Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications

Enze Wang*†, Jianjun Chen†‡✉ , Wei Xie*✉ , Chuhan Wang†, Yifei Gao*, Zhenhua Wang*,
Haixin Duan†‡, Yang Liu§, and Baosheng Wang*
* *National University of Defense Technology*
† *Tsinghua University*
‡ *Zhongguancun Laboratory*
§ *Nanyang Technological University*

***Abstract*—Server-Side Request Forgery (SSRF) vulnerability poses significant security risks to web applications, enabling adversaries to exploit web applications as stepping stones for unauthorized access of internal-only services or even performing arbitrary commands. Despite its recent emergence as a distinct category in the 2021 OWASP Top 10 web security risks and its increasing prevalence in modern web applications, there remains a lack of effective approaches to detect SSRF vulnerabilities systematically.**

We present a novel methodology, SSRFuzz, to effectively identify SSRF vulnerability in PHP web applications. Our methodology consists of three phases. In the initial phase, we designed an SSRF oracle to examine functions in PHP manuals and identify sinks that provide server-side request capabilities. This process yielded a total of 86 sensitive PHP sinks out of 2101 PHP functions. The second stage involves dynamic taint inference and the utilization of the identified sinks to examine the source code of target web applications, pinpointing all feasible input points that could trigger these sinks. The final phase employs fuzzing techniques. We generate testing HTTP requests with SSRF payloads, send them to the previously identified input points within the target web applications, and detect if an SSRF vulnerability is triggered. We implemented a prototype of SSRFuzz and evaluated it on 27 real-world applications, including Joomla and WordPress. In total, we discovered 28 SSRF vulnerabilities, 25 of which were previously unreported. We reported all the vulnerabilities to the affected vendors, and 16 new CVE IDs were assigned.

1. Introduction

The paradigm of retrieving information resources from web services through user-specified URLs has become a common factual feature of modern web applications. Those features are facilitated by *server-side requests* (SSRs), utilizing HTTP requests generated by servers for inter-server communication. Numerous popular web applications, including WordPress [44] and Joomla [18], utilize SSR for diverse functionalities, such as loading remote images as avatars or downloading plugins.

However, this widely adopted SSR mechanism is susceptible to *server-side request forgery* (SSRF) vulnerabilities. This security flaw occurs when a web application inadequately validates a user-provided URL, potentially leading to the redirection of server requests to harmful destinations, thus opening up opportunities for further exploitation of the target system. Due to its growing prevalence and threat, SSRF vulnerability got its own category for the first time in the 2021 OWASP Top 10 web application threats rankings [78]. Additionally, SSRF has been listed in the 25 most common and dangerous software vulnerabilities (CWE Top25) by MITRE [48], which underscores the increasing prevalence of SSRF and the critical need for addressing it.

The exploitation of SSRF vulnerabilities by attackers can lead to severe security consequences, including unauthorized access to private files and internal resources, as well as the capability to scan internal or external networks [35], [38]. Despite efforts by developers to mitigate these risks through measures such as string-filtering checks and URL access restrictions, sophisticated attackers have demonstrated the ability to bypass these defenses using techniques like DNS rebinding and exploiting servers with 302 redirect capabilities [9], [12], [22].

Prior research has proposed different static or dynamic testing techniques to detect various web vulnerabilities, such as SQL injection, XSS, and OS command injection [52], [55], [58], [61], [62], [68], [74]. However, there remains a lack of effective approaches to detect SSRF vulnerabilities systematically. Prior studies have developed a few tools for identifying SSRF vulnerabilities [1], [29]. Yet, these tools just collect known attack exploits, still relying on manual testing to discover novel attacks, which leads to ineffectiveness and incompleteness in testing, leaving many bugs undiscovered.

In this paper, we propose SSRFuzz, a new methodology aimed at addressing this gap. SSRFuzz is designed to efficiently identify SSRF vulnerabilities in PHP-based web applications, the preferred server-side programming language for 80.4% of websites [83].

The detection of SSRF vulnerabilities entails three chal-

✉ *Corresponding author: jianjun@tsinghua.edu.cn*

✉ *Corresponding author: xiewei@nudt.edu.cn*

lenges: (1) Lack of an oracle to identify all sinks related to SSRF vulnerability. Prior work collects a few SSRF sinks from known SSRF bug reports [37], [70] which is ad hoc and incomplete. There is a lack of comprehensive research on how to discover the sinks related to SSRF vulnerability in PHP. (2) Need for a specific method to reduce the input space of fuzzing. Modern web applications involve complex communication between the web front-end and back-end. This complexity is evident in two aspects: the high number of endpoints in web applications and the numerous parameters in requests. This complexity directly expands the input space of fuzzing. Without effective vulnerability information to drive fuzzing, such as HTTP parameters that can trigger SSRF vulnerability, fuzzing can become time-consuming, reducing its efficiency and effectiveness. (3) Crafting effective payloads to trigger SSRF vulnerabilities. As illustrated in Figure 5, the URL generation process involves the combination of multiple components. Only well-constructed payloads can successfully exploit these types of vulnerabilities. Besides, this payload needs to bypass application-specific string-filtering checks in the target web application.

Our methodology consists of three stages, each addressing a specific challenge. (1) In the sink identification stage, we designed an SSRF oracle-based picker to identify sinks in the PHP manual that provide server-side request capabilities. This stage yielded 86 sensitive PHP sinks from 2101 PHP functions. (2) In the inference stage, we employ the dynamic taint analysis to identify all viable input points capable of triggering these sinks. This stage reduces the tested input points and narrows down the fuzzing space. (3) In the fuzzing stage, we introduce payload generation strategies to generate SSRF payloads and infuse them into the previously identified input points within the target web application to produce test HTTP requests. After sending the test HTTP requests to the target web application, the vulnerability detector monitors whether SSRF vulnerabilities are triggered.

We implemented a prototype of SSRFuzz for PHP and evaluated it against 27 real-world web applications, including around 15.6M SLOC of code and 110.9K files. Our evaluation demonstrates that SSRFuzz can effectively and efficiently detect SSRF vulnerability. SSRFuzz identified 25 new SSRF vulnerabilities, of which 16 have been assigned CVEs. The results show that SSRFuzz outperformed existing SSRF vulnerability detection tools.

In summary, the paper makes the following main contributions:

- **New approach**: We present a novel approach, SSRFuzz[1], for discovering SSRF vulnerabilities with vulnerability information-driven web fuzzing. It combines dynamic taint inference with mutation-based fuzzing to augment the efficiency of testing.
- **New SSRF Oracle**: We propose an oracle to identify the sensitive sinks of SSRF vulnerability. Using this approach, we identified 86 sensitive PHP sinks out of 2101 PHP functions, among which 73 sinks are newly discovered.
- **Real-world impact**: We evaluated SSRFuzz with 27 real-world web applications and successfully identified 28 vulnerabilities, including 25 previously unknown ones. We disclosed all discovered vulnerabilities to the affected vendors and were assigned 16 CVE IDs.

1. https://github.com/SSRFuzz/SSRFuzz

2. Background and Motivation

In this section, we first present the workflow of SSRF vulnerability (§ 2.1) and then provide a concrete example of SSRF vulnerability in PHP web applications. (§ 2.2). Finally, we introduce the scope of our research in this paper (§ 2.3).

2.1. Server-Side Request Forgery Vulnerability

2.1.1. Attack Scenario. The SSRF vulnerability occurs whenever a web application (WebApp) provides SSR features but fails to validate the user-supplied URL properly. As shown in Figure 1, the normal workflow of SSR involves three steps. ① First, a user submits input to web applications intended to specify a URL for the server's request. This URL is usually specified in the HTTP request parameters. ② Next, the web application extracts the URL from the HTTP request and fetches the resource information indicated by the URL, which may originate from local, internal, or external servers. ③ Finally, the obtained resource is transferred to web applications. Depending on the different SSR functionality within the web applications, it either directly transfers the fetched resource to users or processes the resource before returning it, such as by only providing the web page title associated with the URL.

Although server-side requests are crucial for the functionality and interoperability of modern web applications, they must be handled securely to prevent SSRF vulnerabilities. For instance, web applications may not adequately validate user-supplied URLs, which allows an attacker to manipulate web applications to send a crafted request to an unexpected destination, even when protected by a firewall or NAT.

