Argus: All your (PHP) Injection-sinks are belong to us.

Rasoul Jahanshahi and Manuel Egele, *Boston University*

https://www.usenix.org/conference/usenixsecurity24/presentation/jahanshahi

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Argus: All your (PHP) Injection-sinks are belong to us.

Rasoul Jahanshahi
Boston University
rasoulj@bu.edu

Manuel Egele
Boston University
megele@bu.edu

Abstract

Injection-based vulnerabilities in web applications such as cross-site scripting (XSS), insecure deserialization, and command injection have proliferated in recent years, exposing both clients and web applications to security breaches. Current studies in this area focus on detecting injection vulnerabilities in applications. Crucially, existing systems rely on manually curated lists of functions, so-called sinks, to detect such vulnerabilities. However, current studies are oblivious to the internal mechanics of the underlying programming language. In such a case, existing systems rely on an incomplete set of sinks, which results in disregarding security vulnerabilities. Despite numerous studies on injection vulnerabilities, there has been no study that comprehensively identifies the set of functions that an attacker can exploit for injection attacks.

This paper addresses the drawbacks of relying on manually curated lists of sinks to identify such vulnerabilities. We devise a novel generic approach to automatically identify the set of sinks that can lead to injection-style security vulnerabilities. To demonstrate the generality, we focused on three types of injection vulnerabilities: XSS, command injection, and insecure deserialization. We implemented a prototype of our approach in a tool called Argus to identify the set of PHP functions that deserialize user-input, execute operating system (OS) commands, or write user-input to the output buffer. We evaluated our prototype on the three most popular major versions of the PHP interpreter. Argus detected 284 deserialization functions that allow adversaries to perform deserialization attacks, an order of magnitude more than the most exhaustive manually curated list used in related work. Furthermore, we detected 22 functions that can lead to XSS attacks, which is twice the number of functions used in prior work. To demonstrate that Argus produces security-relevant findings, we integrated its results with three existing analysis systems – Psalm and RIPS, two static taint analyses, and FUGIO, an exploit generation tool. The modified tools detected 13 previously unknown deserialization and XSS vulnerabilities in WordPress and its plugins, of which 11 have been assigned CVE IDs and designated as high-severity vulnerabilities.

1 Introduction

In recent years, web applications have become an inseparable part of users' daily online lives, providing the means for communication, news, media, and financial services. The plethora of sensitive information held by the databases behind such web applications becomes a lucrative target for cyber-criminals. In 2017, Symantec reported that one in every 13 web requests was malicious [4]. Furthermore, the ever-increasing number of discovered vulnerabilities exposes web applications as well as their users to security breaches [10]. Without loss of generality, we focused on PHP as it powers 77% of all live websites [28].

Injection vulnerabilities (e.g., command injection, insecure deserialization, or XSS) are the most common category of application vulnerabilities in web applications [10]. To exploit such injection vulnerabilities attackers provide malicious inputs to the web application, compromising both the backend systems as well as the clients. The root cause of an injection vulnerability is passing insufficiently sanitized attacker-controlled user-input to *sensitive* APIs. Depending on the injection vulnerability type, the sensitive APIs differ. For instance, PHP's `echo` is recognized as a sensitive API for XSS attacks, while `unserialize` plays the same role for insecure deserialization. Despite these differences, at their core, injection vulnerabilities are data-flow problems where untrusted user-input is accepted at information sources (e.g., HTTP request parameters), propagated throughout the web application's execution, and finally reaching sensitive sinks (e.g., `echo` or `unserialize`).

Existing systems proposed different approaches to detect or exploit injection vulnerabilities using static, dynamic, or hybrid analysis of web applications. Several approaches rely on static taint analysis to track unsanitized user-input to a *predefined* list of sensitive APIs in order to detect different types of injection vulnerabilities [1, 7–9, 21, 32, 42, 42]. Similar to static approaches, prior work also relied on dynamic and hybrid techniques to identify injection vulnerabilities or generate exploits for already detected ones [25, 27].

Despite the effectiveness of prior work in detecting security vulnerabilities, all existing systems possess one common flaw:

They rely on a manually curated or predefined list of sensitive APIs to identify injection vulnerabilities. The accuracy of manually curated lists for sensitive APIs depend on the documentation for the programming language and the expertise of the analyst who identifies the sensitive APIs. As is often the case with human involvement, the listings are not comprehensive which leads to undetected injection vulnerabilities (i.e., false negatives) in web applications. This intuition is not merely hypothetical; our results show, incomplete lists of sensitive sink APIs are commonplace and further lead to false negative results.

Consider, as an example, the vulnerability type of insecure deserialization. Insecure deserialization occurs when an application deserializes untrusted data, such as user-input or generally attacker-controlled data. This vulnerability allows an adversary to manipulate the control-flow of a vulnerable application by injecting a malicious serialized object. As applications commonly interact with objects after deserializing them, an attacker-controlled object can lead to attacks, including arbitrary code execution. Existing studies of insecure deserialization and PHP object injection focus on detection [7, 24, 37] and automated exploit generation [9, 27]. RIPS [7] and Psalm [37] detect PHP object injection (POI) vulnerabilities by statically tracking user-inputs to the `unserialize` API function in PHP applications. Dahse et al. [9] presented the first automated approach to generating POI exploits by tracking the flow of user-input to invocations of the `unserialize` API. While Dahse et al.'s approach identifies POI vulnerabilities, FUGIO [27] utilizes a fuzzing approach to generate a concrete exploit object for an *already detected* POI vulnerability.

Deserialization in PHP is the consequence of invoking PHP API functions that perform deserialization either *explicitly* or *implicitly*. For consistency, we use the term *PHP API* to refer to the set of functions provided by the PHP runtime that can be directly invoked by web applications. Explicit deserialization APIs are described throughout the PHP manual [12]. One can identify the functions of this group by studying or analyzing the PHP documentation to curate a list of explicit deserialization functions – this approach was taken by RIPS [7], Psalm [37], FUGIO [27], and Dahse et al. [9]. However, implicit deserialization functions are not described in the PHP manual, and are therefore more challenging to identify. First described by Thomas for PHP in [36], implicit deserialization happens if the PHP interpreter transparently deserializes data consumed by certain APIs. For example, Thomas found that the stream wrapper for the PHP archive (PHAR) format implicitly deserializes metadata, which led to the identification of multiple PHAR-based deserialization vulnerabilities in popular web applications and libraries such as WordPress and TCPDF (details in Section 2.4). Crucially, implicit deserialization functions are equally as potent for adversaries as explicit ones. However, implicit deserialization APIs are not documented publicly, and the number and identity of these functions depend on the implementation of the PHP interpreter (i.e., the set of deserialization APIs varies with the PHP version). This implies that in order to identify the set of implicit deserialization functions featured by a given PHP version, one must analyze the implementation of the interpreter itself. The same claim also applies to other injection-type vulnerabilities, such as XSS and command injection.

To improve on the error-prone manual efforts on identifying injection APIs, this paper provides a systematic and principled approach to inferring a comprehensive set of APIs that can lead to XSS, command injection, or insecure deserialization. To this end, we design, implement, and evaluate an automated approach called Argus to identify the list of APIs that deserialize user-input, execute OS commands, or write to the output buffer. For consistency, we use the term , *deserialization*, *output* and *exec API* to refer to the set of PHP APIs that perform deserialization, write to the output buffer, or execute OS commands, which can lead to an insecure deserialization, XSS, and command injection vulnerability, respectively. We demonstrate that by incorporating Argus' resulting sink list into existing systems, those systems produce significantly higher-quality results (i.e., more detected vulnerabilities and exploits). Our key observation is that there is precisely *one* function inside the PHP interpreter that deserializes data – `php_var_unserialize`. Crucially, `php_var_unserialize` is at the core for *all* explicit and implicit deserialization APIs. Consequently, we argue that the correct way to identify the set of deserialization APIs is to subject the PHP interpreter to program analysis. In cases of XSS vulnerabilities, a PHP API such as `echo` prints its argument to the output buffer, which contains the HTML response sent back to the user's browser. Our observation shows that there is one function inside the PHP interpreter responsible for writing to the output buffer called `php_output_write`. Similarly, the set of exec APIs will eventually invoke the `execve` system call in order to execute the OS command. For brevity, we use the term *vulnerability indicator functions* (*VIF*) throughout the text to refer to the functions `php_var_unserialize`, `php_output_write`, and the functions inside the PHP interpreter that directly invoke the `execve` system call or use its front-ends in C (e.g., `exec` family).

Argus detects the set of deserialization, exec, and output APIs through an automated hybrid static-dynamic program analysis of the PHP interpreter. Specifically, Argus first generates the call-graph for a given PHP interpreter. Subsequently, Argus performs a reachability analysis to identify the set of PHP API functions whose invocation can trigger either deserialization or output API in the interpreter (i.e., reach the *VIF*). Using this approach, Argus detected more than 280 functions (in PHP 7.2) that deserialize their arguments and 22 functions whose invocation writes their arguments to the output buffer.

We demonstrate the security impact of Argus by integrating its findings as the set of injection-sinks into three PHP analysis tools; Psalm [37] and RIPS [1] (static taint analyses to identify POI, command injection, and XSS vulnerabilities) and FUGIO [27] (a POI exploit generation system). When evaluating

Psalm with the extended list of injection-sinks on over 1,900 popular PHP applications and plugins, our evaluation yielded 10 times more potential for insecure deserialization compared to Psalm's default implementation (which exclusively reported false positives). Furthermore, we integrated Argus' list of deserialization sinks into FUGIO and discovered proof-of-concept exploits for 12 previously unknown deserialization vulnerabilities on the same data set. In summary, this paper makes the following contributions:

- We draw attention to the importance of accurately and comprehensively detecting APIs in PHP that lead to injection vulnerabilities such as insecure deserialization, XSS, and command injeciton.
- We design and implement a novel automated analysis and prototype (called Argus) to identify the set of deserialization, exec, and output APIs applicable to any version of the PHP interpreter.
- Evaluating Argus on the three most popular PHP versions identifies over 280 deserialization APIs, at least an order of magnitude more than the most extensive manually curated list used by prior work (i.e., 26 sinks in FUGIO). An adversary can leverage a call to any of these APIs to exploit a PHP application. Argus also detects 22 output APIs, which is twice the number of APIs that were used in the past to detect XSS vulnerabilities. In addition, Argus detects 9 exec APIs which can be used to detect command injection vulnerabilities by Psalm and RIPS.
- In order to demonstrate the real-world impact of these findings, we incorporated our results into three existing analysis tools, Psalm, RIPS, and FUGIO. Our refinements of the sink-lists used by these tools led to the identification of 13 previously unknown POI and XSS vulnerabilities. Of course, we responsibly disclosed all our findings to the corresponding developers. As such, 11 of the detected POI vulnerabilities have been assigned CVE IDs (CVSS scores between 7.2 and 8.8), and seven vulnerabilities have already been patched.

