SSRF vs. Developers: A Study of SSRF-Defenses in PHP Applications

Malte Wessels and Simon Koch, *Technische Universität Braunschweig;*
Giancarlo Pellegrino, *CISPA Helmholtz Center for Information Security;*
Martin Johns, *Technische Universität Braunschweig*

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

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

SSRF vs. Developers: A Study of SSRF-Defenses in PHP Applications

Malte Wessels*†, Simon Koch*†, Giancarlo Pellegrino‡, Martin Johns†
† *Technische Universität Braunschweig*
‡ *CISPA Helmholtz Center for Information Security*
{malte.wessels, simon.koch, m.johns}@tu-braunschweig.de, pellegrino@cispa.de

Abstract

Server-side requests (SSR) are a potent and important tool for modern web applications, as they enable features such as link preview and web hooks. Unfortunately, naive usage of SSR opens the underlying application up to Server-Side Request Forgery – an underappreciated vulnerability risk. To shed light on this vulnerability class, we conduct an in-depth analysis of known exploitation methods as well as defenses and mitigations across PHP. We then proceed to study the prevalence of the vulnerability and defenses across 27,078 open-source PHP applications. For this we perform an initial data flow analysis, identifying attacker-controlled inputs into known SSR functions, followed up by a manual analysis of our results to gain a detailed understanding of the involved vulnerabilities and present defenses. Our results show that defenses are sparse. The hypermajority of our 237 detected data flows are vulnerable. Only two analyzed applications implement safe SSR features.

Since known defenses are not used and detected attacker-controlled flows are almost always vulnerable, we can only conclude that developers are still unaware of SSR abuses and the need to defend against them. Consequently, SSRF is a present and underappreciated danger in modern web applications.

1 Introduction

Server-side requests (SSRs) are a convenient service-to-service communication pattern in which a web service sends HTTP requests to external entities. Modern web applications use SSRs to implement many user-facing functionalities, such as URL previews [36] or integration of third-party content, such as external calendars [16]. While essential in practice, if not implemented correctly, SSRs can be abused, introducing a wide variety of security risks, ranging from attacks such as network reconnaissance [32] to high severity ones such as remote code execution (RCE) attacks.

*Both authors contributed equally to this research.

SSRF ranks among the OWASP Top 10 [31] and CWE Top 25 [40] security risks. Recent incidents suggest that web applications are still exposed to SSR vulnerabilities. For example, an attacker recently elevated an attacker-controlled SSR vulnerability affecting Microsoft Exchange servers into an RCE vulnerability, bypassing authentication and firewalls, resulting in email theft [3, 17]. Most recently, PyTorch's TorchServe suffered from an arbitrary code execution flaw caused by an SSRF flaw, which allowed the attacker to load the malicious model [4]. Even applications that attempt to defend themselves frequently get it wrong, such as vSphere, which employed URL validation but suffered from an information disclosure attack through SSRF [2].

Unfortunately, we know little about the prevalence of SSR vulnerabilities and the effectiveness of the existing defenses, as the research community has given SSRs limited attention. Up to this point, prior work has been conducted only on a small scale and was focused on either understanding and exploring the security risks (i.e., Pellegrino et al. [32]) or novel defenses (i.e., Jabiyev et al. [21]). This paper addresses this gap by providing a large-scale analysis of SSR threats in PHP web applications.

Analyzing PHP SSRF vulnerabilities, at scale, and the code patterns associated with defenses is particularly challenging. The lack of reliable, scalable and flexible investigation tools hinders such analysis because it requires collecting, analyzing, and reasoning on millions of lines of code. Tools such as PHPJoern [9], a Code Property Graph Generator, have previously been employed in large-scale analysis of PHP programs. However, our evaluation has revealed significant inaccuracies, such as an incorrect control flow graph and consequently inaccurate data flows edges, thereby increasing the likelihood of false positive results. Thus, beside our SSRF study, we also present an up-to-date version of the CPG generator for PHP web applications aiding us in our static analysis.

In this paper, we present the first static security measurement of SSR threats in PHP source code. Starting from a thorough survey of existing academic and non-academic work on SSR vulnerabilities, attacks, and defenses, we create an up-to-date taxonomy of SSR threats. Then, we use our taxonomy to study the prevalence of SSR vulnerabilities and defenses in real web applications by downloading and analyzing 30.870 popular open-source PHP web applications on GitHub.

To infer if protection mechanisms against SSR vulnerabilities are present, we study these applications with SURFER, a flexible static analysis PHP framework tailored to support exploratory analysis of PHP vulnerabilities. SURFER works on top of our novel PHP bytecode code property graph (CPG) [47]. SURFER successfully analyzed 27,078 repositories. In this set, 15.308 applications utilize API sinks, such as `file_get_contents`, that are potentially susceptible to SSR problems. For 1.040 of these applications, SURFER was able to identify at least one potentially suspicious flow into such sinks. To further narrow down the set of vulnerability candidates, we only consider cases in which a direct data flow from user-provided input into the sinks exists, and the adversary controls the significant parts of the passed URL value. After applying these processing steps, we end up with 141 PHP projects that utilize server-side request sinks in a potentially insecure manner.

To thoroughly examine the existence of potential defensive measures and their robustness, we follow up with a manual inspection of these applications. The results of this analysis paint a somber picture: More than half of the identified applications use *no* defensive measures against SSRF and, thus, are trivial to exploit. Only three projects leverage dedicated existing SSRF-secure request mechanism and only two of the remaining PHP projects deployed robust countermeasures against sophisticated attack techniques, such as using DNS rebinding. Thus, our analysis demonstrates a widespread ignorance by developers in respect to SSRF vulnerabilities.

In summary, this paper makes the following contributions:

- A survey and usage study of existing open source SSR abuse mitigation techniques in PHP.
- A PHP Code Property Graph generator based on the modern CPG framework.
- A CPG based static analysis tool chain (SURFER) to identify SSRF vulnerable code.
- A large scale study of 27,078 PHP projects for SSRF vulnerabilities and mitigation techniques.

Organization of the paper We first discuss the difference between SSR and SSRF as well as the challenges of defending against it, deriving our research questions (2). After having established the required background, we detail the current state of SSRF mitigation techniques in popular open-source frameworks (3). We then lay the groundwork for our tooling and present SURFER, as well as our Manual Analysis to perform our large-scale static analysis study (4). The results of our analysis are given next (5), followed by a discussion of

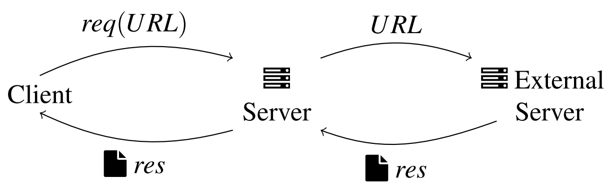


Figure 1: A Server-Side Request.

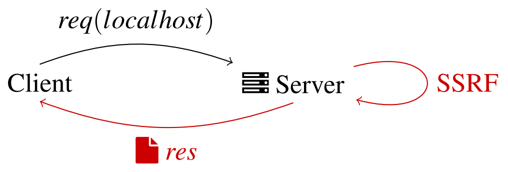


Figure 2: An SSRF attack that accesses local resources.

the implications, including a case study and the answer to our research questions (6). Finally, we provide an overview of related work (7) and conclude with a summary of our key takeaways (8).

2 A Primer on SSRF

We briefly introduced the SSR feature and its evil variation *SSRF*. In this section we are providing a primer on why server-side requests exist (2.1) and how they can turn into SSRF (2.2) arriving at our research questions (2.3).

2.1 Server-Side Requests

An application requires a server-side request as soon as it needs information that is not stored within the application components itself. Common examples are webhooks, previews of links, and interaction with external APIs like payment or authentication providers.

The chain of events leading to a server-side request starts with a (user) request for an application resource, depending on remote data. For example, if the application wants to display a link preview, it must perform a request and retrieve the data required for the preview to answer the client's request. Figure 1 visualizes this event chain.

User-input-controlled SSRs are allowed for several reasons. For instance, webhooks and link previews target user-provided URLs by design. Other SSR use cases allow the user to affect parts of the target URL, such as URL parameters.

2.2 SSRF: Server-Side Requests Going Rogue

Suppose an SSR request can be influenced by user input beyond the scope envisioned by the developer. In that case, attackers can leverage this as an attack vector and guide the request to malicious hosts and services. The developer is responsible for ensuring that user-provided data cannot unintentionally influence a request, e.g., by changing the target to network-internal resources. How an attacker can exploit their ability to influence the request depends on the deployment context, including the network topology, the host machine's configuration, and the influence's scope.

If a server with a vulnerable application runs localhost-accessible services, an attacker can use the SSRF to access normally denied resources. Figure 2 visualizes such an attack. As in our introductory examples, this enables an attacker to conduct network reconnaissance, secret stealing, or even RCE. Cloud services often serve HTTP APIs on the local network, providing configuration and metadata [7, 12, 21]. An SSRF vulnerability exposes this internal API to attackers.

Even if all local resources are adequately protected, an attacker can still abuse an SSRF vulnerability. Vulnerable public access points can be attacked via SSR requests from SSRF-vulnerable servers. As the vulnerable server performs the request, the attacker hides their identity, impeding investigations into attacks or laundering the presumed host to an unsuspecting third party.

