keyword '; DROP TABLE customers; --? The query would be:

SELECT cust\_id, cust\_name, cust\_email FROM customers WHERE category = ''; DROP TABLE customers; --'

## The Drupal philosophy - Escape or filter when appropriate

You may have found the following expert advice in text dealing with security problems: always validate input

So what we really have to do is make sure that, regardless the data, **its content can never be interpreted as SQL.** To do this, we use the escaping functions provided by the database API. We do this escaping in the database layer and not directly after receiving input as it may not be the only escaping (or filtering) we have to do.

/\*\* Example 2 - Insecure

\* SQL injection via $keyword

\* XSS via $keyword and (possibly) $row->cust\_name, $row->cust\_email

\*/

$keyword = $\_REQUEST['keyword'];

$query = "SELECT cust\_id, cust\_name, cust\_email FROM customers WHERE category = '$keyword'";

$result = db\_query($query);

echo "$keyword";

while ($row = db\_fetch\_object($result)) {

echo "$row->cust\_name, $row->cust\_email";

}

Example 2 — though small, riddled with vulnerabilities — demonstrates the issue with input filtering escaping; the $keyword has to be filtered or escaped twice for different purposes: Once to prevent SQL injection and once more to prevent cross site scripting (XSS) attacks. Applying both filters directly to the input $keywordwould result in extraneous slashes showing up in output, or failed search attempted with &quot; in them.

The solution is to **use an appropriate filter when needed**. For example, just before sending plain text to the browser or mixing plain text with HTML, escape it with [check\_plain](http://api.drupal.org/apis/check_plain" \o "check_plain API documentation).

/\*\* Example 2 - corrected

\*/

$keyword = $\_REQUEST['keyword'];

$query = "SELECT cust\_id, cust\_name, cust\_email FROM customers WHERE category = '%s'";

$result = db\_query($query, $keyword);

echo ''. check\_plain($keyword) .'':

while ($row = db\_fetch\_object($result)) {

// Not very elegant, but making a point.

echo check\_plain($row->cust\_name) .','. check\_plain($row->cust\_email) .'';

}

# Avoid using data from $form\_state['input']

$form\_state['input'] and \Drupal::request()->request are actually copies of the raw, unsanitized data from [PHP's $\_POST superglobal](http://php.net/manual/en/reserved.variables.post.php), so the data it contains has not been type-checked.

**Use data from $form\_state['values']** (Drupal 7) or $form\_state->values(Drupal 8) instead.

## High-level overview of the problem

Essentially, the issue comes down to the way that browsers send form data, [the way that PHP interprets GET and POST data](http://php.net/manual/en/language.variables.external.php), and the fact that certain built-in PHP functions, database drivers, Drupal functions, and library functions can accept both strings or arrays (and may act in ways you do not expect when passed data of a type you did not expect).

Specifically, you may have put a textfield on the page in your form, but the raw data returned for the textfield in $form\_state['input'] might not be a string: it could be an uploaded file or an array.

## A more detailed example

Recall that, when you submit HTML like the following...

<form action="/" method="post">

<input type="text" name="foo" value="bar" />

<input type="text" name="baz" value="qux" />

<input type="submit" value="Submit" />

</form>

... your browser will send an HTTP request that looks (roughly) like...

POST / HTTP/1.1

Content-Type: application/x-www-form-urlencoded

Host: example.com

foo=bar&amp;baz=qux

PHP will automatically de-serialize this into the $\_POST super-global as...

$\_POST = array(

'foo' => 'bar',

'baz' => 'qux',

);

Note that the web browser does not pass any information about the type of form control, and that PHP doesn't know about the HTML that generated the response.

However, when you submit a form like the following...

<form action="/" method="post">

<input type="checkbox" name="foo[1]" value="bar" checked="checked" />

<input type="checkbox" name="foo[2]" value="qwe" checked="checked" />

<input type="text" name="baz" value="qux" />

<input type="submit" value="Submit" />

</form>

... that will result in an HTTP request like...

POST / HTTP/1.1

Content-Type: application/x-www-form-urlencoded

Host: example.com

foo%5B1%5D=bar&amp;foo%5B2%5D=qwe&amp;baz=qux

But PHP will interpret that a bit differently than you might expect...

