Melting the Flesh of PHP's Memory Hardening

Yifan Wu
UC Riverside

Xiaochuan Yu
UC San Diego

Zhiyun Qian
UC Riverside

Abstract

Heap allocators are responsible for efficiently allocating or releasing memory on the heap. In addition, they also commonly implement various mitigation measures to defend against heap-based memory corruption. Recently, the PHP project launched a heap hardening initiative aimed at stopping popular heap exploit techniques or restricting the exploit strategy space. In this paper, we conduct the first security study to understand the impact and effectiveness of these new protective measures. We find that while they are effective at stopping current-generation exploits, they fall short against determined attackers who will adapt. Through our analysis, we not only identify the flaw that allows specific mitigations to be bypassed, but also a new suite of novel exploitation strategies that work for the most common vulnerabilities involving out-of-bounds memory write and use-after-free write primitives. Notably, our strategy is generic across the built-in PHP objects and can still work even with a single-byte out-of-bounds memory write primitive. Finally, we evaluate our exploit strategies against five real vulnerabilities in an environment with all evaluated protections enabled. The results show that although the new protection measures can effectively defend against the exploitation of most vulnerabilities, the attack strategies proposed by this work can still make these vulnerabilities exploitable again. The identified flaw has since been patched after our responsible disclosure.

1 Introduction

PHP remains one of the most widely used programming languages for web development, with 72.2% of websites still relying on PHP as of 2026 [62]. Prominent platforms such as WordPress, HotCRP, and numerous other web frameworks are built on PHP. The popularity and extensive usage of PHP highlight its significance in web infrastructure and security.

In recent years, several high-impact PHP exploits, such as CVE-2024-2961 and CVE-2022-31626, underscore the feasibility of directly targeting the PHP interpreter for remote code execution. Unlike vulnerabilities in specific business logic within PHP applications, flaws within the PHP interpreter itself offer broader attack surfaces and higher exploitation potential. Recently, this has prompted PHP developers to introduce heap hardening techniques [59] as a countermeasure to mitigate the exploitation of PHP vulnerabilities. Heap hardening involves a range of strategies designed to fortify PHP's memory management mechanisms, such as making heap metadata read-only and isolating the PHP application heap from the heap that handles HTTP-related objects.

Nevertheless, the effectiveness of such heap hardening solutions has not been scrutinized. In this study, we systematically survey all built-in PHP object types that are generally available to PHP applications and identify feasible generic attack paths that do not depend on application-specific objects. From our investigation of PHP, we uncover implementation flaws that render some defenses vulnerable to bypass. In other cases, we identify alternative and universal exploit paths that are constrained by the defenses. Overall, we are the first to propose a comprehensive security analysis of proposed PHP heap hardening and a generic end-to-end remote PHP exploit strategy under minimal assumptions and weak bug primitives. The techniques work for local exploits, i.e., sandbox [19, 29] escaping [6], which are generally easier to construct, as well. Furthermore, we show the exploit techniques work for weak primitives, i.e., a single-byte OOB write. In response, we propose mitigation strategies and responsibly disclose our findings to the developers, offering recommendations for improving the security of PHP's memory management systems.

We demonstrate the effectiveness of our proposed exploit techniques by reviving four publicly available remote exploits that no longer work under the new PHP heap defenses. In particular, we demonstrated effective and reliable end-to-end exploits that can achieve control flow hijacking and arbitrary code execution in a fully remote setting, which prior academic research [21] did not consider. Furthermore, we demonstrate the same technique also works for local PHP sandbox escape.

The contributions of our work are as follows:

- **Security analysis of PHP's heap hardening defenses**:

We present the first comprehensive study of the latest PHP heap hardening techniques, identifying their weaknesses and limitations. We identify several strategies that enable full bypasses, including one exploiting an implementation flaw that we reported and that has since been fixed upstream.

• **Generic end-to-end exploit strategy under minimal assumptions, supporting remote exploitation.** We identify a feasible exploit path that relies only on built-in PHP object types, avoids application-specific objects, and still works with weak primitives such as a single-byte out-of-bounds write, even when all evaluated heap hardenings are enabled. By combining techniques discussed in this paper, we extend PHP memory-corruption exploitation to a fully remote setting without requiring a separate information leak.

• **Strong evaluation results with real CVEs:** We successfully demonstrated the strategy by developing stable exploits against real-world CVEs and CTF challenges (most are remote settings). We further propose two hardening patches that mitigate the evaluated exploit paths. Our open-sourced artifacts support reproducibility and future defense research.

2 Background & Motivation

In this section, we first briefly introduce the basic workflow of PHP applications. We then define the threat model and describe various vulnerabilities and attack capabilities considered in this study. We then describe the design and behavior of PHP memory management components that are most relevant to implementing new protections and developing bypass strategies. Finally, this section presents a series of heap hardening protections recently introduced by the PHP development team to mitigate memory corruption vulnerabilities.

2.1 Workflow of PHP Application

In a typical PHP web application workflow demonstrated in Figure 1, users access a specific URL through their browsers, and the request is initially received by a web server (also called middleware, such as Nginx, Apache, or httpd). The web server then forwards the dynamically processed part of the request to the PHP interpreter according to its configuration. To meet the demands of high concurrency in modern web applications, PHP usually employs PHP-FPM (FastCGI Process Manager) [20] as a process manager, which creates multiple interpreter instances (worker processes) by using the **fork** mechanism to parse and execute PHP scripts. A typical request handling workflow can be summarized as follows: User (or attacker) → Web server (middleware) → PHP-FPM → Worker process → PHP script parsing and execution.

Each stage of this workflow may expose an attack surface, spanning web-server vulnerabilities, memory errors in the PHP interpreter, and application-level logic flaws. In this work, we exclude middleware issues and script-level logic vulnerabilities (such as SQL injection and XSS), and instead

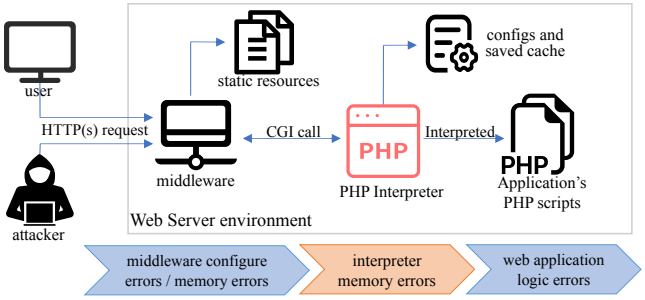

Figure 1: Workflow Overview of a PHP Web Application.

focus on protection-bypass techniques in the latest PHP interpreter, namely PHP-FPM and its worker processes.

From a functional perspective, web applications developed in PHP can be viewed as remotely callable functions. Users send HTTP requests to the server, and the PHP interpreter selects and parses the corresponding PHP script, performs the required operations, and returns the rendered HTML or other types of content. Generally, this process is stateless and non-interactive. PHP developers commonly use session IDs to manage multiple requests from the same user and maintain user-specific data across sessions. However, ordinary users and remote attackers have no direct access to the PHP interpreter or server-side session state. Their interaction is limited to issuing HTTP requests and observing the responses.

As an interpreted language, PHP applications are deployed as separate script files on the server and are parsed and executed at runtime by the PHP interpreter. In practice, only administrators can modify these scripts. As a result, remote attackers typically cannot supply arbitrary PHP code or freely shape heap allocations to aid exploitation, unlike in JavaScript or Python engine exploits. Instead, they must reach and trigger interpreter vulnerabilities indirectly through existing, legitimate application code paths. For example, CVE-2022-31626 can be triggered when an application calls `mysqli_real_connect()` to connect to a remote database and cause an overflow copy of the password.

Attackers have different capabilities in cloud computing environments. A typical scenario is that cloud service providers deploy servers running PHP interpreters, allowing users to upload and host their own PHP scripts for personal websites or remotely callable cloud functions. In such scenarios, PHP interpreters provide comprehensive sandboxing mechanisms (enabled via configuration files). These mechanisms allow server providers to restrict CPU and memory resources used by each script, control file access permissions, and disable certain sensitive built-in functions (such as `system()` and `socket()`), thereby preventing users from executing potentially malicious scripts. In this scenario, since attackers can upload and execute arbitrary PHP scripts, they possess greater flexibility for exploitation and can freely create various PHP

objects to assist their attack process.

2.2 Threat model

Corresponding to the two scenarios mentioned above, our work analyzes two primary attack scenarios targeting memory corruption vulnerabilities in the PHP interpreter: (1) remote code execution and (2) sandbox escape. In the remote code execution scenario, attackers cannot directly modify any content on the server or deploy new PHP code. Instead, they can only craft specific HTTP requests and leverage existing legitimate PHP scripts on the server to shape memory layouts, triggering and exploiting vulnerabilities within the PHP interpreter. In contrast, the sandbox escape scenario assumes that attackers can freely deploy PHP scripts on the server, running arbitrary PHP code within a restricted environment (sandbox) provided by the PHP interpreter. These two scenarios differ in the abilities of attackers to deploy PHP scripts. However, both aim at the same ultimate goal: to execute arbitrary system commands on the target server.

We assume that in both attack scenarios, attackers can achieve initial memory corruption via a vulnerability in the PHP interpreter, such as an out-of-bounds (OOB) or use-after-free (UAF) issue. Under our threat model, the attackers aim to exploit this initial memory corruption further and ultimately gain the ability to execute arbitrary system commands as the PHP interpreter. Our threat model and exploitation strategy are mainly independent of the PHP interpreter's deployment mode (e.g., embedded as a shared library or working standalone), as interpreter-level mechanisms govern them.

We also assume that the target PHP interpreter has all four protection mechanisms from the Heap Hardening initiative, which have been merged into the latest official release or have existing prototype implementations [2, 60]. The details of these protection mechanisms will be elaborated in §2.4. Additionally, apart from the known memory corruption vulnerability in the PHP interpreter, we do not assume other vulnerabilities, e.g., from other server components. The server environment is also assumed to enable all mainstream attack mitigation mechanisms, including Address Space Layout Randomization, stack canaries, and Non-eXecutable memory.