Sometimes, vulnerabilities such as SSR occur in authenticated areas of an application. They should not be ignored. Firstly, not all authenticated users should possess power over the whole system, e.g., regular users vs. admins. Secondly, superuser rights inside an application, such as admin rights in a CMS, do not equal to any rights on the host OS. This applies especially in managed hosting environments, where users might have superuser rights inside their application but not on the underlying machine. However, in both cases, a malicious user or web admin could exploit an SSRF vulnerability to gain access to sections of the application they are not authenticated for or the underlying host OS.

2.3 Research Questions

SSRF poses a risk to anybody using SSR features, but the state of SSRF is an orphaned domain in current research. To address this, we formulate four research questions that each provide a distinct insight:

RQ1: What is the current state-of-the-art of SSRF defenses?
RQ2: Are web application developers using existing SSR mitigations?
RQ3: Are web application developers using homegrown SSR mitigations?
RQ4: How many web applications are prone to SSR abuses?

3 Survey Of Attacks and Defenses

SSRF is a multifaceted vulnerability class that allows for varied exploitation options. The same holds for defending against SSRF abuses. This warrants a detailed discussion of known attacks and defenses according to the literature to frame our search later on.

We start our systematization from Pellegrino et al. [32]'s work on SSR attacks and amend it with results from an academic literature survey. Additionally, we explored non-academic sources systematically by searching the web for 'SSRF' with keywords such as 'Defense' and 'best practice'. This resulted in the non-academic sources [8, 18, 25, 26, 30, 39, 44] as well as the academic sources [21, 28, 32]. A systemized overview of our results on SSR attacks and application-level defenses is given in Table 1. Table 2 augments it with details about possible defense evasions and the respective fixes.

3.1 Attacks

Across the literature, we identified six distinct classes of attacks that leverage SSRF vulnerabilities to perform malicious and unintended activities:

(A1) Recon Attack: The first class of attack tries to gain information about a server's network using the SSRF vulnerability to reach behind the firewall and gain access to network internals. An attacker can identify deployed services and available machines using return values or timing-based side channels.

(A2) Origin Laundering: In the second class of attack, the attacker uses the SSRF vulnerability to misuse the server as a proxy to serve malicious data from another website. This can be used to circumvent block lists implemented by the browser. An SSRF-vulnerable website can be misused as a proxy serving otherwise blocked malicious content.

(A3) Denial of Service: The next attack category is Denial of Service (DoS) attacks, usually split into three distinct subtypes across the literature.

Consider an SSR service using a GET parameter as input and reflecting the SSR's response. An attacker can craft a URL directing the service to a domain hosting illegal or known malicious content. They then provide this prepared link to a web scanner. When the scanner requests the prepared link, the SSR service requests the embedded target and mirrors the content. The scanner then flags the SSR service due to malicious content. Consequently, an attacker can use an SSRF vulnerability to put the victim server on block lists and thus achieve a denial of service. Such a DoS is also possible on a non-technical level. For example, if the prepared link is reported to the authorities, the SSR provider could face legal action.

The remaining two kinds leverage amplification of the request. In case the SSRF does not trigger a single but multiple requests to a target, the vulnerable service can be abused as

Attacks		D1 URL Valid.				D2 DNS Valid.	D3 Secure Conf.		D4 Response Modification				
		Scheme	Domain	Port	Path	IP input validation	HTTP(s) only	No OOP	wrap result	C–D header	rate limit	fixed RT	
A1 Recon attack	Port scan			●								○	
	Network scan		△			●						○	[21, 30, 31, 32, 49]
A2 Origin Laundering			△			△		●	●	●			[32]
A3 DoS attacks	by blocklist		△		●	△							[30]
	attack ES					△					○		[32]
	attack Ampl.					△					○		[32]
A4 Bridging Attack		●					●						[32, 44]
A5 Exploiting SSBs			△			△							[28]
A6: Local Res. Leak			△		○	△							[30, 39]

Table 1: Overview over SSR attacks and application-level defenses. ●: Defense successfully prevents the attack. ○: Defense mitigates some version of the attack. △: Defense works in an allow list scenario.

	Evasion Technique	Evaded Def. Technique	Fix	
D1	URL parser confusion attacks	insufficient parsers	well-established, hardened parser	[8, 21, 29, 31, 39, 44]
D2	DNS rebinding	IP validation	IP pinning	[8, 21, 31]
D3	redirects	singular checks	rechecking on each redirect or disabling redirects	[30]

Table 2: Defense evasion and respective fixes.

an amplifier to conduct DoS attacks. The same is possible the other way around if a targeted service serves large or multiple responses, leading to a DoS of the vulnerable server.

(A4) Bridging Attack: The fourth kind of attack is bridging attacks. Bridging attacks can happen when an attacker can control the protocol of a request, allowing the attacker to bridge between different protocols. A notorious example is the Gopher protocol. Since Gopher is simplistic, HTTP requests can be interpreted as Gopher requests. If the SSR client supports Gopher, an SSR can quickly turn into Remote Code Execution via bridging attacks, as shown by Gupta [18].

(A5) Exploiting SSB: The fifth class of attack exploits the client used to perform SSR. Musch et al. [28] established the risk of using a full browser as a client for conducting SSRs. Given the constant struggle of browser vendors to keep up with the most recent exploits, an outdated browser executing arbitrary requests can easily become an attacker's gateway to the server.

(A6) Local Resource Leak: The sixth and final class of SSRF covers exploitation accessing otherwise non-reachable local resources via a request to an internal IP or localhost. This is the most commonly known variation of SSRF and was featured in Figure 2.

3.2 Defending Against SSRF

Across the literature, we identified four distinct approaches to defend against SSRF, each with its caveats and drawbacks.

(D1) URL Validation: The most immediate solution to ensure that only intended targets are used for the SSR is validation based on the string on which the request is based. This validation can cover parts or the complete set encompassing a URL – *scheme, domain, port, path*, and *query* – using either an allowlisting or a denylisting approach.

While this is the most apparent defense, it has severe limitations. The correctness of the allow/deny procedure is paramount, and past events have shown that validating URLs is not trivial [1]. This results in Parser Confusion attacks, a well-documented attack on insufficient parsers. These attacks bypass an implemented mitigation via parser bugs or unexpected encodings. Therefore, to parse user input, well-established correct parsers should be used [21, 31]. Consequently, the whole URL has to be validated, as only prefixing protocol and domain in front of the user input might not be sufficient, as the user input can contain characters that confuse the parser [44].

A robust and complete URL allow-listing can defend against SSR abuses, as it limits the requests to well-known benign targets [30]. Denylisting is insufficient since attackers can always register or overtake new domains.

(D2) DNS Validation: Using DNS validation expands on checking the destination URL by validating the actual target, as it ensures that only a predetermined set of IPs can be requested. As the URL is resolved, the corresponding IP is validated against an allow or deny list, ensuring only intended targets. However, this defense has its challenges, as a cunning attacker could change the targeted IP between the check and the actual request time.

A proper implementation has to resolve the domain only once and then keep using the resolved and validated IP. This advanced defense is called IP pinning and provides the only reliable defense against attacks targeting local resources without unduly restricting the versatility of the SSR feature. IP Pinning protects against the DNS Rebinding validation bypass attack.

In a variant of the DNS validation approach, the target is validated against a denylist of unwanted targets, such as localhost. This can be used as a defense for applications that take arbitrary user input as a target by design, e.g., URL previews. However, this technique is limited by the ability of the developer to ascertain the deployment context. If they miss targets in complex deployment scenarios, this technique is insufficient. For example, suppose only typical local targets are blocked, but the application is deployed in a cloud environment. In that case, attackers can still access cloud-internal IPs hosting meta-data and configurations.

(D3) Secure Configurations: Shifting the focus from actual application code to feature flags and configurations of application components, the third type of defense covers secure configurations. To avoid bypasses resulting from HTTP redirects, the used HTTP client should either reject redirects in general or apply the target validation after each redirect [39]. Additional important security settings are visiting only HTTP(s) URLs, mitigating A4, and not exposing an Open Origin Policy (OOP). I.e., they shouldn't set the `Access-Control-Allow-Origin` header to `*`; otherwise, they are susceptible to being used for Origin Laundering.

Because XML External Entities can fetch external resources, previous work included disabling XML External Entities as an SSRF mitigation. Still, since this is a) a general security issue and b) off by default since PHP 8 [41], we will not discuss it further in this work.

(D4) Response Modification: The fourth and final approach to mitigate the effect that turns an SSR feature into SSRF can be achieved by modifying the response an SSR provides. The response can be wrapped to not directly reflect the accessed content or use the `Content-Disposition` header to prevent a browser from directly rendering the response [27]. Either approach reduces the ability to abuse the SSR feature as a proxy (A2). Finally, imposing a rate limit and fixing the response time would severely hamper an attack's ability to conduct a recon attack (A1) or a denial-of-service attack (A3). However, no response modification prevents A4 to A6.

