Security

TABLE OF CONTENTS

Overview

Drupal is a highly secure platform mostly due to the tireless efforts of the security team.

Sanitizing on output to avoid Cross Site Scripting (XSS) attacks

The Twig theme engine now auto escapes everything by default. That means, every string printed from a Twig template (e.g. anything between ``) gets automatically sanitized if no filters are used.

See Filters - Modifying Variables In Twig Templates for the Twig filters available in Drupal. Notably, watch out for the “raw” filter, which does not escape output. Only use this when you are certain the data is trusted.

When rendering attributes in Twig, make sure that you wrap them with double or single quotes. For example, class="{{ class }}" is safe while class={{ class }} is not safe.

In order to take advantage of Twig’s automatic escaping (and avoid safe markup being escaped) ideally all HTML should be outputted from Twig templates.

.htaccess magic

Disallowing access to users coming from a domain

Requests from specific domains can be blocked by adding the following to .htaccess:

```
RewriteCond %{HTTP_REFERER} domain-name\.com [NC]
RewriteRule .* - [F]
```

For multiple domains, use something similar to the following:

```
RewriteCond %{HTTP_REFERER} domain-one\.com [NC,OR]
RewriteCond %{HTTP_REFERER} domain-two\.com RewriteRule .* - [F]
```

Blocking core Drupal pages

Files such as CHANGELOG.txt can be used to quickly identify security vulnerabilities in your Drupal installation to a malicious script or user. While there are a number of ways to identify the version of Drupal that you are running, one quick addition to your .htaccess file can make it slightly le obvious.

```
# Various alias rules
Redirect 404 /CHANGELOG.txt
Redirect 404 /COPYRIGHT.txt
Redirect 404 /cron.php
Redirect 404 /INSTALL.mysql.txt
Redirect 404 /INSTALL.pgsql.txt
Redirect 404 /INSTALL.sqlite.txt
Redirect 404 /INSTALL.txt
Redirect 404 /install.php
Redirect 404 /LICENSE.txt
Redirect 404 /MAINTAINERS.txt
Redirect 404 /PATCHES.txt
Redirect 404 /README.txt
Redirect 404 /update.php
Redirect 404 /UPGRADE.txt
Redirect 404 /web.config
```

Blocking file resources from all but a handful of sites

You may want to keep a specific directory from being accessed by the general public, unless it's being pulled by a particular website. This example shows blocks requests to the /sites/default/files directory unless the request comes from www?, prod-kb, or the kb subdomains of example.com.

```
RewriteCond %{REQUEST_URI} ^/sites/default/files
RewriteCond %{HTTP_REFERER} !^http://prod-kb.example.com [NC]
RewriteCond %{HTTP_REFERER} !^http://kb.example.com [NC]
RewriteCond %{HTTP_REFERER} !^http://(www.)?example.com [NC] RewriteRule .* - [F]
```

Time-based blocks

If you are only allowed to expose your website for a specific time period, you can do that. This condition and rule blocks access until 4 PM.

```
RewriteCond %{TIME_HOUR} ^16$
RewriteRule ^.*$ - [F,L]
```

Blocking HTTP commands

You may not want to allow certain types of commands to be proceed by your site.

This blocks any HTTP request that is not a GET or a POST request.

```
RewriteCond %{REQUEST_METHOD} !^(GET|POST)
RewriteRule .* - [F]
```

Block specific user agents

If your website is the victim of a DDoS attack, and you want to block a group of IP addresses using the same User Agent, the following code may be helpful. Replace the UserAgent with the name of the agent you want to block:

```
RewriteCond %{HTTP_USER_AGENT} UserAgent
RewriteRule .* - [F,L]
```

You can also block more than one User Agent at a time with the [OR] ('or next condition') flag, and the [NC] ('no case') flag renders the string case insensitive. Here are some examples of some user-agents with properly escaped regexes:

```
RewriteCond %{HTTP_USER_AGENT} Baiduspider [NC,OR]
RewriteCond %{HTTP_USER_AGENT} HTTrack [NC,OR]
RewriteCond %{HTTP_USER_AGENT} Yandex [NC,OR]
RewriteCond %{HTTP_USER_AGENT} Scrapy [NC,OR]
RewriteCond %{HTTP_USER_AGENT} Mozilla/5\.0\ \(compatible;\ Yahoo [NC,OR]
RewriteCond %{HTTP_USER_AGENT} AppleNewsBot [NC,OR]
RewriteCond %{HTTP_USER_AGENT} Googlebot [NC,OR]
RewriteCond %{HTTP_USER_AGENT} Mozilla/5\.0\ \(compatible;\ YandexBot [NC]
RewriteRule .* - [F,L]
Important
```

Properly escape characters inside your regex (regular expressions) to avoid website errors.

HTTP_USER_AGENT can use regex as an argument. As seen in the example above, many User Agents will require regex due to the complexity of their name. Rather than creating the rule manually, websites such as https://www.regex-escape.com/regex-escaping-online.php can help construct a properly-escaped regex quickly.