2.3 Memory management in PHP

To improve memory allocation performance, reduce fragmentation, and better support large objects, PHP has a custom memory manager known as Zend Memory Manager (ZendMM) [41] to manage heap memory. From a high-level perspective, ZendMM categorizes memory allocation into three cases based on allocation size:

• **Small**: Memory allocations smaller than 3 KB are managed by 30 singly linked lists, each containing fixed-sized slots ranging from 8 bytes to 3072 bytes.

• **Large**: Memory allocations larger than 3 KB but smaller than 2 MB are allocated from a larger cache structure called `zend_mm_chunk` in units of *pages*.

• **Huge**: Huge allocations that exceed 2 MB are directly allocated using the `mmap` system call.

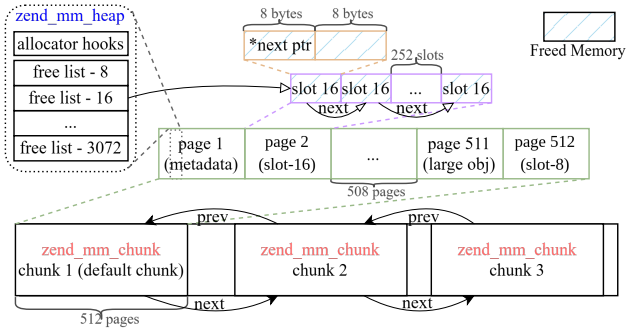

Figure 2: Overview of PHP Interpreter Memory Allocation.

As shown in Figure 2, from an implementation perspective, ZendMM manages memory through a multilevel structure: At the top level, there is exactly one `zend_mm_heap` object. This object maintains singly linked freelists for each slot size and manages a doubly linked list of multiple `zend_mm_chunk` structures, each with a fixed size of 2 MB. Each `zend_mm_chunk` contains a bitmap to record the availability of its 512 pages (each page is 4 KB in size).

For small-sized memory allocations, the freelists of slots follow a last-in-first-out (LIFO) strategy. Since all slots within the same freelist are of equal size, the allocator does not need to store size metadata. As a result, when a slot is freed, only the first 8 bytes are used to store the next pointer in the singly linked list. When a freelist is exhausted, ZendMM will allocate new memory in units of pages from the available space in a `zend_mm_chunk`. Similarly, large allocations will also request new memory in units of pages from `zend_mm_chunk`. However, when a large object is freed, it will be returned directly to `zend_mm_chunk`. The allocation from `zend_mm_chunk` uses a deterministic best-fit strategy. If the current `zend_mm_chunk` is exhausted, a new `zend_mm_chunk` is allocated and added to the top-level doubly linked list in `zend_mm_heap`.

Before processing each new request, the PHP interpreter initializes a default `zend_mm_chunk`. At the end of each request, the interpreter releases all extra `zend_mm_chunk` structures except the default one and clears all freelists in the default chunk. Consequently, data or heap layouts are not shared across requests. In other words, the initial state of the PHP heap memory is identical for each new request under normal circumstances. This uniform reset mechanism after each request, combined with the best-fit and LIFO allocation strategy, makes PHP memory allocations deterministic. That is, given

the same PHP script, the same requests can always produce an identical heap layout.

2.4 New defenses of PHP

The PHP development team has implemented four protection mechanisms as part of the "Heap Hardening" initiative since April 30, 2024. Two of these protections, Shadow Pointer and Unlink Abuse Prevention, have already been merged into official release since PHP 8.4.0, while the other two, Read-only Metadata and Heap Isolation, remain unreleased proposals or prototypes. Appendix A summarizes their timeline, release status, and upstream references.

Shadow Pointer (PHP 8.4.0+): Similar to pointer encryption named Safe-Linking [23] used by GNU libc or Pointer Authentication Codes (PAC) [30] by ARM, PHP's Shadow Pointer Protection provides integrity checks for singly linked freelists. Small allocations are organized using bins. Bins of the same size are grouped together as a singly linked freelist. An attacker can corrupt the `next` pointer and force the Zend Heap allocator to return a specific memory chunk on the very next allocation and write to it. This gives a powerful exploit primitive of *arbitrary allocation and writes*. PHP's Shadow Pointer aims at verifying the integrity of the freelist. To do this, this mitigation stores an additional encrypted copy of the `next` pointer at the end of each free memory slot. We illustrate this in Figure 3. The XOR-based encryption and decryption rely on a secret called the *shadow key* stored within the heap metadata and refreshed automatically by worker processes. To accommodate this extra piece of data, PHP increased the minimum slot size from 8 to 16 bytes — the first 8 bytes can continue serving as the `next` pointer of the freelist, while the additional 8 bytes at the end store the encrypted pointer copy. Each time a bin is retrieved from the linked list, the allocator

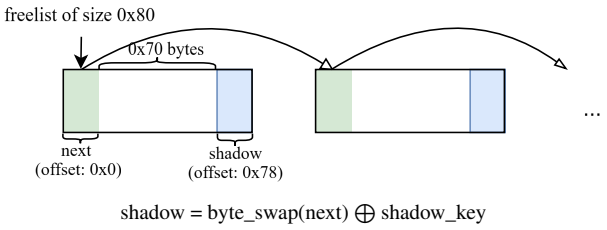

Figure 3: Singly Linked Freelist with Shadow pointer

checks whether the next pointer has been tampered with by comparing it against the decoded shadow pointer. Therefore, it prevents the hijacking of freelists, for example, through OOB write primitives from achieving *arbitrary address allocations and writes*. The ZendMM verifies the integrity of the pointer copy whenever allocating memory from the freelist and will immediately abort execution upon detecting a mismatch.

Unlink Abuse Prevention (partially merged): To prevent memory corruption attacks against doubly linked lists such as `zend_mm_chunk`, PHP introduced an integrity check mechanism similar to Safe-Unlinking [58], with a related `php_stream_bucket` check still under review. Specifically, when elements are unlinked from doubly linked lists, this mechanism verifies that the forward pointer `fd` and backward pointer `bk` satisfy the fundamental doubly linked list constraints: `P->fd->bk == P` and `P->bk->fd == P`. This mechanism prevents attackers from manipulating doubly linked list pointers to perform arbitrary address read/write operations or forge memory management structures, i.e., `zend_mm_chunk`.

Read-only Metadata (not merged): The Read-only Metadata protection aims to safeguard the integrity of function pointers in heap metadata at runtime. The heap management structure used by PHP, `zend_mm_heap`, is located within the first memory page of the default `zend_mm_chunk`. This structure contains several critical memory management function hooks (such as pointers to `malloc`, `free`, and `realloc`). If attackers corrupt these metadata pointers, subsequent memory allocation or deallocation operations could allow attackers to hijack the PHP interpreter's control flow easily. Thus, the PHP maintainers plan to dynamically alter the read-write permissions of the metadata page, granting temporary write permissions only during legitimate metadata updates to prevent the compromise of these critical pointers. As summarized in Appendix A, the custom-heap-disabling idea was rejected and the later Read-only Metadata prototype is not merged. We treat the later version as a proposed defense and included it in our evaluation as well.

Heap Isolation (pending merge): The Heap Isolation protection aims to separate attacker-controlled data structures, such as HTTP request headers and POST request data, from the application logic (i.e., memory allocations made by PHP scripts written by developers). By allocating request data to a dedicated heap separate from the main heap used by the application, Heap Isolation prevents attackers from directly manipulating the application's heap layout via crafted HTTP key-value pairs, making it more challenging to position overflowable objects adjacent to sensitive application objects. This separation mitigates targeted memory corruption through layout manipulation. In the future, the PHP maintainers plan to extend Heap Isolation at the object and function levels for stronger separation guarantees.

The roadmap also contains broader ideas such as guard pages, freelist randomization, and finer-grained type or size isolation. We discuss them only as future directions in §8.1 because they are not yet close to production deployment and there is no prototype available.

3 Overview

3.1 High-Level Goals

To transform memory corruption primitives such as use-after-free (UAF) and out-of-bounds (OOB) accesses into complete

remote code execution or local sandbox escape exploits, attackers typically need to perform multiple stages of primitive transformation. For example, converting an OOB primitive into an arbitrary address write primitive, or using a UAF primitive to derive an infoleak primitive. Additionally, some of these new primitives also simultaneously grant postconditions that serve as preconditions for subsequent primitives.

Memory protection mechanisms such as Heap Hardening reduce the exploitation strategy space by imposing additional preconditions or making existing preconditions tighter, i.e., harder to fulfill, at various known primitive transformation opportunities. To bypass these defenses, attackers need to systematically analyze the imposed constraints on primitive transformations and explore potential new exploit paths. Attackers typically follow two approaches: (1) achieving the same primitive and/or postconditions under the more stringent preconditions; (2) circumventing the restricted transformation by converting existing primitives into different, functional, but less constrained primitives. Eventually, attackers leverage these newly acquired primitives or postconditions to complete the exploitation process.

In the rest of the section, we will first thoroughly analyze the restrictions imposed by heap hardening protection mechanisms on primitive transformations. We will then explore possible new exploit paths and combine the two bypass strategies described above to propose a comprehensive and practical exploitation methodology.

3.2 Technical Challenges

Specifically, we find that the new heap mitigations render classical exploitation strategies ineffective and alternative strategies difficult, which we discuss below.

Freelist metadata corruption is no longer effective. Achieving a write-anywhere primitive via freelist hijacking is now particularly difficult. Specifically, attackers must forge both the freelist entry's next pointer and shadow pointer, which is encoded by the shadow key as illustrated in Figure 3. This renders attacks that rely on overwriting the next pointer [5, 10–12] infeasible.

Corrupting heap metadata is no longer effective. Hardening custom heap management function hooks significantly raises the bar for attackers seeking to hijack function pointers with controllable arguments. Exploitation techniques that target those read-only metadata regions [11] have to adapt.