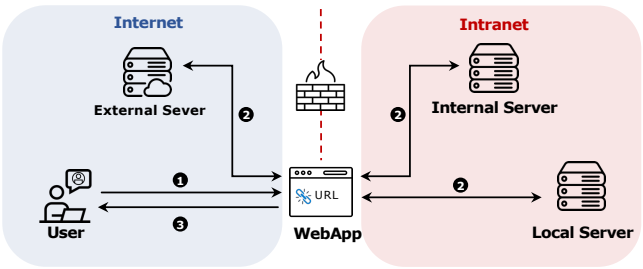

Figure 1: Normal SSR communication model.

2.1.2. Threat Model. In our threat model, we assume that a remote attacker can interact with a PHP web application

normally and has the ability to modify the HTTP request sent to the PHP web application. The security risks caused by SSRF vulnerability can be classified into three categories depending on the different resource servers that the attacker-specified URL targets.

- Exploiting Local Server via WebApp: The adversary manipulates the URL of the resource server to point towards a local server, allowing the *WebApp* access and retrieve resources from the local server. These resources include web interfaces that are usually restricted to local users in an unauthorized state and local file resources (such as application configuration files containing sensitive information). Once these resources are accessed, it can lead to potential data leakage.
- Exploiting Intranet Server via WebApp: The adversary can exploit SSRF vulnerability for activities like port scanning on internal servers (e.g., checking if port 6379 is open on a server within the same intranet as the *WebApp*) or identify application service (for example, to identify whether Tomcat is deployed on the internal server).
- Exploiting External Server via WebApp: The adversary can use an SSRF vulnerability to make *WebApp* act as an attack proxy, thereby hiding their real IP addresses when accessing an external server. The SSRF vulnerability can escalate to DDoS attacks on the external server when the attacker controls multiple *WebApps*. Attackers can also use the SSRF vulnerability to make *WebApp* obtain malicious code from his external server. If *WebApp* executes the malicious code, it will cause other security threats.

It is worth noting that existing defenses against SSRF attacks often rely on allow list access control policies [53] to restrict the resources that an attacker can access (e.g., domains and IPs). However, this defense might be bypassed using DNS rebinding attacks [11], [56] or HTTP redirections [69], [76].

2.2. Example

Listing 1 provides a PHP code example with an SSRF vulnerability. The web application allows users to select an image as their profile picture. Users can provide the image URL to the web application in the GET parameter called `picurl`. Once the server-side PHP application receives the URL, it will automatically fetch it with the `curl_exec` function and immediately return the response to the user. It is worth noting that the one who requests the image is the server rather than the browser.

The SSRF vulnerability arises since the code in Listing 1 does not restrict which URL can be requested. Thus, if an attacker submits a malicious payload (e.g., "`file:///etc/passwd`") to the application via the `picurl` parameter, the code will faithfully output the corresponding password file contents to the attacker.

2.3. Scope

Prior to exploring methods to detect SSRF vulnerabilities, we conducted an empirical study to understand the types of input data that trigger these vulnerabilities.

In the empirical study, we randomly selected a total number of 100 SSRF vulnerabilities from the CVE list [66] of the National Vulnerability Database (NVD) over the past three years. We analyzed these vulnerabilities based on disclosed information such as CVE descriptions, vulnerability reports, patches, etc. SSRF can be classified according to different standards. In this paper, we focus on classifying input data formats that trigger SSRF vulnerabilities. We observed that there are mainly two types of input formats: (1) 99% of the inputs are string values that conform to the RFC 3986 URL specification [75]. This string value is typically used as the value of HTTP parameters, including in parameters of JSON format data. (2) 1% of the input is a URL string in XML format data. Thus, our research is primarily concerned with detecting SSRF vulnerabilities caused by inadequate validation of URLs by web applications. Unless otherwise stated, we restrict the research scope to the former type of SSRF vulnerability and discuss identifying SSRF vulnerability caused by XML formatted input data as future work in Section 6.

Define the SSRF vulnerability within the scope. RFC 3986 defines the composition of URLs into six main parts, as shown in Figure 5. In this research, we aim to identify SSRF vulnerability where user-supplied data can corrupt elements of the URL, such as schemes (e.g., `file://`", `gopher://`"), domains, ports, and paths.

```php
<?php
function get_picture($pic_link){
    $curlobj = curl_init();
    curl_setopt($curlobj,CURLOPT_URL,$pic_link);
    curl_setopt($curlobj, CURLOPT_RETURNTRANSFER, 1);
    $result=curl_exec($curlobj);
    $data='image/png;base64,'.base64_encode($result);
    echo '<img src="data:'.$data.'">';
    curl_close($curlobj);
}

$pic_link = $_GET['picurl'];
if (!empty($pic_link)){
    get_picture($pic_link);
} else {
    $msg = "Invalid Resource";
    echo $msg;
}
?>
```

Listing 1: A code snippet for SSRF vulnerability

3. Overview

3.1. Challenges in Discovering SSRF Vulnerabilities

Given the serious security threat posed by SSRF vulnerabilities, we aim to develop an effective methodology

to identify SSRF vulnerabilities in web applications. To achieve this goal, we face three key challenges:

Challenge 1: Lack of an oracle to identify all sinks related to SSRF vulnerability. Prior work collects SSRF sinks from known SSRF bug reports [37], [70], but those bug reports do not cover all possible SSRF vulnerability scenarios. There are many other functions in the PHP documentation that provide server-side request capabilities and could lead to SSRF vulnerability. Unfortunately, identifying sensitive sinks associated with SSRF vulnerabilities is challenging due to the lack of available oracles.

Challenge 2: Reducing the input space for fuzzing. Web applications often provide users with a large number of input points. A significant portion of these input points is not pertinent to SSR functionality. Consequently, interactions with these specific inputs are not expected to precipitate SSRF vulnerabilities. Blindly fuzzing all input points can waste much testing time and reduce fuzzing efficiency. Therefore, we need a way to identify all potential input points that can trigger SSRF vulnerabilities before fuzzing.

Challenge 3: Generating effective payloads to trigger SSRF vulnerabilities. The payload that can trigger an SSRF vulnerability should be a string that conforms to the URL structure specification. In addition, since the target web application may have string-filtering checks, the payload needs to be mutated to bypass string-filtering checks. However, there is no available mutation strategy to generate payloads that both conform to the URL structure specification and can bypass string-filtering checks.

3.2. Our Solution

In light of these challenges, we propose a vulnerability information-driven web fuzzing called SSRFuzz. Figure 2 illustrates an overview of SSRFuzz. The overall input for SSRFuzz contains the PHP manual and web application source code, while the output is a report about the detected SSRF vulnerability. The workflow of SSRFuzz is as follows.

Solution for Challenge 1: Sink Identification Stage. We designed an SSRF oracle based on the definition of SSRF vulnerabilities within the scope of our study. Then, we utilize the SSRF oracle to examine functions in PHP manuals and identify sinks that provide SSR functionality. The workflow of this phase is as follows: ① Given the PHP manual, the test case generator can create 2101 code snippets as test cases for each of the 2101 functions in the PHP manual. ② With these test cases, the SSRF oracle-based picker filters out sensitive sinks that could lead to SSRF vulnerability. This stage only needs to be run once when testing different versions of PHP (such as PHP5 and PHP7) rather than for every web application tested.

Solution for Challenge 2: Dynamic Taint Inference Stage. In this phase, we leverage dynamic taint inference to examine the source code of target web applications, pinpointing all feasible input points that could trigger these previously identified sinks. This approach can effectively narrow down the input space of fuzzing. The workflow of this phase is as follows: ③ After hooking the SSRF sinks, SSRFuzz builds an instrumented web application runtime environment. ④ Web crawler interacts with the instrumented web application runtime environment. ⑤ The dynamic tainted inference module infers the input points that can trigger the sinks, such as the user-controllable parameter name. ⑥ Then, the identified input points (HTTP parameter name) are forwarded to the request builder for further testing.

