WHITE PAPER

The Costs of Building PHP In-House

Executive Summary

Teams that build PHP in-house often experience unforeseen costs and consequences. This white paper discusses those costs and how teams can avoid them.

Contents

Why Do Teams Want to Build PHP In-House?

There are a range of reasons why teams might want to build PHP in-house. It could be as simple as a team wanting to use a different PHP version than the one provided by their operating system. As an example, Linux operating systems generally provide a single PHP version per operating system version. Ubuntu 20.04, for instance, ships with PHP 7.4. If you wanted to use a different PHP version, you would either need to switch to a version of the operating system that shipped that PHP version, which could impact other system dependencies and security, or you would need to manually build PHP on your existing system. Similarly, teams might want to pin to a very specific PHP version regardless of which operating system they are on. That decision is often made to insulate the team from operating system changes.

As an example, a team might need to upgrade to a newer version of an operating system after their previous version reached end of life. However, they might not want to perform a PHP migration at the same time. For risk-averse organizations, like those in the financial or healthcare verticals, performing those migrations in tandem might introduce an unacceptable level of risk. Pinning to a specific version for long enough will lead to another use case for building PHP in-house: end of life patching. Teams that want to stay on an end of life PHP version will either need to find commercial LTS or build from source and backport security patches themselves.

A less common reason why teams might want to build PHP in-house is performance optimization. In certain situations, streamlined PHP runtimes (with pared down integrations and features) can lead to reduced memory usage and better overall performance. That said, this is an older approach, and is largely obsolete with modern packaging, which is modular and allows installing and enabling only the PHP extensions you require.

As we discuss later, all of these approaches can introduce both risk and costs to teams that are not equipped to handle them.

What Does Building PHP Look Like in Practice?

Building PHP in-house might seem simple at first. Getting started is as simple as downloading your desired version from php.net. You will find several tarball archives with varying compression schemes, including gzip, bzip, and xz; file sizes change dramatically between these different zipping mechanisms. For teams that want to build based on end of life versions, they will need to find their version in the archives

> For the purposes of this white paper, we will discuss from the perspective of a team that wants to compile on Linux. For teams that want to compile on IBM I or Windows, there will be issues specific to those environments that they will need to consider.

Once you have your archive extracted, you will run **`./configure`**, which is a script that works with **`autoconf`** in order to prepare the package for compilation. Run **`./configure --help`** to get a list of the various compilation options. The first set of options presented are common options you see anytime you configure an application using **`autoconf`**, and includes things such as where cache files are stored, the prefix used for installation, etc.

But this is only one screen of those options. This is just the tip of the iceberg, however; there are 14 additional pages (assuming a standard terminal size) of compile options, typically relating to individual extensions or core PHP features. That means you need to know exactly which ones you want to enable and disable.

However, just building that basic PHP version from an existing version is relatively easy. The true complexity, and the accompanying costs, come next.

The Costs of Building PHP In-House

Building PHP in-house might seem like a cheaper alternative to paying for long-term support at first, but there are a number of costs teams will need to account for before moving forward. In the following sections, we detail some of those costs, including dependencies, testing, installation and deployment, packaging, and more.

DEPENDENCIES

Building PHP requires standard build tools: **`autoconf`**, **`make`**, GCC, the C++ compiler, and more. Typically, your operating system provides these to you via a single package. However, some features of the PHP core functionality, as well as individual extensions, have additional dependencies. Some are packaged with the PHP source, but others are not.

As an example, if you need MySQL or MariaDB support, you'll need the development packages for those. If you were to pass the appropriate flag to **`./configure`** to enable MySQL or MariaDB support without first installing the development libraries, the script will fail and report an error indicating you are missing the libraries. The messages provided are generic, however, and do not indicate the system package you need; you will need to determine that on your own.