Limited application objects. Besides heap metadata, an attacker can turn to application objects for heap spray and corruption. However, due to heap isolation, objects that are originally attacker-controlled, e.g., `$_GET`, `$_POST`, and `$_COOKIE` are now put in a separate heap region (i.e., user input heap). This significantly limits the available heap objects that an attacker can spray on the application heap, which is where vulnerabilities occur [38, 39]. We summarized only commonly allocated and controllable types in §6 and listed them in Table 1. The application may allocate other object types. However, to preserve generality, our exploitation technique considers only the objects listed in Table 1 as helpers.

Data Type	Descriptions
`zval`	PHP value reference or inline storage
`zend_string`	Inline storing a string
`zend_array`	Hash table in Zend
`zend_object`	Represents an object in PHP

Table 1: Basic Data Types Allocated When Parsing Requests

Data encoding restrictions. Previously, attackers could inject URL-encoded raw string data via POST request bodies to spray arbitrary raw bytes (with varying lengths) into the now-isolated heap region. This path is no longer available. Attackers can instead rely on common PHP application features to spray attacker-controlled objects. Under our threat model, usable generic options reduce to JSON parsing and XML parsing. We present a detailed analysis of those formats in §6. An attacker can craft such payloads and trigger object creation on the application heap during decoding. Specifically, to inject large buffers with attacker-controllable input, an attacker can choose string objects that will be parsed into `zend_string`. Unfortunately, PHP's string decoding function imposes restrictions. For example, `json_decode` emits escape sequences for characters above `U+007F`. This limits the number of bytes available for spraying.

One-shot interaction. In most cases, attackers can only interact with PHP applications indirectly through stateless HTTP requests and CGI calls. Such stateless indirect interaction presents unique challenges for exploitation. Due to PHP-FPM's share-nothing architecture [40], it will reset the Zend heap after handling a request. The attacker is thus limited to a one-shot interaction. When *shadow pointers* and *heap isolation* are enabled, this constraint makes it even harder to achieve the necessary memory layout for exploitation. In short, heap fengshui has to be done in a single HTTP request.

4 Bypass Shadow Key Protection

We develop two high-level solutions to bypass this protection. First, we identify a critical implementation flaw that allows an attacker to predict the shadow key value. Second, even if the implementation flaw is fixed (as it was after our reporting), we show novel strategies that no longer corrupt the `next` pointer, leveraging commonly found OOB and UAF primitives.

4.1 Shadow Key Derandomization

After analyzing the shadow key implementation, we find that it is first created in the main process and then inherited by

all worker processes. To prevent the shadow keys from being predicted, the shadow key of each worker process will be updated when the heap is torn down, after handling a request.

Unfortunately, there are two implementation issues: First, refreshing the shadow keys in worker processes does not refresh the key in the main process since they are in different memory spaces. Second, a newly spawned worker process does not update the inherited shadow key from the parent until after *handling* a request. Consequently, exposing the shadow key of a worker that handles its very first request effectively reveals the shadow key of the main process, which in turn compromises the key for every newly spawned future worker.

Specifically, we define an overflow that can overwrite both the next pointer and the shadow-key-encrypted pointer within a bin as a *long overflow*. We can first construct an out-of-bounds read primitive from a long overflow with the existing method shown in Figure 14 to leak the encrypted `next` pointer. As the shadow key is XORed with the next pointer, it is sufficient to leak the encrypted next pointer to recover the shadow key. Since corrupting any shadow pointer will force a new worker with predictable key, we can now corrupt the freelist with a fake `next` pointer with a matching encrypted `next` pointer (shown in Figure 3). This effectively revives freelist poisoning to derive arbitrary allocation and write primitives.

After responsibly disclosing this issue to the PHP developers, this implementation flaw has since been patched.

4.2 Pivot from Buffer Overflows

In this section, we present new exploit strategies for heap buffer overflows of any size in the PHP interpreter. At a high level, our strategy is to avoid corrupting the `next` pointer. Instead, we will identify other targets to spray for corruption and derive alternative exploit paths. Specifically, we will introduce a universal exploitation approach in detail that can systematically transform these limited vulnerabilities into full exploitation by completely controlling the general `zval` structure. Ultimately, this universal exploit strategy enables an attacker to achieve arbitrary read and write capabilities even after the shadow key derandomization issue has been patched.

Our strategy applies to overflows of any size. For large overflows that can reach the next free slot (and potentially overwrite the `next` pointer), we will spray additional objects to occupy that slot. For small overflows, `next` pointer corruption is not a concern, but pivoting to stronger primitives becomes harder. Notably, our strategy still works even for 1-byte overflows: although such short overflows generally cannot be pivoted into out-of-bounds reads (as longer overflows can), and thus cannot leak the shadow key. Despite this, we can still achieve control flow hijacks.

Next, we describe how to derive stronger primitives without corrupting the `next` pointer, using short overflows as initial primitives, since they represent the more challenging cases.

Pivot to large index out-of-bounds `zval` access. Given the severe constraints of a short overflow, directly overwriting an entire zval or its reference is infeasible. Therefore, our first objective is to escalate this minimal primitive into a broader out-of-bounds access by targeting some critical objects. As we discussed in §3.2, we aim to find our target object types in the table. Specifically to corrupt an adjacent `zend_array`.

A `zend_array` can operate in two modes. In the integer-indexed array mode, where all keys are integers, the `arPacked` pointer points to the data storage of the array. Alternatively, when the array has string keys, it will operate in hashtable mode. And `arData` pointer will be used in this mode. The `arData` is a butterfly-like structure, i.e., it points to the middle of a contiguous block of allocated memory, as shown in Figure 4. The memory for `arData` is allocated with a total size equal to the sum of the index array and the bucket storage. The `arData` pointer references the first bucket. Memory preceding it stores the index array (4-byte integers indexing buckets). In both modes, two distinct heap objects are allocated: the `zend_array` and the `arData`-backed allocation. *We corrupt the latter* (specifically, the index array) via a short overflow.

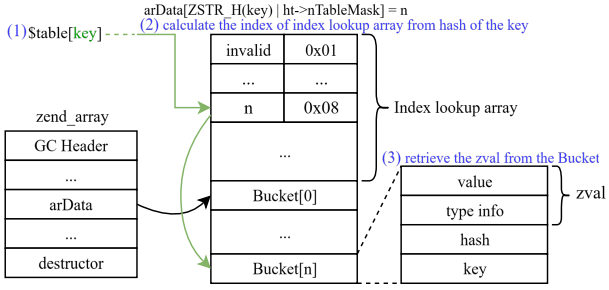

Figure 4: The process of index lookup in zend_array

In particular, we will allocate the `zend_array` object in hashtable mode. We illustrate the critical part of the data structure of a `zend_array` and lookup process in Figure 4. When it (1) evaluates any code such as `$table[key]`, the Zend Engine will (2) compute the hash of the key and apply a bitwise OR with `hashtable->nTableMask` to produce an offset into the index array. Finally, (3) it retrieves the element from the array as the index into the `Bucket` buffer to access the corresponding `zval`.

As illustrated in Figure 5, once we place the `arData` buffer right next to the buffer overflow victim, we will be able to overwrite the very first index (since we have only a short overflow) with a large value. This enables us to craft a fake bucket at a higher address (via heap spraying), which contains a `zval` that is under the attacker's control.

Pivot to limited-- & limited++. Assuming we have successfully corrupted the index lookup array as illustrated in Figure 5, we can turn this into a powerful primitive to decrement or increment an attacker-controlled location (decided by the fake `zval`'s `value` pointer). This is because an at-

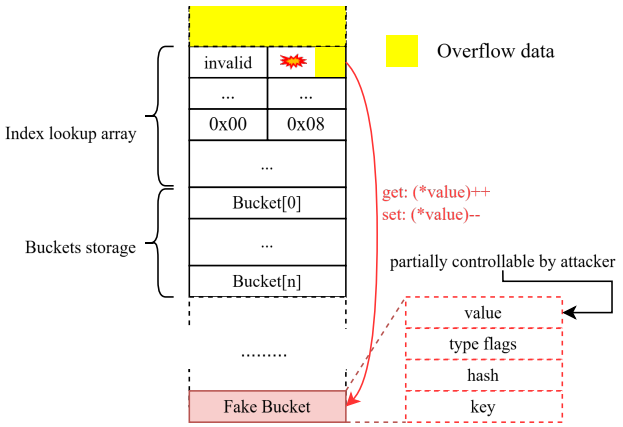

Figure 5: Corrupt the index lookup array

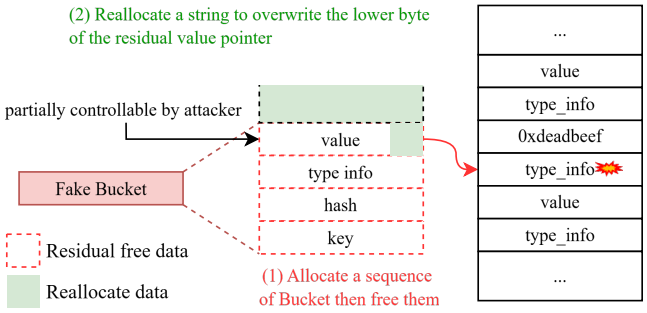

Figure 6: Fake Zval points to Type Flags of Another Zval

tacker can forge a `zval` object that is refcounted, by setting its `type_flags`. This way, during a hashtable store operation that overwrites an existing `zval` (in our case it is the fake one) with a new one, the reference count of the existing object pointed to by the `value` pointer will be decremented (since a reference is lost). Similarly, when a read operation occurs that retrieves the object, the reference count will be incremented (as a new reference is created). Since the `value` pointer is attacker-controlled, this primitive can decrement or increment memory at an attacker-specified address.

One limitation, though, is that due to data decoding restrictions on data encoding (as mentioned in §3.2), the attacker cannot inject arbitrary values for the `value` pointer when spraying string objects, and hence we call the decrement and increment primitives "limited".