Solution for Challenge 3: Fuzzing Stage. During this phase, we tailored a specialized vulnerability detector for SSRF vulnerabilities and developed new mutation strategies to generate testing payloads. The workflow for this phase is as follows: ⑦ The payload generator generates testing payloads as described in Section 4.3.1. ⑧ The requests builder then injects these payloads into the corresponding user-controllable parameter to form a new HTTP request. This request is sent to the test environment, which deploys the instrumented web application. ⑨ The vulnerability detector uses the vulnerability detection strategies described in Section 4.3.2 to determine if an SSRF vulnerability is present. If an SSRF vulnerability exists, a vulnerability report is generated. The vulnerability report includes the call stack information, the vulnerable code, and the HTTP request that can trigger the SSRF vulnerability.

4. SSRFuzz Design

4.1. SSRF Sink Identification

Prior research collects a limited number of SSRF sinks from SSRF bug reports, missing many functions susceptible to SSRF exploitation. To bridge this gap, our goal is to scrutinize every function in the PHP manual to identify all possible sinks. However, pinpointing sinks related to SSRF vulnerabilities is challenging due to the absence of detection oracles and the PHP manual's lack of explicit documentation on SSR behaviors for PHP functions.

To surmount this obstacle, we have developed an SSRF oracle for this task. First, we collect a list of URL schemes by examining the PHP manual and the source code of the PHP engine Zend because SSRF vulnerabilities are usually triggered by user-supplied URLs, as presented in Section 2.3. We collected 19 URL schemes in total. Next, we analyze the SSR functionalities of these URL schemes to understand their expected behaviors, such as file access or network socket interaction. This analysis led to the creation of 21 probe payloads designed to activate these SSR functionalities, as detailed in Table 3.

After that, we construct test cases for each function in the PHP manual for examination. The model output in Appendix A.1 shows a test case for the `fopen` function. Subsequently, we develop an SSRF oracle-based picker that dynamically loads and executes each test case using previously crafted probe payloads. If a test case target exhibits the expected SSR behavior during the testing process, we consider the corresponding function as an SSRF sink. For instance, if executing the test case for the `fopen` function

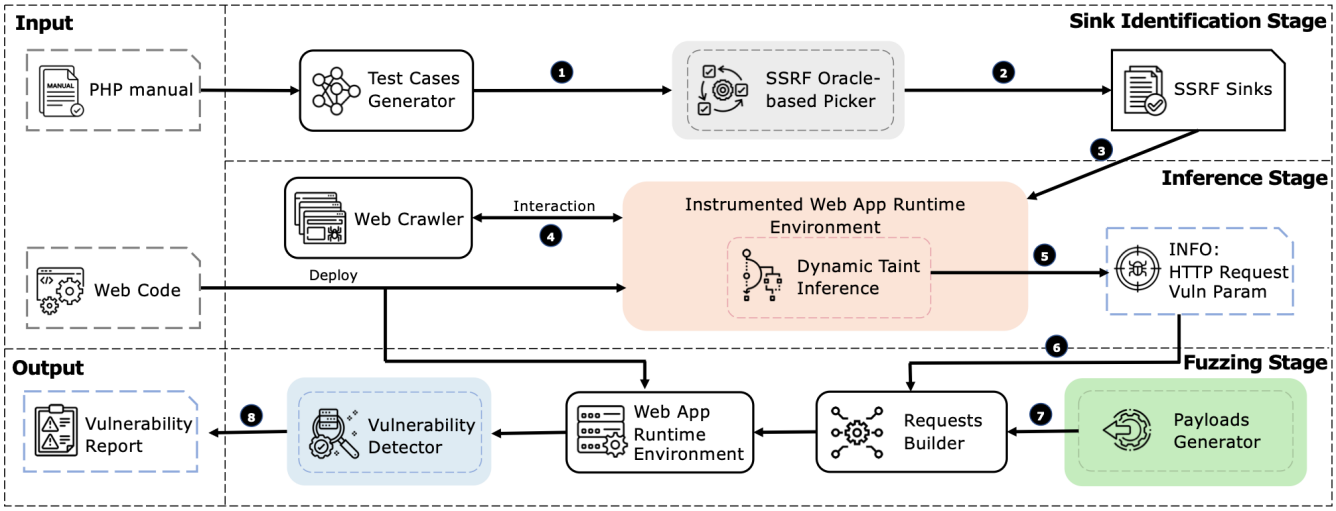

Figure 2: SSRFuzz architecture: A workflow overview of SSRFuzz for SSRF vulnerability.

with the input "file:///flag" results in accessing the flag file, we classify the `fopen` function as an SSRF sink. To minimize manual efforts in test case generation, we employ the few-shot prompting technique using OpenAI's ChatGPT API, enabling automatic test case generation (further details in Appendix A.1). Ultimately, our analysis identifies 86 out of 2101 built-in PHP functions that can potentially lead to SSRF vulnerability.

We further compared sinks identified by SSRFuzz with existing collections of SSRF sinks and found that SSRFuzz has identified all known sinks so far and has also discovered 73 new ones. The SSRF sinks identified by SSRFuzz were compared against other collections, as depicted in Figure 3. This collection includes two SSRF vulnerability labs (SSRF Vulnerable Lab with 0.5k Github stars [37] and OpenRasp Testcases with 0.25k Github stars [70]), a comprehensive vulnerability lab (Pikachu vulnerability testing platform with 2k Github stars [73]), and SSRF sinks referenced in non-academic articles [27], [36], [40] from the top 30 Google search results.

To the best of our knowledge, the list of SSRF sinks we found is the most comprehensive to date[2]. As we will show in Section 5.3, these newly discovered sinks have aided us in identifying multiple new vulnerabilities.

4.2. Dynamic Taint Inference for SSRF

4.2.1. Instrumentation for Dynamic Taint Inference. SSRFuzz's instrumentation is designed to perform dynamic taint inference. The dynamic taint inference module has two core functions. The first function involves tracking input sources with taint tags and establishing whether they can flow into sensitive sinks and form a vulnerable taint flow. The second function is to ascertain the potential of a vulnerable taint flow to activate an SSRF vulnerability.

2. https://github.com/SSRFuzz/SSRFSinks

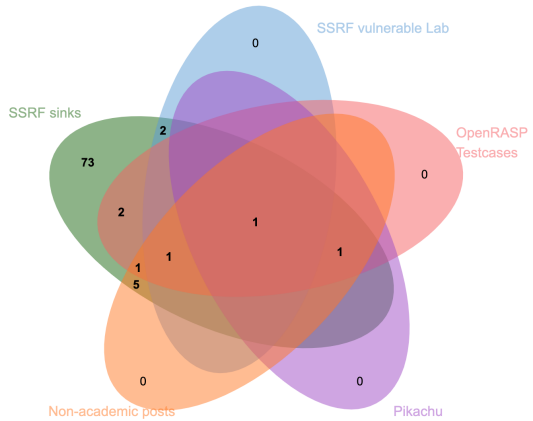

Figure 3: Venn Diagrams showing covered sinks between the vulnerability labs and our work.

The instrumentation enables dynamic taint inference to track and analyze tainted data, allowing it to identify parameters likely contributing to an SSRF vulnerability accurately. This allows the fuzzing engine to focus only on these parameters, which reduces the overall input space used for fuzzing.

The instrumentation contains three functionalities: (1) attaching taint tags to string variables, (2) propagating and tracking taint tags during program execution, and (3) hooking SSRF-sensitive sinks. We have modified the implementation of string-related functions, string variables, and sensitive sinks within the Zend virtual machine to achieve the above three functionalities. Therefore, the instrument does not need to modify the web application, making it transparent to web applications and facilitating smoother deployment.

Figure 4 illustrates how the instrumentation works while executing our vulnerable example in Listing 1. Initially, the input source (`$_GET['picurl']`) is attached with tainted tag. The user specifies the value received by this

input source in the HTTP request parameter `picurl`. As the code executes dynamically, the tainted tag also moves along the execution path shown in Figure 4, spreading from source operands to destinations. The instrumentation pre-hooks the SSRF sinks (e.g., the `curl_exec` function in code Listing 1). Thus, when the tainted tag reaches the `curl_exec` function, the instrumentation detects the behavior of a sensitive sink invoked over the tainted malicious string. Subsequently, the instrumentation records the input source information from the dynamic taint tracking. However, we cannot determine whether the variable's value becomes safe after sanitization during the propagation [55]. To bridge this gap, we need to use dynamic taint inference to assess the possibility of an SSRF vulnerability.