$\_POST = array(

'foo' => array(

1 => 'bar',

2 => 'qwe',

),

'baz' => 'qux',

);

Note that, in both examples, PHP's $\_POST array contains an entry named foo, but in the first example it is a string, and in the second example, it is an array.

## The security issue

This can cause unexpected behavior in your PHP code if it expects $\_POST['foo']to be a string but it is actually an array, and vice-versa.

Famously, attacks for [SA-CORE-2014-005](https://www.drupal.org/SA-CORE-2014-005) leveraged a [weakness in the form API input processing which has since been fixed](https://www.drupal.org/node/2380053), because the database driver would act differently if passed a string versus an array.

## The solution

The Form API's [drupal\_validate\_form()](https://api.drupal.org/api/drupal/includes!form.inc/function/drupal_validate_form/7) and [\_form\_validate()](https://api.drupal.org/api/drupal/includes!form.inc/function/_form_validate/7) functions take care of populating a safe set of data in $form\_state['values'], using sanitized data from $form\_state['input'].

Notes

* The [Drupal Best Practice sniffs for PHPCodeSniffer](http://cgit.drupalcode.org/coder/tree/coder_sniffer/DrupalPractice) will warn you if you use $form\_state['input'].

# Create forms in a safe way to avoid cross-site request forgeries (CSRF)

Cross-site request forgery (CSRF or XSRF) is a process where a request is made to a site which takes an action when the user did not intend to take that action. This can be achieved in a variety of ways, but in Drupal it is simple to protect against this type of attack.

The [HTTP 1.1 specification](http://rfc-editor.org/rfc/rfc2616.txt) makes a clear distinction that POST requests can modify data in the site (section 9.5) while GET requests should not modify data (section 9.3). Modules that modify data should require a POST request (i.e. a form). In versions of Drupal prior to the release of [SA-2007-017](http://drupal.org/node/162360) it was possible to create a specifically formed page that contained image tags where the image "src" element was a link to certain menu "disable" URLs on a Drupal site. If a site admin visited that page when they were logged into their site then their browser would request the URL of the menu disable page which would then disable their menu items.

### Protecting against CSRF in Drupal

The Drupal [Form API](https://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7) provides protection against CSRF using special tokens in the forms which are added automatically. If your module uses the Form API for all requests that modify data and if you properly follow the Form API documentation then your module is protected from CSRF.

### Bad code

Bad code could take two forms:

* Using the $\_POST variables directly and creating a form via HTML instead of the Drupal Form API
* Using a link and a menu callback to handle a action that modifies data (especially destructive modifications like deletion)

You can see the [Drupal Core CSRF vulnerabilities fixed in 5.2](http://drupal.org/node/162360) for an example of "bad code" and how to fix it.

### Good code

See the [Forms API](https://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7) documentation.

#### Examples

If you've discovered that you have menu callbacks that are vulnerable to CSRF, the simplest solution may be to add a confirmation form via confirm\_form() for each of your menu callbacks. Drupal does most of the work for you.

# Database access

Drupal provides several functions to send queries to the database. The canonical form is [db\_query](http://api.drupal.org/apis/db_query). Always use functions provided by Drupal to access the database to guard against SQL injections attacks. However, just using the functions is not enough as the following example illustrates:

/\*\* Example 1 - Insecure

\* SQL injection via $type

\* Display node titles of type $type (input supplied by the user via a form textfield)

\*/

$result = db\_query("SELECT n.nid, n.title FROM {node} n WHERE n.type = '$type'");

$items = array();

while ($row = db\_fetch\_object($result)) {

$items[] = l($row->title, "node/{$row->nid}");

}

return theme('item\_list', $items);

Example 1 displays a list of titles depending on the type argument supplied by a user. A list of page nodes will be retrieved when $type is page, a list of story nodes when $type is story. Unfortunately, the example is vulnerable to SQL injection.

The vulnerability can be used on databases with [UNION](http://dev.mysql.com/doc/refman/5.0/en/union.html) support (MySQL 4.1+) to gain administrator access to the site by supplying as type: story' UNION SELECT s.sid, s.sid FROM {sessions} s WHERE s.uid = 1 -- .

This will cause the following query to be executed:

SELECT n.nid, n.title FROM {node} n WHERE n.type = 'story' UNION SELECT s.sid, s.sid FROM {sessions} s WHERE s.uid = 1 -- '

As the snippet will now display valid session ids for the administrator user account, an attacker can instruct his/her browser to use the id and have full permissions on the site.