3.3 Further Defenses and Threat Model

Multiple mitigation-level approaches that expand past the application level and touch the SSR's configuration have been pitched. On a network level, the SSR can be routed through a proxy [21, 39], ensuring requests can neither access local resources nor devices on the internal network. Furthermore, a robust network segmentation can ensure that no local resources can be accessed. Authentication should be enabled for all services [39]. This approach only affects A6 and potentially impacts A1, making all other attacks feasible.

Tennant [39] proposed the concept of a *SSRF Jail*, where DNS and networking calls are hooked on the OS level. However, they note that it is not suitable as a practical solution since applications have to request internal, i.e., deny listed, resources for valid reasons, which a naive hooking solution would prevent.

Threat Model In this work, we study the SSRF mitigations that developers deploy. Therefore, we assume a bug and vulnerability-free PHP standard library.

If we mention allow-listing approaches, we assume developers have a complete, up-to-date list of IPs or domains they control and trust. To summarize our exploration of the SSRF attacks and defenses and the resulting thread model: *We assume an attacker who can access a service that affords the capability to trigger and influence the target of a request.*

3.4 Existing SSRF Defense Implementations

We now understand the attacks an SSR feature must be defended against and a set of working defenses and mitigations. The remaining question is whether libraries used to make SSRs provide developers with the means to do so securely. To answer this question, we survey existing PHP frameworks, libraries, and HTTP clients with SSR capabilities.

Table 3 lists all PHP Frameworks included in our survey and the standalone PHP HTTP clients. We compiled this list by searching the web for the most used and popular PHP frameworks and clients. Additionally, we included all clients listed by the HTTPlug project [43]. The first column indicates if a framework offers an HTTP client and, therefore, SSR capabilities.

To evaluate SSR capabilities as well as SSRF defenses, we manually checked the documentations for mentions of Server-Side Request Forgery. We found that only five HTTP clients offer any defense:

Symfony Since Version 5.1. Symfony provides the `NoPrivateNetworkHttpClient` decorator, which ships mitigation against SSRF, which blocks requests to internal networks (D2) [37, 38]. Technically, the decorator hooks into the request process and is called after the DNS resolution to check if the resolved IP address is on a predefined denylist of internal IP addresses.

SafeCurl SafeCurl [14] blocks requests to internal IPs (D2) and pins the protocol to HTTP(S) (D3). Optionally, it provides DNS Rebinding protection by implementing a check of the resolved IP vs. a denylist (D2).

Framework	SSR capability	Defense
WordPress	✓	†
Laravel	✓via Guzzle	✗
Symfony	✓	D2
Yii	✗	✗
CakePHP	✓	✗
FuelPHP	✓	✗
Windwalker	✓	✗
Zend	✓	✗
Laminas	✓	✗
CodeIgnite	✓via Curl	✗
PHPixie	✓	✗
HTTP Client		
Guzzle	✓	✗
SafeCurl	✓	D2, D3
SafeURL	✓	D2, D3
HTTPlug	✓	D2, D3 (via plugin)
PECL HTTP	✓	✗
ReactPHP Sockets	✓	✗
WordPress Requests	✓	✗
Buzz	✓	✗
Httpful	✓	✗
PHP stdlib	✓	✗
PHP curl ext.	✓	✗

Table 3: PHP frameworks and HTTP clients included in our survey. The two columns indicate if they offer SSR capabilities and implement SSRF mitigations. ✓= Exists, ✗= Non-existent, † = exists, but is vulnerable.

SafeURL IncludeSec team [20] presented a set of safer HTTP clients in 2016, including *SafeURL* for PHP. It is a fork of SafeCurl.

HTTPlug plugin The HTTP client abstraction HTTPlug can be used with plugins to extend its functionality. Benoist [10] published a plugin that introduces SSRF mitigation capabilities for HTTPlug. It is 'inspired by SafeCurl' and has the same capabilities as SafeCurl.

Guzzle Guzzle is a popular third-party HTTP library. It does not provide an SSR defense, however the issue was already raised in the issue tracker [34] and a domain allow list was proposed. However, the issue was closed due to inactivity.

WordPress

We will now present our findings on WordPress. **`wp_http_validate_url` vulnerabilities** WordPress provides functions such as `wp_safe_remote_get` that promise safe requests via the `wp_http_validate_url` function: 'The URL is validated to avoid redirection and request forgery attacks.' [46]. We tested the first part of this statement with a status 307 redirection, but all functions of the function family followed the redirection except the `_head` variation. We checked the second part of the statement (request forgery protection): WordPress attempts to sanitize the URL with respect to internal IPs but does not pin the resolved IP address. It is, therefore, vulnerable to DNS Rebinding attacks. Since we could bypass both promised safety features, we don't consider it a safe client for this work and included it as a regular sink for our later study.

Disclosure We constructed a Proof-of-Concept and disclosed the DNS Rebinding issue to WordPress. However, the issue was closed as a duplicate of a report from 2017. We also reported the issue of unexpected redirection to WordPress.

Safe to use Sanitizer? Additionally, WordPress provides the function `esc_url_raw`. The documentation states, 'The resulting URL is safe to use in [...] HTTP requests.' [45]. The last statement is wrong and dangerously misleading since the function does not defend against SSRF and SSR abuses. In our later studies, we found data flows solely relying on this function to sanitize the input of an SSR sink; refer to Section 5.3. *Disclosure:* We reported the issue to WordPress.

Overall, only one framework and three clients support protection against SSRF, while several others support SSR capabilities without warning users about their risks or providing SSR defenses. Note that since most HTTP clients are supported by the HTTPlug system, they could be combined with the plugin to make them safe.

3.5 Usage Study

To evaluate if applications in our data set use the HTTP clients with some form of offered protection, we used ripgrep [15] to conduct a string-based search for usage of the corresponding HTTP clients. We constructed search expressions from usage examples, i.e., we searched for `use` statements, `new` statements, and fully qualified function names.

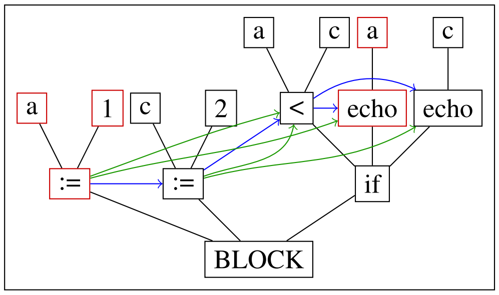
We found four repositories using safer HTTP clients in our dataset of 30.870 applications. One of them is the composer backend `packagist`. It is using a safe SSR request in its GitHub migration code. Another application, with safe HTTP client usage, is authored by one of the maintainers of one of the HTTP clients. This means that effectively, only **three** third-party open-source applications are using an HTTP client with a defense available.

Only a fraction of the frameworks that provide SSR functionality discuss SSRF in their documentation, and only Symfony provides an actual SSRF defense. Only the SafeCurl family provides an SSRF defense out of all the pure HTTP clients. If used through the HTTPlug framework, most clients could be secured using a plugin inspired by SafeCurl. But nobody is actually using these defenses.

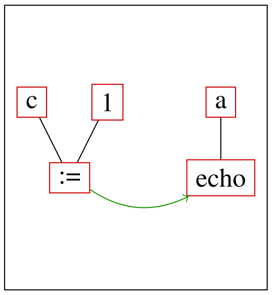
This study shows that developers do not use existing countermeasures against SSRF. This raises the question of whether they implement SSR defenses themselves. Or are they not using any defense at all? We need to inspect the source code of the applications for homegrown SSRF mitigations to solve this question.

```
a := 1
c := 2
if a < b
    echo a
else
    echo c
```

(a) Computation of a max value. Highlighted statements might influence the slicing criterion `echo max`.



(b) The CPG representation of Figure 3a with the nodes that might influence the slicing node `echo max`. The black edges represent the AST, the blue edges the CFG, and the green edges the DDG.



(c) The extracted program slice for `echo a` of Figure 3b. The green lines represent data dependency and the blue line represents control dependency.

Figure 3: A sequence of visualizations showing the transformation of a computer program (3a) into a Code Property Graph (3b) and finally an extracted program slice (3c).

4 Identifying SSRF Vulnerable Code

Given the complexity of SSRF, we want to study its prevalence and its expression in common software. For this, we develop a static analysis methodology that leverages a Code Property Graph (CPG) code representation [47] and interprocedural data flow analysis to identify interfunctional data flow from user-controlled sources into a known SSRF sink. Subsequently, we manually analyze identified data flows to gain an in-depth understanding of vulnerable flows and any present attempts to defend against SSRF exploitation.

We first lay the theoretical groundwork establishing our static analysis methodology (4.1), which is followed by describing our subsequent manual analysis of any identified data flow (4.2).

4.1 Automatic Static Analysis

To present our static analysis methodology, we start by giving a brief introduction to CPGs (4.1.1) and Data Flow Analysis (4.1.2), followed by an explanation of how we leverage those techniques to detect SSRF vulnerabilities (4.1.3).

4.1.1 PHP Code Property Graphs

A CPG is a combination of multiple different graph representations of program source code [47] – most prominently the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Dependency Graph (DDG), and Call Graph (CG). The ASTs of each method of the analyzed program are the forest that forms the basis of the CPG. Each remaining graph (CFG, DDG, CG) is layered into the ASTs by reusing the existing AST nodes and adding corresponding edges. For example, by adding `CFG` edges between subsequently executed statements. Figure 3b provides a visualization of a CPG representing the code in Figure 3a.

