Poster: Whether We Are Good Enough to Detect Server-Side Request Forgeries in PHP-native Applications?

Yuchen Ji
ShanghaiTech University
Shanghai, China
jiych2022@shanghaitech.edu.cn

Ting Dai
IBM Research
Yorktown Height, USA
ting.dai@ibm.com

Yutian Tang
University of Glasgow
Glasgow, United Kingdom
yutian.tang@glasgow.ac.uk

Jingzhu He*
ShanghaiTech University
Shanghai, China
hejzh1@shanghaitech.edu.cn

Abstract

Server-side request forgeries (SSRFs) are inevitable in PHP web applications. Existing static taint analysis tools for PHP suffer from both high rates of false positives and false negatives in detecting SSRF because they do not incorporate application-specific sources and sinks, account for PHP's dynamic type characteristics, and include SSRF-specific taint analysis rules, leading to over-tainting and under-tainting. In this work, we propose a technique to accurately detect SSRF vulnerabilities in PHP web applications. First, we extract both PHP built-in and application-specific functions as candidate source and sink functions. Second, we extract explicit and implicit function calls to construct applications' call graphs. Third, we perform a taint analysis based on a set of rules that prevent over-tainting and under-tainting. We have implemented a prototype and evaluated it with different types of PHP web applications. Our preliminary experiment shows that we detect 24 SSRF vulnerabilities in 13 different types of applications. 20 of the vulnerabilities are known and 4 of the vulnerabilities are new.

CCS Concepts

• **Security and privacy → Web application security**.

Keywords

PHP; Server-Side Request Forgery; Taint Analysis

ACM Reference Format:
Yuchen Ji, Ting Dai, Yutian Tang, and Jingzhu He. 2024. Poster: Whether We Are Good Enough to Detect Server-Side Request Forgeries in PHP-native Applications?. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3658644.3691419

*Jingzhu He is the corresponding author.

1 Introduction

PHP is the dominating programming language to build web applications [9]. PHP web applications allow developers to use server-side requests to interact with third-party applications [13]. Attackers manipulate the user inputs to send forged server-side requests, making server-side request forgery (SSRF) vulnerabilities inevitable. Exploitation of SSRF vulnerabilities often causes severe consequences to the applications, such as denial of service (DoS), leakage of sensitive data, and remote code execution [13]. In 2019, the exploitation of an SSRF vulnerability in Capital One's service caused the leakage of credit card information of more than 100 million consumers [1]. Since 2021, SSRFs have been ranked as the top 10 vulnerabilities by OWASP, based on the occurrence, impacts, incident rates, number of CVEs and other factors [2].

Static analysis tools are widely adopted by developers to detect SSRFs. Existing taint-analysis-based static detectors such as Rips [11] and TChecker [12] track the flow of (untrusted) user input, originating from source functions, and detect whether the tainted data reaches sink functions to send forged server-side requests. However, they all have limitations.

First, those tools only consider PHP built-in sources or sinks. In our preliminary study, we find that 46% of applications use third-party APIs to handle user input and send server-side requests. Without taking into consideration application-specific sources and sinks, existing tools suffer from false negatives.

Second, existing static tools fail to accurately construct call graphs with PHP's dynamic type features. They do not support implicit call flows such as magic methods [6], variable classes, or variable methods [10]. For example, Rips generates call graphs by matching function signatures, disregarding object-oriented methods. Thus, it overlooks method calls. TChecker generates call graphs using type inference and takes into account object-oriented methods. When it fails to infer the type of a variable, any method call on that variable is ignored. In addition, magic methods are ignored.

Third, existing tools contain both over-tainting and under-tainting rules. Over-tainting rules overlook data flow paths and string sanitizations, which may result in non-vulnerable code being flagged as tainted. Under-tainting rules exclude certain data structures, code blocks, and indirect paths, which can lead to missed vulnerabilities. For example, Rips recklessly marks the return value of a function as tainted if any argument is tainted, regardless of whether the argument actually impacts the return value through data flows. TChecker treats functions that are not connected in the call graph as dead code and omits their analysis. However, bypassed functions can be invoked by reflection in third-party libraries and contain vulnerabilities. Both Rips and TChecker taint a string when any of its components is tainted, even if the string is not a URL.