## Parameterized query prevents SQL injection

Preventing SQL injection is easy; db\_query provides a way to use parameterized queries. Drupal's database functions replace the [sprintf](http://www.php.net/manual/en/function.sprintf.php)-like placeholders with the properly escaped arguments in order of appearance.

db\_query("SELECT n.nid FROM {node} n WHERE n.nid > %d", $nid);

db\_query("SELECT n.nid FROM {node} n WHERE n.type = '%s'", $type);

db\_query("SELECT n.nid FROM {node} n WHERE n.nid > %d AND n.type = '%s'", $nid, $type);

db\_query("SELECT n.nid FROM {node} n WHERE n.type = '%s' AND n.nid > %d", $type, $nid);

Valid placeholders are documented in the [API documentation](http://api.drupal.org/api/HEAD/function/db_query):

* %d - integers
* %f - floats
* %s - strings, enclose in ''
* %b - binary data, do not enclose in ''
* %% - replaced with %

That leads us to a correction of Example 1:

/\*\* Example 1 - Corrected

\* Display node titles of type $type (input supplied by the user via a form textfield)

\*/

$result = db\_query("SELECT n.nid, n.title FROM {node} n WHERE n.type = '%s'", $type);

$items = array();

while ($row = db\_fetch\_object($result)) {

$items[] = l($row->title, "node/{$row->nid}");

}

return theme('item\_list', $items);

# Do not use /e in preg\_replace() - use preg\_replace\_callback() instead

Security advisories such as [SA-CONTRIB-2010-027](https://www.drupal.org/node/745390) and [SA-CONTRIB-2010-030](https://www.drupal.org/node/751934) have pointed out the risk of passing user input through regular expressions that use the [/e flag](http://www.php.net/manual/en/reference.pcre.pattern.modifiers.php) which causes matches to be evaluated as PHP code. Mistakes in escaping strings can lead to arbitrary code execution vulnerabilities - this will likely lead to total compromise of your Drupal site.

For this reason, the /e flag should be avoided as insecure. It is generally possible to instead use [preg\_replace\_callback()](http://www.php.net/manual/en/function.preg-replace-callback.php) to transform the matches and generate a replacement string while avoiding the risk that user input may be executed as PHP.

# File uploads, downloads and management

Allowing users to manage files on your server is a potentially dangerous operation.

You need to make sure that users cannot

* view arbitrary files.
* delete arbitrary files.
* overwrite 'critical' files.
* upload and execute arbitrary files.
* completely fill a device (or disk quota).

Note that "arbitrary" means "any file on the server". So, for example, if you limit them to files in the "files" directory then that is not arbitrary. But if the code for writing files allows the user to somehow affect the file path, then they could insert "../../" into the filename which will get it back out of the "files/" directory and into other directories on your server.

# Directories

For starters, always make sure that actions on uploaded files (upload, view, download, delete) are taking place in the 'files' directory or another designated directory. Beware that in the following code samples a hardcoded directory "files" is used to simplify the examples, in reality this directory is configurable.

Users have no business reading or deleting important system files (such as /etc/passwd or sites/default/settings.php). While the examples concentrate on deletion, keep in mind that reading arbitrary files is also undesirable.

/\*\* Example 1 - Insecure

\* Arbitrary file deletion.

\*

\* $file is path/filename (eg files/myfile.txt) provided by the user.

\*/

file\_delete($file);

A malicious user can abuse the trust granted to him/her in Example 1 by providing filenames in different directories, such as /sites/default/settings.php. The attack is clearly limited by the permissions of the user account that executes Drupal (often the webserver) on these files.

/\*\* Example 2a - Insecure

\* Arbitrary file deletion.

\*

\* $file is a filename (eg. myfile.txt) provided by the user.

\*/

file\_delete("files/$file");

/\*\* Example 2b - Insecure

\* Arbitrary file deletion.

\*

\* $file is path/filename (eg files/myfile.txt) provided by the user.

\*/

// Check whether $file is files/file

if (strpos($file, "files/") === 0) {

file\_delete($file);

}

Both Example 2a and 2b try to mitigate the attack by either prepending the filename with a fixed directory or by checking whether the supplied path begins in files/. Both examples are vulnerable to attack with *parent* paths (..).

What would happen if a malicious user would provide Example 2a with ../sites/default/settings.php and Example 2b with files/../sites/default/settings.php as path?