We present an up-to-date PHP CPG generator that is not based on the source code but utilizes the PHP interpreter's internal bytecode representation. The PHP interpreter provides a debug function that dumps the bytecode representation of syntactically valid PHP source code. Our CPG generator parses this bytecode dump into an AST structure. The main advantage of using the bytecode representation is that the dump comes with a CFG, which we use to add our CFG edges. This leads to a high degree of CFG correctness as the CFG is taken directly from PHP. Based on the CFG, we generate the data dependency graph using a standard algorithm [5]. Another advantage of using the bytecode representation provided by the interpreter is that each function and method reference is represented with a fully qualified name, including possible namespaces or class names in the case of static methods. Therefore, we create the call graph by matching the qualified names of functions and methods with their definitions if they are unique. In cases where the names are not unique, e.g., method names shared across class definitions, we do not create a call edge. Our tool is implemented against the publicly available specifications and framework for the Code Property Graph [48] and is publicly available[1].

PHPJoern We decided against using the existing PHPJoern ([9]) due to shortcomings we encountered. Besides being unmaintained and deprecated by its authors, we will discuss the two major shortcomings:

PHP Compatibility: PHPJoern can only generate CPGs for code compatible with PHP 7.1. PHP 7.1 has been deprecated and superseded by several minor and major releases introducing new (syntax) incompatible features. As a result, PHPJoern is unable to analyze modern PHP code and skips corresponding files that can thus not be analyzed.

CFG: PHPJoern generates the control flow graph itself, which leads to inconsistencies. `exit()` and `die()`, commonly

[1] https://github.com/SSRF-vs-Developers

used by PHP developers to stop a script prematurely in if a check fails, are the culprits for the inconsistent behavior. Implemented checks are commonly access controls, exit on failures, or user input validations. Figure 4 provides a minimal code example. If a condition (line 3) is met, the process ends with an exit call (line 4). Otherwise, line 6 is executed. The echo is unreachable if the condition is satisfied due to the exit call. PHPJoern generates a control flow edge between the exit()-node (4) and the echo (6). This is wrong.

Our bytecode CPG does not suffer from this issue because we get the CFG directly from the PHP interpreter. Therefore, we eliminate the potential for derivations between the actual PHP control flow as interpreted by the PHP engine and our CFG.

```
1   <?php
2   echo "init";
3   if($condition) {
4       exit();
5   }
6   echo "code";
```

Figure 4: PHP code that triggers the CFG bug in PHPJoern.

4.1.2 Data Flow Analysis

To perform the data flow analysis, we extract the subgraph that forms the input value to a given sink to perform the data flow analysis. Figure 3c provides a visualization of a data flow extracted from the CPG in Figure 3b.

Our data flow implementation starts by following back the data dependency edges leading into the sink call and collecting the involved nodes and edges recursively until all subsequent data dependency edges are consumed. After finishing the intrafunctional analysis, we identify each function call and function parameter usage and follow the call edges to the target and source nodes. The algorithm is then restarted from those nodes again. Our procedure stops as soon as there are no further nodes and edges to be added. The result is a subgraph containing all nodes involved in forming the input value for a given start call, i.e., a program slice.

4.1.3 SURFER – Detecting SSR(F) with Program Slicing

To identify potential SSRF vulnerabilities, we start out by searching for common SSRF sinks (Appendix 8.1) within a given CPG. Each such statement then serves as a starting point for our program slicing.

We reconstruct the possible input strings to the detected sinks based on the extracted program slice by traversing the slice across its data dependency edges. Starting at the slice leaves, i.e., the initial input values, we collect the applied transformations for each step until we reach the sink. This process results in a tree structure. Leaves represent the input values, e.g., constants and variables. Nodes represent the transformations, and the root is the sink. We traverse the tree, starting at the leaves, and apply the transformations we encounter. This procedure depends on the semantics of each transformation and leverages the individual bytecode instructions stored in the nodes. Thus, if multiple leaves lead to a concatenation, we combine them. If they lead to an assignment, we create a list of multiple outcomes, as multiple incoming dataflows indicate that multiple values are possible.

As this procedure is only used as a filter for our later manual analysis, we are liberal in the transformations we apply, and if we do not know the semantics of an instruction, e.g., an unknown function call, we pass the arguments through so as not to miss a possible vulnerability. Global variables, e.g. the super globals `$_REQUEST,$_POST, and $_GET`, are marked in the output. This allows us to recognize user-controlled sections of the input values passed into the sink and reason about a possible SSRF vulnerability due to user-controlled input.

As discussed in 2.2, we assume that existing sources and sinks are reachable, e.g., if they are behind some access control system, we assume that an attacker has access to them. This attack scenario is interesting in more complex deployment scenarios such as managed hosted services where regular users might have admin rights in the web app but not on the barebone machine.

4.2 Manual Investigation of Candidates

Applying SURFER reduces the initial set of projects of web applications to those also containing a data flow into a sink with an abstract representation of the value passed into the sink. However, a simple data flow analysis is not sufficient to achieve our goal of identifying the prevalence of SSRF and accompanying insufficient defenses. A defense does not necessarily reside on the data flow but can be contained in the direct (conditionals) or indirect (assertions) control flow associated with it. Consequently, we performed a manual in-depth analysis of each identified flow.

We take the candidates found by the previous static analysis and analyze them manually for possible defenses against SSR attacks and to determine if they are vulnerable. We start by filtering the apps into three distinct categories to remove any applications that are vulnerable by design or cannot be considered a proper web application, to begin with (Section 4.2). Next, we analyze whether a detected data flow is trivially exploitable (e.g., direct application of a sink on user inputs) or if a data flow is a false positive. Finally, we perform a manual in-depth exploitation and defense analysis to establish how a detected data flow can be exploited and whether any form of defense is present (Section 4.3).

Filtering for Apps Our overarching goal is to answer RQ4 and RQ3, i.e., are developers of applications using homegrown defenses, and how many applications are vulnerable to SSR abuses? This entails filtering any flow that is either irrelevant to our attacks or not representative of real applications.

Based on our reconstructed inputs, we filter out any flow for which an attack does not have control over the domain. We approximate this by only taking these candidates into consideration where attacker-controlled input is at the start of the reversed string. All of the discussed attacks require the attacker to control the domain, consequently, any flow for which this is not the case is not interesting for our overarching research questions.

Next, we survey all remaining flows and their corresponding applications to determine the type of project they belong to. We discard flows sourced from examples or test code. But most importantly, we remove any flow that are detected in 'hacking tools'. Hacking tools are projects that do not contain a real application but are used for red and blue teaming, such as Capture-the-Flag tasks and solutions, applications that are vulnerable by design for educational purposes, and malicious applications such as web shells. These are not representative of the regular developer and application and are thus out of the scope of our research.

4.3 Detailed Analysis

The remaining flows are inspected in-depth in a manual analysis for all known defense techniques detailed in Section 3.2 by manually retracing the data and control flow.

D1 URL Validation We analyze if any validation of the input URL takes place and distinguish between allow-list and deny-list approaches. Additionally, we categorize its implementation (e.g., via a regular expression) and which parts of the URL are checked, e.g., if only non-HTTPS requests are prevented. Additionally, we note any usage of a proper URL parser or homegrown solution. Finally, we assess whether breaking any present URL validation is possible.

D2 DNS We check for the presence of DNS validation and distinguish between DNS allow and denylisting, as well as if IP pinning is used.

D3 Configuration If we detect any configuration of the used sink we analyze how it affects redirection (i.e., enabled or disabled), if only http(s) requests are possible, and if there is a Open-Origin-Policy configured.

D4 Response Modification Finally, we analyze how the response of the SSR is used and, more significantly, if it is returned. If it is returned, we check if the result is wrapped or reflected as-is and if a Content-Disposition header is set. Additionally, we check for rate-limiting routines and fixed response time mechanisms.

5 Results

We have established SSRF as an intricate and relevant security issue and proposed a methodology and an implementation – SURFER – to detect SSRF vulnerable code and its possible protections. In this section, we describe the data set we searched with SURFER (5.1). We then discuss the parameter as well as the metadata surrounding our search (5.2) and present the raw results of SURFER, as well as the result from our manual analysis (5.3).

5.1 Data Set

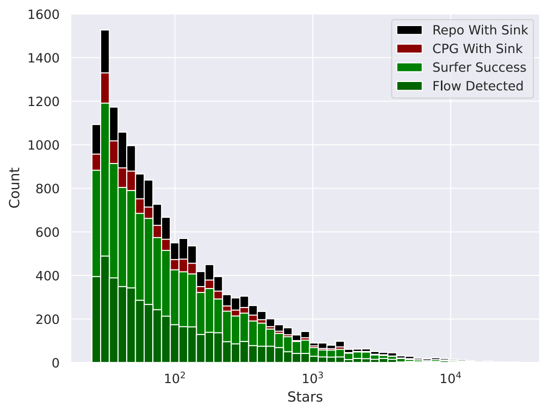


Figure 5: Stars of the repositories available, converted to a CPG, successfully analyzed, and with a flow detected. Only repositories that contain a sink are included.