How to test that the block is working

Test that the site is responding:

```
curl -H "host:www.url_you_are_testing.url http://localhost/
```

Test that the user-agent (Pcore as an example) is indeed blocked:

```
curl -H "host:www.url_you_are_testing.url" -H "user-agent:Pcore-HTTP/v0.25.0" http://localhost.com/
```

Block traffic from robot crawlers

While a robot crawler may not technically be an attack, some crawlers can cause real problems. You can use this when the robots do not obey the robots.txt file, or if you need an immediate block, because robots.txt is generally not fetched immediately by crawlers.

```
RewriteCond %\{HTTP_REFERER\} ^$
RewriteCond %\{HTTP_USER_AGENT\} "<exact_name_for_the_bot>"
RewriteRule ^(.*)$ - [F,L]
```

Blocking hotlinks

The last thing most website owners want is other websites stealing their content, or worse - hotlinking to their images and stealing their bandwidth. Here s a simple bit of code that prevents it-modify domain.com to your domain name:

```
RewriteCond %{HTTP_REFERER} !^$
RewriteCond %{HTTP_REFERER} !^http://(www\.)?domain.com/ .*$ NC
RewriteRule \.(gif|jpg|swf|flv|png)$ /feed/ R=302,L
```

Reviewing Server logs

Find the most frequent IP addresses

```
awk '{print $1}' access.log| sort | uniq -c | sort -nr| head
```

Find the most frequent User Agents

```
awk -F\" '{print $6}' access.log | sort | uniq -c | sort -nr | head
```

Find the most frequent URLs visited on your site

```
awk -F\" '{print $2}' access.log| awk -F? '{print $1}' | sort | uniq -c | sort -nr | head
```

This query strips the variables (anything after a question mark in your URL).

Read more on Acquia.com

General overview of harding your Drupal site

from Acquia.com

Ensure up-to-date backups are safe and secure

- Initiate a production database backup
- Download a copy of recent database backups, and keep updated copies offsite
- If possible, also take backups of the file system

Ensure Drupal Core and Installed Modules are up to date Drupal Core updates often contain security patches. Outdated, unmaintained modules often contain known security vulnerabilities.

- Look for projects and modules covered by the Drupal Security Advisories
- Remove obsolete and unused modules
- Check for available updates under the Drupal admin console, or by using drush or composer.

Perform a user audit

- Ensure permissions are restricted and implemented correctly
- Remove any old or unneeded admin or privileged accounts

If a breach has occurred or internal threat, an attacker or internal threat may have added user(s) to retain access.

- Check for any new or unexpected user accounts

Password Checks

Bad passwords are the most common cause of site compromise.

- Ensure strong password requirements are enforced. A community contributed module that offers this functionality is Password Policy.
- Perform a check for bad passwords. A community contributed module that offers this functionality is Drop the Ripper

2-Factor Authentication

- Enforce 2-factor authentication (especially for admin and/or privileged accounts) to mitigate the threat of compromised passwords.
- **Review Site Functionality**
- Check that file uploads are restricted to intended file extension type (e.g. Do not allow .html uploads for an image)
- Ensure any sensitive data files are uploaded to secure directories only (e.g. Do not place personal data (PII) such as CVs or job applications in public ‘files’ directories)
- Review controls on web forms

Attackers will often target forms that generate outbound emails (e.g. “refer a friend” or “contact-us”)

Try to keep messages generated from forms generic Ensure CAPTCHA controls are used to prevent abuse

Web Application Firewall (WAF)

If a WAF is not already in place, Acquia strongly recommend implementing one.

Acquia Cloud Edge Protect is Acquia’s WAF offering.

Edge Protect provides advanced security controls to restrict and block attacker traffic before it reaches the application stack. Common attack methods are identified and blocked automatically. WAFs are extremely effective in mitigating (D)DOS attacks.

API functions best practices

From Writing secure code for Drupal

- Use t() and \Drupal::translation()->formatPlural() with placeholders to construct safe, translatable strings. (See Translation API overview for more details.)
- Use Html::escape() for plain text.
- Use Xss::filter() for text that should allow some HTML tags.
- Use Xss::filterAdmin() for text entered by admin users that should allow most HTML.
- Use UrlHelper::stripDangerousProtocols() or UrlHelper::filterBadProtocol() for checking URLs - the former can be used in conjunction with SafeMarkup::format() - Oops, SafeMarkup was removed from Drupal 9. Rather use FormattableMarkup

Strings sanitized by `t()`, `Html::escape()`, `Xss::filter()` or `Xss::filterAdmin()` are automatically marked safe, as are markup strings created from render arrays via Renderer.