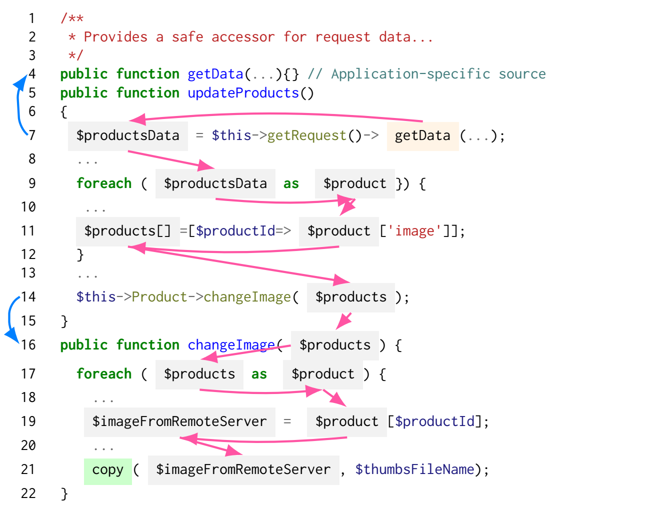
2 Methodology

To address the limitations of existing static detection tools, we propose a three-step technique to detect SSRFs in PHP applications:

Step 1. Identifying Sources and Sinks: Source and sink functions are crucial in detecting SSRFs. The source functions return user input, while the sink functions send server-side requests. Existing tools gather lists of sources and sinks from the PHP standard library. However, modern PHP applications often rely on third-party libraries to streamline or enhance functionality, providing more flexibility and abstraction from the underlying implementation by encapsulating built-in sources with input encoding, validation, and sanitization and encapsulating built-in sinks with argument validation, response parsing, and error handling, such as timeout and retry. These third-party library sources and sinks are widely used in modern PHP applications. We analyze them once offline and cache the results, allowing us to quickly retrieve data from the cache online instead of reanalyzing each time. To identify them, we extract their function signatures and corresponding PHPDoc documentation [8], then prompt a large language model (LLM) to determine if they are classified as sources or sinks.

Step 2. Constructing Call Graphs: To construct call graphs with regard to PHP's dynamic type features, it is necessary to recognize both explicit and implicit call relationships. In explicit function calls, the method name is specified, the class name can be inferred, and the function's definition is available. In implicit function calls, the definition might be absent, or the class and method names can be variables. Four cases of implicit function calls are considered. First, when both the method name and the class name are known, but there is no corresponding definition in the related class, we consider magic methods (including `__call` and `__callStatic`, which handle calls to undefined methods). Second, when a call target has a specific method name, but the class name cannot be inferred by type inference, all methods with the matching name and the same parameter count with compatible types, including magic methods, are considered. Third, method names can be variable. When a call target has a known class name but a variable method name, all methods within that class that match the parameter count and types, along with magic methods, are taken into account. Finally, for call targets with variable class names and method names, all methods, including magic methods, that have the same number of parameters and compatible types are considered.

Step 3. Performing Taint Analysis: We design taint analysis rules to mitigate the issues of over-tainting and under-tainting prevalent in existing tools. To prevent over-tainting, we apply taint-clearance rules to terminate the tracking of tainted data when it is no longer relevant or has been neutralized. For example, casting a variable to a non-string type clears its tainted state because it is no longer relevant to SSRF vulnerabilities, where tainted strings are used to craft malicious URLs. Additionally, we ensure string safety



```
/**
 * Provides a safe accessor for request data...
 */
public function getData(...){} // Application-specific source
public function updateProducts()
{
  $productsData = $this->getRequest()-> getData (...);
  ...
  foreach ( $productsData as $product ) {
   ...
   $products[] =[$productId=> $product ['image']];
  }
  ...
  $this->Product->changeImage( $products );
}
public function changeImage( $products ) {
  foreach ( $products as $product ) {
    ...
    $imageFromRemoteServer = $product [$productId];
    ...
    copy ( $imageFromRemoteServer , $thumbsFileName);
}
```

Figure 1: A new vulnerability (CVE-2023-46725) detected by our method. Taint propagates from identified source to PHP built-in sink. → represents the function call flows. → represents data flows.

by identifying whether they are complete domain names or cannot be represented as URLs, as these strings cannot be manipulated to cause SSRF vulnerabilities. To mitigate over-tainting in array operations, we implement fine-grained taint rules that track the state of array elements with concrete keys.To prevent under-tainting, we model PHP syntax constructs that impact taint propagation and account for implicit data flows created by the `extract` function. Additionally, we analyze each function within the application, since third-party libraries might call application functions via reflection. The choices of which functions to invoke depend on custom configurations and runtime user input. Consequently, it is impossible to statically ascertain that any application function is inactive code, leading us to approximate by analyzing every function within the application.

3 Preliminary Experiment

We have implemented a prototype following the methodology in Section 2 using the Phan [3] framework and GPT-4 model. We collect applications from CVE database [5], GitHub [4], and WordPress plugins repository [7]. The applications collected fall into 13 categories: asset management, bookmark-sharing, content management, customer management, E-commerce, file management, forums, library management, learning management, marketing, online programming, project management, and miscellaneous systems. We randomly select one application from each category, resulting in 13 applications for our preliminary experiment. We test our prototype on the selected applications and compare it with Rips [11] and TChecker [12]. We first run the two tools without adjustments and then incorporate the sources and sinks that we identified. As shown in Table 1, our approach detects **24** (**20** known and **4** new[1]) SSRFs,

[1]CVE-2023-5877, CVE-2023-46725, CVE-2023-46730, CVE-2023-48006

Table 1: Preliminary experiment results. $*$ denotes that the vulnerabilities cannot be detected by the tool out of the box. They are detected after applying sources and sinks identified by our method.