2 Background and Motivation

In this section, we describe XSS, command injection, and deserialization in PHP and how these vulnerabilities arise (e.g., implicitly from file operations). This background allows us to shed light on our results and put the evaluation presented in Section 4 in context. Finally, we elaborate on our assumptions about XSS and insecure deserialization attacks on PHP applications and our motivation to design an automated approach to identify a comprehensive set of deserialization, exec, and output functions in an interpreter.

2.1 PHP interpreter and Cross-site Scripting

A cross-site scripting (XSS) vulnerability is an injection vulnerability that allows an attacker to compromise the interactions of the victim with a vulnerable application [21]. This vulnerability allows the attacker to execute malicious scripts in the victim's web browser by including malicious code in a legitimate web application.

In the life-cycle of a request sent to a web-server such as Apache or Nginx, the PHP interpreter plays an important role in providing the output shown to the user. When a web-server receives a request for a PHP script, the web-server invokes the PHP interpreter to determine the output. The PHP interpreter executes the PHP script, which can include interaction with a database, the file system, or the underlying operating system. After the execution of the script, the PHP interpreter provides the output in the form of HTML to the web-server. The output is then sent back to the user as the response.

Thus, the PHP interpreter determines the response that the web-server sends back to users. In order for a PHP script to assemble the response, the PHP interpreter provides a set of built-in functions (i.e., PHP API), which PHP scripts can use. One of the APIs that is used to modify the response of a web-server is `echo`. This function accepts one or more strings, which are then sent verbatim to the output buffer. However, the set of APIs that can modify the response of a web-server is not limited to only one API. According to prior work such as RIPS [7], there are 12 functions in the PHP interpreter that can modify the output buffer.

In a cross-site scripting attack, the attacker is able to modify the response that is sent back to the user's browser. If an attacker has control over the arguments passed to an API such as `echo`, an XSS attack is a certainty. This capability of the `echo` API is provided by an internal function of the PHP interpreter, which allows APIs to write into the output buffer (i.e., the HTML response). An analysis of the source-code of the PHP interpreter reveals that all write opration to the output buffer goes through a function called `php_output_write`. As a result, all invocations of the `php_output_write` by the PHP APIs can modify the response sent back to the web-server, which is the superset of all the APIs identified in prior work.

2.2 PHP interpreter and command injection

Command injection in PHP applications occurs when a malicious actor gains the ability to execute arbitrary commands (e.g., through a shell) [3]. Command injection attacks result in a range of consequences, such as compromised data confidentiality and integrity or unauthorized access to the system hosting the application. An attacker can leverage the exploited application to execute a malicious payload and gain access to additional resources.

As mentioned in Section 2.1, in the life-cycle of a request, the PHP interpreter executes a PHP script. During this execution,

the PHP interpreter communicates with the resources through the operating system to determine the output.

The operating system provides mediated access to resources through the system call API. Applications such as the PHP interpreter can access OS resources by invoking system call APIs. A PHP interpreter executes OS commands by invoking the `execve` system call or its wrappers in the C libraries, such as the family of `execv` functions [22]. Consequently, a command injection attack only occurs when the PHP application uses a PHP API that invokes the `execve` system call and passes insufficiently sanitized user-input. As a result, a PHP API is part of the *Exec API* if its underlying implementation invokes the `execve` system call.

2.3 PHP Object Injection

PHP object injection (POI) is a security vulnerability that leverages insecure deserialization in PHP applications. To exploit such a vulnerability, an adversary must control the properties of an insecurely deserialized object. By exploiting a POI vulnerability, an attacker can potentially hijack the program's execution by controlling the properties used in automatic calls to the `__wakeup` and `__destruct` methods.

The snippet in Listing 1 presents a PHP script that contains a deserialization vulnerability. We observe that at Line 9, user-input is passed to the `unserialize` function without sanitization. In order to exploit this vulnerability, the attacker needs to satisfy two conditions.

- There needs to be at least one class in the application which implements the class methods `__wakeup` or `__destruct` to carry out the attack.
- All of the classes used in the exploit need to be defined (or the application must support automatic loading of classes) when the `unserialize` function is called on Line 9.

Exploiting a POI vulnerability is inherently a code-reuse attack, where an attacker recombines the already existing code to achieve a malicious outcome by introducing a malicious object. To exploit a POI vulnerability the attacker needs to identify the user-defined functions and methods (i.e., gadgets) in the PHP app that can be used to achieve his goals [27]. As an example, we describe how an attacker can choose the gadgets to link and perform a remote code execution attack in Listing 1. Looking at Listing 1, the script defines two classes prior to the deserialization: `Example` and `Exec`. The destructor of class `Example` calls a function named `getValue` from the variable `obj`. If an attacker sets the variable `obj` to an object of class `Exec`, then the destructor will call the class method `getValue` at Line 7. Looking at the implementation of class `Exec`, the method `getValue` invokes the function `system` on the `_cmd` property. Hence, the attacker can run an arbitrary command be setting the value of the `_cmd`. The code snippet in Listing 3 contains the exploit for the vulnerability in Listing 1.

```
1  class Example {
2    protected $obj;
3    function __destruct() {
4        return $this->obj->getValue();} }
5  class Exec {
6    private $_cmd;
7    function getValue() {
8      system($this->_cmd); } }
9  $user_data = unserialize($_POST['data']);
10 file_exists($_POST['file']);
```

Listing 1: A deserialization vulnerability leading to arbitrary code execution. An adversary can execute any command by crafting a PHP object which modifies the value of `_cmd` property.

2.4 Stream Wrappers

In this section, we explain the concept of PHP stream wrappers and how an attacker can abuse stream wrappers to cause a PHP object injection. A stream in the PHP interpreter is a generalization of a data source which implements a set of common file operation functions such as `fopen` and `copy`. PHP Stream wrappers allow developers to use consistently-named file-related functions such as `fopen` for different types of file resources. The types of resources are identified analogous to URL schemes and can vary from classic local files (e.g., `/etc/passwd`), network reachable resources (e.g., `https://example.com/text`), to PHAR archive types (e.g., `phar://usr/share/app.phar`). Importantly, once the resource's type is identified, the PHP interpreter, maps each type to a corresponding stream wrapper which allows the application developer to transparently perform (supported) file operations on the resource (e.g., `read`, `seek`, etc.).

PHP Archives (`phar`) allow developers to package an entire PHP application in a single file. To interact with phar files, PHP provides a built-in stream wrapper. Each phar file contains the following sections:

- **Stub:** A PHP file that instructs the interpreter how to load the application.
- **Manifest:** Includes the number of files in the phar, as well as the file permissions, type of compression, and serialized metadata. The metadata includes a description for the existing files in the archive in a *serialized* format.
- **Contents:** The content of files in the phar archive.
- **Signature:** An optional signature for the file's integrity.

Exploiting phar wrappers. Thomas in [36] demonstrated how an attacker can exploit an invocation of a file operation API and perform a PHP object injection. He showed that the PHP interpreter deserializes the metadata upon any file operation on a phar file. Considering the aforementioned information, an adversary can achieve arbitrary code execution by leading the PHP interpreter to perform file operations (e.g., `file_exists`) on a phar file with a malicious metadata field.

The second part of Listing 3 shows how an attacker can generate a phar file with malicious metadata (set on Line 12). Looking at the snippet in Listing 1, we observe that the PHP script checks the existence of a file by passing an unsanitized user-input at line 10. In order to exploit the vulnerability at Line 10 of Listing 1, the attacker can set the `$POST` variable `file` to `phar://path-to-malicious-phar-file`.

2.5 Observations, Motivation and Assumptions

Our work is motivated by the error-prone human efforts to aggregate lists of deserialization, exec, and output APIs in PHP. Consider the case of CVE-2022-2437, an insecure deserialization vulnerability that Argus identified in the popular `Feed Them Social` WordPress plugin (detailed discussion in Section 4.3.1). This vulnerability exists because of the implicit (i.e., undocumented) deserialization performed by the `get_meta_tags` PHP API function. As this API is missing from all sink lists of prior systems, the vulnerability went unnoticed. Argus automatically identified this function (along with over 280 others) as a deserialization API, and by incorporating this knowledge in existing POI analysis systems these systems readily detected and created POC exploits for this vulnerability.

As mentioned in the introduction, Argus' analysis relies on a key observation regarding the invocation of deserialization, exec, or output APIs inside the PHP interpreter. In the case of deserialization, the PHP interpeter uses its own template and formatting for serialized data. The PHP interpeter uses a customized *yacc* [20] parser to first parse the serialized data and then perform the deserialization. Our analysis of the PHP source-code shows that there is only *one* function inside the PHP interpeter, which uses the *yacc* parser. Hence, there is only *one* function responsible for deserialization: `php_var_unserialize`. In the case of output APIs, we analyzed the source-code of the PHP interpreter. Specifically, we analyzed the underlying implementation of output APIs such as `echo` and `print_r` and identified the set of invoked functions inside the interpreter. For a more comprehensive study of the output APIs, we executed the official PHP testsuite and analyzed all the function traces for each output API that leads to writing to the output buffer (i.e., using the `write` system call). Similar to our observation for deserialization, the PHP interpreter uses the `output` module and specifically the function `php_output_write` inside this module to write to the output buffer (i.e., the HTML response).

Many implicit deserialization and output APIs are vulnerable if an attacker can invoke them on a malicious file (e.g., PHAR archives). For such attacks to succeed, the malicious file must reside on the web application's file system. Thus, to demonstrate exploitability, we assume an attacker already uploaded a malicious file to the underlying web-server. Note that, while this assumption is realistic (see Section 5 for a more detailed explanation), the assumption purely exists to demonstrate exploitability and is orthogonal to Argus' goal of identifying deserialization and output APIs.

3 System Design

In this section, we discuss the salient characteristics of our approach – Argus – and how it identifies the set of deserialization, exec, and output PHP APIs. Figure 1 illustrates the overall process. First, Argus combines static and dynamic analysis techniques to generate a call-graph of a PHP interpreter in Step ①. Subsequently, for Step ②, Argus uses the call-graph to perform a reachability analysis to determine the set of API functions that invoke the *VIF*s. Furthermore, Step ③ discusses the validation mechanism in Argus to confirm the injection-sinks. Finally, we discuss how we incorporate Argus' results into existing program analyses to detect and exploit previously unknown vulnerabilities.

3.1 Call-graph Generation

In Step ①, Argus generates a call-graph for the PHP interpreter. To achieve this, Argus performs a static analysis on the PHP interpreter to generate the initial call-graph, which it then refines using dynamic execution traces.