While it can also sanitize text, it's almost never correct to use `check_markup` in a theme or module except in the context of something like a text area with an associated text format.

checkPlain

This is from the file web/modules/custom/rsvp/src/Controller/ReportController.php

Here we pass `SafeMarkup::checkPlain` to array_map to call it on each entry in the array. The entry array looks like:

```
$entry['name']="fred",
$entry['email']="fred@bloggs.com"
```

$rows = array_map('Drupal\Component\Utility\SafeMarkup::checkPlain', $entry);

Here is the whole function.

```
$rows = $this->getAllRSVPs();
foreach ($rows as $row) {
  // Sanitize each entry.
  $rows_array[] = array_map('Drupal\Component\Utility\SafeMarkup::checkPlain', $row);
}
```

Html::escape

If you have html like this: `<script>alert(2)</script>` which will be output as a mess with these sorts of characters: `&amp;`, `&lt;` etc. use Html::escape to avoid this.

$rows_array[] = array_map('Drupal\Component\Utility\Html::escape', $row);

Drupal.checkPlain()

It is best practice to use the server to sanitize text, but there may be situations where you need to lean on the client (browser) to provide additional or temporary sanitization, such as an HTML element that updates on the client side as the user enters text.

These examples are for use cases of outputting text to the DOM. If you're sending text to the server, such as when making an API call, you should review Back End practices.

You can use `Drupal.checkPlain()` to escape basic characters and prevent malicious elements being introduced into the DOM, avoiding some basic Clickjacking techniques.

Bad Practice:

```
var rawInputText = $('#form-input').text();
```

Good Practice:

```
var rawInputText     = $('#form-input').text();
var escapedInputText = Drupal.checkPlain(rawInputText);
```

Use the database abstraction layer to avoid SQL injection attacks

Bad practice: Never concatenate data directly into SQL queries.

```
\Database::getConnection()->query('SELECT foo FROM {table} t WHERE t.name = '. $_GET['user']);
```

Good Practice:

Use proper argument substitution. The database layer works on top of PHP PDO, and uses an array of named placeholders:

```
\Database::getConnection()->query('SELECT foo FROM {table} t WHERE t.name = :name', [':name' => $_GET['user']]);
```

For a variable number of argument, use an array of arguments or use the select() method. See examples of each below:

```
$users = ['joe', 'poe', $_GET['user']];
\Database::getConnection()->query('SELECT f.bar FROM {foo} f WHERE f.bar IN (:users[])',  [':users[]' => $users]);
```

```
$users = ['joe', 'poe', $_GET['user']];
$result = \Database::getConnection()->select('foo', 'f')
  ->fields('f', ['bar'])
  ->condition('f.bar', $users)
  ->execute();
```

When forming a LIKE query, make sure that you escape condition values to ensure they don't contain wildcard characters like `"%"`:

```
db_select('table', 't')
  ->condition('t.field', '%_' . db_like($user), 'LIKE')
  ->execute();
```

Make sure that users cannot provide any operator to a query's condition. For example, this is unsafe:

```
db_select('table', 't')
  ->condition('t.field', $user, $user_input)
  ->execute();
```

Instead, set a list of allowed operators and only allow users to use those.

`db_query`, `db_select`, and `db_like` were deprecated and removed from Drupal 9 - instead you should use a database connection object and call the query, select, and escapeLike methods on it (the parameters are the same).

CSRF access checking

CSRF (Cross-Site Request Forgery) protection is now integrated into the routing access system and should be used for any URLs that perform actions or operations that do not use a form callback. In previous versions of Drupal, it was necessary to add a generated token as a query parameter to a URL and check this token manually in either the callback or the access callback. Now you can simply use the '_csrf_token' requirement on a route definition. Doing so will automatically add a token to the query string, and this token will be checked for you.

example.routing.yml

example:

```
  path: '/example'
  defaults:
    _controller: '\Drupal\example\Controller\ExampleController::content'
  requirements:
    _csrf_token: 'TRUE'
```

Note that, in order for the token to be added, the link must be generated using the `url_generator` service via route name rather than as a manually constructed path.

```
$url = Url::fromRoute(
  'node_test.report',
  ['node' => $entity->id()],
  ['query' => [
    'token' => \Drupal::getContainer()->get('csrf_token')->get("node/{$entity->id()}/report")
  ]]);
```

See API reference: CsrfTokenGenerator::get

To validate token manually (e.g. without adding `_csrf_token: 'TRUE'` to your `mymodule.routing.yml` file) at the route destination you can use the token and value used for generating it.

```
// Validate $token from GET parameter.
\Drupal::getContainer()->get('csrf_token')->validate($token, "node/{$entity->id()}/report");
```

Note. regarding anonymous users. Currently the `_csrf_token` check fails for users without an active session, which includes most anonymous users. See: #2730351: CSRF check always fails for users without a session

Resources

- Security on Drupal.org
- Blocking access using rewrites (Acquia.com)
- .htaccess documentation (Acquia.com)
- Ban module in Drupal core overview for blocking IP addresses (Acquia.com)
- Advanced Ban module
- Restricting website access (Acquia.com)
- Writing secure code for Drupal from Drupal.org update August 2022
- Twig Filters - Modifying Variables In Twig Templates
- Translation API overview on Drupal.org updated August 2022
- CSRF access checking on Drupal.org updated March 2023

Back to top

Page last modified: Jul 31 2023.

Edit this page on GitHub