Once you have the correct package tracked down, you will install the package and start the whole process over again. Because it is a linear configuration process, it will always provide an error for the first issue it runs into, then stop. This means, in a best case scenario, you will go back and forth configuring, hunting down packages, and installing packages until you have all the packages you need in place. We say this is a "best case" because there are going to be cases where you find that versions of dependencies available on your operating system are not even compatible with PHP, so you will need to compile them separately then tell configure where they live. Not everyone will know how to overcome these issues, and it can be a big time-sink for teams completing the process for the first time.

After you are able to successfully configure, you will call **`make`** to generate a **`Makefile`**. This will take time. Even with a modern multi-core machine with lots of memory and an SSD, it can take upward of 30 minutes depending on the extensions you build.

OpenSSL or MySQL, as an example, are large and can add significant time to the build process. Even when configuration succeeds and all dependencies are in place, it's not uncommon to encounter a compilation error. This is often due to slight differences in dependencies, and the result will be additional time spent troubleshooting error messages, correcting dependencies, and reconfiguring the build.

TESTING

Once you finish running **`make`**, it will tell you to run **`make test`**. This command runs the PHP test suite, which generates a long list of results, which you should pipe to a log file for review. At the tail end of the results, you will see a tally of how many tests were run, including how many were *skipped*, how many *failed*, and how many generated *warnings*. It is typically safe to ignore these, but you will always want to test your application against your build to make sure that it is actually going to run – especially if there are extensions where you have warnings or failures against tests that were specific to that extension, or for any functionality that you consume within PHP itself.

There is a lot that can go wrong here. You will always see a number of skipped tests, as well as a number marked as "expected" test failures. This means you will need to distinguish between tests that are safe to ignore, and tests that are NOT safe to ignore. Establishing a baseline of acceptable failures is not a onetime act, either. Teams will need to evaluate their baseline every time they encounter new failures when building.

This means that there will need to be some level of automation, and not all teams will be equipped to set up that automation. The output from this test file is a relatively irregular structure, so understanding the relevant differences and how to parse them will be difficult.

If you are backporting security patches, you're not quite done with testing. For each patch you deploy, you need to ensure that it's patching the vulnerability and then verify that the test passes. To do that, you'll need to know the test name, and you'll need to look at logs from your test results to make sure it passed. If it didn't, that means that it was either incorrect or didn't patch the issue cleanly. If that happens, you'll need to have enough C or C++ to go through and analyze and fix those issues.

INSTALLATION AND DEPLOYMENT

At this point, you have built PHP and are ready to deploy, posing your next challenge. One thing to note here is that you should never build on your deployment platform, and you should never have build tools on your production machine. Those tools can make it easier for attackers to gain access to the system and build malware. This can lead to things like privilege escalation, which can lead to even worse outcomes.

Because of that, you will want to build on a dedicated machine, and push to your production machines. But doing that is easier said than done. Teams can roll their own packaging, but getting the package to the deployment system is a different story. And once there, you will also need to ensure that the appropriate system dependencies are in place. Any development library you installed in order to fulfill a compilation dependency will require that the target system install the associated shared libraries. This will add complexity to your deployment solution.

Packaging

You might think that packaging would just entail creating an archive file containing the PHP binary and other artifacts that you can drop onto a production system. This could be a zip file or a compressed tarball that you unarchive on the target system to a known tree. However, you also need to consider the various dependencies you identified while building PHP, and account for those as part of your packaging process.

When it comes to your PHP build dependencies, there are generally two types of dependency packages, dynamic library packages, which are small ones that can be consumed; and development packages, which have the source code that you are going to link against and compile with.

When you build, you use the development packages. On your target (production) machines, you use dynamic library packages. As an example, if you are compiling with cURL on a Debian-based operating system, you might use "libcurl-dev" to compile PHP; your production machine will require the "libcurl" package. Because of that, when creating your PHP package, it will need to have a script in place to install all required dynamic library dependencies.

As such, creating the package isn't enough; you will also need mechanisms for running these scripts for installing dependencies. And, better, you should include some sort of script to verify the installation when complete; it should validate the PHP version installed, the location of configuration files, and that the expected modules are enabled.