Both would attempt to delete sites/default/settings.php.

To properly check the real path of a file, use the Drupal function [file\_check\_location](http://api.drupal.org/api/HEAD/function/file_check_location).

/\*\* Example 3

\* No longer vulnerable to parent path (..) attacks.

\*

\* $file is path/filename (eg files/myfile.txt) provided by the user.

\*/

// Check whether $file is files/file

**if (file\_check\_location($file, 'files') {**

**file\_delete($file);**

**}**

# Handle text in a secure fashion

When handling and outputting text in HTML, you need to be careful that proper filtering or escaping is done. Otherwise, there might be bugs when users try to use angle brackets or ampersands, or worse you could open up [XSS](http://en.wikipedia.org/wiki/Cross_site_scripting) exploits.

When handling data, the golden rule is to store exactly what the user typed. When a user edits a post they created earlier, the form should contain the same things as it did when they first submitted it. This means that conversions are performed when content is output, not when saved to the database (be sure to read the [db\_query()](https://api.drupal.org/api/drupal/includes%21database%21database.inc/function/db_query/7) documentation on how to use the database API securely).

To help you see where checks are needed, it is handy to mentally 'color' in each string depending on which format its data is in. Is it plain-text, HTML, BBcode or Textile? Then, whenever you concatenate two strings, you need to make sure they are both in the same format. If they are not, an appropriate check, conversion or filtering must be applied.

User-submitted data in Drupal can be divided into three categories:

Some places require that you first sanitize any text:

**Plain-text**

This is simple text without any markup. What the user entered is displayed exactly on screen as is, and is not interpreted in any form. This is generally the format used for single-line text fields.

When outputting plain-text, you need to pass it through [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7) before it can be put inside HTML. This will convert quotes, ampersands and angle brackets into entities, causing the string to be shown literally on screen in the browser.

Most themeable functions and APIs take HTML for their arguments, and there are a few that automatically sanitize text by first passing it through [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7):

**Examples:**

* [t()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/t/7): the placeholders (e.g. '%name' or '@name') are passed as plain-text and will be escaped when inserted into the translatable string. You can disable this escaping by using placeholders of the form '!name' but only if you are sure that the string is safe.
* [l()](https://api.drupal.org/api/drupal/includes%21common.inc/function/l/7): the link caption should be passed as plain-text (unless overridden with the $html parameter).
* menu items and breadcrumbs: the menu item titles and breadcrumb titles are automatically sanitized.
* [theme('placeholder')](http://api.drupal.org/api/function/theme_placeholder): the placeholder text is plain-text.
* Block descriptions (but not titles--see below)
* User names when printed using [theme\_username()](https://api.drupal.org/api/drupal/includes%21theme.inc/function/theme_username/7) (Drupal 6 and earlier only. D7 expects the name to be sanitized already).
* Form API (FAPI) #default\_value element
* Form API (FAPI) #options element when the type is a select box (in Drupal 7   
  #options element is always sanitized).

$form['safe'] = array(

'#type' => 'textfield',

'#default\_value' => $u\_supplied, // FAPI will pass through check\_plain(),

);

$form['also\_safe'] = array(

'#type' => 'select',

'#default\_value' => 0, // FAPI will pass through check\_plain(),

'#options' => node\_get\_types('names'), // FAPI will sanitize the '#options' attribute with check\_plain() for select boxes.

);

// In Drupal 7 this is XSS safe as options are run through filter\_xss\_admin().

$form['drupal6\_unsafe'] = array(

'#type' => 'checkboxes',

'#default\_value' => 0, // FAPI will pass through check\_plain(),

'#options' => node\_get\_types('names'), // In Drupal 6, FAPI will NOT sanitize the '#options' attribute on other elements than select boxes.