Lift to arbitrary- - & arbitrary++. We demonstrate one approach to further lift the limited decrement & increment primitives to arbitrary ones. As we show in Figure 6, we can point the `value` pointer of the fake `zval` (part of the fake bucket as described in Figure 5) to the `type_flags` of yet another `zval` object (one extra layer of indirection). This is possible through careful heap spraying as follows. First, we (1) construct a legitimate `value` pointer (part of the fake bucket) which can be obtained by allocating actual `zval` objects and freeing them (leaving residual value in memory). Second, we (2) spray a string with its tail overlapping with only the lower bytes of the legitimate `value` pointer (e.g., by overwriting the lowest byte or the second lowest byte). This effectively allows us to partially control the `value` pointer (its lower bytes specifically) such that it will point to nearby memory that is controlled by the attacker.

In this case, we will spray `zval` array objects so that the partially controlled `value` pointer will point to a `type_flags` field in one of the `zval` objects in the array.

By applying our decrement & increment primitive on the partially controlled `value` pointer, which points to the `type_flags`, we can force it to become a reference-counted type. Then, an attacker can trigger an operation on the ref-counted `zval` object. For example, one can trigger the `zval` array to be freed, which will go through each element and trigger a decrement operation on this refcounted `zval`, i.e., at the address of `0xdeadbeef` in this example. We will discuss how to spray raw scalar data later in §6 so that we can construct an arbitrary address (instead of `0xdeadbeef`).

4.3 Pivot From Use-After-Free

Since we seek a universally applicable exploit chain under our PHP threat model, we restrict our analysis to use-after-free bugs in the basic PHP object types enumerated in Table 1. We divide these UAFs into two groups: `zval` UAFs and UAFs on other object types. Either group enables control of a `zval` structure.

- UAF on `zval` yields direct control of a `zval`.
- UAF on other objects yields indirect control of a `zval`.

For a `zend_array` as shown in Figure 4, its data storage buffer holds a sequence of `zval`. When the interpreter frees the `zend_array`, it also releases the elements buffer. Reclaiming the buffer allows the attacker to control `zvals` in the buffer as well. Other objects, including `zend_object` and `zend_function`, contain a `HashTable` (alias of `zend_array`) member that describes essential properties. Note that the object defined by the web application is still materialized as a `zend_object`. This places UAFs in application-defined objects within our scope.

Once we obtain control over a direct or indirect `zval`, discarding and using the `zval` brings us back to the decrement and increment primitives we mentioned previously in §4.2.

5 Bypass Read-only Metadata and Doubly linked List Protection

Now that we have achieved arbitrary increment and decrement primitives, we aim to achieve either control flow hijack

or arbitrary write primitives. Without the Read-only Metadata protection, we could have used the primitive to corrupt a function pointer (in allocator hooks as shown in Figure 2), and achieve control flow hijack by decrementing it or incrementing it many times. Similarly, without the doubly linked list protection, we could have corrupted such `next` and `prev` pointers in `zend_mm_chunk` to achieve arbitrary write. However, due to such protections, these are no longer feasible. At a high level, these restrictions impose additional preconditions that make an obvious primitive transformation become harder. This means we will need to look for alternative exploit paths, i.e., by finding alternative objects and their fields to corrupt and derive more powerful exploits.

In this section, we will describe exploit strategies that take arbitrary increment and decrement primitives as input, and produce either control flow hijack or arbitrary write primitives. Finally, to realize end-to-end exploits, we will also describe how to achieve infoleak using the same input primitives.

5.1 Achieve Control Flow Hijack

5.1.1 Metadata corruption

As mentioned, one widely documented method to achieve control flow hijacking before the implementation of the new mitigations is to target the heap metadata [11]. The Zend heap memory allocator exposes a mechanism to register custom memory management functions. This includes the ability to override core operations such as `malloc`, `free`, `realloc`, garbage collection, and heap shutdown implementation. Those function pointers are stored in the main heap `_zend_mm_heap` metadata, and can be enabled by `_zend_mm_heap::use_custom_heap`. With our heap decrement and increment primitives, we could overwrite those pointers and eventually invoke arbitrary functions. Unfortunately, these function pointers are marked as read-only by the *Read-only Metadata* mitigation (as described in §2.4).

5.1.2 Destructor function pointer corruption

Alternatively, we seek alternative function pointers that are outside of the protection of read-only heap metadata. Specifically, we aim to find heap objects with function pointers that can be allocated by an attacker. Out of the available PHP interpreter objects listed in Table 1, we find that `zend_array` has a destructor function pointer that will be called by the Zend Engine to ensure proper cleanup of all elements in the array by iterating over each of them in its storage. Specifically, the function pointer, i.e., `array->pDestructor`, will be dereferenced and called for every element in the array.

As shown in Figure 7, the idea is to first reuse the previously described capability of creating a fake bucket object (described in the previous section). Then we configure the `type_flags` of the fake bucket so that the PHP interpreter thinks the `value` pointer points to a reference-counted object

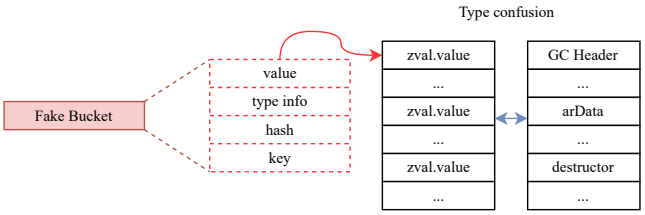

Figure 7: Trigger Destructor of Fake Array with Fake Bucket

(at the rightmost of the figure). We then leverage the partially controllable nature of the `value` pointer (as mentioned in §4.2 when pivoting to arbitrary decrement & increment primitives) and have the pointer point to a nearby address (by changing its lower bytes). Thus, it is likely going to point to attacker-controlled memory instead. In this case, we will again spray an array of `zval` objects.

In the `zval` array, we configure the first `zval.value`, which would be interpreted as the GC header's refcount field, to a value of `1`. This setup ensures that when the PHP interpreter executes a replacement operation on the hashtable, it would cause the decrement of the refcount to 0 and thus invoke the destructor function. Crucially, because both `arData` and `pDestructor` fields overlap with specific entries in a sprayed `zval` array, we obtain full control over the function calling destination as well as the arguments passed to it. In other words, we can now achieve a control flow hijack. One might wonder whether there is any restriction on the target addresses, as data encoding may impose such restrictions (as mentioned in §4.2). However, as will be described in §6, this restriction no longer applies since we can prepare scalar data (e.g., integers) in the sprayed `zval` array.

5.2 Information Leak

To obtain the necessary function addresses, e.g., `system()` in libc, `zif_system()` in PHP, and other payload addresses under ASLR, implementing an address-leak primitive is a critical component of the entire exploit chain. Even though not a focus of our novel exploit strategies, we describe our infoleak for completeness. Overall, we consider two options. First, as an optional step, we can leak some address (e.g., on heap) as an anchor which can speed up the subsequent probing process. Second, we probe the address space using previously derived primitives and identify the location of the .text section in libc or PHP.

5.2.1 Leak some heap address

We can leverage our arbitrary decrement and increment primitives to target some critical fields in basic objects such as the length of a string as discussed in previous sections, which can

derive out-of-bounds read primitives. However, typically reading a string and returning it through HTTP responses (e.g., via JSON or XML payloads) is subject to data constraints as mentioned in §3.2. Alternatively, we can also manipulate the type of `zval` by applying the arbitrary decrement and increment primitives to its type tag to achieve type confusion.

For example, we can convert the `zval` from a string type into a double, as illustrated in Figure 8. In this way, we can leak the address of the string because the `value` field will be interpreted as double and thus can be correctly encoded in the HTTP response (e.g., via JSON or XML). In other words, we know where the Zend heap is.

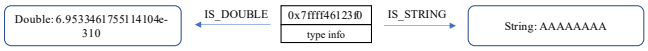

Figure 8: Zval Interpretation Based on Type info

Alternatively, we can try to leak a libc heap address. In order to improve the performance, the Zend Engine will preallocate several internal strings on libc heap [40]. For example, an empty string will be preallocated on libc's heap instead of handled by Zend. Using the same technique above, but with an empty string, we can then leak libc's heap address as well.

Knowing an address in either the Zend heap or the libc heap allows the attacker to subsequently scan the memory address space and locate the base address of the .text section for either the PHP application or libc.

5.2.2 Memory probing from known heap addresses

Our arbitrary decrement and increment primitives imply an arbitrary address probing ability. We can probe the memory page by page to discover the boundaries of the writable memory as shown in Figure 9. Specifically, when probing memory that is unmapped or not writable, the worker will crash because of a segmentation fault and notify the attacker with a 502 HTTP error code. Given that each newly spawned worker inherits the same address space layout as the main process (a known issue [53]), an attacker can attempt the probing many times and gradually narrow down the boundary.

If we have already leaked the address of the Zend heap or the libc heap, we can simply scan towards higher or lower addresses, to identify the boundary. Because the Zend heap and libc heaps reside near well-known memory mappings (dynamic libraries and the main executable, respectively), starting the probing from the anchors will be significantly faster than probing the entire possible range of a .text segment.

As an example shown in Figure 9, when we start probing from a Zend heap address, say `0x00007ffff4600000` towards higher addresses, there will be no crash at `0x00007ffff4800000-0x1000` but then crash at `0x00007ffff4800000` (since the corresponding page is unmapped). This allows us to decide where other libraries such as (g)libc will be located relative to Zend heap. We make the assumption that there are only a few popular stock glibc versions that simplify the probing process. Finally, note that the memory probing can also be done without knowing any valid heap address; instead, one can simply brute force all possible locations of the heap. However, the probing will take much longer and likely trigger many more crashes.

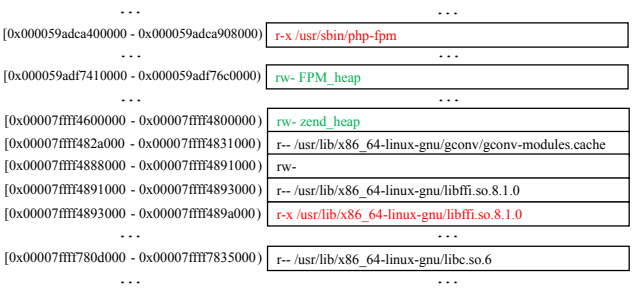

Figure 9: Memory Layout of PHP process

6 Bypass Heap Isolation

In order to successfully achieve the previously described primitive transformations, one key step is heap fengshui. Given the new mitigation that isolates the heap under direct attacker control (e.g., HTTP request fields) from the application heap, this step becomes challenging. In particular, under this mitigation, requests related global data, such as `$_GET`, `$_POST`, and `$_COOKIE` are allocated in a separate heap zone, preventing crafted requests from manipulating the main application heap.