3.1.1 Static Analysis of the PHP Interpreter

To construct the call-graph, Argus analyzes the PHP interpreter. The PHP interpreter's core consists of approximately 120K lines of C code. Additionally, the interpreter relies on extensions to deliver features such as image processing, database communication, and communication protocols such as LDAP and IMAP. These extensions are free to augment the PHP API, including adding additional injection-sinks, and frequently do so. For example, in a standard PHP deployment, the *GD* graphics library, the *PDO* database communication extension, and the FTP extension are provided as separate libraries and all add additional injection-sinks to the runtime. To complicate matters further, extensions can be written in different programming languages (e.g., [6, 41]), provide their own build environments, and are usually simply loaded by the interpreter as shared dynamic libraries. However, injection vulnerabilities can arise from any API provided by the runtime, including APIs provided by the core interpreter and those provided by extensions. As such, it is imperative to analyze the interpreter's core along with the code that comprises the extensions. At first glance, the open source nature of the PHP interpreter would suggest a source-based analysis to infer the call-graph. However, the variety of frameworks, languages, and build systems used for extensions would require an analysis catering to all these characteristics. Thus, instead of deriving call-graph information from various interconnected source-based analyses, Argus instead performs its call-graph analysis on the compiled

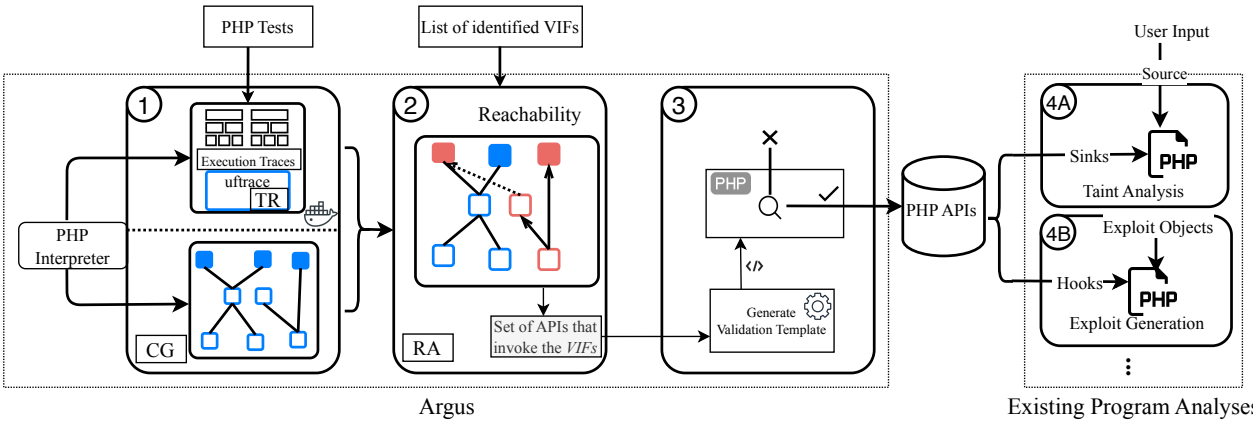

Figure 1: Argus performs a hybrid static-dynamic analysis on the PHP interpreter to generate a call-graph. Next, Argus identifies a comprehensive set of output, exec, and deserialization APIs through reachability analysis and validation tests. The output of Argus can be used to improve the existing program analysis tools to identify POI and XSS vulnerabilities.

binaries of the interpreter and its extensions. To facilitate this analysis, we build the runtime and include debug symbols.

The call-graph analysis (`CG` in Figure 1) first disassembles the PHP interpreter and all of its shared libraries using the `objdump` tool. Argus builds the call-graph by adding a node for each binary symbol in the disassembled PHP interpreter. Subsequently, Argus performs a linear scan over the interpreter and library disassembly. For every call instruction, `CG` draws an edge in the call-graph from the caller (i.e., the currently analyzed symbol) to the callee (i.e., the target of the call). This analysis works well for direct calls and calls to symbols provided by extensions. That is, direct calls will invoke symbols that have corresponding names in the debug information. Argus handles calls to imported symbols by launching the PHP interpreter with the `LD_DEBUG=binding` environment variable set to infer symbol binding information from extension libraries. The `LD_DEBUG` option allows Argus to resolve the external symbols to the library and address where the symbols are implemented. Unfortunately, indirect calls (e.g., those that are used to implement the concept of stream wrappers) elide this analysis.

3.1.2 Refining Call-graph using Dynamic Analysis

Argus uses dynamic analysis to handle indirect calls in the PHP interpreter and refine the statically generated call-graph created in the previous step. For instance, PHP's `fopen` can be used to access local or remote files over protocols such as HTTP, HTTPS, or FTP. Depending on the argument passed to `fopen`, the PHP interpreter decides which stream wrapper (see Section 2.4) should handle the underlying resource. Internally, PHP stream wrappers rely on function pointers to dispatch operations (e.g., `fread()`) to functions that handle the protocol corresponding to the opened resource. The static analysis in Argus cannot handle such cases implemented in the PHP interpreter and runtime. To address this issue, Argus improves the statically generated call-graph by tracing the execution of the PHP interpreter while executing its high-quality test-suite (i.e., the PHP test-suite achieves 70% function coverage on average for our dataset of three different PHP interpreters). Argus then uses this dynamic information and adds any edges not already detected by the static analysis to the call-graph.

To achieve this, we compile the PHP interpreter with the `-pg` flag. This flag instruments each function with two additional hook functions at the entry and exit of each function, which allow Argus to perform dynamic tracing [35]. The first function call occurs just after each function entry, which invokes the function `__cyg_profile_func_enter`. The next function call invokes the function `__cyg_profile_func_exit` before exiting each function. After the recompilation of the PHP interpeter, Argus uses the `uftrace` tool [23] (`TR` in Figure 1) to implement both hook functions and record dynamic traces. Finally, Argus executes the PHP unit tests while `uftrace` records the execution traces for each test-case.

After recording the execution traces, Argus iterates over the sequence of invoked functions by each test-case and examines the statically generated call-graph for the missing edges. For every invoked function during the dynamic analysis, Argus draws an edge between the pair of functions in the execution trace if there is no edge representing the recorded invocation.

At the end of this step, Argus has assembled a static call-graph of the PHP interpreter and refined it using dynamically-collected traces of PHP unit tests.

3.2 Reachability Analysis

In Step (2), Argus performs a reachability analysis on the generated call-graph, which requires the identification of sources and sinks on the call-graph. In this analysis, Argus identifies the set of PHP APIs that reach the VIF functions. We define VIFs as the minimal set of PHP internal functions, that user-input must pass through to trigger a vulnerability. As

stated in Section 1, the PHP interpreter uses a single internal function that is responsible for all deserialization operations, called `php_var_unserialize` (*VIF*). The PHP interpreter uses a custom parser to parse serialized strings, which are then converted to PHP objects. Our analysis of this custom parser across PHP's source-code yielded a single deserialization function inside the PHP interpreter. In the case of output APIs that write to an output buffer, we observed a similar pattern during our analysis of PHP's code, where the function `php_output_write` is exclusively responsible for outputting the buffer. As mentioned in Section 2.2, any PHP API that executes an OS command requires the `execve` system call. As a result, in order to identify the functions inside the PHP interpreter (VIFs) for command injection, Argus needs to identify the functions that call the `execve` system call. To achieve this, we leveraged Saphire [3], to identify functions that invoke the `execve` system call. We marked these functions as the VIFs for the exec APIs. These PHP internal functions are `php_exec`, `zif_shell_exec`, `zif_popen`, `phpdbg_do_sh`, `php_mail`, `zif_pcntl_exec`, `_php_imap_mail`, and `zif_proc_open`. With the *VIF* identified, Argus labels all API functions in the call-graph as sources.

Unfortunately, the symbols for API functions are indistinguishable from those of internal functions, and text-based techniques that parse documentation are rarely, if ever, accurate. However, a running PHP process must be aware of any and all APIs exposed to the web applications running on top of it. Thus, to identify the set of APIs, Argus uses a PHP extension that, once loaded into the PHP interpreter, iterates over all available APIs. Specifically, the extension first invokes PHP's `get_defined_functions` API to obtain the list of all API functions. Unfortunately, the results of `get_defined_functions` cannot be directly mapped to the call-graph. The reason is that the name of an API function available to a web application is commonly different from the name of the symbol that implements the actual functionality. For example, the `session_decode` PHP API is implemented by a function called `zif_session_decode`. Unfortunately, the `zif` prefix is not a consistent pattern. As the nodes in the call-graph correspond to symbol names rather than API names, the API names have to be translated. To this end, the extension leverages an interpreter-internal data structure (i.e., `executor_globals.function_table`), which maps API names to the names of the functions that implement the APIs' functionalities. Finally, the extension relays this information to Argus, which labels the symbols that map to API functions in the call-graph accordingly.

Once all APIs are labeled as sources, Argus traverses the call-graph for each source node and follows any call edges captured in the graph. Argus identifies an API as a deserialization, exec, or output API if this traversal includes the graph node corresponding to its *VIF* function.

3.3 Validation

The reachability analysis presented above might inappropriately label an API as an injection-sink if the underlying implementation in the runtime performs input sanitization or filtering. Thus, Step ③ filters APIs and only passes those that propagate their input to *VIF* unmodified. To this end, Argus automatically generates PHP snippets to test each identified API for this characteristic. More specifically, these snippets contain a class definition (i.e., `test`) that, if deserialized (i.e., its `__wakeup` method is invoked), prints the content of one of its properties (i.e., `msg`) as a success message. Subsequently, the template calls the API in question with a serialized `test` object that has `msg` set to "SUCCESS". Thus, if the execution of the PHP snippet prints the success message, Argus validates that the API in question passes the input argument unmodified to *VIF*, and hence the API is for sure a deserialization API. For each API, the validation step iterates over various patterns of passing inputs including explicit (e.g., serialized data) and implicit (e.g., Phar file). When one input leads to deserialization which invokes our test function and prints out the "SUCCESS" message, Argus marks the API as a deserialization API. The psuedo-code in Listing 4 (Appendix B) shows the process of validating the reachable APIs identified.

As our evaluation in Section 4 will show, Argus confirmed 284 deserialization APIs in the PHP interpreter, which warranted an automated validation step. However, the number of confirmed output APIs in the PHP interpreter is an order of magnitude less (i.e., 22), which prompted us to manually validate whether the invocation of the API with user-input can cause an XSS attack. To this end, we created a Docker container with a running Nginx web-server for each PHP interpreter version. For each output API, we created a PHP template that invokes the API and passes a constant user-input containing a Javascript snippet (i.e., `<script>alert(1)</script>`). We visited the generated PHP template with a browser, and if the browser displays a dialog, Argus marks the tested API as an output API. In case of the exec APIs, we manually validated the set of PHP APIs that execute commands in the PHP interpreter. For each exec API, we created a PHP script that invokes the API and passes a constant user-input containing an OS command (i.e., `ls -lh` which lists the directory content). The PHP interpreter under test executes each of the generated PHP scripts, and if the PHP script prints out the directory information, Argus marks the PHP API as an exec API.

3.4 Extend Security Analysis Tools

Argus' results comprise a comprehensive list of injection-sinks. This is in contrast with the exclusively manually-crafted lists of injection-sink used by all existing XSS and POI detection and automatic exploit generation systems. Thus, to demonstrate the value of Argus' principled approach, we extend three existing

program analysis tools, Psalm [37], RIPS [7], and FUGIO [27] as examples of downstream analysis that benefit from our work.

3.4.1 Psalm and RIPS Extension

Psalm and RIPS are two static analysis tools featuring code refactoring and taint analysis [37]. For the taint analysis, Psalm and RIPS attempt to find unwanted flows between user-controlled inputs (e.g., `$_GET` variables) and a set of sink functions (e.g., `system`). The set of sink functions in Psalm and RIPS differs depending on the type of vulnerability that the user is trying to detect. For instance, to identify insecure deserialization, they exclusively consider `unserialize` as a sink for the core PHP interpreter.