);

tag however, all tags will be stripped out.  
Examples:

* Drupal 6 and earlier only: page titles set through [drupal\_set\_title()](https://api.drupal.org/api/drupal/includes%21path.inc/function/drupal_set_title/6), if the page title is displayed in the HTML, where it makes sense to use tags like <em> for clarity. When the page title is displayed in the HTML

drupal\_set\_title($node->title); // XSS vulnerability in D6, correct in D7

drupal\_set\_title(check\_plain($node->title)); // Correct in D6

* Drupal 6 and earlier only: block titles passed in through [hook\_block()](https://api.drupal.org/api/drupal/developer%21hooks%21core.php/function/hook_block/6). For the same reason as the page title, using HTML here is commonly done.
* Watchdog messages  
  **Examples:**

Drupal 6/7 (The message and variables are passed through t() by the watchdog function):

watchdog('content', "Deleted !title", array('!title' => $node->title)); // XSS

watchdog('content', "Deleted %title", array('%title' => $node->title)); // or @

* Form elements #description and #title  
  **Examples:**

$form['bad'] = array(

'#type' => 'textfield',

'#default\_value' => check\_plain($u\_supplied), // bad: escaped twice

'#description' => t("Old data: !data", array('!data' => $u\_supplied)), // XSS

);

$form['good'] = array(

'#type' => 'textfield',

'#default\_value' => $u\_supplied,

'#description' => t("Old data: @data", array('@data' => $u\_supplied)),

);

* Drupal 6 only: Form elements - #options when #type = checkboxes or #type = radios  
  **Examples:**

// This is XSS safe in Drupal 7+.

$form['drupal6\_bad'] = array(

'#type' => 'checkboxes',

'#options' => array($u\_supplied0, $u\_supplied1),

);

$form['good'] = array(

'#type' => 'checkboxes',

'#options' => array(check\_plain($u\_supplied0), check\_plain($u\_supplied1)),

);

* Form elements - #value of #type markup and item need to be safe. Note that the  
  default form element #type is markup!  
  **Examples:**

$form['unsafe'] = array('#value' => $user->name); //XSS

$form['safe'] = array('#value' => check\_plain($user->name));

or

$form['safe'] = array('#value' => theme('username', $user));

* **Rich text**

This is text which is marked up in some language (HTML, Textile, etc). It is stored in the markup-specific format, and converted to HTML on output using the various filters that are enabled. This is generally the format used for multi-line text fields.

All you need to do is pass the rich text through [check\_markup()](http://api.drupal.org/api/function/check_markup) and you'll get HTML returned, safe for outputting. You should also allow the user to choose the input format with a format widget through [filter\_form()](http://api.drupal.org/api/function/filter_form) and should pass the chosen format along to [check\_markup()](http://api.drupal.org/api/function/check_markup).

Note that you must make sure that the author of a post is allowed to use a particular input format, typically by checking with [filter\_access()](https://api.drupal.org/api/drupal/modules%21filter%21filter.module/function/filter_access/7) when the content is being submitted. Note that in Drupal 6 [check\_markup()](http://api.drupal.org/api/function/check_markup) performs this check for the current user by default. However, because content is filtered on output, this is often not the person who originally wrote the content. In that case, you can disable this check by passing $check = false to [check\_markup()](http://api.drupal.org/api/function/check_markup).

* **Admin-only HTML**

As of Drupal 4.7 there is a third way of dealing with text. There are some places in the administration section where it is impractical to invoke the filter system (for rich text), but where some simple markup is desired, such as a link or some emphasis (so plain text is not acceptable).

Examples include the mission statement, posting guidelines, and forum descriptions.

For such cases, you can use a regular text-area, and pass the text through [filter\_xss\_admin()](https://api.drupal.org/api/drupal/includes%21common.inc/function/filter_xss_admin/7) when you output it. This will allow most HTML tags to pass through, while still blocking possibly harmful script or styles.

URLs across Drupal require special handling in two ways:

1. If you wish to put any sort of dynamic data into a URL, you need to pass it through [urlencode()](http://www.php.net/manual/function.urlencode.php). If you don't, characters like '#' or '?' will disrupt the normal URL semantics. [urlencode()](http://www.php.net/manual/function.urlencode.php) will prevent this by escaping them with %XX syntax. Note that Drupal paths (e.g. 'node/123') are passed through [urlencode()](http://www.php.net/manual/function.urlencode.php) as a whole since Drupal 4.7 so you don't need to urlencode individual parts of it. This convenience does not apply to other parts of the URL like GET query arguments or fragment identifiers.
2. When using user-submitted URLs in a hyperlink, you need to use [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7) rather than just [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7). [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7) will call [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7), but also perform additional XSS checks to ensure the URL is safe for clicking on.