4.2.2. Dynamic Taint Inference. We developed a dynamic taint inference method based on instrumentation to evaluate the existence of SSRF vulnerability. This approach is based on a key observation that SSRF vulnerability is essentially caused by invoking an SSRF-sensitive sink on tainted malicious characters. If there are no effective sanitizations in the vulnerable tainted flow, then the value at the taint source and sink tend to be highly similar or even identical. It is worth noting that sanitization is essentially a process for strings involving add, delete, and replace operations. Thus, these similarities can be compared to determine the potential risk of activating SSRF vulnerabilities.

Algorithm 1 shows our dynamic taint inference method to discover the correlation between values at taint sources and sinks. The following paragraphs will describe these phases using the symbols outlined below.

- X = a source (e.g., `$_GET['picurl']`)
- Y = a SSRF sink (e.g., `curl_exec`)
- X_v = string value at source X
- Y_v = string value at SSRF sink Y
- H = an HTTP request message
- p = an HTTP parameter that can trigger an SSRF vulnerability

The core of the algorithm is to iterate over each source. The input sources that are under the control of users are deemed untrusted [54]. The `AddVariableWithTaintTag` function is employed to mark the untrusted input source as tainted, thereby highlighting the potential security risks associated with the data originating from these sources.

A pivotal aspect of this algorithm involves the determination of the correlation between the values of the taint source and the sink. The `IsTaintMetaInfoIntoSink` function is utilized to ascertain whether the taint tag has propagated into SSRF sinks. To evaluate the correlation between the values of tainted sources and sinks, conditional checks involving the `Substring` and `StringSimilarity` functions are employed. This evaluation facilitates the early exclusion of tainted flow that incorporates strict string-checking logic, thereby optimizing the efficiency of the fuzzing.

Function 1: Substring. This phase is focused on identifying an exact substring match between string values originating from the source (X) and terminating at the sink (Y). If a substring match is found, the dynamic taint inference module immediately reports a potential taint flow that can trigger an SSRF vulnerability. In scenarios where an exact substring match is not present, the `StringSimilarity` function comes into play, assessing the probability of the taint flow leading to an SSRF vulnerability.

Function 2: String similarity. This phase performs approximate matching on X_v and Y_v. The Levenshtein distance between X_v and Y_v is calculated to obtain a similarity score. This score is the result of dividing the Levenshtein distance by the length of whichever string is longer, X_v or Y_v. This similarity score is then compared with the threshold β for the purpose of discarding taint flows unlikely to cause SSRF vulnerabilities. The formula 1 shows how the similarity score between X_v and Y_v is calculated. To minimize the error of incorrectly identifying two distinct strings as the same, the similarity threshold is intentionally set high, at 0.9.

$$Similar\left(X_v, Y_v\right) = 1 - \frac{levenshtein\left(X_v, Y_v\right)}{max\left(strlen\left(X_v\right), strlen\left(Y_v\right)\right)} \quad (1)$$

Algorithm 1 Dynamic Taint Inference Algorithm for Identifying Potential Input Sources Triggering SSRF Sinks.

```
Input: Sources: Untrusted input sources of PHP (e.g., $_GET, $_POST)
Output: H_p: an HTTP request message with parameter p which can trigger SSRF Sinks
Begin
InitializeInstrumentedWebEnvironment()
for each X ∈ Sources do
    AddVariableWithTaintTag(X)
    for each Op ∈ Operations do
        if InvolvesTaintedVariable(Op) then
            PropagateTaintTag(Op)
        end if
    end for
    if IsTaintMetaTagIntoSink(Y) then
        if Substring(X_v, Y_v) is not True then
            if StringSimilarity(X_v, Y_v) is True then
                return H_p
            end if
        else
            return H_p
        end if
    end if
end for
End
```

4.3. SSRF Fuzzing

SSRFuzz employs a vulnerability-driven fuzzing approach to identify SSRF vulnerabilities and generate Proof of Concept (PoCs). During the inference stage, the dynamic taint inference module pinpoints potential SSRF vulnerability taint flows (more details in 4.2.2). Upon detecting an

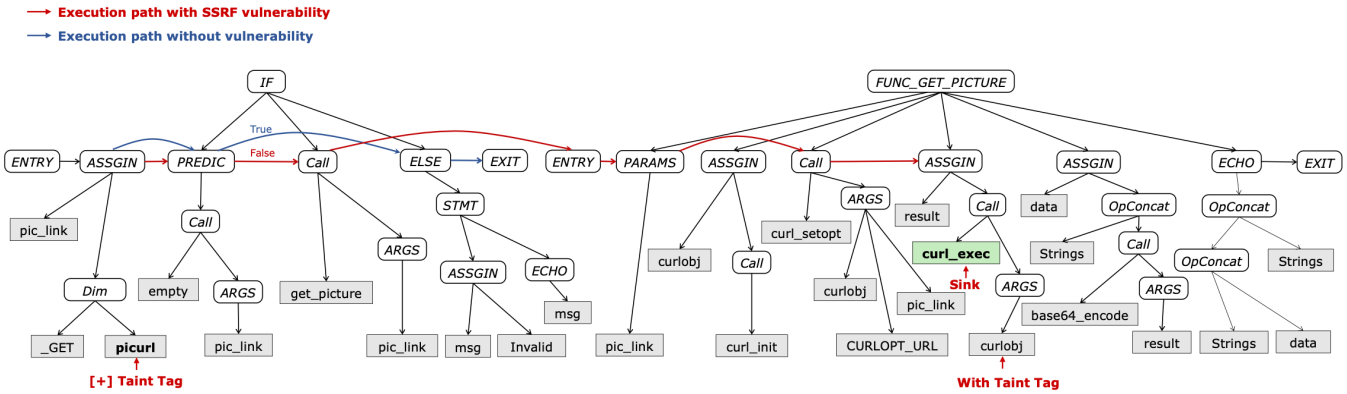

Figure 4: The syntax tree of the PHP code snippet in the Listing 1.

SSRF vulnerability taint flow, SSRFuzz records two key information: the HTTP request triggering the vulnerability and the specific parameter name facilitating the taint flows into the sensitive sink. This key information related to vulnerabilities is relayed from the inference stage to the fuzzing stage. In the fuzzing stage, the request builder receives this key information and uses the payload generator to generate the SSRF payload as input to the corresponding HTTP request parameters. This process constructs a series of HTTP requests with different SSRF vulnerability payloads. After that, the fuzzing starts for the same web application runtime environment without the instrumentation module. Then, the vulnerability detector uses predefined detection strategies (detailed in 4.3.2) to identify SSRF vulnerabilities. Upon successful detection, a comprehensive vulnerability report is generated, encompassing three critical pieces of information: (1) the HTTP request capable of triggering the SSRF vulnerability (along with the input parameter that can cause the vulnerability marked in the request), (2) the code implicated in the SSRF vulnerability, and (3) the call stack detailing the SSRF taint flow.

4.3.1. Payloads Generator. The primary objective of mutations is to generate payloads that effectively trigger SSRF vulnerabilities and bypass string-filtering checks. The dynamic taint inference module has pinpointed the user-controllable HTTP parameters associated with potential SSRF vulnerabilities. These parameters are thus targeted as key objects for mutation testing. SSRFuzz generates a range of payloads tailored to test these user-controllable parameters. These payloads are strategically injected into the parameters. The vulnerability detector is then employed to confirm or refute the presence of SSRF vulnerabilities.

Mutation vectors. SSRFuzz mutates the position of the components in a URL (as shown in Figure 5) to craft payloads that can activate SSRF vulnerabilities. The mutation is divided into two steps: (1) The initial stage involves constructing initial payloads formulated per established testing strategies. (2) The subsequent stage involves performing mutation operations on the initial payloads. During the fuzzing procedure, these generated payloads are utilized

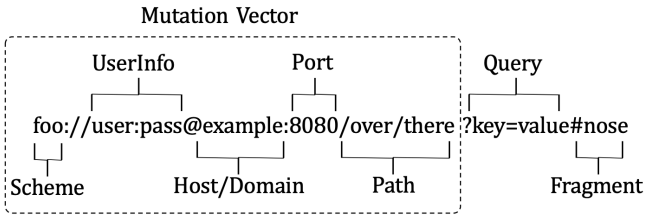

Figure 5: Structure of URL and mutation vectors

in a random sequence as inputs for fuzzing. The fuzzing engine follows a rule that stops further experimentation on the remaining payload once the SSRF vulnerability is successfully triggered.