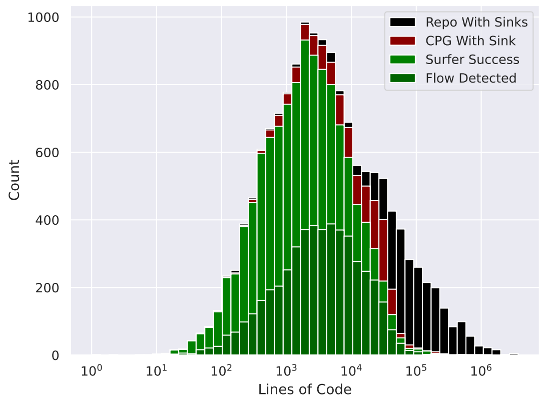


Figure 6: Lines of Code of the repositories available, converted to a CPG, successfully analyzed, and with a flow detected. Only repositories that contain a sink are included.

Our collection of open-source PHP applications was retrieved from GitHub – the largest code hosting platform. We used the public API to acquire all PHP applications with 26 or more stars[2] and started the download on July 31th, 2023. The

[2]this occurred organically through the limitations of the GitHub API

process took 6 hours and 40 minutes and occupied 411 GB of RAID SSD storage. Our final data set consists of 30.870 repositories.

We applied our CPG generator to each project and successfully created 28.325 CPGs. The CPG generation was run on a AMD EPYC 7702P with 504 GB RAM with 10 parallel processes. Each generation was allocated 20 GB of RAM and forcefully terminated after 10 minutes. To determine the coverage of relevant repositories in our dataset, we conducted another ripgrep-based search for calls of our SSR sinks. This is a rough over-approximation. 15.308 of our 30.870 repositories had at least one match. The distribution of stars and lines of code of the analyzed projects is displayed in Figure 5 and Figure 6, respectively. We have only included those that contain some SSR sink.

5.2 Applying SURFER

We compiled PHP functions that can trigger network requests in default configurations and included the popular 'curl' extension [42]. Additionally, we amended the list with functions from WordPress, as it is the most-used PHP framework. Sinks requiring a specific configuration and protocol were excluded, e.g., if `allow_url_include` is set to true, the include and require functions of PHP can trigger network requests. Our final list includes 15 function calls we use as sinks. The calls are listed in Appendix 8.1.

We ran SURFER with a timeout of 40 minutes and 20 GB of JVM heap space per analysis on a AMD EPYC 7702P machine. SURFER was done after 11 hours. SURFER successfully analyzed repositories that sum up to 107.4 mil. lines of PHP source code in 1.4 mil. files[3]. By average, each repository had 17.8k lines of code and 50.5 files.

5.3 Manual Analysis

In this section, we will discuss the results of our manual analysis. A visualization of the first few steps can be found in Figure 7 as a Sankey plot.

After the first pass of our manual analysis, we categorized our dataset. The dataset consists of 1.040 apps with some data flow. After filtering for dataflows with user input at the start of the string, we obtained a dataset of 237 flows. It consists of 158 flows we categorized as an app, 71 as a hacking tool, 7 as test code, and 1 as an example.

Of all the flows labeled as belonging to an application, 39 are trivially exploitable, 18 were false-positives, and 101 required further analysis. The false positives were mostly due to over-approximations in our static analysis for unknown function calls and objects. In 65 flows, the host part of the URL was attacker-controllable; in 36, it was not. A full overview of the first steps of our manual analysis pipeline can be found in the Sankey diagram in Figure 7.

[3]all lines of code and num. of files are measured via `cloc` [11]

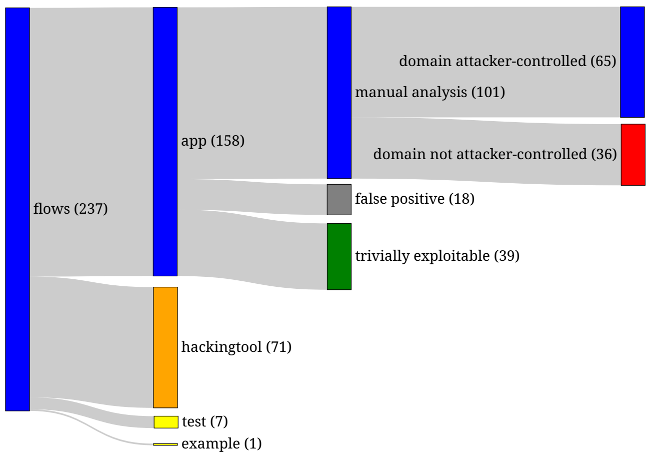


Figure 7: Sankey plot of analyzed flows.

Table 4 provides an overview of our findings regarding defense techniques 3.2 and 3.2. Potentially vulnerable flows fall into one of three categories: those without URL validation, allow-list style validation, or deny-list style validation. The most used implementation type was regular expressions. Additionally, five flows were found that use a good URL parser, and 14 validations were identified as broken. 216 flows validated no special part of the URL, while 10 did validate the scheme.

(D2) DNS Defenses No DNS resolution or pinning was used.

(D3) Configuration Four candidates were found that changed the redirection behavior. They all disabled it. No candidate set an Open Origin Policy. Table 5 provides an overview of the configuration defaults of the sinks. Table 6 provides an overview of the frequency of sinks. The core PHP provided sinks, i.e., `file_get_contents`, `getimagesize`, and `get_headers` accept non-HTTP(S) URLs by default. As discussed in Section 3.4, the WordPress functions do not disable redirects, except for the head function. Unsurprisingly, the WordPress HEAD functions and the PHP built-in `get_headers` function do not follow redirections. As their primary purpose is to return headers, redirection headers could never be polled by these functions if they were executed instead of returned.

(D4) Response Modification 15 flows returned the SSR result, and 13 did so in a wrapped manner, i.e., they wrapped the result in a JSON object and returned that or used some other processing step. No flow was found using a Content-Disposition header, rate-limiting, or fixing the response time to a constant value.

		#
category	no validation	38
	allowlisting	26
	denylisting	10
implem. type	Regex	12
	`is_file`	10
	`strpos`	6
	`substr`	5
	WP's `esc_url_raw`	4
	`opendir`	4
	`in_array`	2
	WP's `clean_url`	1
	`is_readable`	1
	wp_http_validate_url	1
Other	broken validation	14
	Secure, well-established URL parser	5
validated URL parts	none	216
	only scheme	10
	only host	2
	only path	3
	only query	1
	scheme and host	4
	scheme and query	1
	... all other combinations	0

Table 4: Analysis results for URL validation (D1).

Sink	Default Configuration	
	redir. disabled	HTTP(s) only
file_get_contents	✗	✗
get_headers	✓	✗
getimagesize	✗	✗
curl (init and setopt)	✓	✓
`wp_*` exc. `head`	✗	✓
`wp_(safe_)remote_head`	✓	✓

Table 5: Default configuration behavior of the SSR sinks.

Proper Defenses We found two apps (three flows) that implement a proper defense. Both use an allow-listing approach. One app uses `parse_url` to extract and compare the host against an allow-list. The other app defines a regular expression that only allows user input in parts of the query, like this: `https://example.com/path/.*/foo.txt`.

6 Discussion

This section will discuss our results, including valuable insights for other work. We will present two case studies, discuss our limitations, and finally answer our Research Questions.

Survey Result I: Community Knowledge We have established that the techniques to exploit SSRs are widely discussed in the literature and community. SSRF can be used as a stepping stone to deliver payloads for other vulnerabilities, circumvent firewalls, etc. The range of SSR abuses is exorbitant.

Sink	#
`file_get_contents`	85
`curl_init`	24
`getimagesize`	24
`get_headers`	11
`wp_remote_get`	10
`wp_remote_post`	2
`requests::get`	1
`wp_remote_head`	1

Table 6: Frequency of sinks in candidates classified as 'app.'

Survey Result II: SSRF Defenses Ready-to-be-used defenses against SSRF exist in Symfonfy's HTTPClient and the SafeCurl family of tools. WordPress tries to implement some defense but fails to do so. Since Symfony is a major framework and SafeCurl is effectively a drop-in replacement, we expected to find some usage. Surprisingly, they are used barely at all.

We developed a novel PHP CPG and SURFER to answer whether developers use home-grown defenses. There are two ways of defending against such attacks: Deny and allow listing.

Manual Analysis: Deny Listing To implement a proper deny list-style approach, DNS resolution and DNS pinning are required. Otherwise, an attacker could either register a new domain not on the denylist to attack a target with an SSR or provide a short-lived DNS entry to perform a DNS rebinding attack.

We did not encounter any DNS-based defenses. Since properly defending against SSRF attacks in a denylist scenario requires DNS rebinding, we were surprised by the absence of DNS requests. It is not entirely unexpected that DNS pinning routines are missing since it is a complex technique, but we find it noteworthy that no single DNS request call was found.