Faced with the constraints of Zend heap isolation, we shift our heap fengshui or grooming strategy from the HTTP request to the decoding process of the application's PHP scripts. Since the decoding occurs within the PHP script of the web application, those allocations take place in the application's heap. We specifically choose to focus on JSON and XML decoding, which are pervasive in web applications (Appendix Table A2).

We performed static analysis of their respective parsing functions with the source code using CodeQL. Table 2 summarizes the types of objects that can be allocated by the parsing functions. Consequently, such objects are always allocatable by an attacker as long as the application supports requests that contain XML or JSON and decodes them.

6.1 One-shot Heap Fengshui

To address the challenges of one-shot interactions as discussed in §3.2, we use a one-shot heap grooming strategy. All grooming operations are done within a single request by leveraging the fact that the encoded JSON or XML data (i.e., key/value pairs) can contain multiple identical keys: While each key's

Format	Allocated Types	Functions
JSON	`zend_array`, `zend_string`	`json_decode`, `simdjson_decode`, `simdjson_key_value`
XML	`zend_array`, `zend_string`, `zend_object`	`xml_parse`, `xml_parse_into_struct`, `simplexml_load_file`, `simplexml_load_string`
YAML	`zend_array`, `zend_string`	`yaml_parse`, `yaml_parse_url`, `yaml_parse_file`
Mixed	`zend_array`, `zend_string`, `zend_value`, `zend_object`	`unserialize` (not recommended to use with untrusted user input)

Table 2: Parser formats, their result data types, and corresponding PHP parsing functions.

value is parsed and allocated, only one key's value is retained after parsing, and all other values are freed. We demonstrate the idea in Figure 10. We perform heap grooming at the page-level and every block in Figure 10 and Figure 11 represents one page. A significant problem in small-size allocations arises from the fact that many similarly small-size allocations, as well as metadata of various objects, during the application runtime may be handled by the same freelist, which introduces noise into our heap grooming process. Our page-level spraying strategy can reduce such noise and make the layout process more generic. Furthermore, it is resilient against potential future mitigations such as freelist randomization. When a request size exceeds one page, ZendMM searches for the most suitable contiguous pages to satisfy the allocation, as discussed in §2.3. We exploit this behavior to precisely obtain the allocation layouts we require.

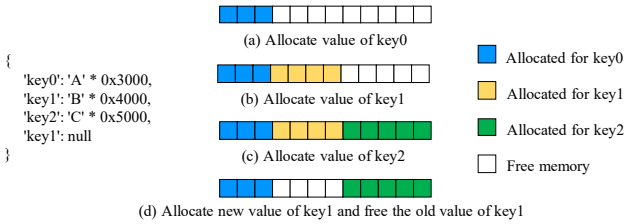

Figure 10: Page-level heap grooming in one request

We use page-level fengshui for two purposes: positioning the victim object so that its corruption reaches the intended target, and positioning the forged objects that a corrupted pointer is later redirected to. We discuss them in turn.

The first purpose applies primarily to out-of-bounds overflows, where the victim object must precede the region we intend to corrupt. By leveraging this behavior, we first allocate appropriately sized chunks before our corruption target region, then free those chunks and reallocate them as the overflowable victim object. This ensures that our overflow spans into the target memory region. For example, we can use `arData` as described in §4. We illustrate this idea in Figure 11. When the vulnerable object cannot fill a page on its own, we spray objects of varying sizes such as `zend_string` and `zend_array` on the same page, so that the vulnerable object and these helpers together occupy a page-level region we can position against the target.

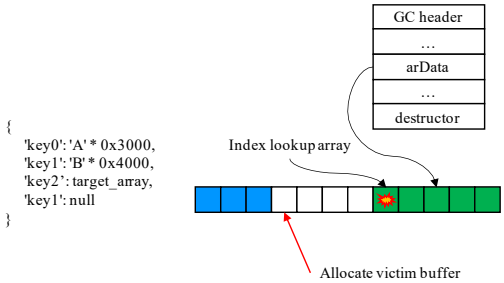

Figure 11: Overwrite index lookup array in one request

In the second purpose, we place a forged object on a chosen page so that a single partial pointer overwrite can reach it. Since objects within the same chunk share their high-order address bits, a pointer that already references some page in the Zend heap can be redirected to an attacker-prepared object by rewriting only its low bytes, as long as that object has been groomed onto the intended page. Page-level fengshui places the forged object exactly where the low-byte overwrite in §4.2 lands, so that the partially controlled `value` pointer there resolves to the attacker-controlled object instead of an unintended location. The destructor-based control flow hijack in §5.1.2 depends on the same arrangement. This purpose applies to both OOB and UAF primitives, as our method first converts either one into the same arbitrary decrement primitive in §4 before the forged object is placed. This operates entirely at the granularity of pages, where the spraying unit can be a single multi-page object or an entire freelist backed by contiguous pages.

6.2 Raw Data Spraying

Recall our earlier observation that string-based spraying is restricted by encoding rules of those data decoding functions, we turn to a more flexible approach: **packed array**

buffer spraying. In particular, when an array is operating as an integer-indexed array (packed array), its `arPacked` pointer will point to a contiguous sequence of `zval` rather than `Bucket`. By setting each `zval`'s value to the desired integer, we effectively spray arbitrary 8-byte patterns directly into memory, thereby sidestepping all string-encoding constraints. This technique helps us with full control over the raw bytes needed for building the fake array in §5.

Despite the power of packed array buffer spraying to put any byte pattern into memory, its utility is limited by the fact that each 16-byte chunk only grants control over the first 8 bytes, namely the `zval.value` field, leaving the `type_flags` immutable. We designed a novel **hash spraying** technique as a complementary raw byte spraying strategy. The key idea is to exploit PHP's DJBX33A hash algorithm [8, 40]. In particular, we find that this hash algorithm is highly reversible: for a given 8-byte hash value, one can construct an input string (in UTF-8) of a specific size by solving each character in a "backwards" fashion with Algorithm 1. In contrast to the packed array method, this hash-based spraying offers a significant benefit: the sprayed content includes a controllable 8-byte hash (with the highest bit set) and partially controllable length fields and string literals, as illustrated in Figure 13.

7 Evaluation

7.1 Experiment Setup

To systematically evaluate the effectiveness of our exploit strategy under Heap Hardening protections, we consider every PHP memory corruption CVE with a public proof-of-concept exploit released between 2019 and 2025: CVE-2024-2961 [39], CVE-2023-3824 [32], CVE-2022-31626 [5], CVE-2019-11043 [36], and CVE-2019-6977 [37]. We added two closely related PHP memory corruption challenges from public CTFs to cover a broader range of bug patterns, giving seven test targets. However, real-world web applications often contain business logic that may accidentally benefit the exploit. For example, the public PoC [10] for CVE-2022-31626 relies on Adminer [61] decoding base64 input, which makes spraying and leaking raw data easier. To remove such external noise and highlight interpreter-level challenges, we built a minimal reproduction web app for two remote targets: CVE-2022-31626 uses a simplified *update bookmark* feature extracted from real-world application phpMyAdmin [48], and CVE-2024-2961 uses a crafted application that logs file hashes. All other CVEs and the two CTF cases keep the shortest official or community trigger path. All targets were run inside the same base container image: 64-bit Ubuntu 22.04, Docker 24.0.5, with all four evaluated protections enabled on our test branch based on PHP 8.4.0. The attacker machine uses Ubuntu 22.04 and identical hardware (24-core Intel Ultra9-285HX @4.60GHz, 64GB RAM @4400MT/s). The exploit scripts are executed with Python 3.10.12.

7.2 Experiment Results

We first reproduced each public exploit chain in our test environment. The results in Table 3 show that the original exploits of CVE-2024-2961 [11], CVE-2022-31626 [5], and CTF Case A [50] can no longer achieve arbitrary command execution because the new Heap Hardening protections completely blocked their exploit strategy. We further manually analyzed the exploit logic and confirmed that the failures were caused by reliance on freelist corruption, rather than differences in application-layer behavior. The original exploits for CVE-2019-6977 [9] and CTF Case B [33] also failed because they either assumed ASLR was disabled or could only tamper with application data, which does not satisfy our goal of command execution. In contrast, CVE-2023-3824 [32] and CVE-2019-11043 [26] still succeed because they rely on powerful but rare primitives not yet covered by current defenses: in-structure overwrite of a data pointer or a negative overflow during request parsing, respectively.

ID / Case	Goal	Primitive	Origin	Recon
CVE-2024-2961	RCE	OOB	No	Yes
CVE-2023-3824	SBE	OOB	Yes	-
CVE-2022-31626	RCE	OOB	No	Yes
CVE-2019-11043	RCE	OOB	Yes	-
CVE-2019-6977	SBE	OOB	No	Yes
CTF Case A	RCE	OOB	No	Yes
CTF Case B	RCE	UAF	No	Yes

RCE: remote code execution. SBE: sandbox escape. OOB: out-of-bounds write. UAF: use-after-free. Case A/B are modified from public CTF challenges. "Origin" and "Recon" indicate whether public exploits and those reconstructed using our strategy succeed in the test environment, respectively.

Table 3: Existing and Rebuilt Exploits under Heap Hardening

For the five scenarios where the original exploits failed or were limited, we applied our exploit and defense bypass strategies to craft new exploits that can regain command execution ability. We performed stability tests on three representative CVEs (CVE-2024-2961, CVE-2022-31626, and CVE-2019-6977). As shown in Table 4, all tests succeeded across 100 runs. The new exploits required fewer than 300 HTTP requests on average in the remote scenario and only two requests in the local sandbox-escape scenario with arbitrary script uploads.