Phase I: Initial Payloads. In this phase, SSRFuzz first creates a comprehensive set of testing strategies that outline methods for combining URL components. These strategies are highly comprehensive, covering various combinations of URL components, with the aim of thoroughly examining every possible combination. Therefore, the payload generator produces a diverse collection of initial payloads, each injecting different values corresponding to the respective attributes of the URL components. Table 4 illustrates the specific filling operations. These initial payloads are specifically crafted to effectively trigger SSRF vulnerabilities, particularly effective without string-filtering checks.

Phase II: Mutating Payloads. The main goal of mutation is to transform the initial payload generated in Phase I so that the mutated payload can trigger SSRF vulnerability and bypass string-filtering checks. To achieve this goal, we investigated known CVEs, and existing evasion techniques from the Internet [26], [32], [39]. Based on these investigations, we designed 31 operations summarized in Table 5 to bypass strings filtering logic.

The M29 mutation operation, for instance, was conceptualized following our analysis of current SSRF defense methods [34]. Our research reveals that developers often have a limited understanding of the complexities involved in URL parsing, which in turn affects the effectiveness of their sanitization functions. To address this, we introduced the M29 mutation operation specifically to bypass whitelist-

based input filters. This approach enabled us to discover an SSRF vulnerability in the open-source software WonderCMS, a feat not achieved by other state-of-the-art tools (as detailed in Section 5.5.1).

SSRFuzz uses a combination of mutation operations to generate a chain list. Each chain in this chain list entails a list of mutation operations that SSRFuzz applies to mutate an initial payload. Each mutated payload is thus a computation result of applying mutations in a chain to an initial payload. We predefine a set of validity conditions for each mutation operation and exclude operations that do not satisfy these conditions when creating the chain list. For example, when the initial payload does not contain the string "127.0.0.1". In this case, mutation operations M1-M11 will be considered invalid.

4.3.2. Vulnerability Detection Strategy. We designed the six vulnerability detection (VD) strategies based on the threat model described in Section 2.1.2. In this threat model, attackers typically exploit SSRF vulnerabilities to access various types of resources. As a result, vulnerability detection strategies can be categorized into network and file monitoring strategies, depending on the attacker's intentions.

The network monitoring strategy employs the out-of-band application security testing (OAST) technique to monitor HTTP and DNS interactions, referred to as HTTP OOB and DNS OOB in this paper. Additionally, the network monitoring strategy also includes the monitoring of port access and the analysis of HTTP response packets. These four monitoring strategies correspond to the generation strategies for URL components during the phase of mutation payloads, namely Scheme, Userinfo, Host/Domain, Path, and Port.

VD1 The HTTP OOB monitor focuses on tracking the HTTP requests received by the server hosting the vulnerability detector. It identifies SSRF vulnerabilities by detecting a specific URL (e.g., `http://vuln.detector/a6s9emny`). The path component of URLs is introduced by the payload generator during Phase I of the initial payload.

VD2 The DNS OOB monitor is centered on observing DNS requests received by the server hosting the vulnerability detector. The detection of SSRF vulnerabilities is facilitated by justifying if a specific subdomain DNS query exists (e.g., `a6s9emny.vulndetector.com`). The subdomain part is introduced in the Userinfo and Host/-Domain components by the payload generator.

VD3 The Port monitor operates by activating multiple ports on the server where the web application is hosted, monitoring for any incoming connections. Connections to these ports signal the presence of an SSRF vulnerability.

VD4 The HTTP response Monitor analyzes HTTP response messages. For example, it examines responses to the payload "`file:///flag`" to check if they contain the content of the flag file.

The file monitoring strategy operates directly on the web application server and involves monitoring HTTP log files and feature files. These two monitoring methods correspond to the generation strategies aimed at the Scheme and Path components of the payload generator for URLs.

VD5 The HTTP log monitor detects URLs with specific strings in HTTP logs, and if such URLs are found, the vulnerability detector reports the presence of an SSRF vulnerability. This monitoring policy focuses on malicious requests destined for local IP addresses. It is designed to identify cases where the Path component of the URL is susceptible to manipulation via user inputs. It identifies SSRF vulnerabilities by detecting a specific URL in the HTTP log (e.g., `GET 127.0.0.1 - [DATE] "GET /a6s9emny" 200 Status Code`).

VD6 The specific file monitor checks whether the web application has interacted with specific files, including actions like access or modification. An example of using this strategy is creating a "flag" file in the root directory and monitoring it for any interactions after sending an HTTP request using a request builder. If any interactions are detected, it signifies the presence of SSRF vulnerabilities.

5. Evaluation

In this section, we first introduce the SSRFuzz prototype (§5.1) and the experimental setup (§5.2). We then evaluated SSRFuzz for finding SSRF vulnerabilities. (§5.3). We also compared it against a state-of-the-art black-box vulnerability scanner (§5.4). Finally, we discuss a few interesting vulnerabilities SSRFuzz detects (§5.5).

5.1. SSRFuzz Prototype

We implemented a prototype of SSRFuzz to find SSRF vulnerabilities. The test case generator is built on the Python LangChain library and uses gpt-3.5-turbo model [25] to generate test cases automatically. The SSRF oracle-based picker is developed in Python, while the web crawling functionality is provided by crawlergo [7], a highly regarded open-source tool. The crawlergo can recursively discover URLs within the same domain and queue them for exploration. Further, it provides an array of advanced built-in features, including the ability to automatically submit forms and smartly trigger JavaScript events, thereby enabling in-depth exploration of web pages. As crawlergo navigates through web pages, it simulates user interactions such as clicks and form fill-ins to initiate network requests. For instance, it automatically determines the value types of form fields, and inputs predefined usernames, email addresses, and other information to meet the format requirements of the form. The instrumented web app runtime environment is set up on an Apache web server with PHP version 7.2.34. The instrumentation module, developed in C language and based on xmark [45], provides an interface for PHP. This allows us to use PHP code to attach taint tags to untrusted sources and hook SSRF sinks. The capabilities of the taint tags tracking are built upon modifications to the Zend virtual machine.

The dynamic taint inference module was developed using PHP and C. The payload generator, request builder, and vulnerability detector are implemented in Go. These three modules discover SSRF vulnerabilities by sending HTTP requests with inserted payloads to the web application runtime environment.

5.2. Experimental Setup

Dataset. We evaluated SSRFuzz on 27 real-world PHP applications, which collectively consist of 15.6 million Source Lines of Code (SLOC) and over 110.9 thousand files, as referenced in Figure 6. Our criteria for selecting the applications include two categories of applications. First, we collected popular PHP applications based on the ranking from W3Techs [42], to cover widely-used and complex applications such as WordPress, Drupal, and Joomla. We gathered 17 PHP applications that either have over 0.1% market share or have over 500 stars on GitHub. Second, we supplemented 10 more applications that were evaluated by prior works [50], [64], [81], [88]. Based on statistical data from ZoomEye [47], the total number of web applications in our dataset, as deployed across the Internet, amounts to 16.9 million.

Setup. SSRFuzz was deployed on a Linux VM with Intel 8-cores of 2.3 GHz CPU with 16 GB RAM. We installed Ubuntu 18.04, Apache 2.4, and PHP 7.2.34 at the target system under testing, accounting for 15.1% of web server settings among the Alexa top 10 million websites using PHP [41]. To conduct this assessment, we created separate virtual machine snapshots for each web application, including the SSRFuzz instrument module, started the web application, and ran SSRFuzz.

5.3. Discovering Real-World SSRF Vulnerabilities

In total, SSRFuzz finds 28 unique vulnerabilities from those 27 web applications, among which 25 were previously unknown. We have responsibly disclosed all the identified vulnerabilities to the corresponding vendors and have received 16 CVEs to date. The remaining vulnerabilities are waiting for CVE numbers to be assigned. Furthermore, we analyzed the functionalities that caused SSRF vulnerabilities and found 14 were related to download functionalities (involving plugin downloads, template codes, etc.), 5 were related to image editing, 8 were related to file management, and 1 was related to social networking.