Application		# of Vulns		Rips		TChecker		Ours	
Category	**Name & Version**	**Known**	**New**	**Known**	**New**	**Known**	**New**	**Known**	**New**
Asset Mgmt	snipe-it v5.3.3	1	0	0	0	0	0	1	0
Bookmark	LinkAce v1.12.2	0	1	0	0	0	0	0	1
Content Mgmt	WonderCMS v3.1.3	3	0	0	0	1	0	3	0
Customer Relationship Mgmt	Group Office v6.4.196	1	1	0	0	0	0	1	1
E-Commerce	FoodCoopShop v3.6.0	0	1	0	0	0	0	0	1
File Mgmt	Responsive FileManager v9.13.1	4	0	4	0	4	0	4	0
Forum	phpbb 3.2.0	2	0	0	0	2*	0	2	0
Library Mgmt	SLiMS v9.4.2	3	0	1	0	1	0	3	0
Learning Mgmt	Chamilo 1.11.18	1	0	1	0	1	0	1	0
Marketing	affiliate-toolkit v3.2.0	0	1	0	1	0	1	0	1
Misc	rconfig v3.9.4	3	0	0	0	3	0	3	0
Programming	Codiad v1.7.8	1	0	0	0	0	0	1	0
Project Mgmt	gopeak v2.1.5	1	0	0	0	0	0	1	0
Total		20	4	6	1	12	1	**20**	**4**

while Rips detected 7 (6 known and 1 new) SSRFs, and TChecker detected 13 (12 known and 1 new) SSRFs.

Of the 24 identified vulnerabilities, 7 use sources and sinks from third-party libraries. Despite applying identified sources and sinks to Rips and TChecker, neither tool can identify all 7 vulnerabilities. Rips misses all 7 vulnerabilities due to the involvement of method calls, a feature of object-oriented programming that Rips does not support. Conversely, TChecker overlooks 5 vulnerabilities, as it considers the vulnerable methods dead code, although the methods are actually invoked through reflection during runtime.

3.1 Case Study

In Figure 1, we present a new SSRF (CVE-2023-46725) to demonstrate how our method detects this vulnerability while state-of-the-art tools fail the detection. A tainted input originates from an application-specific source function at line #7. Our LLM-based source identification module marks the `getData()` method as a source since the documentation indicates it returns request data, fitting the source function definition. The taint propagates to variables `productsData`, `product`, and `products` consecutively. When `products` is passed as an actual argument to the `changeImage()` method during its invocation at line #14, the formal argument `products` becomes tainted. In function `changeImage()`, the taint contaminates variables `product` and `imageFromRemoteServer`, and eventually flows into the sink function at line #21. Rips fails to detect this SSRF due to its lack of support for object-oriented features, thus it cannot resolve the method call to `changeImage()` at line #14. TChecker skips analyzing the `updateProducts()` function as it is considered dead code due to not being called in the application. However, `updateProducts()` is invoked in third-party libraries via reflection, determined at runtime.

4 Conclusion

In this work, we propose a technique to efficiently identify SSRFs in PHP applications by integrating precise taint analysis rules with a large language model (LLM). We implement a prototype and test it on 13 open-source PHP applications, successfully identifying **24** vulnerabilities (20 known, 4 new), outperforming existing tools. However, our technique still produces false positives caused by input validation. For example, whitelists can be used to restrict input strings to prevent SSRF. To reduce false positives caused by input validation, we plan to research how to extract constraints on user input and verify whether the constraints can be fulfilled while triggering SSRF.

Acknowledgement

This research was supported in part by the Shanghai Sailing Program 22YF1428600 and National Natural Science Foundation of China under grant 62202306.

References

[1] 2019. What We Can Learn from the Capital One Hack. https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack.

[2] 2022. OWASP Top 10 - 2021. https://owasp.org/Top10/.

[3] 2022. phan-plugin. https://github.com/wikimedia/mediawiki-tools-phan-SecurityCheckPlugin.

[4] 2023. Awesome-Selfhosted. https://github.com/awesome-selfhosted/awesome-selfhosted.

[5] 2023. CVE database. https://cve.mitre.org/index.html.

[6] 2023. Magic Methods. https://www.php.net/manual/en/language.oop5.overloading.php.

[7] 2023. Popular Plugins. https://wordpress.org/plugins/browse/popular.

[8] 2023. PSR-5: PHPDoc. https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc.md.

[9] 2023. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.

[10] 2024. Variable Functions. https://www.php.net/manual/en/functions.variable-functions.php/.

[11] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis.. In *NDSS*, Vol. 14. 23–26.

[12] Changhua Luo, Penghui Li, and Wei Meng. 2022. Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* 2175–2188.

[13] Giancarlo Pellegrino, Onur Catakoglu, Davide Balzarotti, and Christian Rossow. 2016. Uses and abuses of server-side requests. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19.* Springer, 393–414.