These results show that the four implemented Heap Hardening protections are still not universal for all memory corruptions, but they can efficiently mitigate traditional freelist-based attacks. Our general bypass strategy fills this gap and restores reliable exploitability for multiple memory-corruption bugs. To ensure reproducibility, we have released all exploit scripts and the complete container image so the research community can build on them further.

ID	Goal	Mean	Worst	Stability
CVE-2024-2961	RCE	245.76	507	100 %
CVE-2022-31626	RCE	284.96	507	100 %
CVE-2019-6977	SBE	2.00	2	100 %

Mean and *Worst* indicate the number of HTTP requests sent to achieve a successful exploit. *Stability* indicates the Success Rate.

Table 4: Stability Test (100 Runs per Target)

7.3 Experiment Case Study

We demonstrate an end-to-end attack with CVE-2024-2961. Although this vulnerability is capable of producing overflows of up to three bytes, in most real-world application configurations it is practically limited to a single-byte overflow. Moreover, the value of the overflowed byte is constrained to the range from `0x48` to `0x4D`.

7.3.1 Initial memory preparation

While our prior discussion followed a conceptual progression from primitives to full exploitation, our actual exploitation workflow begins differently. In practice, we start with the initial memory preparation. Given that each heap zone (application heap and isolated user request heap) is aligned to 2 MB, and ASLR cannot randomize the offset within a page, partially overwriting a valid pointer grants us the ability to pivot it to somewhere else in the Zend heap space. As demonstrated in §5, we first allocate a sequence of Buckets, then free it, then there will be a large amount of residual `zval` as well as hash and key pointers on the heap. This is an important step, as we cannot use arbitrary data as a reference-counted `zval` for now due to the encoding issues discussed in §3.2. Then we allocate a string to partially overwrite the value pointer of a residual `zval` to pivot it to somewhere. Since the escape character will be inserted into the lower address of the string, we can safely overwrite the pointer without the encoding issue.

7.3.2 Access our fake bucket and fix the access key

With the exploitation technique in §5 and §6, we use the vulnerability to corrupt the first index of the index lookup array, making it point to the fake bucket built in §7.3.1.

Then we spray multiple strings to re-occupy the memory pointed by the residual key pointer in the fake bucket (which is a pointer to `zend_string`). The string is carefully crafted as shown in §6 to make sure that it has the same hash value as the residual hash in the bucket. Now operations such as `$table[$key]` and `$table[$key] = $v` will be directed to our fake bucket. We leverage our heap fengshui technique in §6 to precisely control the memory layout. Specifically, we arrange allocations such that a `zval` whose type tag references the `zend_string` destined for return, resides at an address pointed by our overwritten residual value pointer. Then the decrement primitive will be applied to the type tag.

This grants us an address leak. All these memory allocating and freeing operations of this and the previous step can be done in one request as shown in §6.

7.3.3 Arbitrary code execution

We craft fake array by spraying a sequence of `zval` with repeated payload `[0x700000001, arg0, pc]` in Figure 12. Since the array `payload["key"]` is integer-indexed, the parsed `zend_array` will be set to packed mode. When the array is in packed mode, the storage buffer will be a simple sequence of `zval`. Three `zval`s take `0x30` bytes of memory, and their `zval.value` fields overlap with the GC header, `arData`, and `pDestructor` of a `zend_array` respectively. For the `0x0000000700000001` in payload, `1` is the reference count, `0x7` means this object should be destroyed with `zend_array_destroy`, which will invoke `array->pDestructor`, i.e., the `pc` in payload. Note that we keep the payload of step one in the request data, but spray our fake array into the memory pointed by the residual value pointer. Then, when the interpreter executes `$table[$key] = $v`, the reference count of our fake array will be decreased to 0 and invoke `zend_array_destroy`. Since we have set up our `array->pDestructor`, it will then call `pc(arg0)`.

```
payload = {
    "key": [
        0x0000000700000001, arg0, pc,
    ] * 0x400
}
```

Figure 12: Payload Used to Craft Fake Array

8 Mitigation Discussion

This section discusses mitigation directions in two parts. We first revisit the PHP heap hardening roadmap and analyze how its unreleased or proposal-stage defenses would affect the generic attack path identified in this paper. We then present two targeted hardening checks that we implemented and evaluated to block this path.

8.1 Roadmap Defense Directions

The PHP heap hardening roadmap can be grouped into three stages of maturity. Shadow Pointer and Unlink Abuse Prevention are already merged into stable releases. Both have proven effective and forced attackers away from freelist poisoning, which motivates the built-in-object path studied in this paper. Read-only Metadata and Further Heap Isolation have prototypes but are not yet merged, and Guard Pages and

Freelist Randomization remain proposals without prototype code. We focus on these four items below. For the prototypes, we analyze what they contribute and where they fall short. For the proposals, we discuss how they could be implemented and to what extent they would block the generic attack path that we identified.

(1) *Read-only Metadata* protects allocator hooks in `zend_mm_heap`, with a prototype that has reached a stable design. An earlier custom-heap variant was rejected out of concern [3] for the debugging value these pointers provide. The later Read-only Metadata prototype remains unmerged, with maintainers questioning [4] whether its roughly 0.6% runtime cost is justified by the protection it provides. We view this protection as offering limited benefit under our threat model. Once an attacker can corrupt allocator metadata, the underlying write primitive is already strong enough to be redirected to writable interpreter objects, as in §5.1.2, and the ASLR probe in §5.2.2 can further expose writable structures inside the PHP and libc images. Read-only Metadata therefore raises exploitation cost without reducing reliability for attackers who reach this stage.

(2) *Further Heap Isolation* is still under development, it plans to put objects of the same types into the same chunk, avoiding the mix of different types of objects (this is similar to AutoSlab [28] in the Linux kernel). Further Heap Isolation directly addresses our threat model, since it disrupts remote heap shaping before the attacker reaches an object-corruption path. Our attack assumes that parser-allocated and victim objects can be arranged at predictable page-level positions. If the objects listed in Table 1 are placed in dedicated heaps that do not share pages, this assumption no longer holds and generic remote exploitation against built-in PHP objects would no longer be feasible.

(3) *Guard Pages* insert unmapped pages between freelists or large allocations to prevent linear access across them and mitigate cross-page overflows. Our exploit does not rely on either primitive, so fixed-size guard pages contribute little to closing the path in this paper. A randomized variant [54, 55] is more relevant, since the information leak in §5.2.1 and the control flow hijack in §5.1.2 both depend on page-level fengshui. Randomized guard pages would inject page-level entropy at low engineering cost. The tradeoff is some address-space fragmentation and runtime overhead.

(4) *Freelist Randomization*, inspired by the `SLAB_FREELIST_RANDOM` [16] mitigation in the Linux kernel, has also been proposed but not yet implemented. It randomizes slot order within each freelist at low cost, since randomization is only required at freelist initialization. This breaks a core assumption of our exploit as the relative distance between two same-size objects will be no longer predictable, so a short overflow cannot reliably reach a chosen field in a neighboring object. The primitive transformation in §4.2 also breaks, since secondary targets such as the fake `Bucket` in Figure 5 can no longer be placed at known offsets. LIFO reuse is generally preserved, so UAF primitives remain available, but their secondary targets become equally hard to predict. The remaining gap is page-level spraying: an attacker who exhausts a freelist can still treat the page-aligned region as a single spraying unit, and the page-level fengshui in §6.1 can partially bypass intra-freelist randomization. Freelist randomization is therefore best paired with isolation or randomized guard pages.

Overall, isolation and randomization are the directions that most directly affect this path, while targeted metadata protection offers more limited benefit.

8.2 Targeted Hardening for the Remaining Generic Path

Beyond the roadmap, we implemented and evaluated two targeted checks. Each independently breaks the generic attack path in this paper.

• **Patch-hashtable**: Validate that a requested hashtable index does not exceed `num_used` before the selected `Bucket` is used. This blocks the forged `Bucket` step in §4.2, which we rely on to convert a weak one-byte overflow into stronger increment and decrement primitives.

• **Patch-refcnt**: On every hashtable operation that updates a refcount, such as replacing or reading an entry, validate that the target address lies inside the PHP heap and points to a correctly aligned `zend_refcounted` object. This invalidates the arbitrary increment and decrement primitives and also blocks the ASLR probe in §5.2.2 and the destructor-based control flow hijack in §5.1.2.

The two checks sit at different points in the exploit chain. Patch-hashtable targets primitive transformation and only needs to harden one generic object, which keeps its cost low. It is worth adopting on its own, since `zend_array` is the only built-in object we surveyed that enables generic remote primitive transformation. Patch-refcnt sits further along the path and covers strong primitives that do not pivot through a hashtable. The tradeoff is that it instruments a hotter code path and carries higher cost. We view Patch-refcnt as a stopgap that becomes less necessary if PHP adopts the isolation and randomization defenses discussed above.

We measured both checks under the environment in §7.1. Patch-hashtable mitigates three cases in Table 4 with a 0.17% overhead. CTF Case A and Case B start from stronger primitives that do not pivot through a hashtable, so they fall outside its scope. Patch-refcnt mitigates four cases with an 8.86% overhead, consistent with the broader code path it covers. As for the last case, CTF Case A exploits the shadow pointer implementation flaw discussed in §4.1, and PHP maintainers have addressed it with two subsequent patches [46, 47].

9 Related Work

We first review prior attack research and point out how our approach differs. We then examine existing defense work and explain how it relates to PHP's heap hardening features.

Exploiting memory corruption in PHP interpreter. Gollum [21] aims to automatically generate exploits for memory corruption bugs in PHP. It uses its custom allocator, ShapeShifter, to mimic PHP's ZendMM allocation behavior and locate the desired heap layout. With the help of Shrike and a genetic algorithm, Gollum produces a set of PHP code fragments that recreate the same layout under ZendMM. After controlling the positions of the overflow source and the target object, Gollum overwrites the target's function pointer and redirects it to a One Gadget to finish the exploit. While Gollum can build the expected layout for relatively large overflows when the attacker can run arbitrary PHP code, it cannot create a valid layout for a PHP remote exploit. In these real-world cases disclosed in the last five years, the available primitives are relatively weak, and new mitigations make earlier heap manipulation tricks ineffective. Moreover, Gollum assumes that ASLR is disabled, and, as noted in §3.2, using One Gadget to exploit the PHP interpreter does not provide arbitrary command execution. Consequently, the proofs of concept produced by Gollum still need extra adjustment, such as combining them with the techniques in §5, before they become full exploits. The ideas in this work can enlarge Gollum's search space and help it produce exploits that work in more restrictive, real-world scenarios. This work offers an opportunity to strengthen tools like Gollum.