Security Risk of Identified Vulnerabilities. Table 1 presents an overview of identified vulnerabilities across various applications, including associated CVE numbers (if available) and the corresponding CVSS scores from the National Vulnerability Database (NVD) [23]. Among the 16 vulnerabilities assigned CVE numbers, NVD has evaluated and classified 13 publicly disclosed vulnerabilities based on the Common Vulnerability Scoring System (CVSS) [6]. Three vulnerabilities were given a HIGH rating with a CVSS score of 8.8. The other ten vulnerabilities were rated as CRITICAL, with seven scoring 9.1 and three scoring 9.8 on the CVSS scale. The 25 new identified vulnerable applications include popular PHP applications, such as Joomla, Drupal, and Z-BlogPHP. The estimated number of websites deploying these three applications ranges from 22,763 to 128,638 sites [47].

False Negatives and False Positives. SSRFuzz identified 3 out of 4 known vulnerabilities. The reason for missing one vulnerability in WordPress is that SSRFuzz does not support crafting a payload in XML format. As this scenario is outside the scope of this study, as presented in Section 2.3, SSRFuzz is unable to identify this vulnerability. The detailed exploration of this particular vulnerability is further elaborated upon in Appendix A.2. To evaluate the false positive rate of our findings, we conducted a thorough manual review of the identified SSRF vulnerabilities. Our results show that all vulnerabilities were valid and exploitable, with no false positives observed.

5.4. Comparison with State-of-the-Art

We compare SSRFuzz with the state-of-the-art commercial black-box web application vulnerability scanner BurpSuite [29], and an open-source automatic SSRF fuzzing and exploitation tool SSRFmap [1]. We selected these tools due to their popularity in discovering SSRF vulnerabilities. Importantly, SSRFuzz also employs black-box fuzzing techniques in its fuzzing phase, making it a suitable candidate for direct comparison with these tools. To compare our approach with BurpSuite and SSRFmap, we focus our evaluation on two metrics commonly used in previous black-box fuzzing studies [57], [63]: how many vulnerabilities are discovered by each tool and how long it takes to find vulnerabilities in a web application.

Configuration. In setting up our comparative analysis, each tool's unique features were taken into account. Since SSRFmap lacks a web crawling feature, it cannot independently crawl websites and scan SSRF vulnerabilities in the process. To address this, we supplied SSRFmap with the requests found by the SSRFuzz web crawler and also automatically specified the HTTP parameter names in the requests for fuzzing by SSRFmap. BurpSuite was configured to a deep scan to achieve greater coverage and better understand a site's security posture, with the crawling process restricted to a maximum of 2.5 hours. Unlike the crawl, BurpSuite's audit process for identifying vulnerabilities did not have a timeout option and was allowed to run until it was completed. SSRFmap is designed to exploit SSRF vulnerabilities to attack web applications, so there is no dedicated detection module for SSRF vulnerabilities. So, we utilized with its `readfiles`, `network scan`, and `portscan` modules for scanning. Then, we manually assessed the results from these modules to pinpoint SSRF vulnerabilities. SSRFmap also did not have a timeout option and was allowed to run until it was completed. In addition, when using BurpSuite and SSRFmap for testing, the instrumentation module of SSRFuzz does not run to exclude the impact of the instrumentation module on their testing efficiency.

Application	Version	Files	LLoC	Vulns	Sink Func	Characterization	CVSS
Joomla [18]	4.3.4	7,878	976,161	Unassigned Unassigned Unassigned	curl_exec curl_exec curl_exec	Update Download Update Download Extended Download	N/A N/A N/A
Drupal [13]	8.9.20	19,490	1,520,484	Unassigned Unassigned	curl_exec curl_exec	Update Download Update Download	N/A N/A
phpBB [28]	3.3.10	3,593	416,143	Unassigned	curl_exec	Image Editing	N/A
Z-BlogPHP [46]	1.7.2	438	73,204	CVE-2022-40357	get_headers	Image Editing	9.8
WonderCMS [14]	3.1.3	20	3,563	CVE-2024-27561 CVE-2024-27561 CVE-2024-27563	curl_exec curl_exec curl_exec	Theme Download Extended Download Extended Download	N/A N/A N/A
CCV [20]	dca50eb	702	333,882	CVE-2022-31393	curl_exec	Image Editing	9.1
KityMinder [19]	v1.3.5	402	53,391	CVE-2022-31830	curl_exec	Image Editing	9.1
ChatGPT(Web Client) [5]	51eaaa4	19	11,043	CVE-2024-27564	file_get_contents	Image Editing	N/A
Ecommerce [3]	d7900dc	1,185	169,702	Unassigned	ftp_connect ϟ	File Management	N/A
rConfig [31]	3.9.4	271	76,440	CVE-2023-39108 CVE-2023-39109 CVE-2023-39110	file file file	File Management Configuration File Management Configuration File Management	8.8 8.8 8.8
WeBid [43]	1.2.2	653	90,101	CVE-2022-41477	fopen	File Management	9.1
MonstaFTP [21]	v2.10.3	4,440	344,093	CVE-2022-31827	curl_exec	File Management	9.1
JizhiCMS [15]	2.2.5	487	167,090	CVE-2022-31388 CVE-2022-31390 CVE-2022-31390 CVE-2022-27429	curl_exec curl_exec fopen curl_exec	Extended Download Template Download Template Download Extended Download	N/A 9.1 9.1 9.8
Nbnbk [2]	532bfdc	2,951	587,135	CVE-2022-31386	file_get_contents	File Management	9.1
iCMS [16]	v7	1,222	193,574	CVE-2022-41496	curl_exec	Resource Loading	9.8

TABLE 1: New SSRF vulnerabilities discovered by SSRFuzz. CVSS scores greater than 9 are considered critical vulnerabilities. ϟ indicates our newly discovered SSRF sink.

Application	BurpSuite	SSRFmap	SSRFuzz
WordPress	0(0)	0(0)	0(0)
Joomla	0	0	3
Drupal	0	0	2
phpBB	0	0	1
Z-BlogPHP	0	0	1
WonderCMS	0(0)	0(0)	4(1)
CCV	1	0	1
KityMinder	0	0	1
ChatGPT(Web Client)	0	1	1
Ecommerce	0	0	1
rConfig	0(0)	3(1)	4(1)
WeBid	0	0	1
MonstaFTP	0	0	1
JizhicCMS	4	0	4
Nbnbk	0	1	1
NavigateCMS	1(1)	0(0)	1(1)
iCMS	0	1	1
Total	6(1)	6(1)	28(3)

TABLE 2: Results of vulnerabilities discovered by BurpSuite, SSRFmap, and SSRFuzz across various web applications. Numbers in parentheses indicate the count of known vulnerabilities verified by each tool.

Performance Comparison. The time consumption by BurpSuite, SSRFmap, and SSRFuzz to detect vulnerabilities in web applications is comparatively presented in Figure 6. SSRFuzz emerges as the more time-efficient tool for detecting SSRF vulnerabilities, significantly outpacing BurpSuite and SSRFmap. Statistical analysis from our test dataset shows that SSRFuzz achieves an average efficiency improvement of 90.3% over BurpSuite and 70.4% over SSRFmap. This superior speed can be attributed to SSRFuzz's dynamic taint inference module, which facilitates rapid identification of potential vulnerabilities. The dynamic taint inference module is specifically engineered to quickly identify user-controllable parameters in HTTP requests that could lead to SSRF vulnerabilities. Once such parameters are confirmed, SSRFuzz progresses to the fuzzing stage, where it employs specialized payload generator and detector modules to verify the presence of SSRF vulnerabilities. In contrast, BurpSuite and SSRFmap indiscriminately perform black-box fuzzing on all input parameters, which is inherently more time-consuming. To ensure the robustness of the data, each web application underwent two rounds of testing, and the average time for vulnerability detection was calculated.