As shown in previous reports [36] and in our evaluation, relying on incomplete lists of sinks results in false negatives (i.e., missed detections). To extend the static analysis tools, we modify the list of taint sinks to include all deserialization, exec, and output APIs identified by Argus.

3.4.2 FUGIO Extension

FUGIO is an automatic exploit generation tool which uses a combination of static and dynamic analysis to generate a proof of concept exploit for *previously known POI vulnerabilities*. In the first step, FUGIO submits requests to a target web application where request parameters (e.g., GET, POST, and COOKIE values) contain serialized data. During the processing of these requests, FUGIO hooks the invocation of deserialization APIs and verifies if the passed arguments correspond to the parameters supplied in the request. To this end, FUGIO hooks a subset of 27 PHP deserialization APIs – the explicit `unserialize` API along with 26 implicit APIs first mentioned by Thomas [36]. If FUGIO detects that parameters are indeed forwarded to deserialization APIs, its second step will attempt to morph the parameter into a complete POP chain, forming a POC exploit. While FUGIO's second step (i.e, the exploit generation itself), is independent of our work, the first step (i.e., recognizing the invocation of vulnerable deserialization APIs) is directly affected by the (in-)completeness of the list of deserialization APIs.

To extend FUGIO, we integrated the set of deserialization APIs identified by Argus such that FUGIO hooks all these APIs in its first analysis step. The extended FUGIO intercepts a comprehensive set of PHP APIs which allows it to identify and exploit previously unknown POI vulnerabilities (see Section 4.3.2 for details).

In summary, Argus generates a call-graph for the PHP interpreter by leveraging hybrid static-dynamic analysis. Furthermore, Argus performs a reachability analysis to identify a comprehensive set of deserialization, exec, and output APIs in the PHP interpreter, and optionally validates APIs that pass their inputs unchecked to the underlying *VIF* deserialization, exec, and output functions. We augmented three existing detection and exploit generation systems as examples that demonstrate the security impact of Argus' results.

Our implementation of the call-graph analysis for the PHP interpreter consists of approximately 700 LoC of Python and C code. In addition, we implemented our extensions to Psalm and FUGIO with less than 600 LoC of PHP.

4 Evaluation

In this section, we evaluate Argus along two orthogonal dimensions. First, we focus on identifying deserialization, exec, and output APIs in the three most popular major versions of the PHP interpreter. The reason for evaluating different interpreter versions is that the number and names of deserialization, exec, and output APIs are implementation and version dependent, calling for an automated solution such as Argus. In the second thrust of the evaluation, we assess how Argus' analysis results improve the accuracy of three example PHP security analysis systems – Psalm, RIPS, and FUGIO. To cover these two dimensions, our evaluation answers the following research questions:
RQ1: In terms of call-graph generation, how precise is the call-graph generated by Argus compared to existing call-graph generation tools such as Joern (Section 4.2.1)?
RQ2: On the interpreter's call-graph, how many PHP APIs reach the *VIF* functions (Section 4.2.2), and how many of the reachable APIs pass their arguments to *VIF* unmodified (Section 4.2.3)?
RQ3: How does the number and identity of deserialization, exec, and output APIs change across PHP versions and what are the reasons for the observed changes (Section 4.2.4)?
RQ4: How do Argus' results improve the current state-of-the-art PHP security analysis that target injection vulnerabilities? Does Argus' comprehensive list of injection-sinks lead to the identification of previously unknown POI and XSS vulnerabilities (Section 4.3)?

4.1 Evaluation Dataset

Our evaluation dataset for Argus is divided into two categories corresponding to the two evaluation dimensions. For our experiments on the PHP interpreter, we evaluated Argus on the three most popular major versions (i.e., versions 5, 7, and 8) of the PHP interpreter. As of June 2023, PHP engines of these versions power 99.8% of all live PHP websites, according to W3Tech data [28]. Furthermore, PHP seven is used by 65.2% of all live websites using PHP, which makes it by far the most popular PHP engine [28]. Our second dataset is used to evaluate the benefit of Argus' results to existing POI, command injection, and XSS detection systems as well as exploit generation systems. As these systems operate on the code of web applications, rather than the PHP interpreter, we aggregated a dataset corresponding to that purpose. We collected the most popular PHP applications and plugins from a variety of sources. On the one hand, we downloaded the 60

most popular PHP applications based on the reported popularity provided by W3Tech [28]. On the other hand, we recognize that large web applications frequently feature a plugin model that allows administrators to customize their sites. As such, we also collected the most downloaded plugins for the popular WordPress, Drupal, and Typo3 web applications from their respective repositories. Overall, we collected 1,977 PHP artifacts (i.e., web applications and plugins). Table 3 provides a detailed breakdown in the first two columns.

4.2 Analysis of the PHP Interpreter

As the PHP language and ecosystem evolves, the interpreter must provide support and functionality accordingly. Unsurprisingly, this evolution also affects the number and identity of the injection-sink functions provided by different versions of the PHP interpreter. To assess these changes, we evaluate Argus on three different versions of the PHP interpreter (versions 5.6, 7.2, 8.0) as detailed in Table 2.

PHP interpreter	Argus	Joern	Argus - Joern	Joern - Argus
PHP 5.6	56,504	31,065	26,024	585
PHP 7.2	68,410	39,560	30,620	1770
PHP 8.0	47,653	33,555	16,636	2538

Table 1: Argus outperforms Joern in terms of detected edges for analyzing the PHP interpreter. The second and third columns show the number of detected edges by each tool. The last two columns report the comparison between the number of detected edges (i.e., subtraction of matching edges).

4.2.1 Argus vs. Joern

Argus uses the call-graph of the PHP interpreter in order to identify injection-sink functions, rendering call-graph generation a crucial step for Argus. For our first evaluation, we investigate the generated call-graphs by Argus and compare the results with Joern, an open-source code analysis tool [19]. During this evaluation, we analyze the PHP interpreter and generate the call-graph using both Argus and Joern. We then compare the generated call-graphs by both tools based on the number of detected edges.

We compared the generated call-graphs in two dimensions: 1) a quantitative comparison of the call-graphs for the number of missing edges, and 2) a qualitative evaluation to investigate the effect of missing edges on identifying injection-sink functions. Our evaluation of the call-graphs generated by Argus and Joern is listed in Table 1. This comparison shows that the call-graph generated by Joern misses 24,426 edges that are included in the call-graphs generated by Argus, on average. There are also cases where Argus misses edges that Joern can detect. However, the number of missing edges by Argus is 15 times less than the number of edges missed by Joern.

For the second part of this evaluation, we investigated the missing edges in the call-graphs of both Argus and Joern. Compared to Argus which analyzes the binary, Joern is a source-code analysis tool. Hence, the compile-time aspects are an important source for the differences between the generated call-graphs by Argus and Joern. In the case of Argus, the missing edges are related to cases of preprocessor directives and compiler optimizations (i.e., these missing edges do not exit in the binary and cannot be exercised). For example, the preprocessor decides to keep or remove blocks of code based on directive conditions (e.g., `#ifdef`); hence, the compiled version of the same source-code can lead to a different binary artifact depending on the condition. As a result, Argus analyzed a version of the PHP interpreter where some function calls were removed due to preprocessor directives, compared to Joern. Furthermore, the analysis of the missing edges by Argus shows that the invoked functions are related to memory management in C, such as `free` and `malloc`. Our analysis shows that the missing edges do not affect the ability of Argus to detect injection-sink functions since the memory management functions in this case are the leaves in the call-graph and do not affect the reachability analysis.

In the case of Joern, the missing edges are mostly related to function pointers in the PHP interpreter. As mentioned in Sections 2.4 and 3.1.2, the PHP interpreter extensively uses function pointers in order to implement its functionality. Joern's call-graph analysis misses the set of indirect calls inside the PHP interpreter, which includes function pointers related to stream wrappers for different file types. Consequently, Joern is not able to detect indirect calls to the `PHAR` module from any file operation APIs such as `fopen`, and using such a call-graph would lead to missing all file operation APIs, which lead to insecure deserialization.

4.2.2 Reachable APIs

Next, we look into the reachability analysis of Argus and the number of PHP APIs that invoke the *VIF* in the analyzed PHP interpreters. The first set of sub-columns in Table 2 labeled as *Detected* for both injection vulnerabilities shows the number of APIs that Argus identified as reaching *VIF* for the three different PHP versions. As the table shows, the number of deserialization APIs for versions 5 and 7 is similar, and two orders of magnitude larger than for version 8. We discuss the difference in the number of reachable deserialization APIs in Section 4.2.4. Furthermore, the number of output and exec APIs for the analyzed PHP versions is almost constant across all three versions.

In our evaluation of Argus' call-graph generation, we explored the contribution of both of our dynamic and static analysis. This analysis demonstrates the advantages of using both static and dynamic analysis while generating the call-graph. To demonstrate the effectiveness, we looked into the number of injection-sinks that Argus can detect by only using the statically generated call-graph. To achieve this, we performed a reachability analysis on the statically generated call-graph of the PHP interpreter before augmenting the call-graph with dynamic information (Step A-2). The numbers

Version	Deserialization API		XSS-leading API		Exec API	
	Detected	**Validated**	**Detected**	**Validated**	**Detected**	**Validated**
PHP 5.6	419 (61)	281 (67%)	54 (51)	22 (41%)	10 (10)	9(90%)
PHP 7.2	425 (63)	284 (67%)	52 (48)	22 (42%	10 (10)	9(90%)
PHP 8.0	20 (13)	13 (65%)	46 (39)	22 (48%)	10 (10)	9(90%)

Table 2: Our analysis of PHP interpreter shows PHP interpreters prior to version 8.0, contained more than 300 PHP functions that deserialize their arguments, execute OS command, or write to output buffer. The numbers in parentheses of *Detected* sub-columns show the number of APIs detected using only the statically generated call-graph.

in parentheses in the sub-column *Detected* for Table 2 show the number of reachable APIs while using only the statically generated call-graph. As an example, we can see that the difference in the number of detected APIs for PHP 5.6 when only using the statically generated call-graph is six times less than when incorporating the dynamic analysis information. Similar to Joern, the missing edges in Argus' static only call-graph relate to function pointers of stream handlers in the PHP interpeter. These results emphasize the benefit of including dynamic analysis to refine the static call-graph.

While using dynamic analysis improves the result of Argus, using only dynamic analysis to generate a call-graph has its own drawbacks. One such drawback is the coverage of dynamic analysis. If the dynamic analysis does not cover all possible functionality of each PHP API, it leads to missing the identification of an injection PHP API. In our evaluation of Argus, we quantified this aspect of dynamic analysis on PHP 5.6 and 8.0. During this evaluation, Argus only used dynamic traces of running PHP high quality unit tests (i.e., 70% line coverage) to generate the call-graph for the PHP interpreter. Next, Argus performs its reachability analysis to identify the injection APIs. Our experiments on PHP 5.6 showed that using only dynamic analysis leads to missing 11 and 5 APIs, which leads to insecure deserialization and XSS. A similar observation holds true for PHP 8.0, which misses 4 insecure deserialization and 7 XSS APIs. As a result, Argus uses a hybrid static-dynamic call-graph generation, since there are drawbacks in both static and dynamic call-graph generations as shown in the aforementioned analyses.