Creating OS Packages

There is already a solution for everything detailed in the previous section: operating system packages. Most Linux operating systems use a package manager for managing system packages and their dependencies, and you can write your own packages. Most package managers allow specifying the file layout, package dependencies, and even pre- and post-installation scripts. This means you do not need to build your own installer or manager; you instead build a package from the artifacts of compiling PHP.

But what happens if you change operating systems or versions? If you change a version of your operating system, you will often find the package name or package version for your dependency has changed. This means you will need to update your package to reflect those dependencies, which will often require re-building PHP.

Creating a package that works with your package manager is only the first step of the solution; you still need to determine how to get it to your production system and install it.

You can push it manually. You will then need to invoke the package manager on the target system in order to install it. Depending on the package manager, this may not actually install dependencies. As an example, on Debian systems, if you use dpkg -i to install a package, it won't install dependencies; you have to run **`apt-get install --fix-broken`** afterwards. While it let's you know that you're missing dependencies (and which ones they are), and you can fix it, it's still an extra step. (You can now use apt install **`./package-name.deb`** in modern Debian systems, which will install dependencies as well.)

Ultimately, you could set up a package repository based on the operating system and package management tool you are using and set up that package repository on your target machine, but that means you will have another piece of infrastructure to maintain. That means keeping that infrastructure up to date and ensuring that it has additional security in place. You will likely need to put your package repository behind a firewall, which means that your build and production systems need to have access to that area. By this point, you are talking about a build server and a package repository, so you have multiple pieces of infrastructure that will need to be maintained – and that's in addition to the days and hours your expert personnel will need to compile all of this.

When to Build

The next question you will need to answer is, how is your team going to know when to create a new build? The release cadence for PHP is such that there is a new release every month. However, if you are only worried about a version that has reached end of life, then you may only be worried about CVEs and whether or not they apply to your application.

That means that when a new release is issued, you will need to read through the CVEs that are patched in that new release and determine if they impact your applications. If they do, then you know you will need to build.

But that brings more questions:

- How do you build the package?
- How do you test and validate it?
- How do you package PHP?

That means you need to have infrastructure for a CI/CD system, and a team with the requisite skills for creating and maintaining that CI/CD system. Of course, it's possible you already have CI/CD set up for your PHP application. Even then, the needs for building the language are different than the needs for building an application on top of that language. That means you will have different tool chains, and that you will need to maintain those toolchains going forward.

Lastly, how quickly can you do all of the above?

If there's a 0-day CVE announced by PHP, how quickly can your team find out that it exists, evaluate it, backport the patches, test everything, and deploy it? Does your team have the time to drop what they are doing and prioritize this? Are you willing, as a manager, to hire someone whose only job is to build your PHP instead of working on the actual applications that you're building? What happens if they are on vacation with no cellular service or internet access when that 0-day CVE comes out?

Personnel Skills

Teams considering building and deploying their own PHP need to honestly evaluate the security and programming expertise of the personnel they have in place. Do they have the skills necessary to evaluate the security patches against the versions you are using? If not, you should not be backporting security patches. Doing so will put your business at risk.

Security patches aside, does your team have skills necessary to resolve issues found during testing and validation? There are times where a new patch will introduce new test failures.

Are you able to put automation in place to understand when a new failure happened against your baseline? Does your team have the expertise needed to determine if it's related to the patch, or something else? If it's unrelated, can they resolve it in a way that will allow your builds to still work?

Another thing to consider for teams is, what happens if someone critical to this process leaves the company? Do you have a contingency plan in place if you are unable to build a patch for a critical CVE? These are all questions you will need to have ironclad answers to before you commit your organization to building your own PHP.

Development Impact

An often-overlooked area for maintaining end of life applications is the lost-opportunity cost of spending your resource hours on building your PHP instead of developing applications. PHP releases happen monthly, which means that, in any given month, there's a possibility that you will need to build PHP. That means you need to ensure your development team either has buffer time built into their schedule, or have dedicated personnel for this.

Ultimately, the decision to build PHP in-house comes down to cost. The up-front costs of dedicated personnel and the costs of maintaining infrastructure for building and patching can be substantial, and that doesn't account for the potential impact to your business opportunity costs, and the lost of trust that can happen if you run a vulnerable PHP version.

