




# Hardening Drupal with WebAssembly

By  Jesús González

At 2023 / 05

10 mins reading

## Intro

[Drupal](#) is one of the most popular Content Management Systems (CMS), powering well-known websites such as the [European Union](#), [NASA](#) or [Tesla](#). Like any other complex software, vulnerabilities are discovered in Drupal or in the underlying PHP components over time, and in many cases, the necessary updates are not promptly implemented. In traditional stacks, where components have direct access to the underlying operating system, certain vulnerabilities could be exploited to compromise the system.

This article explores how Drupal can benefit from the capabilities-based security model offered by [WebAssembly](#), a portable binary format that allows execution of code in a safe and efficient manner. By deploying Drupal within a WebAssembly-based stack, it gains an additional security layer, protecting against a wide range of vulnerabilities, including those that may not be public yet but can be preemptively mitigated through these mechanisms.

In addition, we provide a practical implementation of deploying Drupal in Apache with [mod\\_wasm](#), showcasing the seamless integration of the [WebAssembly build of PHP](#). This novel stack not only improves Drupal's security measures, but also unlocks the ability for integration with other WebAssembly-powered components.

# TL;DR:

Try out Drupal running into a WebAssembly environment:

```
docker run -p 8080:8080 ghcr.io/vmware-labs/httpd-mod-wasm:latest
```

Go to <http://localhost:8080/drupal>

# Understanding WebAssembly and mod\_wasm

[WebAssembly](#) (Wasm for short) is an open, portable and efficient binary instruction format, supported by many [programming languages](#). It follows a secure-by-design approach, implementing a deny-by-default security model that requires explicit authorization for accessing external resources like the file system. This inherent security makes WebAssembly an excellent choice for running untrusted code while ensuring the integrity of the main application and the underlying system.

All major browsers already provide built-in WebAssembly support. Additionally, it can be run on the server-side using a dedicated runtime. This versatility means WebAssembly can execute near-native performance code in a secure environment on both the web and the server. It also provides the ability to seamlessly interoperate binary modules written in different programming languages.

At [Wasm Labs](#), we aim to enable the execution of traditional applications within a WebAssembly environment with little to no modifications. The goal is to bring the benefits of Wasm to as many developers as possible without a steep learning curve. This led us to develop [mod\\_wasm](#), an Apache server extension that allows Wasm binaries to handle HTTP requests. This extension facilitates such a handling in virtually any programming language with WebAssembly support.

With `mod_wasm`, the stack to run PHP applications like Drupal closely resembles a traditional stack with a few modifications. The Apache HTTP server and the Drupal packages remain unchanged. However, instead of loading the `libphp.so` extension module, it incorporates `mod_wasm.so`. In addition, instead of relying on the traditional PHP interpreter, it utilizes a PHP build in the WebAssembly binary format. For this purpose, we can utilize the binary provided by the [WebAssembly Languages Runtime](#) project. This [pre-built binary](#) bundles all the necessary [PHP requirements](#) for Drupal, including PDO, DOM, GD, and more. As of the time of writing, the latest PHP Wasm build available is [8.2.0](#).

# Running Drupal in Apache with `mod_wasm`

The quickest way to try by yourself the full WebAssembly-based stack (Apache + `mod_wasm` + PHP Wasm + Drupal) is checking out the [Docker container](#) we prepared for demo purposes. It includes Drupal in different flavors among other [examples](#).

# Pre-initialized Drupal 10 instance

This provided instance of Drupal 10 is pre-configured and pre-initialized with some initial content. This pre-initialized state allows for a quick demonstration of the fully functional Wasm stack and showcases how all Drupal's PHP dependencies are met and fully operational.

As usual, you can easily install new Drupal extensions and themes via `composer`, provided their required version dependencies are met.

To access the administrator account, simply log in with the credentials "admin/admin".