4.2.3 Validated APIs

The second set of sub-columns labeled *Validated* in Table 2 shows the number of APIs that Argus successfully validated to directly pass their input argument to *VIF*. That is, if an adversary can control input to any of these APIs, the existence of an injection vulnerability (i.e., insecure deserialization or XSS) is a certainty. Validated APIs are a strict subset of reachable APIs. The table shows that Argus was able to consistently validate around 66%, 43%, and 83% of the deserialization, output, and exec APIs, respectively. A closer look at the reachable APIs that failed the validation test shows that either the user is not in control of the input to the *VIF* or the input is sanitized. For instance, Argus detects the function `highlight_string` reaches the output *VIF* function (i.e., `php_output_write`), however, the input is sanitized by replacing `"<"` with `"<"`. As a result, the attacker's input does not cause an XSS attack and the function `highlight_string` fails the validation test. In case of deserialization, the `SplTempFileObject::__construct` opens a temporary file object that the user cannot control. As a result, an attacker cannot trick the API to open a malicious PHAR file and validation failed. Table 5 (in the Appendix) contains the complete list of deserialization APIs for PHP versions analyzed. Note that the set of APIs in version 7.2 is a strict superset of the APIs in version 5.6. The table also highlights the APIs that still show deserialization capabilities in version 8.0 by typesetting their names in bold. As shown in Table 2, all three versions of the PHP interpreter have 22 validated output APIs, which are exactly the same among all the versions.

For the set of Exec APIs, Argus correctly detected one PHP API that reaches the exec VIFs. However, the user-input does not influence the executed command (i.e., a false positive). This PHP API, named `error_log`, provides the option of sending the error logs through email using the mail functionality in the PHP interpreter. The mail functionality in PHP in turn allows users to execute OS commands by passing an extra argument. However, user-input does not influence the arguments passed to the mail function in the `error_log` API. Furthermore, compared to RIPS, Argus' set of exec APIs does not include three PHP APIs. The first API, `expect_popen`, is not packaged with PHP source-code. The `expect` extension is installed through `PECL` package management, which is not installed by default on Debian. In addition, installing the PHP interpreter using Debian's `apt` package tool, does not install `PECL` package management. As a result, Argus cannot detect a PHP API that is not installed and compiled with the PHP interpreter. The other two APIs are `w32api_invoke_function` and `w32api_register_function`, which are conditionally compiled and solely available in the PHP interpreter for the Windows OS. Since our evaluation environment relied on the Linux OS, these two APIs were not included in the compiled version of the PHP interpreter. While Argus did not detect any exec APIs beyond RIPS, Argus identified two exec APIs that are not listed in Psalm; `mail` and `mb_send_mail`.

4.2.4 Reasons for Differences

Comparing the results for PHP 5.6 with those from 7.2 reveals three additional deserialization APIs (all of which Argus validated). The reason for this increase is the addition of support for the `BMP` image format in PHP 7.2's GD standard graphics library. Specifically, the new `createimagefrombmp` and `imagebmp` functions serve as implicit (i.e., undocumented) deserialization APIs. The last implicit deserialization API missing from PHP 5.6 is the `ftp_append` API which is supported in PHP versions 7.2 and above. All deserialization APIs

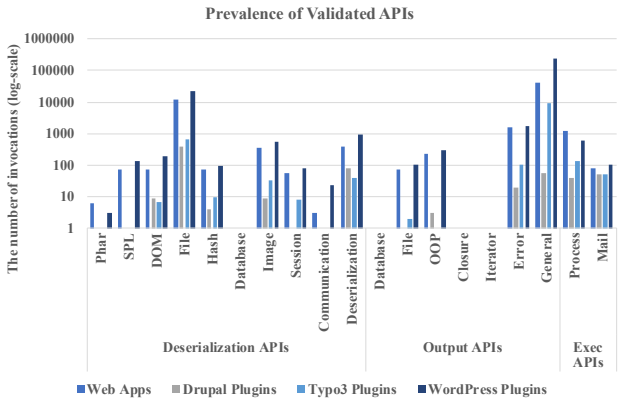

Figure 2: The prevalence of validated APIs in real-world applications.

available in version 5.6 also exist in version 7.2. In contrast to the small change of deserialization APIs between versions 5.6 and 7.2, the drop from 284 to merely 13 deserialization APIs in version 8.0 is significant. As discussed in Section 2.4, prior to version 8.0, any file operation on a phar archive results in the implicit deserialization of the archive's metadata. Fortunately, the PHP developers recognized the negative security consequences this behavior entails in 2020 and voted unanimously to change the default behavior of the phar stream wrapper [11]. Thus, since PHP 8.0 metadata in phar archives is only deserialized upon an explicit call to the `getMetadata` function in the `Phar` module, and not implicitly on any file operation on the archive. While this change certainly benefits the security of web applications, PHP 8.x is still not widely used by PHP-powered websites (less than 5% at the time of writing) [28]. The challenging process of migration prevents most web applications from easily adopting PHP 8 (see details in Section 5). Therefore, most websites still rely on older versions of the PHP interpreter that include 284 deserialization APIs.

4.2.5 Qualitative Analysis of Identified APIs

In this experiment, we assess the prevalence of deserialization and output APIs in our dataset of applications. It is crucial to investigate how many of the identified APIs are actively used in PHP applications since the validated APIs are at the core of injection vulnerabilities. For this evaluation, we grouped different categories of validated APIs listed in Table 5 for different sets of injection vulnerabilities. Figure 2 shows the number of invocation for each API category. As shown in Figure 2, in the case of deserialization APIs, we observe more usage for categories such as file operations and image processing APIs. Similarly, in the case of output APIs, the applications in our dataset often use more error handling APIs as well as general output APIs such as `echo` compared to categories such as Database, Closure, and Iterator APIs.

Furthermore, we enumerate the set of distinct applications that invoke at least one of the newly identified vulnerable APIs. For our dataset of 1,977 applications, 1,355 (i.e., 69%) and 1,218 (i.e., 62%) of applications invoke at least one newly identified deserialization and output APIs, respectively.

Finally, we looked into the pre-condition required for an attacker to exploit each of the newly detected vulnerable APIs. The first pre-condition is that the attacker needs to upload a malicious file to the server hosting the vulnerable application prior to passing malicious arguments to the vulnerable APIs. The second pre-condition is that there should not be a static prefix for file operation APIs, so that an attacker can specify `PHAR` as the stream wrapper. In the case of deserialization APIs, there are 273 APIs (i.e., 96%) that require a file upload and lack of static prefix pre-conditions prior to exploitation. Furthermore, five APIs (i.e., 23%) from the set of output APIs require the pre-condition of file upload prior to XSS exploitation. The APIs that have pre-conditions are indicated in Table 5 in the Appendix. In our evaluation, we enumerated the number of newly identified sinks that PHP applications in our dataset invoke. In the case of deserialization APIs, the most common used API in our dataset was the function `copy` which requires the pre-conditions mentioned above. However, the most common output API used in our dataset was `class_alias` API, which does not require any pre-conditions. In case of exec APIs, we did not perform any qualitative analysis, since Argus did not detect any new APIs compared to RIPS.

4.3 Extending Prior Security Analysis Tools

Argus' value arises from the comprehensive list of output, exec, and deserialization APIs it identifies within a PHP interpreter. To demonstrate the security relevance of this information, we extend two PHP security analysis systems – Psalm and RIPS, both static data flow analysis systems, and FUGIO, a dynamic automatic exploit generation system targeting POI vulnerabilities.

4.3.1 Psalm and RIPS Extension

Psalm and RIPS are two static analysis tools for PHP applications, providing taint analysis and code refactoring capabilities [37]. Taint analysis operates based on a set of configuration files that specify the taint sources and sinks in the PHP application. For our evaluation, we downloaded the latest available versions of both Psalm[1] and RIPS[2] at the time of writing from their GitHub repositories.

Psalm's taint analysis identifies exactly one PHP API function as a taint sink for insecure deserialization: `unserialize`. Furthermore, Psalm includes six functions as taint sinks for XSS vulnerabilities. Argus identified and confirmed 283 and 16 additional sinks that are missing in Psalm related to deserialization and XSS, respectively. To improve Psalm's taint analysis, we extended the set of taint sinks for

[1] Psalm 4.x-dev@832fc35d8da6e5bb60f059ebf5cb681b4ec2dba5
[2] master@ccdd2a56dbc0077cbffd08d4aa9b14af0809831d

Repo. Group	# of Apps	Deserialization				XSS				Command Inj.			
		P	P+A	R	R+A	P	P+A	R	R+A	P	P+A	R	R+A
Web Apps	60	35	354	58	511	3687	3693	538	544	25	32	14	14
Drupal plugins	521	0	0	40	47	1	1	8	8	0	0	0	0
Typo3 plugins	400	0	13	22	80	43	43	35	35	0	0	0	0
WordPress plugins	996	28	289	253	1386	1658	1667	3707	3747	4	4	4	4
Total	1977	**63**	**656**	**373**	**825**	**5,389**	**5,404**	**4,288**	**4,334**	**29**	**36**	**18**	**18**

Table 3: Extending static analysis tools such as Psalm (Labeled as P) and RIPS (Labeled as R) using Argus' results (Labeled as A) improved their detection rate.

both XSS and insecure deserialization to include the APIs Argus identified for PHP 7.2. Subsequently, we performed a comparative evaluation between upstream Psalm, and our modified version incorporating the APIs identified by Argus on the set of 1,977 PHP artifacts described in Section 4.1.

Our findings in Table 3 show a significant increase (i.e., over 10X) in the number of detected insecure deserialization vulnerabilities by the extended version of Psalm. To compare the quality of the results produced by upstream Psalm and our extended version, we manually analyzed all 656 insecure deserialization reports. As Psalm is a static analysis, we expect the results to contain false positives. Furthermore, as the extended version features 284 times as many deserialization sinks, it is unsurprising that it reports 10 times as many potential vulnerabilities. However, what we did not expect is that all 63 reports (i.e., 100%) arising from upstream Psalm are false positives. False positives can arise from web applications that sanitize inputs or, more prevalent in our POI vulnerability analysis, arise from the fact that the application sets a fixed prefix for file-paths. A "fixed" file-path-prefix, even if it is derived from an API such as `dirname` essentially thwarts any attack that relies on the phar module, as the attacker will no longer be able to specify the `phar://` prefix that triggers the stream wrapper. In order to analyze Psalm's results, we investigated the reason behind the false positives in Psalm's taint analysis. To achieve this, we randomly chose 50 reported deserialization vulnerabilities by Psalm, analyzed the report, and reviewed the source-code of the application. Our investigation shows 49 cases of false positives, where 31 false positives were reported due to over-approximation in Psalm's taint analysis as well as not detecting the sanitization process. Furthermore, 18 false positives were reported due to the fact that the pre-condition was not met. In all these cases, tainted variables had a hard-coded prefix passed to vulnerable APIs, meaning that an attacker cannot trigger the phar module by specifying the `phar://` prefix. Psalm's variable-level taint analysis only taints entire variables and hence cannot differentiate variables with a hard-coded prefix. Finally, one reported case was a true positive.