Vulnerability Discovery Comparison. We summarize the number of vulnerabilities found for each tool in this experiment in Table 2. SSRFuzz emerged as the most ef-

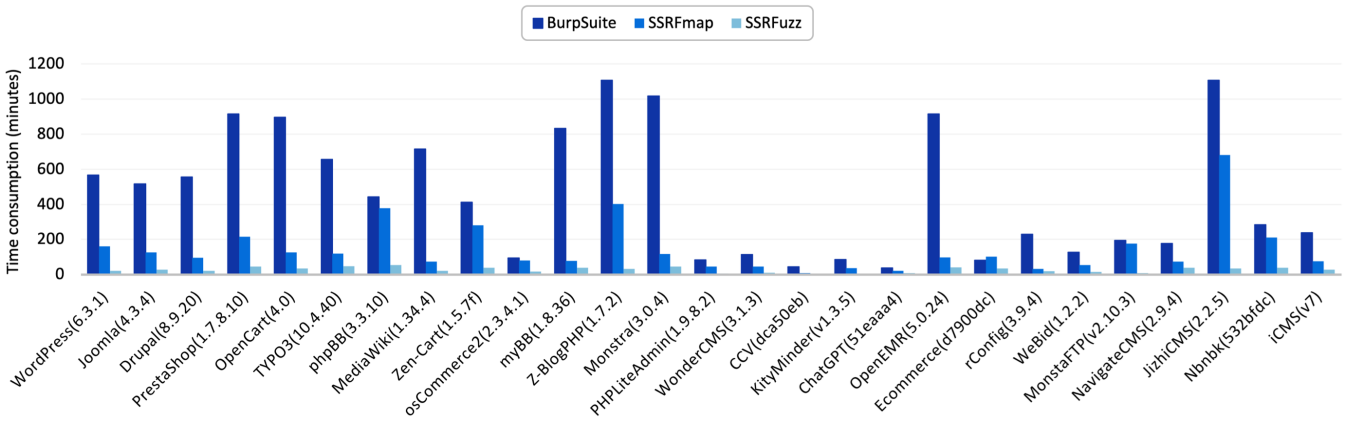

Figure 6: Time consumption comparison among BurpSuite, SSRFuzz, and SSRFmap across various web applications.

fective, discovering 28 vulnerabilities, whereas BurpSuite and SSRFmap detected only 6 each. There are two main reasons behind SSRFuzz's superior performance, particularly in detecting 0-day vulnerabilities. First is SSRFuzz's comprehensive payload generation strategies, tailored to various SSRF attack scenarios. Its mutation strategy is particularly effective in generating payloads that can bypass string-filtering checks, which was crucial in discovering the SSRF vulnerability in WonderCMS (more details in Section 5.5). In contrast, BurpSuite and SSRFmap rely on a static set of payloads and lack the capability for payload mutation, making them less effective against web applications equipped with string-filtering checks. The second reason for SSRFuzz's success is that it incorporates six different detection strategies, covering broader attack scenarios compared to BurpSuite's three strategies and SSRFmap's single strategy. While BurpSuite mainly uses the OAST method for detecting blind SSRF vulnerabilities, this approach is vulnerable to false negatives caused by network delays, as observed in its failure to detect the SSRF vulnerability in MonstaFTP in the initial test. SSRFuzz, however, mitigates this issue by employing a combination of network and file monitoring strategies.

Vulnerability Causes. The sinks for each SSRF vulnerability identified in our research are enumerated in Table 1. We have marked the newly discovered sinks in this study with a lightning symbol (ϟ). We can see that the newly discovered sinks enable SSRFuzz to discover a vulnerability previously overlooked by BurpSuite and SSRFmap. In addition, we quantified the use of various vulnerability detection strategies across two rounds of testing. The statistical analysis showed that the VD3 strategy was implemented with a likelihood of 1/7, highlighting its significance in identifying SSRF vulnerabilities associated with socket-oriented sinks such as `socket_connect` and `ftp_connect`.

5.5. Case Studies

We now discuss some interesting SSRF vulnerabilities that SSRFuzz detected.

5.5.1. WonderCMS. The SSRF vulnerability discovered by SSRFuzz in WonderCMS (3.1.3) [14] is shown in Listing 3. In this case, the input provided via the `pluginThemeUrl` parameter is assigned to the `$url` variable. It then passes through various functions, ultimately navigating to the `getFileFromRepo` function, where it becomes part of an HTTP request executed via `curl_exec`. The system's defensive mechanism, intended to validate URLs by identifying "https://github.com/" and "https://gitlab.com/" using the `strpos` function, proves insufficient. SSRFuzz skillfully bypasses this defense by constructing payloads containing these keywords, such as "http://vulndetect.server/#https://gitlab.com/fuzz." This instance serves as a dual revelation. Firstly, it exposes a notable deficiency in understanding SSRF vulnerabilities among web application developers. This is evident from their inability to create effective sanitization functions. Secondly, it demonstrates the capabilities of SSRFuzz in generating complex inputs tailored to trigger SSRF vulnerabilities in web applications.

5.5.2. WeBid. WeBid (1.2.2), an extensively used open-source auction software with over 100,000 downloads [43], was found to have a critical SSRF vulnerability by SSRFuzz. This vulnerability is associated with our study's identified SSRF sink called `fopen`. NVD classified this vulnerability as CRITICAL, with a CVSS score of 9.1. The code snippet in Listing 4 reveals an SSRF vulnerability within WeBid, resulting from inadequate security validation for user input in the `fromfile` parameter. This oversight permits unfiltered input to proceed directly into the sink function called `fopen`. Initially, the developer's intention was seemingly to employ `fopen` to check file existence, not realizing its capability to support multiple protocols (e.g., `http(s)://`, `ftp://`, `file://`), inadvertently introducing security threats. This vulnerability could be exploited by attackers for various malicious purposes, such as scanning internal networks or combining it with other vulnerabilities to launch more impactful attacks. This case underscores that SSRFuzz's identified sinks can unveil more SSRF vulnerabilities, which are often highly hazardous.

5.5.3. Joomla. Joomla (4.3.4) [18] is a web application that offers a wide range of sophisticated features, which contributes to its widespread popularity. The complexity of Joomla is primarily reflected in two aspects. First, the complexity of its codebase, which heavily relies on PHP's dynamic characteristics, poses challenges for static analysis tools. Second, the diverse types, large quantities, and various combinations of HTTP request parameters in Joomla expand the scope of fuzzing. In Joomla's program execution path, the value introduced by the HTTP request parameter `customurl` has to go through 17 function calls to reach the `request` function invoked by the `$uri` parameter. This `$uri` parameter is then used when the `curl_exec` function initiates the HTTP request. This complex program execution path poses a considerable challenge for static analysis tools. Especially in PHP web applications, it is difficult for static analysis tools to construct such complex sequences and find vulnerabilities within them. This case demonstrates SSRFuzz's ability to quickly point out input parameters that potentially trigger an SSRF vulnerability in complex web applications and utilize fuzzing techniques to reveal the SSRF vulnerability.

5.5.4. rConfig. rConfig (3.9.4) [31] is a robust network configuration management software used in enterprise environments that helps users efficiently manage configurations on heterogeneous networks. SSRFuzz discovered an SSRF vulnerability in rConfig. The application gets the user-specified input value through the HTTP parameter `path_a` and stores this input in the `$path_a` variable. Then, it uses the `diff` class to compare files referenced by `$path_a` and `$path_b`. The SSRF vulnerability emerges when the `doDiff` method in the `diff` class invokes the `file` function to read the file without sanitizing the `$path_a` value beforehand. This vulnerability is categorized as a high-security risk for two primary reasons. First, this SSRF vulnerability is caused by the `file` function, which supports using various protocols to initiate network requests, including `file://`, `http://`, and `ftp://`. Second, rConfig is an automated network management software, but by exploiting this SSRF vulnerability, we can scan internal networks and even read sensitive configuration files. Therefore, NVD has given it an 8.8 HIGH CVSS score. This vulnerability case demonstrates that the SSRF vulnerability can pose serious security threats in specific scenarios, especially to web applications with network management capabilities.

6. Discussion and Limitations

Languages supported. Security threats of SSRF, an adversarial object, have existed in not only PHP but also other programming languages, including Python [17], Java [30], Ruby [8], and .NET [33]. Depending on SSRF security risk, an adversary conducts various malicious behaviors. Although this paper is an in-depth study of SSRF vulnerability in PHP web applications, the approach of finding and modeling SSRF vulnerability in this paper can also be extended to other programming languages.

Fuzzing technology. We did not employ the coverage-guided fuzzing due to the consideration of balancing efficiency and the identification of vulnerabilities. Because we focus on testing SSR-related paths rather than the whole web application, coverage-guided fuzzing could inadvertently spend significant time probing paths unrelated to SSR functions. Prior studies underscore that such an approach can considerably extend fuzzing durations [82]. Therefore, we developed SSRFuzz to enhance the efficiency of vulnerability detection, utilizing dynamic taint inference and black-box fuzzing. The dynamic taint inference enables SSRFuzz to provide developers with the vulnerable code segments and the program execution path after discovering the SSRF vulnerability. Additionally, our black-box fuzzing approach, tailored for SSRF vulnerabilities, can autonomously generate PoCs. These insights assist developers in rapidly identifying and rectifying SSRF vulnerabilities and deepening their understanding of such vulnerabilities.