URL: <http://localhost:8080/drupal>

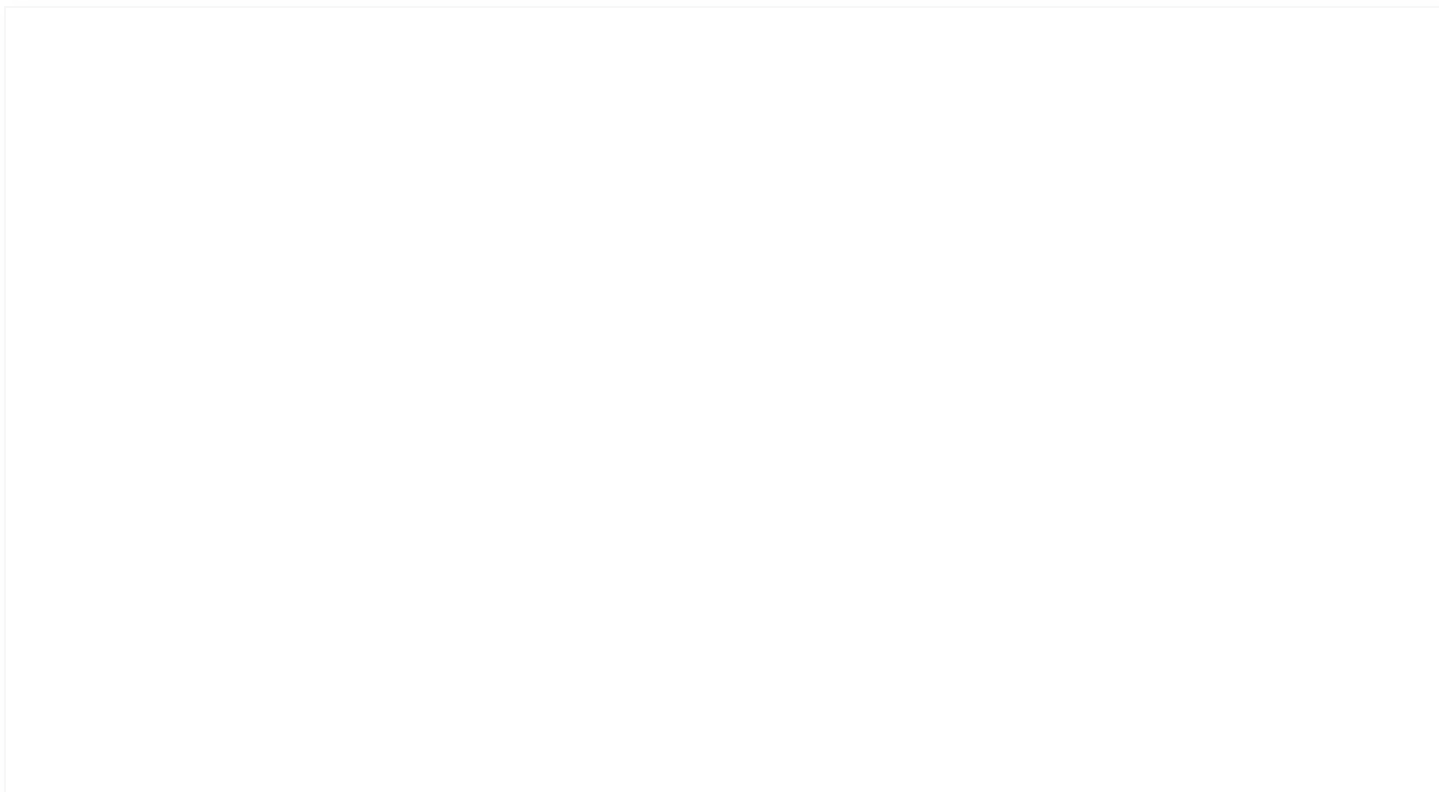


## Drupal 10 from Scratch

This particular instance of Drupal is uninitialized, providing users with the opportunity to explore and test the Drupal setup process within the WebAssembly stack.

During the setup process, Drupal may issue warnings regarding disabled OPcode caching and limited date range. We will address these limitations later on, but for now, you can safely ignore them.

URL: <http://localhost:8080/drupal-10-zero>



## Custom Setup

To try the WebAssembly stack on your own setup, you will first need to download the `mod_wasm` extension and drop it into your Apache's modules directory.

- If you are running Apache on Linux, you can get the latest release directly from [GitHub](#).
- For Windows users, `mod_wasm` can be downloaded from [Apache Lounge](#).
- Alternatively, you have the option to compile it yourself by following the provided [build instructions](#).