We confirmed that our extension to Psalm's taint analysis detected 12 previously unknown POI vulnerabilities (i.e., 2% true positives) in our dataset (see Table 4). We categorized the POI vulnerabilities into three groups: (i) unauthenticated, (ii) authenticated, and (iii) CSRF to Phar. The first two types are authenticated and unauthenticated Phar deserialization, which refers to the required privilege in order to exploit the POI vulnerabilities. In the case of an unauthenticated deserialization vulnerability, the attacker can reach and exploit the vulnerable functionality in the application without providing any administrator credentials for the vulnerable application. The last vulnerability type is CSRF to Phar deserialization, where a malicious actor tricks an administrator of a WordPress app into performing an action such as clicking on a link leading to Phar deserialization.

In addition, we confirmed that the extended Psalm detected one previously unknown XSS vulnerability in the core of the WordPress web application. As we will show in Section 4.3.2, FUGIO generated POC exploits for all 12 POI reports supporting the notion that these are actual vulnerabilities. As a case study, we will describe three of the vulnerabilities that we discovered among WordPress and its plugins and how Argus' comprehensive results were necessary to detect them.

In the case of exec APIs, we only extended the list of exec sinks for Psalm static analysis, as Argus only detected more exec APIs compared to Psalm. According to Table 3, we observed that Psalm detected more potential command injections compared to RIPS. In addition, Psalm+Argus detected seven more command injections compared to the unmodified Psalm. Our investigation of the newly identified vulnerabilities showed that the cause of the vulnerabilities was passing user-input to the PHP function `mail`, which was not detected by the unmodified Psalm. Furthermore, since the applications were using OOP, RIPS was unable to detect the tainted data-flow and did not detect the potential command injection vulnerabilities. However, our analysis shows that the newly identified vulnerabilities were false positive as the applications were passing user-input to the `mail` function after sufficient sanitization.

Case Study - Feed Them Social

The detected vulnerability in Feed Them Social is an unauthenticated insecure deserialization which resides in the functionality of the module's Twitter feed. The Twitter feed in this plugin retrieves and shows the content of tweets including any referenced media on a WordPress page. Whenever a tweet contains a URL, the plugin attempts to retrieve the URL's title, image, and description to display on the WordPress page. To do this, the plugin uses the function `get_meta_tags` with unsanitized user-input directly from the tweet to retrieve the metadata of the specified URL. Listing 2 shows the simplified version of this vulnerability in this plugin, where the

unsanitized user-input is passed to the implicit deserialization API `get_meta_tags` on line 4.

In order to exploit this vulnerability, an attacker sets the `fts_url` request parameter to the path of a phar file with malicious metadata. When the plugin tries to read and parse the metadata of the passed URL, it will automatically deserialize the metadata of the malicious phar file. `get_meta_tags` is an implicit deserialization API identified by Argus and not taken into consideration by prior work demonstrating the necessity of Argus' comprehensive analysis.

```
1  function fts_twitter_share_url_check() {
2   $twitter_url=$_REQUEST['fts_url'];
3   // ...
4   $tags=get_meta_tags($twitter_url);}
```

Listing 2: The feed them social plugin passes unsanitized user-input to the function get_meta_tags.

According to the history of the RIPS git repository, the latest modification to its static analysis was nine years ago [7]. A concern that is also raised by the authors of RIPS is that it does not support new features added to the PHP interpreter, such as object-oriented programming (i.e., OOP). Despite its age, Table 3 shows that the extension of RIPS (i.e., RIPS+Argus) leads to identifying more potential vulnerabilities. Further investigation into RIPS' analysis shows that it raises warnings related to the use of OOP in 1,760 applications (i.e., 89% of our dataset), which leads to false negatives. The reason behind false negatives is that RIPS [7] is not able to track tainted data (i.e., data from `$_GET` and `$_POST` parameters) to and from objects instantiated from classes in the PHP applications. In addition, due to the complex and large codebase for some applications in our dataset, RIPS was not able to complete the analysis for 135 applications (i.e., 7% of the dataset). As explained, we identified several drawbacks to the RIPS analysis that have implications for its vulnerability detection. In order to demonstrate these implications, we analyzed the results of RIPS+Argus to identify whether it was able to identify the vulnerabilities discovered by Psalm+Argus. Our analysis shows that RIPS+Argus only identified eight out of the 13 vulnerabilities (i.e., 60%) listed in Table 4.

4.3.2 FUGIO Extension

FUGIO [27] is an automatic exploit generator for previously identified deserialization vulnerabilities in PHP applications. FUGIO's exploit generation hooks a set of predefined deserialization functions while sending serialized objects as request to the web application under test. Our analysis of FUGIO shows that FUGIO hooks into 26 file operation functions in the PHP interpreter as well as the `unserialize` function to intercept deserialization of user-input. Similar to Psalm, FUGIO obtained the list of hooked functions through manual analysis of PHP documentation and prior works such as Thomas [36]. For our evaluation, we downloaded FUGIO from its GitHub repository at `https://www.github.com/WSP-LAB/FUGIO`.

One should note that FUGIO states that it is not a vulnerability detection tool. Rather its core contribution is to generate exploits for *already known deserialization vulnerabilities* [27], such as those identified by Psalm. As a result, we evaluated FUGIO on the 12 vulnerabilities that our extended version of Psalm detected. To extend FUGIO, we modified its source code to hook the comprehensive set of deserialization API functions identified by Argus. The last two columns in Table 4 show the results of extending FUGIO using Argus when generating exploits for the discovered vulnerabilities by Psalm+Argus.

As a dynamic analysis system, FUGIO requires a runtime environment. To this end, we created an experimental environment for WordPress plugins consisting of Nginx, PHP 7.2, MySQL 8, and WordPress 5.4. FUGIO creates attacks by stitching together so-called gadgets into a POP-chain. However, WordPress alone does not contain any gadgets that could be used for remote code execution attacks. In practice, administrators customize their WordPress installations using plugins and themes. Thus to ensure that FUGIO has gadgets to work with, we installed the latest versions of the top ten most popular plugins in WordPress in our experimental environment [39]. During our experiment, FUGIO without Argus' results does not hook into the image functions listed in Table 5. As a result, FUGIO was unable to generate an exploit for two of the discovered vulnerabilities in Table 4. However, the extended FUGIO+Argus successfully generated exploits for *all* the discovered vulnerabilities listed in Table 4. On this small sample, this indicates the comprehensive set of sinks provided by Argus leads to a 20% increase in the number of generated exploits.

Web App	Plugin	Vuln. Type	CVE	Function	P	P+A	R	R+A	F	F+A
Xoops	-	1	-	*imagecreatefrombmp*	✗	✓	✗	✗	✗	✓
WordPress	Feed them Social	1	CVE-2022-2437	*get_meta_tags*	✗	✓	✗	✓	✗	✓
	ImageMagick	2	CVE-2022-2441	*is_executable*	✗	✓	✗	✓	✓	✓
	String locator	2	CVE-2022-2434	*file_exists*	✗	✓	✗	✓	✓	✓
	Ajax load more	2	CVE-2022-2433	*file_exists*	✗	✓	✗	✓	✓	✓
	Broken link checker	3	CVE-2022-2438	*file_exists*	✗	✓	✗	✓	✓	✓
	wp editor	3	CVE-2022-2446	*is_dir*	✗	✓	✗	✗	✓	✓
	Visualizer	3	CVE-2022-2444	*fopen*	✗	✓	✗	✗	✓	✓
	Easy digital download	3	CVE-2022-2439	*file_exists*	✗	✓	✗	✓	✓	✓
	Theme Editor	3	CVE-2022-2440	*unlink*	✗	✓	✗	✓	✓	✓
	wPvivid Backup	3	CVE-2022-2442	*file_exists*	✗	✓	✗	✓	✓	✓
	Download manager	3	CVE-2022-2436	*file_exists*	✗	✓	✗	✗	✓	✓
	-	XSS	-	*readfile*	✗	✓	✗	✗	-	-
Total	-	-	-		0	13	0	8	10	12

Table 4: We verified the reports of Psalm+Argus by discovering 13 previously unkown POI and XSS vulnerabilities. The vulnerability types 1, 2, and 3 refers to Unauthenticated Phar deserialization, CSRF to Phar deserialization, and Authenticated Phar deserialization, respectively.

Disclosure. We responsibly reported all the vulnerabilities to their corresponding developer teams and notified the WordPress plugin review team of our findings. Seven teams already patched their WordPress plugins, and WordFence assigned CVE numbers to the vulnerabilities as shown in Table 4.

Artifact Availability: Argus is open-source and available at `https://github.com/BUseclab/Argus`. We provide the source-code of our tool along with the instructions for reproducing the experiments. These artifacts were major components of our evaluation and we believe that they can be useful for future research in this space.

5 Discussion

In this section, we discuss the limitations, challenges, and observations of Argus.

Completeness: Argus does not guarantee completeness in its analysis of the PHP interpreter as well as the identified set of deserialization, exec, and output APIs. Argus relies on the call-graph of the PHP interpreter for its analysis, which uses a hybrid static-dynamic analysis. As mentioned in Sections 2.4 and 3.1.2, the PHP interpreter extensively uses indirect calls, such as function pointers, which challenges any static analysis, including Argus. In order to minimize the drawbacks of indirect calls in the generated call-graph by Argus, we use the official unit tests of the PHP interpreter for its dynamic analysis, features a 70% line coverage over the PHP interpreter. As a result, Argus uses a hybrid static-dynamic approach to reduce the drawbacks of each technique. However, Argus cannot guarantee the completeness of its analysis due to the challenges of analyzing a complex codebase such as the PHP interpreter.

Reachability: Argus relies on a reachability analysis on the call-graph to identify the serialization, exec, and output APIs in the PHP interpreter. The reachability analysis does not reason about any sanitization or filtering the PHP interpreter might perform. Hence, the reachability of an API to VIF does not necessarily imply that an attacker can exploit the API. However, we perform a validation step to verify the output of the reachability analysis. While it seems more pertinent to perform a data-flow analysis than a reachability analysis, we argue that Argus needs to reason about the PHP interpreter and its extensions that it is linked against. Ignoring additional challenges to practicality (e.g., extensions relying on non-C code), our analysis needs to scale to millions of lines of code across PHP (one million lines of C code alone). Needless to say, resolving function pointers is still a prominent challenge for existing data-flow analysis, including the state-of-the-art SVF tool [33], which leads to imprecise control-flow graphs. As a result, we opted for a reachability analysis and subsequent validation in Argus to identify injection-sinks in the PHP interpreter.