Note that all Drupal functions which return URLs ([url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/url/7), [request\_uri()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/request_uri/7), etc.) output plain URLs which have not been HTML escaped in any way (in other words, they are plain-text). Remember to use [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7) to escape them when outputting HTML (or XML). Don't use [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7) in situations where a real URL is expected, e.g. in the HTTP Location: ... header.

## In practice

All the rules above can be summed up quite easily: no piece of user-submitted content should ever be placed into HTML. If you are unsure of whether this is the case, you can always test it by submitting a piece of text like <u>xss</u> into your module's fields. If the text comes out underlined or mangles existing tags, you know you have a problem.

Here are some examples of good and bad code. $title, $body and $url are assumed to be user-submitted fields containing a title, a piece of marked up text and a URL respectively. They are fresh from the database and thus contain exactly what the user submitted without any changes.

Bad:  
<?php print '<tr><td>$title</td><td>'; ?>  
<?php print '<a href="/..." title="$title">view node</a>'; ?>

Good (the title is plain-text and may not be placed into HTML as is):  
<?php print '<tr><td>'. check\_plain($title) .'</td></tr>'; ?>  
<?php print '<a href="/..." title="'. check\_plain($title) .'">view node</a>'; ?>

Bad:  
<?php print l(check\_plain($title), 'node/'. $nid); ?>

Good ([l()](https://api.drupal.org/api/drupal/includes%21common.inc/function/l/7) already contains a [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7) call by default):  
<?php print l($title, 'node/'. $nid); ?>

Bad:  
<?php print '<a href="/$url">'; ?>  
<?php print '<a href="/'. check\_plain($url) .'">'; ?>

Good (URLs must be checked with [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7)):  
<?php print '<a href="/'. check\_url($url) .'">'; ?>

## Writing filters

When writing a filter which translates from another markup language into HTML, you need to ensure you don't open any holes yourself. Generally, the same rules apply: check URLs with [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7) and ensure no literal HTML can be injected by escaping appropriately using [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7).

# Handling Private/Personally Identifiable Information

When working with certain kinds of sensitive data, it is important to carefully evaluate Drupal's handling of that information and determine whether it meets your needs. As you might expect, the relative level of scrutiny you should devote to Drupal's handling of your data depends on the purpose of your site, its goals, and context. For example, on a site that provides information to groups persecuted by a government it could be considered inappropriate to store the IP address of a user, while on a site that includes health care information, a requirement may exist that you save the IP address of every user for a specific period of time. To match your site's purpose, goals and context, you will need to verify that Drupal is storing or omitting these data as appropriate.

Below are some general guidelines to consider when building/developing a site.

Consider what data in your site is private or sensitive

Review all the fields collected by a site. It can be easiest to do this by looking at the table definitions and some data from the tables in your database. As you audit each row, consider:

* Should we even collect this information? You may find cases where Drupal is collecting information that you don't want it to.
* What handling practices are required for this data, by 'best practices' or regulation? Personally identifiable information, health care data, financial data, and other classes of data may have specific requirements.
* What data are being collected outside of the database and what does that mean for the security of sensitive data? Load balancer or web server logs may, for example, collect information about IP addresses and information accessed which may be a concern in certain situations.

**Consider encrypting/anonymizing private data**

If a site has any data that requires special handling, consider when and where to encrypt the information.

* Consider using HTTPS (or another transport layer control like VPN only access) to encrypt the data as it travels between the site and the browser.
* Consider encrypting it inside the Drupal database (and storing the key somewhere safe) so that exposure of the database contents alone will not result in the exposure of this information. Note that it is also necessary to consider whether the key is being stored in a cache table in the database rendering this measure less meaningful.
* If encrypting the data inside the Drupal database consider expiring and purging keys for data when they are no longer needed.
* Consider [sanitizing Drupal database dumps](https://drushcommands.com/drush-9x/sql/sql-sanitize/) before distributing them to developers.
* Consider whether sensitive data are best stored in a separate data storage location and accessed via an API that can be further secured rather than incorporated into the main site.
* You may want to evaluate the use of whole-disk or folder-level encryption to secure your site's private data. If you do not have access to the physical storage medium on which your data rests, your hosting provider may be able to assist in ensuring that your disk, or virtual machine, is encrypted.

# Session IDs

Session support in PHP allows one to preserve data across subsequent accesses. A visitor accessing your website is assigned a unique ID, the so-called session ID. The session ID is stored in a cookie on the user side and sent to your website on every page request.