Setting up Apache for `mod_wasm` and Drupal is a straightforward process. Simply update your Apache's `httpd.conf` configuration file by loading the `mod_wasm.so` extension module, specifying the path to the PHP Wasm binary, and configuring specific [directives](#) for PHP and Drupal.

```
LoadModule wasm_module modules/mod_wasm.so
```

```
<Location /drupal>  
  AddHandler wasm-handler .php  
  WasmModule /path/to/wasm/php-cgi-8.2.0.wasm  
  WasmDir /var/www/drupal  
  WasmMapDir /tmp /var/www/drupal/tmp  
  WasmEnv TMPDIR /tmp  
  WasmEnableCGI On  
</Location>
```

# Enhanced Security Through Sandboxing

As we mentioned earlier, WebAssembly follows a deny-by-default strategy, limiting access to different system resources unless explicitly authorized. In the previous configuration example, directives like `WasmDir` or `WasmMapDir` exemplify this approach. By authorizing access only to `/var/www/drupal/` and mapping a virtual `/tmp` directory to `/var/www/drupal/tmp`, the PHP Wasm interpreter is restricted from accessing other directories. This setup applies specifically to `.php` resources within the `/drupal` location. A different configuration could be applied to different directories, providing a secure environment for multi-tenant deployments.

These additional protections provide an extra layer of security without relying solely on operating system-level permissions. They grant developers and sysadmins improved control over access permissions, reducing the attack surface and ensuring better isolation between unrelated modules.

WebAssembly has proven effective in mitigating various vulnerabilities, including those not yet public. Dive deeper into this topic with our article ["Mitigating PHP Vulnerabilities with WebAssembly"](#) and discover the benefits of WebAssembly for enhancing PHP security.

# Considerations and Limitations

Before deploying Drupal into a Wasm-based stack, it's important to be aware of certain limitations.

The primary limitation is the current lack of socket support in `mod_wasm`, which prevents PHP Wasm from accessing the network and using client-server databases like MySQL and PostgreSQL. All different instances in the demo container were configured to use SQLite which is embedded in the PHP Wasm build. This may be good enough for certain web sites, but it is not suitable for many others.

There are ongoing efforts to make socket support mainstream in WebAssembly. The [standardization](#) process is making good progress, including [functional demos](#). Some runtimes, such as [WasmEdge](#), already offer socket support. In fact, we successfully [connected a WordPress instance to a MySQL database](#) using this specific runtime.

The absence of sockets also means that PHP in Wasm cannot perform outgoing HTTP requests for downloading Drupal extensions or checking for updates.


In addition, the current WebAssembly version 1.0 only supports 32-bit memory addresses. Consequently, PHP Wasm can only [handle 32-bit integers](#). Fortunately, the [WebAssembly 2.0](#) draft includes a [64-bit memory space](#), addressing this limitation. Also, PHP's [Opcode caching](#) feature is disabled in PHP Wasm. Drupal promptly detects these limitations during the installation process.



# Conclusions and Future Work

We explored deploying Drupal within a WebAssembly-based stack. By adopting a capabilities-based security model and integrating Drupal with `mod_wasm`, you can mitigate certain vulnerabilities. The integration of WebAssembly with the Apache server enables the execution of PHP code in a secure environment.

Moving forward, we will work on introducing socket support in `mod_wasm` to enable network access to traditional databases and outgoing HTTP requests, further expanding the capabilities of Drupal deployments.

Embracing WebAssembly opens doors to a more secure Drupal ecosystem. Try out our [Drupal demos](#) or directly [download mod\\_wasm](#) for your own setup. And feel free to share your comments with us on [Twitter](#) and [GitHub](#). If you liked this project, [give us a](#) !

ABOUT THE AUTHORS





Jesús González

Staff 2 Engineer at OCTO

SHARE IT

Twitter

LinkedIn

Facebook

Do you want to stay up to date with  
WebAssembly and our projects?

Follow us on Twitter

vmware®

© 2023 VMware, Inc

[Terms of Use](#)

[Your California Rights](#)

[Privacy](#)

[Accessibility](#)

[Trademarks](#)



*This Website does not use Cookies or any personally identifiable user data.*