Validation: During the validation step, Argus determines whether user-input gets passed to the VIF function inside the PHP interpeter (i.e., `php_var_unserialize` and `php_output_write`) unmodified. The presence of sanitization logic for a specific API does not necessarily mean the API cannot be exploited by attackers. Saner [2] demonstrates that sanitization logic might be implemented incorrectly. In this paper, Argus only reports the set of APIs that pass arguments unmodified to a VIF, which means that Argus' results are a lower bound of vulnerable APIs. Analyzing the correctness of sanitization logic is an orthogonal research challenge, which we consider outside the scope of this paper.

VIF identification: The foundation of Argus' analysis is based on our key observation that an underlying function is responsible for performing the action of either deserializing user-input or writing user-input to output buffers (i.e., the HTML response). In the case of XSS and insecure deserialization, there is one VIF for Argus to start the reachability analysis from. However, other types of injection vulnerabilities, might require the identification of multiple VIFs. In the case of command injection, there are eight VIFs which we could directly obtain from Saphire. Similarly, supporting SQL injection would require the identification of multiple VIFs. The reason is that the PHP interpreter supports a variety of database engines (e.g., SQLite, MySQL, Oracle, etc.) through individual extensions which can communicate SQL statements to the respective back-end. Owing to this diversity, the SQL injection VIFs are located in different database extensions and require individual identification. However, once a VIF for a given database extension is identified, Argus can immediately identify the set of (SQL injection) vulnerable API functions for the corresponding database engine.

The efforts and time required by analysts to identify the set of VIFs for each vulnerability vary, depending on the type of vulnerability. This process starts by identifying the cause of the vulnerability using analysis tools (e.g., command injection and XSS) or manual inspection of the code. The manual inspection contains reasoning about the cause of the vulnerability (e.g., the serialization format) and detecting the parser function inside the PHP interpreter that uses the serialization format. In the case of XSS, we use analysis tools to understand how the PHP interpreter prints user-input to the output buffer. To this end, we inspected the sequence of function calls in the PHP interpreter that involve printing to the output buffer. On average, it took less than 10 hours to analyze the PHP interpreter to identify the process of deserializing and printing user-input. For command injection vulnerabilities, any API that can invoke the `execve` system call is a potential exec API. Considering that, Argus uses prior research, Saphire [3] to analyze and enumerate the set of VIFs that invoke `execve` system call. For this analysis, we spent less than four hours preparing Saphire's environment and running its analysis, as well as inspecting the source code of PHP to identify the mechanism of command injection.

Precondition: Furthermore, our evaluation identified two sets of injection-sink APIs for PHP: 1) APIs that operates directly on the value of their arguments and 2) APIs that operate on malicious files. As mentioned in Section 2.4, the phar stream wrapper in the PHP interpreter only operates on local phar files. As a result, to exploit any APIs in the latter category, the attacker needs to upload the phar file prior to invoking the insecure deserialization. Therefore, in order to confirm the detected vulnerabilities, we made the assumption that the attacker had already uploaded the malicious phar file to the web application's server. We argue that this assumption is realistic since there are a plethora of approaches where an attacker can upload malicious phar files, which include exploiting arbitrary file upload vulnerabilities [16, 17]. Furthermore, web applications and their plugins provide upload functionality for many purposes, such as uploading plugins, profile pictures, and PDF files, which an attacker can exploit.

Finally, as our evaluation demonstrates, the PHP developers noticed the security consequences of automatic deserialization of phar files and fixed this issue in PHP 8.0 (released in November 2020). However, the PHP usage statistics indicate that, at the time of writing, only 10.75% of all websites that rely on PHP actually operate on PHP 8.0 [28]. The reason for this low adoption rate is probably that transitioning to PHP 8.0 is a non-trivial procedure for most PHP-powered websites. The major changes in the PHP interpreter 8.0 compared to previous versions lead to backward incompatiblilities [13] which can potentially cause fatal errors in the web applications. The challenge of (in-)compatibility is evidenced by the most popular PHP application – WordPress. Although efforts within the WordPress project to support PHP version 8.0 began on December 2020, WordPress still warns users that even its latest stable version (released in May 2023) is not fully compatible with version 8 yet [40]. While the PHP interpreter has addressed the threat arising from the automatic deserialization of phar files in version 8, history suggests that web sites relying on older versions of PHP are likely to remain publicly accessible on the Internet for the foreseeable future. These will continue to include the over 280 vulnerable deserialization APIs provided by their PHP runtimes.

6 Related Work

In this section, we review the related literature on detecting security vulnerabilities or defending against malicious behavior.

Deserialization in PHP Application: In light of new attack scenarios introduced by Esser, new research has emerged on detecting deserialization vulnerabilities and detecting such attacks on PHP applications. RIPS [7] performs an intra-procedural data flow analysis to detect injection vulnerabilities, including POI. Dahse [9] proposed an automatic approach to identify gadget chains to exploit POI vulnerabilities. Furthermore, FUGIO [27] introduced an automatic exploit generation tool to create exploit objects for POI vulnerabilities. In an orthogonal and complementary direction, our work detects the set of PHP API functions that lead to insecure deserialization, command injection, or XSS. Crucially, prior works rely on an exclusively manually curated list of sinks for taint analysis or exploit generation tools. Unlike prior work, Argus performs an automatic analysis to identify the set of PHP API functions that lead to injection vulnerabilities such as insecure deserialization. In our evaluation, we showed how our results directly improved prior work in detecting previously unknown vulnerabilities.

Deserialization on Other Platforms: Deserialization vulnerabilities threaten various platforms such as Java, Python, and .NET. The research in this area focuses on detecting such vulnerabilities or defending against deserialization attacks. SerialDetector [30] leverages call-graph analysis to identify injection vulnerabilities in .NET libraries. The key difference between SerialDetector and Argus is that we aim to detect functions at the PHP interpreter level, whereas SerialDetector finds new object injection patterns at the library level (i.e., as part of the web application rather than the application framework). Tanaka presents attacking patterns in Python's `Pickle` library, which lead to denial of service (DoS) attack [34]. Look-ahead object input stream (LAOIS) is a defense mechanism against Java deserialization vulnerabilities, allowing the type check of the serialized stream before deserialization, as implemented in Apache's Common IO library [5] and Java Serialization Filtering [26].

Other Vulnerabilities: There are multiple studies on detecting vulnerabilities in PHP applications. Several approaches rely on taint analysis to track unsanitized data and detect injection vulnerabilities [1, 7, 8, 21, 32, 38, 42]. Dynamic analysis and hybrid techniques also play an important role in detection and defense systems [3, 14, 15, 18, 25, 29, 31]. Prior works exclusively analyze the web application code and many rely on hand-crafted list of sinks. Argus analyzes the underlying PHP interpreter and generates these lists in a principal manner which could improve existing systems, as demonstrated in our evaluation. Compared to defense mechanisms, Argus takes a more proactive approach in order to detect injection vulnerabilities rather than defend against such POI attacks.

7 Conclusion

In this paper, we proposed Argus, an automated static-dynamic analysis approach to identify the set of PHP API functions that deserialize, execute, or output their arguments in a PHP application. Argus statically analyzes the PHP interpreter and its modules to generate a call-graph. Next, we refine the statically generated call-graph by using the recorded dynamic trace of the publicly available unit test of the PHP interpreter. Argus then iterates over the call-graph and identifies a comprehensive set of PHP APIs that can invoke the internal deserialization, execute OS command, or output functions. In our experiments on three of the most popular versions of the PHP interpreter, we discovered more than 300 functions that can deserialize user-input, execute OS command, or write user-input to an output buffer, expanding prior knowledge by an order of magnitude. We draw attention toward the fact that prior works rely on a purely ad-hoc curated list of functions for their static or dynamic analysis, whereas Argus automatically generates a comprehensive list. In addition, we demonstrate that Argus' findings are highly security relevant. Our findings show that, extending Psalm by Argus' results, we detected 13 previously unknown XSS and deserialization vulnerabilities in PHP applications.

Acknowledgements

We thank our anonymous shepherd and the reviewers for their helpful feedback. This work was supported by the National Science Foundation (NSF) under grant CNS-2211576.

References

[1] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *IEEE European symposium on Security and Privacy*, 2017.

[2] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, 2008.

[3] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing PHP applications with tailored system call allowlists. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[4] G. Cleary, M. Corpin, O. Cox, H. Lau, B. Nahorney, D. O'Brien, B. O'Gorman, J. Power, S. Wallace, P. Wood, and Wueest C. Internet security threat report. Technical Report 23, Symantec Corporation, 2018.

[5] Apache Commons. ValidatingObjectInputStream. https://github.com/apache/commons-oi, 2021.

[6] copernica. A c++ library for developing PHP extension. http://www.php-cpp.com/documentation/, 2022.

[7] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *Network and Distributed Systems Security Symposium*, 2014.

[8] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[9] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[10] edgescan Corporation. 2022 vulnerability statistics report. Technical Report 7, edgescan Corporation, 2022.

[11] The PHP Group. PHP:rfc phar stop autoloading metadata. https://wiki.php.net/rfc/phar_stop_autoloading_metadata, 2020.

[12] The PHP Group. PHP: PHP Manual. https://www.php.net/manual/en/index.php, 2022.

[13] The PHP Group. PHP:The Backward Incompatible Changes. https://www.php.net/manual/en/migration80.incompatible.php, 2022.

[14] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 2008.

[15] Byron Hawkins and Brian Demsky. Zenids: Introspective intrusion detection for php applications. In *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[16] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2019.

[17] Jin Huang, Junjie Zhang, Jialun Liu, Chuang Li, and Rui Dai. Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.

[18] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[19] Joern. The Bug Hunter's Workbench. https://joern.io, 2023.

[20] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[21] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2006.

[22] Michael Kerrisk. The Linux Programming Interface. https://www.man7.org/linux/man-pages/man3/exec.3.html, 2022.

[23] Namhyung Kim. Function graph tracer for c/c++/rust. https://github.com/namhyung/uftrace, 2022.

[24] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. Objectmap: Detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*, 2019.

[25] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[26] OpenJDK. Jep 290: Filter incoming serialization data. https://openjdk.org/jeps/290, 2021.

[27] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. FUGIO: Automatic exploit generation for PHP object

injection vulnerabilities. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[28] Q-Success. Usage Statistics and Market Share of PHP for Websites. https://w3techs.com/technologies/details/pl-php, 2022.

[29] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[30] Mikhail Shcherbakov and Musard Balliu. Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Network and Distributed Systems Security Symposium*, 2021.

[31] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, 2013.

[32] Sooel Son and Vitaly Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for ecurity*, 2011.

[33] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, 2016.

[34] Kousei Tanaka and Taiichi Saito. Python deserialization denial of services attacks and their mitigations. In *International Conference on Computational Science/Intelligence & Applied Informatics*, 2018.

[35] GCC team. Code Gen Options - using the GNU Compiler Collection. https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/Code-Gen-Options.html, 2022.

[36] Sam Thomas. File Operation Induced Unserialziation via the phar Stream Wrapper. In *21st Blackhat - USA*, 2018.

[37] Vimeo. Psalm - a static analysis tool for PHP. https://psalm.dev, 2021.