Drupal stores the session ID alongside user IDs in the database. On every page access Drupal receives the session ID from the visitor's browser. It then checks the session table to find the associated user ID. The user ID determines which permissions the visitor has on the site.

To keep the system secure it is imperative to keep the session ID secret. If you write a module you should never output session IDs so they can be read by other users. This would allow users to hijack the session of someone else.

Be aware that while your output may not be visible in a page, for example because you send it as part of an AJAX request, it can still be read when the user employs a sniffer such as Wireshark or Fiddler.

# Use of hash functions

For Drupal 7 and later core and contributed modules, the md5() and sha1()hash functions should never be used in any code, since they are considered obsolete and potentially insecure for some applications. This is a settled policy for Drupal core. For a normal hash function use sha-256 by calling hash('sha256', $data).

Even if the use of such functions are not for security purposes, any use of them at all can cause third party security audits of the codebase to raise flags. This can be a problem if, for example, Government entities require such audits - which would then require additional documentation to verify that they are indeed, not a security issue.

## Drupal 7

Drupal 7 presents wrapper functions to get shorter, base-64 encoded hashes to use in URLs, etc. See:

* [drupal\_hash\_base64($data)](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/drupal_hash_base64/7)
* [drupal\_hmac\_base64($data, $key)](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/drupal_hmac_base64/7)

## Background information

Any time you need to authenticate the content of a string or file by combining a secret key (e.g. a session ID) with the string, you should avoid using a single hash function which may be vulnerable to string-extension attacks. The preferred approach is to calculate a Hash-based Message Authentication Code (HMAC) using hash functions or relying on PHP's hash extension for PHP 5. An acceptable (but less preferred) alternative is to apply the hash function twice.

An example using PHP's hash library

$hmac = hash\_hmac('sha256', $data, $secret\_key);

An example using Drupal 7's hmac function:

$hmac = drupal\_hmac\_base64($data, $secret\_key);

# Using PHP with eval() or drupal\_eval()

Using [eval()](http://us.php.net/manual/en/function.eval.php) or [drupal\_eval()](http://api.drupal.org/api/function/drupal_eval/6) in your module's code could have a security risk if the PHP input provided to the function contains malicious code.

It is a best practice to add a new permission in your module just for using PHP so it's more clear of the security risk of assigning the permission to a user role. You should also add a warning for any form elements where the PHP input is entered.

For example, the following is how Drupal core's block module handles using PHP to control block visibility:

block.module:

function block\_perm() {

return array('administer blocks', 'use PHP for block visibility');

}

Drupal 7 uses a more generic permission (use PHP for settings) that should be used from any modules that allow a user to use PHP code in their settings pages.

# When to use db\_rewrite\_sql or ->addTag()

[**db\_rewrite\_sql()**](https://api.drupal.org/api/drupal/includes--database.inc/function/db_rewrite_sql/6) in Drupal 6 and **[db\_select()](https://api.drupal.org/api/drupal/includes!database!database.inc/function/db_select/7)** [tagged](https://api.drupal.org/api/drupal/includes!database!select.inc/function/SelectQuery::addTag/7) with (for example) ['node\_access'](https://www.drupal.org/node/310075) in Drupal 7 provide a method for modules to extend your SQL queries. For example, a module which controls access to nodes will need to limit the results of your queries, removing any nodes for which a visitor does not have the required set of access permissions.

If you do not make use of [db\_rewrite\_sql()](http://api.drupal.org/api/drupal/includes--database.inc/function/db_rewrite_sql/6) or the [->addTag method](https://api.drupal.org/api/drupal/includes!database!select.inc/function/SelectQuery::addTag/7), access control modules won't be able to modify or extend your SQL queries, and you may inadvertently expose content that is meant to be restricted.

*It's good practice to always make use of db\_rewrite\_sql() or ->addTag().*

Possible exceptions include:

* Queries which carry out internal module work, but which aren't responsible for showing content to users (e.g. queries within cron tasks).
* Queries for administrative pages where it is necessary to show an unfiltered list, and where the user is guaranteed to already have full privileges.

# Why does Drupal filter on output?

Some web applications process/filter the user input in the name of security before storing it in the database. Historically, Drupal has preserved user input as is, and filter it on output only. This is occasionally debated within the Drupal community.