Charles Fol et al. proposed "Generic Remote Exploit Techniques for the PHP Allocator". [10] Their method overwrites the least-significant bytes of a freelist's next pointer, allowing the pointer to be hijacked to almost any nearby address. This hijack creates overlapping slots and lets the attacker modify arbitrary data on the heap. Before heap hardening was deployed, this was a powerful remote exploitation technique, but PHP's new protections can effectively detect attempts to corrupt the freelist pointers and thus disable the attack. In addition, the technique proposed by Charles Fol et al. assumes the attacker can forge structures such as `zend_array` by spraying raw data directly into the web application's heap. However, heap isolation prevents raw spraying and mitigates this structure-forging trick. Our work introduces several new techniques that can still achieve arbitrary heap writes. We also use the method in §6 to regain the ability to spray raw data into the application heap under restricted conditions.

Bypassing safety defenses. Prior work has also studied how limited memory-corruption primitives bypass allocator or platform defenses. SLUBStick [31] turns cross-cache attacks into arbitrary read/write primitives through a slab timing side channel. System Register Hijacking [35] modifies privileged registers to bypass kernel defenses such as SMEP, SMAP, KASLR, and (Fine-)IBT [15]. On The Effectiveness of Address-Space Randomization [53] exploits the low entropy limitation of ASLR on 32-bit systems to bypass it within a few hundred seconds. However, on 64-bit systems, the entropy of ASLR has been strengthened by six orders (2^{20}), making this technique less effective. K-leak [27] combines memory errors in 64-bit Linux kernels to bypass KASLR. ARCHEAP [66] and HeapHopper [7] automatically discover exploits that bypass allocator checks in heap allocators like ptmalloc2 [17], but their search scope is limited by memory and time constraints.

Heap Hardening. SeaK [64] and Uriah [22] isolate selected protectable objects into separate heaps, while randomized slab caches [18] and SeMalloc [63] reduce the chance that vulnerable and target objects become adjacent by selecting among multiple heap caches. These directions are relevant to future ZendMM isolation if they can be reproduced at low cost and prevent cross-cache attacks. Memcheck [52] and Address Sanitizer (ASAN) [51] provide broader runtime checks, but are mostly limited to testing environments due to high overhead. Scudo [49] uses checksummed chunk headers and size-based isolation, making it a useful reference for PHP.

10 Conclusion

This work presents the first systematic evaluation of PHP's memory protection mechanisms. Within this scope, we systematically surveyed all built-in PHP object types and the generic attack paths available to remote attackers. We identified implementation flaws and proposed a generalized exploitation strategy that transforms weak primitives, including single-byte overflows, into reliable arbitrary command execution, bypassing all evaluated heap hardening defenses. Our techniques are effective in both remote code execution and sandbox escape scenarios. We validated our approach by reviving real-world CVEs with all evaluated protections enabled and confirmed its reliability through automated stability testing. We further discussed deployable mitigations to offer insights into practical strategies for improving PHP's memory safety. Our results highlight the limitations of existing mitigations and motivate the development of comprehensive models for securing PHP's memory.

Acknowledgments

We thank the anonymous reviewers and our shepherd for the careful and thorough feedback, which substantially improved this paper. We are also grateful to Trent Jaeger for his early guidance on this project.

Ethical Considerations

We structure this discussion around stakeholder analysis and two phases of this work: the research process and publication of results. We then discuss mitigations for the risks and explain why conducting and publishing this study is justified.

Stakeholder Analysis and Research Process Impact

This work has five stakeholder groups. The first group is the PHP development team, especially maintainers responsible for heap hardening. The second group is PHP operators, hosting providers, and cloud platforms that deploy PHP interpreters. The third group is PHP application and framework developers, whose code paths may expose decoding and allocation behavior that affects exploitability. The fourth group is the security research community, which studies interpreter exploitation, heap hardening, and memory-safety defenses. The fifth group is end users and organizations that rely on PHP-based infrastructure, because exploitation and mitigation outcomes affect their data, services, and operations.

The study uses only existing PHP interpreter memory corruption CVEs and introduces no new vulnerabilities. We also followed responsible disclosure for the implementation flaw found in the heap hardening mechanism. The PHP developers patched the issue after our report. This protects maintainers, operators, and end users before publication. It also gives the community a clearer view of which results depend on a patched implementation flaw and which results remain relevant to PHP's interpreter behavior.

Impact of Publication: Positive impacts

For maintainers, this work evaluates current heap hardening under a realistic remote attacker model, shows which traditional paths are blocked, and identifies where determined attackers can still adapt. For operators and hosting providers, it reduces false confidence and supports complementary defenses such as sandboxing, monitoring, WAFs, rate limiting, and anomaly detection. For application and framework developers, it explains why exposed JSON and XML decoding paths matter after an interpreter memory corruption bug. For the research community, the artifacts provide Docker files, reference commands, and standalone implementations as a reproducible baseline. For end users and organizations, the paper clarifies that heap hardening narrows exploitation but does not eliminate it, helping them assess the security posture of service providers and advocate stronger layered defenses.

Impact of Publication: Negative impact

The main risk is misuse. The paper describes exploit strategies that can help attackers adapt after PHP heap hardening is deployed. These strategies may lower the engineering effort needed to turn a future PHP memory corruption vulnerability into command execution. The study does not introduce a new PHP memory corruption vulnerability or application entry point. The remaining risk comes from lowering the effort needed to adapt future exploits after such vulnerabilities already exist. Open-source artifacts may also increase this risk if they are used outside local test environments.

Mitigations

We use several mitigations to reduce harm. First, we disclosed the implementation flaw to PHP maintainers before publication and separate that patched flaw from the remaining techniques. Second, we pair the attack analysis with defense discussion: we explain which existing mitigations are effective, which paths remain open, and why additional hardening of built-in PHP objects can raise exploitation cost. Our artifact also includes two prototype checks that independently break the attack path identified in this paper, giving maintainers concrete short-term hardening options. Third, deployment-level mitigations. PHP's own mitigations should be complemented by sandbox restrictions, monitoring, WAFs, rate limits, and anomaly detection. Finally, future PHP hardening should evaluate built-in PHP objects and decoding-based allocation paths, because current hardening mainly focuses on heap metadata while our results show that built-in objects may also need protection.

Justification for Conducting and Publishing the Research

Move before attackers. Withholding the results would leave maintainers and operators with less information about the limits of current hardening. Similar strategies could still be discovered independently, and defenders would have fewer concrete examples for improving PHP. Publication after responsible disclosure gives the community a safer path: the implementation flaw is patched, the assumptions are explicit, and the paper provides mitigation directions and targeted patch implementations.

Proactively addressing the misuse risk. We address the risk through responsible disclosure and concrete defense recommendations. In particular, our proposed mitigations, already implemented in the artifacts, give PHP maintainers concrete options for stopping the generic exploit strategy discussed in the paper.

Longer-term impact. Overall, we believe publishing this paper will bring more awareness of PHP's security posture to the security community, which will inform future hardening directions. Even though there is some risk of misuse in the short term, it will lead to a more secure PHP interpreter in the long run. In other words, the long-term benefits to PHP maintainers, application developers, operators, end users, and the research community outweigh the short-term risk.

Open Science

Our artifacts have been released at the following link: https://zenodo.org/records/20401797. All experimental environments include Docker files and reference commands that facilitate easy and stable reproduction. We also provide standalone implementations for all pseudocode that appear in the paper.

References

[1] Adobe. Magento open source. https://www.magento.com, 2025.

[2] Arnaud Le Blanc. The php interpreter. https://github.com/arnaud-lb/php-src/tree/mm-zones, 2024.

[3] bwoebi. PHP pr 14570: Make some parts of _zend_mm_heap read-only at runtime (the first comment). https://github.com/php/php-src/pull/14570#issuecomment-2168457515, 2024.

[4] bwoebi. PHP pr 14570: Make some parts of _zend_mm_heap read-only at runtime (the second comment). https://github.com/php/php-src/pull/14570#issuecomment-2248354450, 2024.

[5] CFandR-github. CVE-2022-31626 analysis. https://github.com/CFandR-github/PHP-binary-bugs/blob/main/cve_2022_31626, 2022.

[6] SSD Secure Disclosure. PHP spldoublylinkedlist uaf sandbox escape. https://bugs.php.net/bug.php?id=80111, 2020.

[7] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Bringing bounded model checking to heap implementation security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, Baltimore, MD, August 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/eckert.

[8] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44(6):681–698, 2014. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2179, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2179, doi:10.1002/spe.2179.

[9] Charles Fol. imagecolormatch() oob heap write exploit. https://github.com/cfreal/exploits/tree/master/CVE-2019-6977-imagecolormatch, 2019.

[10] Charles Fol. Generic remote exploit techniques for the php allocator, and 0days. https://www.blackalps.ch/ba-22/talks.php, 2022.

[11] Charles Fol. Iconv, set the charset to rce: Exploiting the glibc to hack the php engine (part 1). https://blog.lexfo.fr/iconv-cve-2024-2961-p1.html, 2024.

[12] Charles Fol. Iconv, set the charset to rce: Exploiting the glibc to hack the php engine (part 2). https://blog.lexfo.fr/iconv-cve-2024-2961-p2.html, 2024.

[13] Cake Software Foundation. Cakephp: The rapid development framework for php - official repository. https://cakephp.org, 2025.

[14] CodeIgniter Foundation. Open source php framework (originally from ellislab). https://github.com/bcit-ci/CodeIgniter, 2025.

[15] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Fineibt: Fine-grain control-flow enforcement with indirect branch tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 527–546, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3607199.3607219.