[38] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[39] WordPress. Plugins Categorized as Popular. https://wordpress.org/plugins/browse/popular/, 2022.

[40] WordPress. Server Environment: Make WordPress Hosting. https://make.wordpress.org/core/handbook/references/php-compatibility-and-wordpress-versions/, 2022.

[41] Zephir. Building php extensions with zephir. https://docs.zephir-lang.com/, 2022.

[42] Y. Zheng and X. Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *35th International Conference on Software Engineering*, 2013.

A PHP Object Injection

```php
// PART ONE: modify properties
class Exec {
   private $_cmd = "cat secret"; }
class Example {
   protected $obj;
   function __construct() {
      $this->obj = new Exec; } }
print urlencode(serialized(new Example));
// PART TWO: create Phar file
$phar = new Phar('exploit.phar');
$phar->startBuffering();
$phar->setMetadata(new Example());
$phar->stopBuffering();
```

Listing 3: Adversary can exploit file operations by generating a malicious phar file.

B Validation Process

```php
$pre_code = "code snippet of the exploit";
$payloads = array
    ("phar"=>"path-to-phar-file","direct
    "=>"serialized_data",...); // different
     pattern of input to deserialize APIs
$list_funcs = []
    // the list of functions to be validated
foreach($list_funcs as $func) {
 // generate the phar file
 if $func {
  $ref = new ReflectionFunction($func);
  // get the list of params using reflection
  foeach($payloads as $key => $payload) {
  $snippet
       = "..."; // invoke $func with $payload
  file_put_content
       ("tmp.php", $pre_code . $snippet) }
  $cmd
       = $PHP_BINARY." tmp.php 2> /dev/null";
  $res = shell_exec($cmd);
  // checking the result.
  if (strpos($res, "SUCCESS") !== false) {
   echo $func. " is vulnerable\n";
   break } } }
```

Listing 4: Psuedo-code of the validation process in Argus

Deserialization API	
Category	**PHP API functions**
Phar†	*phar::__construct phar::unlinkArchive*, *phar::loadPhar*, *phar::setAlias*, *phar::delete*, *phar::offsetSet*, *phar::setSignatureAlgorithm*, *phar::isValidPharFilename*, *phar::buildFromIterator*, *phar::setDefaultStub*, *phar::mount*, *phar::getType*, *phar::covertToExecutable*, *phar::offsetUnset*, *phar::stopBuffering*, *phar:getATime*, *phar::setStub*, *phar::isLink*, *phar::addFromString*, *phar::isFile*, *phar::addFile*, *phar::compress*, *phar::extractTo*, *phar::hasChildren*, *phar::getInode*, *phar:getFileInfo*, *phar::decompressFiles*, *phar::mapPhar*, *phar:isReadable*, *phar::addEmptyDir*, *phar::compressFiles*, *phar:getOwner*, *phar:getGroup*, *phar::offsetGet*, *phar::setMetadata*, *phar:getPerms*, *phar::isExecutable*, *phar::loadPhar*, *phar::copy*, *phar::convertToData*, *phar::isWritable*, *phar:getSize*, *phar:getCTime*, *phar:getMTime*, *phar:isDir*, *phar::getStub*, *Phar::delMetadata*, *PharFileInfo::__construct*, *PharFileInfo::chmod*, *PharFileInfo::getContent*, *PharFileInfo::getType*, *PharFileInfo::isReadable*, *PharFileInfo::isDir*, *PharFileInfo::isWritable*, *PharFileInfo::openFile*, *PharFileInfo::decompress*, *PharFileInfo::compress*, *PharFileInfo::getInode*, *PharFileInfo::getCTime*, *PharFileInfo::getMTime*, *PharFileInfo::getSize*, *PharFileInfo::isExecutable*, *PharFileInfo::isLink*, *PharFileInfo::isFile*, *PharFileInfo::getATime*, *PharFileInfo::getGroup*, *PharFileInfo::getPerms*, *PharFileInfo::getOwner*, *PharFileInfo::getFileInfo*, *PharFileInfo::setMetadata*, *PharFileInfo::delMetadata*, *PharData::unlinkArchive*, *PharData::loadPhar*, ***phar::getMetadata***, ***PharFileInfo::getMetadata***,
SPL	*FileInfo::openFile*†, *FileInfo::getCTime*†, *FileInfo::getSize*†, *FileInfo::getATime*†, *FileInfo::getFileInfo*†, *FileInfo::getGroup*†, *FileInfo::getType*†, *FileInfo::getPerms*†, *FileInfo::getOwner*†, *FileInfo::isWritable*†, *FileInfo::isDir*†, *FileInfo::getMTime*†, *FileInfo::isReadable*†, *FileInfo::getInode*†, *FileInfo::isExecutable*†, *FileInfo::isFile*†, *FileInfo::isLink*†, *SplFileObject::__construct*†, *SplFileObject::getType*†, *SplFileObject::isReadable*†, *SplFileObject::isDir*†, *SplFileObject::openFile*†, *SplFileObject::getInode*†, *SplFileObject::isWritable*†, *SplFileObject::getFileInfo*†, *SplFileObject::getCTime*†, *SplFileObject::getPerms*†, *SplFileObject::getOwner*†, *SplFileObject::getGroup*†, *SplFileObject::getATime*†, *SplFileObject::getGroup*†, *SplFileObject::isExecutable*†, *SplFileObject::isFile*†, *DirectoryIterator::__construct*†, *DirectoryIterator::getType*†, *DirectoryIterator::isReadable*†, *DirectoryIterator::isDir*†, *DirectoryIterator::openFile*†, *DirectoryIterator::getInode*†, *DirectoryIterator::isWritable*†, *DirectoryIterator::getFileInfo*†, *DirectoryIterator::getATime*†, *DirectoryIterator::getCTime*†, *DirectoryIterator::getPerms*†, *DirectoryIterator::getOwner*†, *DirectoryIterator::getGroup*†, *DirectoryIterator::isLink*†, *DirectoryIterator::isFile*†, *DirectoryIterator::isExecutable*†, *RecursiveDirectoryIterator::__construct*†, *RecursiveDirectoryIterator::getType*†, *RecursiveDirectoryIterator::isReadable*†, *RecursiveDirectoryIterator::isDir*†, *RecursiveDirectoryIterator::openFile*†, *RecursiveDirectoryIterator::getInode*†, *RecursiveDirectoryIterator::isWritable*†, *RecursiveDirectoryIterator::getFileInfo*†, *RecursiveDirectoryIterator::getCTime*†, *RecursiveDirectoryIterator::getPerms*†, *RecursiveDirectoryIterator::getOwner*†, *RecursiveDirectoryIterator::getGroup*†, *RecursiveDirectoryIterator::isLink*†, *RecursiveDirectoryIterator::current*†, *RecursiveDirectoryIterator::isFile*†, *RecursiveDirectoryIterator::isExecutable*†, *RecursiveDirectoryIterator::hasChildren*†, *FileSystemIterator::__construct*†, *FileSystemIterator::getType*†, *FileSystemIterator::isReadable*†, *FileSystemIterator::isDir*†, *FileSystemIterator::openFile*†, *FileSystemIterator::getInode*†, *FileSystemIterator::isWritable*†, *FileSystemIterator::getFileInfo*†, *FileSystemIterator::getPerms*†, *FileSystemIterator::getOwner*†, *FileSystemIterator::getGroup*†, *FileSystemIterator::getATime*†, *FileSystemIterator::current*†, *FileSystemIterator::getSize*†, *FileSystemIterator::isLink*†, *FileSystemIterator::getMTime*†, *FileSystemIterator::isExecutable*†, *FileSystemIterator::isFile*†, ***SplQueue::unserialize***, ***SplStack::unserialize***, ***SplDoublyLinkedList::unserialize***, ***ArrayIterator::unserialize***, ***RecursiveArrayIteratorunserialize***, ***SplObjectStorage::unserialize***, ***ArrayObject::__unserialize***
DOM & XML†	*DOMDocument::loadHTMLFile*, *DOM::C14NFile*, *DOMDocument::load*, *DOMDocument::loadXML*, *DOMDocument:saveHTMLFile*, *DOMDocument:relaxNGValidate*, *DOMDocument:validate*, *DOMDocument:save*, *xmlwrite_open_uri*, *xmlreader::open*, *SimpleXMLElement::__construct*, *simplexml_load_file*, *simplexml_load_string*
File Operation†	*get_meta_tags*, *is_dir*, *scandir*, *is_writable*, *is_file*, *opendir*, *file*, *move_uploaded_file*, *rmdir*, *fileowner*, *touch*, *gzfile*, *file_get_contents*, *mkdir*, *finfo_file*, *fileatime*, *bzopen*, *fileperms proc_open*, *readgzfile*, *is_link*, *file_put_contents*, *finfo_buffer*, *gzopen*, *getdir*, *unlink*, *is_readable*, *filegroup*, *finfo_open*, *filectime*, *filemtime*, *rename*, *fileinode*, *copy*, *filesize*, *mime_content_type*, *stat*, *filetype*, *fopen*,*readfile*,*file_exists*, *is_executable*
Hash†	*md5_file*, *hash_hmac_file*, *sha1_file*, *hash_file*
DataBase†	*PDO::pgsqlCopyFromFile*, *PDO:pgsqlCopyToFile*, *pg_trace*
Image Processing†	*imageloadfont*, *exifimagetype*, *exif_read_data*, *read_exif_data*, *exif_thumbnail*, *getimagesize*, *imagecreatefromjpeg*, *imagecreatefrompng*, *imagecreatefromgd2*,*imagecreatefromgif*, *imagecreatefromwebp*, *imagecreatefromgd*, *imagecreatefromxbm*, *imagecreatefrombmp*, *imagecreatefromwbmp*, *imagecreatefromavif*, *imagejpeg*, *imagepng*, *imagegif*, *imagegd*, *imagegd2*, *imageavif*, *imagebmp*, *imagewbmp*,*imagexbm*, *imagewebp*
Session Function	***session_decode***, ***session_start***
Communication	*ftp_nb_put*†, *ftp_nb_get*†, *ftp_get*†, *ftp_append*†, *ftp_put*†, ***msg_recieve***
Deserialization	***unserialize***
Output API	
Database	*pg_loreadall*, *pg_lo_read_all*, *odbc_result_all*
File Operation†	*fpassthru*, *readfile*, *readgzfile*, *gzpassthru*, *SplFileObject::fpassthru*
OOP	*class_alias*
Closures	*Closure::bind*, *Closure::bindTo*
Iterators	*CachingIterator::offsetGet*, *RecursiveCachingIterator::offsetGet*
Error Handling	*trigger_error*, *user_error*, *die*, *exit*
General	*echo*, *print*, *print_r*, *vprintf*
Exec API	
Mail	*mail, mb_send_mail*
Process	*system, shell_exec, exec, proc_open, popen, pcntl_exec, passthru*

Table 5: The categories of exec, output and deserialization API. The functions or category of functions specified by † require the precondition of uploading a malicious file prior to exploitation. The functions specified in **bold** are the set of vulnerable deserialization APIs in PHP 8.