Manual Analysis: Allow Listing SSR abuses can also be prevented using proper URL parsing and a complete allow list (compare our threat model 3.3). Most applications (38) did not validate the URL in any way, but 26 did so in an allow list manner. However, since rarely any flow uses a proper URL parser, and most flows don't validate the arguably most important part of the URL, the domain, allow-list-based defenses do not seem to reflect the status quo as well. We could only find two apps that do properly allow list-based defending.

Technically, prefixing user input with a qualified URL can be interpreted as a type of allow listing. However, it cannot be considered a deliberate countermeasure. Instead, it is a fundamental design choice to realize a specific functionality of the application. Therefore, we argue that this is not a sufficient indicator of SSRF awareness.

Developers We have established that there is prior work available on the dire consequences of SSR abuses. But nobody is using existing defense solutions nor properly deny list-based defenses. We could only find two apps using proper allow-list-based defenses. From this, we can only conclude that SSRF is still not present enough in developers' minds and applications.

PHP: SSR by Accident? PHP is a language in which it is easy to implement SSR functionality 'by accident'. Many functions that deal with local files can also request remote resources. Most prominently, `file_get_contents`, but even more obscure functions such as `getimagesize` can trigger requests.

The implementation types `is_file` and `opendir` – both PHP builtins – indicate that developers try to limit the SSR sinks to their local features only. But both functions allow FTP requests. *Takeaway:* We propose that APIs should be designed so that SSR functionality has to be explicitly requested. Otherwise, developers introduce unexpected SSR features or have to implement checks to stop SSRs, introducing risk and complexity.

GitHub as a Data Source We want to provide insights into using GitHub as a data source for security research and surveys to help future work. One interesting result of our manual analysis is that many of our findings are in repositories that are not real applications but hacking tools, such as web shells or programs that are vulnerable by design, e.g., for CTF competitions. It is important to disregard those when reasoning about developer awareness since they do not reflect the status quo of real applications and developer awareness. Researchers must exclude these repositories when using GitHub as a data source.

We also encountered archived repositories as well as abandoned projects (19). Although this is not necessarily a sign of vulnerable applications, we expect that unsupported applications do not represent the current state of attacks and defenses.

Case Study: Insufficient Domain Validation We encountered a vulnerability in `LibreX` during our work[4] that serves as a suiting example for an insufficient attempt at protecting against SSRF. The vulnerable code is shown in Figure 8.

The code defines a domain allowlist in line 3. It then attempts to check the user input from line 1 against the allow list before passing it to the custom sink-wrapper `request()` in line 10. To do so, it first calls the userland function `get_root_domain`. Thus, it is implementing D2: Domain URL Validation.

However, the implementation of `get_root_domain` is defective. The developers utilize a home-grown solution instead of using a well-established URL parser, such as PHP's built-in `parse_url()`. It splits the provided URL using the forward slash as the delimiter. The third element is reversed and split again, this time at the dots. The second and first elements of the resulting array are reversed and concatenated. For example, `http://www.example.org` is split into `["http:","","www.example.org"]`. The third element is reversed to `gro.elpmaxe` and split `["gro", "elpmaxe"]`. Then, the second and first elements are reversed and concatenated: `example.org`.

To bypass the allowlist, an attacker can append `?.example.org` to an arbitrary scheme, port, and domain: `http://evil.com:22?example.org`. Since the last step only takes the first and second element in the reversed input, the function returns `example.org`, which is accepted by the allowlist. Thus, an attacker can control the scheme, domain, and port of the SSR request, making the application vulnerable to A1 (port scanning, network scanning) and A4 Bridging Attacks.

```
1  // we simplified this snippet.
2  $url = $_REQUEST["url"];
3  $requested_root_domain = get_root_domain($url);
4  $allowed_domains=["qwant.com", "wikimedia.org"];
5  if (in_array($requested_root_domain, $allowed_domains)) {
6      $image = $url;
7      $image_src = request($image);
8      header("Content-Type: image/png");
9      echo $image_src;
10 }
11 function get_root_domain($url){
12     $split_url = explode("/", $url);
13     $base_url = $split_url[2];
14     $base_url_main_split = explode(".",
        ↪  strrev($base_url));
15     $root_domain = strrev($base_url_main_split[1]) . "."
        ↪  . strrev($base_url_main_split[0]);
16     return $root_domain;
17 }
```

Figure 8: Vulnerable code snippet of the metasearch engine.

Case Study: unsafe `esc_url_raw` We found a vulnerable WordPress plugin that depends on the broken sanitizer function `esc_url_raw`. Figure 9 shows its sanitization function, which uses some user input as a source. The input is passed through `sanitize_text_field` from WordPress, which has no special effect on URLs. The plugin only depends on `esc_url_raw` as a sanitizer for SSR abuses. As established in Section 3.4 `esc_url_raw`'s documentation promises that

[4] `https://github.com/hnhx/librex`. We notified the developers of the problem. The repository was deleted from GitHub.

its result is safe for HTTP requests, but this is not true since it is not performing any SSR validation. This underlines the importance of clear and correct documentation.

```php
// We simplified this snippet.
function get_response( $url ) {
    $result = wp_remote_get(esc_url_raw( $url ));
    return $result;
}
```

Figure 9: Vulnerable sanitizer function of a WP plugin.

6.1 Limitations

While we aim at being complete, our approach contains limitations:

Static Analysis Static analysis suffers from inherent limitations that can lead to missed data flows due to programming patterns that are inherently difficult, if not impossible, to reconstruct. These patterns are a well-known limitation of static analysis that is under active research [e.g. 23, 24], and may lead to missed data flows. Static analysis is also vulnerable to false positives; however, as we manually verify each of our detected flows, we eliminate this risk reliably. Additionally, our static analysis failed for very complex applications as CPG creation can experience exponential growth, which may lead to time-outs or memory exhaustion depending on the structure of the application, limiting our data set (ref. Figure 6). Finally, we had to decide on one of the different and only partially compatible PHP major versions against which to implement our CPG creation. We decided on the, at the time, most recent PHP 8.2. This will inevitably lead to projects with legacy code that are only partially translatable into bytecode, with the files containing the legacy code left out. Using static analysis potentially reduces the overall amount of analyzed and reported SSRF-related data points.

Missed Second Order Vulnerabilities We do not consider second-order data flows when performing our manual analysis, as we filter out any flow without attacker control. Second-order vulnerabilities are notoriously challenging to detect [13] and, given the typical usage pattern of SSR, are of minor relevance to our overarching research question.

Focus on Common Sinks and Exploits We use popular HTTP sinks with protocol-agnostic exploitation patterns as our starting point for the static analysis. This excludes more exotic attacks involving technically complex and situational exploitations, e.g., leveraging `open_dir` via `ftp`. While those exploits are technically feasible, they do not represent this research's common and focused exploit scenario and should be considered for future work.

6.2 PHPJoern

We have discussed PHPJoern's shortcomings we identified before conducting our study (4.1.1). We, a posteriori, evaluated if they would have impacted our results if we had used PHPJoern. Since we mitigate false positives through our subsequent manual review, we focus on the version mismatch.

To estimate the impact of errors due to version mismatches, we conducted an experiment: Using PHP's syntax check feature, we measured the PHP 7.1 compatibility of our dataset. We found 4.990 repositories that use modern PHPJoern-incompatible features, i.e., they pass PHP 8.2's syntax check but not PHP 7.1's. We manually cross-checked these with our vulnerable flows and identified one vulnerable code path missing from PHPJoern's CPG. Therefore, we would be unable to find it if we based SURFER on PHPJoern instead of our new CPG generator. The breaking feature used by the vulnerable code is *DNF Type Declarations*.

6.3 Research Questions Answered

We will now answer our research questions from Section 2.3 and summarize our learnings.

How Popular are SSRs? 49.6 % of the repositories in our dataset contained at least one SSR sink. Since this number is string-search-based, it is a rough approximation. Filtering results for only those with user input in the data flow to the SSR sink, we get 141. This shows that application developers are using SSR sinks mostly with static targets.

Current State of SSRF defenses (RQ1) From an academic and documentation standpoint, SSR abuses are well documented. The literature provides enough sources on defenses, and OWASP is providing cheat sheets. The common framework Symfony implemented a safer HTTP client, and there are drop-in replacements for safer curl usage in PHP. WordPress attempts to provide a defense but is flawed and vulnerable to DNS rebinding. Other frameworks supporting SSR lack defense capabilities and do not discuss the risks of SSRs in their documentation.

Web developers are not using readily available defenses (RQ2) We discussed the availability of ready-to-be-used safer HTTP clients that defend against certain SSR abuses in Section 3.4. However, only a negligible amount of PHP applications on GitHub are using them.

Applications are lacking custom defenses (RQ3) Our static analysis results, in combination with an in-depth manual analysis, showed that proper defense and mitigation attempts are rare. No application is equipped with DNS validation, which is essential for a safe and proper deny-list-based defense against SSR abuses. Only two applications properly used allow listing.

State of SSR vulnerabilities in applications (RQ4) Considering that open-source web application developers are not using existing SSR defenses and are not implementing proper

SSR defenses, we conclude that SSR awareness has not arrived in the mainstream of application developers.