Alternatives to Building PHP In-House

If you have made it this far, and you were considering building PHP in-house, chances are you are second-guessing that opinion. The good news is that there are some good, cost-effective alternatives. In the following sections, we look at a few of those options, including using versions shipped with your operating system, community PHP versions, or, our favorite, commercial long-term support.

USE THE PHP VERSION SHIPPED WITH YOUR OPERATING SYSTEM

Generally speaking, an operating system provides three to five years of LTS. During this time, it will provide security patches for all the software it contains, including its official repositories.

The issue with that approach is that if you want to take advantage of newer PHP versions, you can't. You're locked into the version of PHP that ships with your operating system, for better or worse.

The other thing to consider is that some of the libraries that your application depends on may eventually end support for the PHP version you are on, or develop new versions that support PHP versions beyond the one you're currently on.

As an example, if you use Composer, you can use it to update your application libraries. But, you might find that some of those libraries release new versions that no longer support the PHP version you are on. This could mean that you no longer get security patches for those libraries, which means your security is tied to not only the PHP version you are on and the fact it receives security patches via the operating system, but also to whether or not your libraries are getting updated with security patches. If the libraries are not patching versions that support your PHP version, you may still be vulnerable.

USE COMMUNITY PHP VERSIONS

If you can't use the PHP version shipped with your operating system, the next alternative is using community PHP versions. For instance, if you want to run 8.0 or 8.1, and you are on Ubuntu 20.04, you could install the Sury repository. This allows you to run multiple different PHP versions and select which version you want. Or you could run multiple versions in parallel on Red Hat Enterprise Linux (RHEL) or other RHEL variants that are compatible with it.

The thing to understand is that these, generally, don't backport security patches for end of life versions of PHP. So, even though you're using this community version, you actually may not be getting the security patches you need in order to ensure your application is patched against CVEs that have been released.

USE COMMERCIAL PHP LTS VERSIONS

The other option, and one a lot of enterprise organizations turn to, is commercial PHP LTS. This approach best addresses many of the reasons why teams build PHP in-house. Zend, as an example, provides commercial PHP LTS for a minimum of two years after community support end of life, with some versions (like PHP 7.4) going far beyond that. Commercial PHP LTS from Zend also includes SLA-backed support, which can go up to 24/7/365 depending on the package.

Aside from the immediate benefit of being able to pin to PHP versions for longer periods of time, there is also the benefit of ensured compliance. If your organization has PCI or SOX standards that they need to comply with, they require you to be on secure versions of that software.

The last benefit of this approach, and there are others we don't cover here, is that using commercial PHP LTS versions allows teams the agility to upgrade their operating system and their PHP separately. While it might seem like a small thing, for teams managing complex or numerous applications, having the freedom to upgrade these separately can greatly reduce the complexity of managing these processes simultaneously.

Closing Thoughts

Ultimately, the choice between building PHP in-house and finding an alternative means of supported PHP comes down to cost. If you have the infrastructure in place, you have the skilled staff in place with the bandwidth needed to continuously execute on this concept, and you know with 100% certainty that the costs and consequences of maintaining that process are less than the costs and consequences of all available alternatives, then building PHP in-house may be an option. Even then, things can change, and the costs of building PHP in-house can quickly become a burden.

The good news is that there are a number of alternatives that can help teams avoid this burden.

Learn More About PHP LTS Options From Zend

Zend offers expert, dependable, long-term support for PHP that includes includes automated PHP updates — plus on-demand, consultative support from a certified PHP expert for all your applications running end of life PHP.

Learn more about our PHP LTS options by speaking with a Zend representative today.

CONTACT US

zend.com/services/php-long-term-support

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle. Our portfolio includes solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static & dynamic code analysis, version control, and more. With over 9,000 customers, Perforce is trusted by the world's leading brands, including NVIDIA, Pixar, Scania, Ubisoft, and VMware. For more information, visit www.perforce.com.