[Steven Wittens'](http://drupal.org/user/10) excellent article [Safe string theory for the web](http://acko.net/blog/safe-string-theory-for-the-web) provides a full technical explanation of why it is best to preserve the original user input. The type of filtering needed depends on the output context. Acting on input can be quite problematic because you do not know what characters are forbidden without knowing the context in which they will appear.

To make things even trickier, a certain string could appear in more than one context at one time. For example, the same string might be used as HTML text, and as an HTML attribute too:  
<a title="$node->title">$node->title</a>

If you attempt to strip all "special" characters from this string, you'll be unable to output meaningful text in an HTML page. Encoding those characters won't help either. In addition, encoding creates another problem, in that processing the escaped or encoded text is very cumbersome (for example, consider the difficulty of extracting a teaser from an HTML-escaped node body).

The best choice is to store the user input unchanged, and perform proper escaping upon output. As much as possible, variables should be escaped prior to the theme layer in a way appropriate for their most likely use.

|  |  |
| --- | --- |
| [check\_markup](https://api.drupal.org/api/drupal/modules%21filter%21filter.module/function/check_markup/7.x) | Runs all the enabled filters on a piece of text. |
| [check\_plain](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7.x) | Encodes special characters in a plain-text string for display as HTML. |
| [check\_url](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7.x) | Strips dangerous protocols from a URI and encodes it for output to HTML. |
| [drupal\_attributes](https://api.drupal.org/api/drupal/includes%21common.inc/function/drupal_attributes/7.x) | Converts an associative array to an XML/HTML tag attribute string. |
| [drupal\_strip\_dangerous\_protocols](https://api.drupal.org/api/drupal/includes%21common.inc/function/drupal_strip_dangerous_protocols/7.x) | Strips dangerous protocols (e.g. 'javascript:') from a URI. |
| [filter\_xss](https://api.drupal.org/api/drupal/includes%21common.inc/function/filter_xss/7.x) | Filters HTML to prevent cross-site-scripting (XSS) vulnerabilities. |
| [filter\_xss\_admin](https://api.drupal.org/api/drupal/includes%21common.inc/function/filter_xss_admin/7.x) | Applies a very permissive XSS/HTML filter for admin-only use. |
| [filter\_xss\_bad\_protocol](https://api.drupal.org/api/drupal/includes%21common.inc/function/filter_xss_bad_protocol/7.x) | Processes an HTML attribute value and strips dangerous protocols from URLs. |
| [format\_string](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/format_string/7.x) | Formats a string for HTML display by replacing variable placeholders. |
| [get\_t](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/get_t/7.x) | Returns the name of the proper localization function. |
| [st](https://api.drupal.org/api/drupal/includes%21install.inc/function/st/7.x) | Translates a string when some systems are not available. |
| [t](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/t/7.x) | Translates a string to the current language or to a given language. |
| [\_filter\_xss\_attributes](https://api.drupal.org/api/drupal/includes%21common.inc/function/_filter_xss_attributes/7.x) | Processes a string of HTML attributes. |
| [\_filter\_xss\_split](https://api.drupal.org/api/drupal/includes%21common.inc/function/_filter_xss_split/7.x) | Processes an HTML tag. |

# Safely Impersonating Another User

### Impersonating Possibilities

There are many times when you may want your code to "impersonate" another user. An example of this is when a user takes an action that triggers another process. If that other process should be done as a different user then you want to impersonate that other user.

Here is some example code to impersonate another user which is **unsafe:**

global $user;

$original\_user = $user;

$user = user\_load(array('uid' => 1));

// Take your action here where you pretend to be the user with UID = 1 (typically the admin user on a site)

// NOTE: - this is the unsafe part - if your code here fails, then the user suddenly has the permissions of UID 1!

$user = $original\_user;

The safe way to implement this is to use the function [session\_save\_session()](http://api.drupal.org/api/function/session_save_session/5) (D6) or [drupal\_save\_session()](http://api.drupal.org/api/function/drupal_save_session/7) (D7) as follows:

**For D7:**

global $user;

$original\_user = $user;

$old\_state = drupal\_save\_session();

drupal\_save\_session(FALSE);

$user = user\_load(1);

// Take your action here where you pretend to be the user with UID = 1 (typically the admin user on a site)

// If your code fails, it's not a problem because the session will not be saved

$user = $original\_user;

drupal\_save\_session($old\_state);

// From here on the $user is back to normal so it's OK for the session to be saved