[16] Thomas Garnier. mm: Slab freelist randomization. https://lore.kernel.org/lkml/1460138602-85386-1-git-send-email-thgarnie@google.com/T/, 2016.

[17] W GLOGER. Ptmalloc. http://www.malloc.de, 2006.

[18] Ruiqi GONG. Randomized slab caches for kmalloc(). https://lkml.org/lkml/2023/5/8/206, 2023.

[19] PHP Documentation Group. Description of core php.ini directives. https://www.php.net/manual/en/ini.core.php, 2025.

[20] PHP Documentation Group. Fastcgi process manager (fpm). https://www.php.net/manual/en/install.fpm.php, 2025.

[21] Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1689–1706, 2019.

[22] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the heap: Efficient memory error protection of safe heap objects. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1330–1344, 2024.

[23] Eyal Itkin. Safe-linking – eliminating a 20 year-old malloc() exploit primitive. https://research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive, 2020.

[24] Eddie Kohler. Hotcrp conference review software. https://read.seas.harvard.edu/~kohler/hotcrp, 2025.

[25] Laravel. The laravel framework. https://laravel.com, 2025.

[26] Emil Lerner. Phuip-fpizdam. https://github.com/neex/phuip-fpizdam, 2019.

[27] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. K-leak: Towards automating the generation of multi-step infoleak exploits against the linux kernel. In *NDSS*. Internet Society, 2024.

[28] Zhenpeng Lin. How autoslab changes the memory unsafety game. https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game, 2021.

[29] LoadForge. Enhance security by disabling dangerous php functions. https://loadforge.com/guides/disable-dangerous-php-functions-for-enhanced-security, 2025.

[30] ARM LTD. ARMv8 architecture reference manual, for armv8-a architecture profile (arm ddi 0487). https://developer.arm.com/documentation/ddi0487/, 2024.

[31] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. SLUBStick: Arbitrary memory writes through practical software Cross-Cache attacks within the linux kernel. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4051–4068, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick.

[32] maplgebra. CVE-2023-3824: Lucky off-by-one (two?). https://m4p1e.com/2024/03/01/CVE-2023-3824, 2024.

[33] maplgebra. N1ctf24 php master writeup. https://m4p1e.com/2024/11/12/n1ctf24-php-master, 2024.

[34] Trilby Media. Grav - a modern flat-file cms. https://getgrav.org/, 2025.

[35] Jennifer Miller, Manas Ghandat, Kyle Zeng, Hongkai Chen, Abdelouahab Habs Benchikh, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. System register hijacking: Compromising kernel integrity by turning system registers against the system. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 7427–7446, 2025.

[36] MITRE. CVE-2019-11043. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11043, 2019.

[37] MITRE. CVE-2019-6977. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6977, 2019.

[38] MITRE. CVE-2022-31626. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31626, 2022.

[39] MITRE. CVE-2024-2961. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-2961, 2024.

[40] Julien Pauli, Nikita Popov, and Anthony Ferrara. PHP internals book. https://www.phpinternalsbook.com/index.html.

[41] Julien Pauli, Nikita Popov, and Anthony Ferrara. Zend memory manager. https://www.phpinternalsbook.com/php7/memory_management/zend_memory_manager.html, 2024.

[42] PHP Group. PHP pr 13943: Add two checks for zend_mm_heap's integrity. https://github.com/php/php-src/pull/13943, 2024.

[43] PHP Group. PHP pr 14054: Detect heap freelist corruption. https://github.com/php/php-src/pull/14054, 2024.

[44] PHP Group. PHP pr 14304: Remote heap feng shui / heap spraying protection. https://github.com/php/php-src/pull/14304, 2024.

[45] PHP Group. PHP pr 14339: Add an unlink check for php_stream_bucket_unlink. https://github.com/php/php-src/pull/14339, 2024.

[46] PHP Group. PHP pr 16765: Refresh zend mm shadow key on fork. https://github.com/php/php-src/pull/16765, 2024.

[47] PHP Group. PHP pr 19287: Call php_child_init() after fork during preloading. https://github.com/php/php-src/pull/19287, 2025.

[48] phpMyAdmin. A web interface for mysql and mariadb. https://www.phpmyadmin.net, 2025.

[49] LLVM Project. Scudo hardened allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[50] r3kapig. Securinets ctf quals 2024(jeopardy). https://r3kapig-not1on.notion.site/Securinets-CTF-Quals-2024-Jeopardy-118ec1515fb980eeadebc7b6df22a7a1, 2024.

[51] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[52] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[53] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[54] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2389–2403, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3133957.

[55] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 117–133, Baltimore, MD, August 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/silvestro.

[56] Yii Software. Yii 2: The fast, secure and professional php framework. https://www.yiiframework.com, 2025.

[57] Symfony Core Team. The symfony php framework. https://symfony.com, 2025.

[58] Chris Valasek. Understanding the low fragmentation heap. *Black Hat USA*, 11, 2010.

[59] Julien Voisin. Heap hardening. https://github.com/php/php-src/issues/14083, 2024.

[60] Julien Voisin. Make some parts of _zend_mm_heap read-only at runtime. https://github.com/php/php-src/pull/14570, 2024.

[61] Jakub Vrana. Database management in a single php file. https://www.adminer.org, 2025.

[62] W3Techs. Usage statistics of php for websites. https://w3techs.com/technologies/details/pl-php, 2026.

[63] Ruizhe Wang, Meng Xu, and N Asokan. Semalloc: Semantics-informed memory allocator. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1375–1389, 2024.

[64] Zicheng Wang, Yicheng Guang, Yueqi Chen, Zhenpeng Lin, Michael Le, Dang K Le, Dan Williams, Xinyu Xing, Zhongshu Gu, and Hani Jamjoom. {SeaK}: Rethinking the design of a secure allocator for {OS} kernel. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1171–1188, 2024.

[65] WordPress. Blog tool, publishing platform, and cms. https://wordpress.org, 2025.

[66] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1111–1128. USENIX Association, August 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/yun.

Appendix

A Timeline of PHP Heap Hardening

The PHP heap hardening roadmap was opened on Apr. 30, 2024 and lists allocator unlink checks, freelist shadow pointers, read-only metadata, guard pages, freelist randomization, heap isolation, and related isolation ideas [59]. The protections in Table A1 are the ones that have available prototypes and materially affect our evaluation.

Protection	Status
Shadow pointer [43]	Merged to PHP 8.4.0+
Fork-refresh Patches [46, 47]	Merged to PHP 8.5.0+
Unlink Abuse Prevention [42]	Merged to PHP 8.4.0+
Read-only metadata [60]	Closed prototype
Remote heap isolation [44]	Open prototype

Table A1: Available PHP heap hardening protections.

The freelist shadow pointer PR was merged into master on Jun. 12, 2024 [43], and the unlink checks were merged into master on Apr. 23, 2024 [42], while the stream-bucket variant [45] remains open. Two fork-refresh patches were later merged into master on Jul. 29, 2025 [46, 47] to fix the shadow pointer randomization issue. Heap isolation is not released, but it directly targets remote heap shaping, so evaluating it helps estimate the value of this direction. Read-only Metadata has two versions as discussed in §2.4 but neither is deployed or considered an imminent mitigation. We include the later version which is more practical according to the discussion thread [60] only as a representative metadata-protection design. §5 addresses this class of design, while the techniques in §4 and §6 remain relevant regardless of whether PHP adopts read-only metadata.

Algorithm 1 Reverse DJBX33A Hash

Input: Target hash value H, target input length n
Output: String s of length n such that `DJBX33A(s) = H`
$H_0 \leftarrow$ `DJBX33A`(n zero bytes)
$\Delta \leftarrow (H - H_0) \bmod 2^{64}$
seq $\leftarrow []$
while $\Delta \geq 33$ **do**
    seq.append($\Delta \bmod 33$)
    $\Delta \leftarrow \left\lfloor \frac{\Delta}{33} \right\rfloor$
end while
seq.append(Δ)
$s \leftarrow$ `n - len(seq)` zero bytes $\|$ `reverse(seq)`
return s

Name	Stars	JSON	XML	M	Y
Top 5 PHP Frameworks on GitHub					
Laravel [25]	34479	✓	✓	✓	×
Symfony [57]	30922	✓	✓	✓	✓
CodeIgniter [14]	18226	×	✓	✓	×
Yii 2 [56]	14319	✓	✓	✓	×
CakePHP [13]	8785	✓	✓	✓	×
Popular PHP Applications					
WordPress [65]	20071	✓	✓	✓	×
Grav [34]	14903	✓	✓	✓	✓
Magento2 [1]	11768	✓	✓	✓	×
phpMyAdmin [48]	7495	✓	✓	✓	✓
HotCRP [24]	389	✓	✓	✓	×

M refers to the serialized format from the `serialize` function in PHP and Y refers to the YAML serialization language.

Table A2: Comparison of supported serialization formats usage in popular open source PHP frameworks and applications.

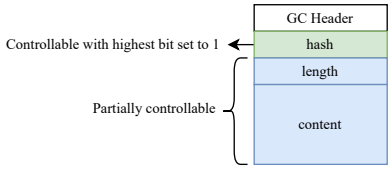

Figure 13: Spray String with Controllable Hash Value and Partially Controllable Content

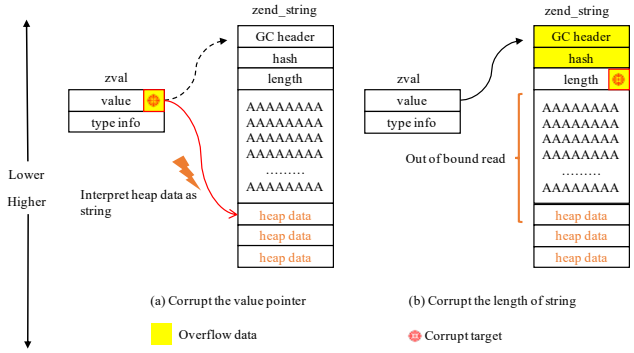

Figure 14: Convert heap overflow to out-of-bounds read