7 Related Work

SSR Studies Previous academic work introduced dynamic scanners to detect vulnerable SSR, Pellegrino et al. [32] proposed a black-box testing tool and scanned 68 services. We leverage the benefits of static code analysis, enabling us to cover all possible code paths without any input except the application itself. This allows us to conduct a large-scale study on 27,078 applications. Jabiyev et al. [21] proposed defenses against SSRF and benchmarked them against known SSRF vulnerabilities. Musch et al. [28] studied the prevalence and security implications of Server-Side-Browsers as SSR clients. Sahin et al. [33] have conducted a CTF experiment to study developers' awareness of different web attack types. SSRF was exploited the least, showing that it is still an unknown vulnerability class. Previous work focused on detecting SSR vulnerabilities or defending against them — we are the first to evaluate existing defenses in the wild.

SSR Systematization Pellegrino et al. [32] introduced a classification of SSRs along the axes of *Flaw*, *Behavior*, *Control*, and *Target*. Additionally, they presented some mitigation techniques they encountered.

In contrast, our systematization, presented in section 3, includes defenses as a first-class citizen. Consequently, attacks are explicitly mapped to suitable defenses. They are classified as *full*, *partial*, or *allow-list only* protections. Additionally, we systematized known defense evasion techniques and linked them with our previous efforts.

Furthermore, our systematization distinguishes between allow listing and deny list cases, which Pellegrino et al. did not cover. However, this differentiation is essential to a complete understanding of SSR defenses. Some defenses are sufficient in the allow list case, e.g., complete URL validation is an adequate host validation. At the same time, more technical effort and knowledge are required in the deny list case, i.e., DNS Rebinding protection is needed.

Additionally, while our systematization contains the attacks from Pellegrino et al., it presents a current and updated picture. It includes recent developments; for example, it encompasses the new attack surface of browsers as HTTP clients [28]. Additionally, we split the *Probe* class into the more suited *Port scan* and *Network scan* categories to better reflect the impact of different defense techniques.

Therefore, our defense-encompassing systematization can be used by both researchers and practitioners. Security researchers can easily classify their potential findings alongside existing mitigations using our systematization. Similarly, developers can leverage it to check if their implementation is vulnerable and are provided with better options.

PHP Static Analysis Previous work covered information flow and taint-style vulnerabilities in PHP applications. Huang et al. [19] detected vulnerabilities via information flow analysis in PHP applications and were the first to do so. Jovanovic et al. [22] proposed a static taint-flow analysis for PHP applications. Kassar et al. [23, 24] are working on pushing the coverage of PHP static analysis tools but overlook SSRF sinks and vulnerabilities in their work. Backes et al. [9] were the first to leverage code property graphs to analyze PHP applications. Alhuzali et al. [6] combined it with dynamic analysis to generate exploits more precisely. Shezan et al. [35] augmented it with cross-language capabilities to search for GDPR violations. Our novel PHP code property graph converter, written against the modern CPG standard [48], works on the CFG provided by PHP itself, making it more reliable than the one proposed by Backes et al. [9].

8 Conclusion

SSRF is a complex and multifaceted vulnerability class, and our survey of the current state of the art shows that developers must consider multiple attack vectors. However, the experiments in this paper reveal that developers do not properly defend against SSRF:

i) Our analysis of popular PHP frameworks and SSR libraries shows that even if SSR capabilities are offered, defenses in any form are commonly missing or defective. In particular, dedicated defenses against SSRF are either broken (WordPress) or are simply not used by the vast majority of PHP applications on GitHub (Symfony, SafeCurl, etc.) – Only four applications are using existing safe countermeasures, as shown by our usage study.

ii) As dedicated defensive measures are not used, we investigated if homegrown countermeasures are implemented instead. For this purpose, we examined 27,078 software projects sourced from GitHub using our CPG-based tool SURFER and a subsequent rigorous manual analysis. Our investigation into the resulting flows and deployed defenses revealed that *only two* applications employ their own safe allow-list defense. Furthermore, we did not find *any* secure deny-list defense since protection against DNS rebinding was absent in all cases.

In a somber conclusion, our results show that, while being comparatively infrequent, SSRF is widespread in the applicable subset of software projects: Almost *all* applications that might be susceptible to SSRF due to their application logic (i.e., they utilize at least one functionality that requires the retrieval of external HTTP resources based on user input) are indeed vulnerable to such attacks. Hence, our results suggest that developers either are unaware of SSRF's dangers or are unwilling/unable to implement effective defenses.

Acknowledgments

We are thankful for the valuable feedback of our anonymous reviewers and shepherd. This work has received funding from the European Union's Horizon 2020 research and innovation programme under project TESTABLE, grant agreement No 101019206. Additionally, it was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy — EXC 2092 CASA — 390781972.

Availability

Our tooling is available as open-source software at https://github.com/SSRF-vs-Developers.

Disclosure

We contacted the developers of affected repositories that were not deprecated or archived. We preferred the contact information from security policies to disclose the issues responsibly. If no security policy was present, we filed issues asking for their preferred way of disclosure or tried to contact the developers via email.

References

[1] Cve-2016-4029. Online https://www.cve.org/CVERecord?id=CVE-2016-4029, 2016. visited 2023-06-02.

[2] Cve-2021-21973. Online https://www.cve.org/CVERecord?id=CVE-2021-21973, 2021. visited 2023-06-02.

[3] Cve-2021-26855. Online https://www.cve.org/CVERecord?id=CVE-2021-26855, 2021.

[4] NVD - CVE-2023-43654. Online https://nvd.nist.gov/vuln/detail/CVE-2023-43654, 2023.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilerbau, Teil 2, Compilerbau*. Oldenbourg Wissenschaftsverlag, 2016.

[6] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392. USENIX Association, August 2018. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/alhuzali.

[7] Amazon Web Services, Inc. Instance metadata and user data - amazon elastic compute cloud. Online https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html, March 2022.

[8] Arr0way. SSRF Cheat Sheet & Bypass Techniques . Online https://highon.coffee/blog/ssrf-cheat-sheet/, 2021.

[9] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[10] Jérémy Benoist. Server-Side Request Forgery (SSRF) protection plugin for HTTPlug. Online https://github.com/j0k3r/httplug-ssrf-plugin, July 2022.

[11] Albert Danial. cloc: v1.81. Online https://github.com/AlDanial/cloc, 2019.

[12] DigitalOcean, LLC. How to access droplet metadata. Online https://docs.digitalocean.com/products/droplets/how-to/retrieve-droplet-metadata, March 2022.

[13] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1125–1142. IEEE, 2021. doi: 10.1109/SP40001.2021.00022. URL https://doi.org/10.1109/SP40001.2021.00022.

[14] fin1te. SafeCurl: SSRF Protection, and a "Capture the Bitcoins". Online https://whitton.io/articles/safecurl-ssrf-protection-and-a-capture-the-bitcoins/, May 2014.

[15] Andrew Gallant. ripgrep (rg). Online https://github.com/BurntSushi/ripgrep, 2021.

[16] Google. Subscribe to someone's Google Calendar - Computer - Google Calendar Help. Online https://support.google.com/calendar/answer/37100, 2023.

[17] Josh Grunzweig, Matthew Meltzer, Sean Koessel, Steven Adair, and Thomas Lancaster. Operation exchange marauder: Active exploitation of multiple zero-day microsoft exchange vulnerabilities. Online https://www.volexity.com/blog/2021/03/02/active-exploitation-of-microsoft-exchange-zero-day-vulnerabilities/, 2021. visited 2023-06-02.

[18] Tarunkant Gupta. Blog on Gopherus Tool. Online https://tarunkant.github.io/2018/08/14/2018-08-14-blog-on-gopherus/index.html, 2018.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004*, WWW '04, pages 40–52. Association for Computing Machinery, 2004. doi: 10.1145/988672.988679.

[20] IncludeSec team. Introducing: SafeURL – A set of SSRF Protection Libraries. Online https://blog.includesecurity.com/2016/08/introducing-safeurl-a-set-of-ssrf-protection-libraries/, 2016.

[21] Bahruz Jabiyev, Omid Mirzaei, Amin Kharraz, and Engin Kirda. Preventing server-side request forgery attacks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, pages 1626–1635. Association for Computing Machinery, 2021. doi: 10.1145/3412841.3442036.

[22] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5): 861–907, August 2010. doi: http://dx.doi.org/10.3233/JCS-2009-0385.

[23] Feras Al Kassar, Giulia Clerici, Luca Compagna, Davide Balzarotti, and Fabian Yamaguchi. Testability tarpits: the impact of code patterns on the security testing of web applications. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL https://www.ndss-symposium.org/ndss-paper/auto-draft-206/.

[24] Feras Al Kassar, Luca Compagna, and Davide Balzarotti. WHIP: improving static vulnerability detection in web application by forcing tools to collaborate. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023. URL https://www.usenix.org/conference/usenixsecurity23/presentation/al-kassar.

[25] Vickie Li. Bypassing SSRF Protection. Online https://vickieli.medium.com/bypassing-ssrf-protection-e111ae70727b, 2019. visited 2023-06-02.

[26] Colm MacCarthaigh. Add defense in depth against open firewalls, reverse proxies, and SSRF vulnerabilities with enhancements to the EC2 Instance Metadata Service. Online https://aws.amazon.com/de/blogs/security/defense-in-depth-open-firewalls-reverse-proxies-ssrf-vulnerabilities-ec2-instance-metadata-service/, 2019.

[27] mozilla.org contributors. Content-Disposition - HTTP | MDN. Online https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Disposition, 2023.

[28] Marius Musch, Robin Kirchner, Max Boll, and Martin Johns. Server-Side Browsers: Exploring the Web's Hidden Attack Surface. In *Proc. of the 17th ACM Asia Conference on Computer and Communications Security (AsiaCCS'22)*, May 2022.

[29] Ivan Novikov. SSRF bible. Cheatsheet. Online https://cheatsheetseries.owasp.org/assets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet_SSRF_Bible.pdf, Jan 2017.

[30] OWASP Contributors. Server Side Request Forgery Prevention - OWASP Cheat Sheet Series. Online https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html, 2022.

[31] OWASP Top 10 team. A10:2021 – server-side request forgery (SSRF). Online https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_(SSRF)/, September 2021.

[32] Giancarlo Pellegrino, Onur Catakoglu, Davide Balzarotti, and Christian Rossow. Uses and abuses of server-side requests. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2016*, January 2016.

[33] Merve Sahin, Tolga Ünlü, Cédric Hébert, Lynsay A. Shepherd, Natalie Coull, and Colin Mc Lean. Measuring Developers' Web Security Awareness from Attack and Defense Perspectives. In *2022 IEEE Security and Privacy Workshops (SPW)*, pages 31–43, 2022. doi: 10.1109/SPW54247.2022.9833858.

[34] sbani. New Option 'pin_base_uri' to Prevent Potential SSRF · Issue #2859 · guzzle/guzzle. Online https://github.com/guzzle/guzzle/issues/2859, 2021.

[35] Faysal Hossain Shezan, Zihao Su, Mingqing Kang, Nicholas Phair, Patrick William Thomas, Michelangelo van Dam, Yinzhi Cao, and Yuan Tian. CHKPLUG: Checking GDPR Compliance of WordPress Plugins via Cross-language Code Property Graph. In *NDSS*, 2023.

[36] Giada Stivala and Giancarlo Pellegrino. Deceptive previews: A study of the link preview trustworthiness in social platforms. In *27th Annual Network and Distributed System Security symposium*, February 2020. URL https://publications.cispa.saarland/3029/.

[37] Symfony. HTTP Client (Symfony Docs). Online https://symfony.com/doc/current/http_client.html, 2023.

[38] Symfony. New in Symfony 5.1: Server-side request forgery protection (Symfony Blog). Online https://symfony.com/blog/new-in-symfony-5-1-server-side-request-forgery-protection, 2023.

[39] Laurence Tennant. Mitigating SSRF in 2023. Online https://blog.includesecurity.com/2023/03/mitigating-ssrf-in-2023/, 2023.

[40] The MITRE Corporation. 2021 CWE top 25 most dangerous software weaknesses. Online https://cwe.mitre.org/data/definitions/1387.html, 2021.

[41] The PHP Group. PHP: Deprecated Features - Manual. Online https://www.php.net/manual/en/migration80.deprecated.php#migration80.deprecated.libxml, 2020.

[42] The PHP Group. PHP: cURL. Online https://www.php.net/manual/en/book.curl.php, 2024.

[43] The PHP HTTP group. HTTPlug. Online https://httplug.io/, 2023.

[44] Cheng-Da Tsai. A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages! Online https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf, 2017.

[45] WordPress. esc_url_raw() | Function, 2016. URL https://developer.wordpress.org/reference/functions/esc_url_raw/.

[46] WordPress. wp_safe_remote_get() | Function. Online https://developer.wordpress.org/reference/functions/wp_safe_remote_get/, 2017.

[47] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE Computer Society, 2014. doi: 10.1109/SP.2014.44.

[48] Fabian Yamaguchi, Markus Lottmann, Niko Schmidt, Michael Pollmeier, Suchakra Sharma, and Claudiu-Vlad Ursache. Github code property graph. Online https://github.com/ShiftLeftSecurity/codepropertygraph, 2023.

[49] Amar Zlojic. Server Side Request Forgery (SSRF) Attacks & How to Prevent Them, 04 2022. https://brightsec.com/blog/ssrf-server-side-request-forgery/.

Appendix

8.1 List of Supported PHP SSR sinks

We compiled PHP functions that can trigger HTTP requests in default configurations. We included the popular 'curl' extension. We amended the list with request sinks from WordPress, since it is the most-used PHP framework. Please note that `get_headers` performs a GET and not a HEAD request.

We chose not to include sinks in this work that require a configuration change to trigger an SSR, e.g., if `allow_url_include` is set to true, the include and require functions of PHP are able to trigger network requests. We include `file_get_contents` due to a similar reasoning. It requires `allow_url_fopen` to be set to true – which is the default.

However, since our methodology is general, the list can be easily modified to broaden the scope of sinks.

- `file_get_contents`
- `curl_init`
- `curl_set_opt`
- `getimagesize`
- `get_headers`
- `wp_http::get`
- `requests::get`
- `wp_remote_request`
- `wp_remote_get`
- `wp_remote_post`
- `wp_remote_head`
- `wp_safe_remote_request`
- `wp_safe_remote_get`
- `wp_safe_remote_post`
- `wp_safe_remote_head`

8.2 Findings

Table 7 lists the apps we identified as vulnerable. We contacted the developers if the repository was not archived or explicitly deprecated.

Name	Stars	Exploitability	Note
10up/safe-svg	193	◑	
A5hleyRich/delightful-downloads	27	◑	
akirk/friends	68	◑	
amzik/officemanage	27	◑	
Arsenal21/all-in-one-wordpress-security	40	◑	†
bigbignerd/WxCrawler	31	◐	
blindsidenetworks/wordpress-plugin_bigbluebutton	26	◑	
captn3m0/jqaas	32	◐	
chyrp/chyrp	203	●	
Codiad/Codiad	2823	◑	†
codingeverybody/makewebapp	33	◐	
csev/dj4e	90	◑	
Cvolton/GMDprivateServer	313	●	
cw1997/Tieba-Posting-Frequency	31	◐	
d3y4n/instagraph	326	◐	
dave-p/TVHadmin	26	◑	
diegolamonica/EUCookieLaw	50	◐	
DSJAS/DSJAS	42	◑	
factmaven/xml-to-json	48	◑	
Feathur/Feathur	72	◑	
fingerQin/Yaf-Server-Admin	51	◐	
Frecuencio/sqlbuddy-php7	40	◐	
friend-nicen/nicen-localize-image	63	◑	
GeSHi/geshi-1.0	162	◑	
greenido/backbone-bira	26	●	
hnhx/librex	652	◐	✗
iandevlin/resimagecrop	41	◑	
inclusive-design/AChecker	68	◑	†
jadijadi/techninjatheme	34	◑	
kodejuice/localGoogoo	41	◑	
lfiore/upld	42	◑	
Licoy/wordpress-theme-puock	1740	◑	
LyLme/lylme_spage	267	◐	
marekrei/encode-explorer	221	◐	
markjaquith/WordPress-Plugin-Readme-Parser	42	◑	
mojeda/QuickGallery	42	◑	
mokecc/VideoUrlParser	30	◑	
MonstaApps/Monsta-FTP	129	◐	†
mpeshev/DX-Plugin-Base	115	◐	
mwt/apfollow	34	◑	
nangge/webchat-robot	41	◑	
naofode/naofo.de	100	◑	
nbhr/php-reverse-proxy	81	◐	
nk932714/yify-movies-php	26	◑	
norbusan/piwigopress	426	◑	
onigetoc/m3u8-PHP-Parser	55	◐	
OpenGamePanel/OGP-Agent-Linux	86	◐	
PhiRhythmus/Tanx	134	◑	
photonstorm/AS3toTypeScript	70	◐	
PHPAuth/PHPAuth	872	◑	
phucvo0709/Clone-Google-Search-Engine	33	◑	
plidezus/aimozhen	39	◑	
qakcn/qchan	207	◑	
quicksketch/timezonepicker	53	◐	
rmorse/Open-Manager	61	◑	†
s3131212/allendisk	37	◑	†
segler-alex/radiobrowser-api	71	◑	†
shiflett/unveil	28	◑	
Simsso/Online-Tools	54	◐	
sixty-nine/PHP_Word_Cloud	39	◐	†
splitbrain/php-epub-meta	58	◐	
su18/Stitch	161	◐	
uksb/vqgen	48	◐	
vedees/wcms	250	◐	
vito/chyrp	232	◐	†
WolfieZero/Markdown-Viewer-PHP	50	◐	
wp-sync-db/wp-sync-db-media-files	520	◑	
wujunze/onlineDisk_search	26	●	
xb2016/kratos-pjax	949	◑	†
Average stars	190,55		
Median stars	54		

Table 7: Applications we identified as vulnerable. ◐: Repo contained a trivially exploitable flow. ◑: Repo contained a non-trivial exploitable flow. ●: Repo contained both trivial and non-trivial flows. †: Deprecated or archived, ✗: Repo was deleted.