Web crawler.. The crawler component within SSRFuzz leverages crawlergo, an advanced open-source crawling tool. However, it inherits certain limitations from crawlergo, including the challenge of handling CSRF tokens and unique identifiers in forms and dynamic links. This limitation can lead to servers rejecting some requests by the server.

Mitigation. The premise of mitigation measures is that developers should review the purpose and intended use of the SSR functionality in their applications. It is crucial to assess whether allowing the web application to load arbitrary resources aligns with its intended behavior. If so, the mitigations developers can take include blocking network access from the web application to other internal systems and fortifying the web application defenses by removing access to services available on the local loopback adapter. If arbitrary resource loading is not part of the intended functionality, developers should establish an allow list and restrict access to any URLs not on this list. However, considering that attackers can use IP (e.g., 127.0.0.1) binding to allow list domains to bypass the allow list restrictions [4], it's essential for developers to verify that resolved IP addresses from user-provided domain names are on the allow list. To prevent DNS rebinding attacks, the DNS resolver used to validate the allow list should be the same as the resolver used to send the actual requests. Moreover, developers can further bolster security by implementing strict network-level policies that restrict outbound requests to known security domains and IP ranges, which can provide an additional layer of security.

Legality and ethicality. This research has not raised any legal or ethical issues. We downloaded the source code from the vendors for local analysis and responsibly reported all vulnerabilities to the CVE Numbering Authority (CNA) [10] and respective vendors. We have contacted all the developers for the SSRF vulnerability found in Section 5.3. To date, 11 vendors have acknowledged and fixed the reported vulnerabilities. We will continue to communicate with the other vendors throughout the vulnerability disclosure process.

7. Related Work

7.1. Web Application Vulnerabilities Discovery

There is a vast volume of studies on finding vulnerabilities in PHP applications, including XSS and SQL injection [62], [85], [86], [87]. Dahse and Holz implemented a tool called RIPS to detect bugs with taint analysis after precisely modeling PHP built-in functions [55]. Backes et al. [52] proposed an interprocedural analysis technique based on code property graphs to identify web application vulnerabilities utilizing programmable graph traversals. Huang et al. introduced WebSSARI to detect insecure information flow using a typestate-based static analysis algorithm [62], [64]. Xie et al. [86] presented a three-tier analysis for capturing information at the intra-block, intra-procedural, and inter-procedural levels. Pixy [61] performed additional alias and literal analysis to provide more comprehensive and precise results. Son et al. [77] presented static analysis techniques that identify semantic bugs and access-control bugs.

There are other works that use fuzzing techniques to find vulnerabilities. In essence, fuzzing is mostly a random process, and thus, vulnerability information-driven is crucial to the success of fuzzing, which significantly contributes to avoiding redundant testing [65], [84]. Eriksson et al. [58] propose a data-driven black-box scanning approach to enhance vulnerability scanning. Gauthier et al. [59] integrated state-aware crawling, type inference, coverage, and taint analysis into a black-box fuzzer to discover vulnerabilities efficiently. FUSE [63] finds file upload vulnerabilities in PHP applications by generating mutated upload requests from seed files, where mutations are designed to bypass content filtering checks while preserving executability constraints. FUGIO [71] is the first automatic exploit generation tool for PHP object injection vulnerabilities that identifies promising property-oriented programming chains and generates exploits through static analysis, dynamic analysis, and feedback-driven fuzzing. However, none of these efforts have delved into discovering SSRF vulnerabilities. Inspired by existing works, SSRFuzz has introduced several essential designs to discover SSRF vulnerabilities effectively, and the evaluation results have demonstrated their helpfulness.

Many symbolic execution studies have attempted to validate various types of vulnerabilities [49], [51], [77], [80], such as file upload, file inclusion, or SQL injection, but no prior study has addressed validating SSRF vulnerabilities. Huang et al. [60] conducted symbolic execution to assess the practicality of uploading arbitrary files with PHP-style extensions. They identified PHP built-in functions associated with file writing, for example, `move_uploaded_file`, as prospective vulnerability points. To achieve this, they formulated a reachability constraint to guarantee the accessibility of these functions from a tainted source, namely, `$_FILES`. Furthermore, they devised extension constraints to confirm the presence of PHP-style file extensions in the uploaded PHP files.

7.2. Server-Side Requests

There are few studies specifically dedicated to server-side requests, and even fewer explore the security risks of server-side requests. Orange Tsai [24] presented their findings in 2017 that differences in URL parsers in trending programming languages can lead to filter bypass and, thus, SSRF vulnerabilities in seemingly safe code implementations. Stivala and Pellegrino [79] in 2020 investigated how link previews on social media platforms can be manipulated to create benign-looking previews for malicious links. While the underlying link preview implementation uses server-side requests to obtain this information, SSR security was not studied as part of their paper. Marius Muschd et al. [67] conducted a wide-ranging study on the security of automated browsers running server-side in 2022 to reveal the security issues that such potentially vulnerable browsers can cause when requesting links provided by users. However, they still have not examined how the program itself contributes to the cause of the SSRF vulnerability. Pellegrino et al. [72] conducted a study on the security threats arising from the abuse of the server-side request (SSR) communication pattern. To evaluate the security implications of SSRs and the prevalence of this issue, they developed a black-box scanner called Guenther for detecting SSR misuses and analyzed the behavior of 68 online services. In contrast, our approach combines dynamic taint analysis and fuzzing to effectively discover SSRF vulnerabilities in 27 popular open-source web applications, which are deployed by millions of online websites across the Internet.

8. Conclusion

In this paper, we conducted the first systematic study of SSRF vulnerability in PHP. We presented an automated tool, SSRFuzz, a vulnerability-driven web fuzzer for SSRF vulnerabilities. SSRFuzz utilizes dynamic taint inference to detect potential SSRF vulnerability. We evaluated SSRFuzz on 27 real-world PHP applications. SSRFuzz discovered 28 SSRF vulnerabilities, including 25 previously unreported, and 16 new CVE IDs were assigned, demonstrating the practical utility of SSRFuzz in SSRF vulnerability detection. We hope our work can aid the community in addressing the rising threats of SSRF vulnerabilities.

9. Acknowledgement

We thank the anonymous reviewers for the helpful comments. This work is in part supported by the NSFC #62272265, the projects of National Key R&D Program of China (No.2021YFF0901000, No.2022YFB3104800), National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions and findings expressed in this material are those of the authors and do not reflect the views of employers or funding agencies.

References

[1] "Automatic ssrf fuzzer and exploitation tool," https://github.com/swisskyrepo/SSRFmap, accessed: June 28, 2023.

[2] "A b2c e-commerce open source php mall system platform," http://www.nbnbk3.com, accessed: June 28, 2022.

[3] "Bootsrap responsive multi-vendor, multilanguage online shop platform," https://github.com/kirilkirkov/Ecommerce-CodeIgniter-Bootstrap, accessed: June 28, 2023.

[4] "Bypassing ssrf protection," https://vickieli.medium.com/bypassing-ssrf-protection-e111ae70727b, accessed: June 28, 2023.

[5] "Chatgpt web client," https://github.com/dirk1983/chatgpt, accessed: June 28, 2023.

[6] "Common vulnerability scoring system," https://nvd.nist.gov/vuln-metrics/cvss, accessed: June 28, 2023.

[7] "crawlergo - a powerful browser crawler for web vulnerability scanners," https://github.com/Qianlitp/crawlergo, accessed: June 11, 2023.

[8] "Cve-2019-11027," https://security.snyk.io/vuln/SNYK-RUBY-RUBYOPENID-449661, accessed: Oct 11, 2022.

[9] "Cve-2022-25876," https://security.snyk.io/vuln/SNYK-JS-LINKPREVIEWJS-2933520, accessed: June 28, 2023.

[10] "Cve program," https://www.cve.org/About/Overview, accessed: Oct 11, 2022.

[11] "Dns rebinding attack: How malicious websites exploit private networks," https://unit42.paloaltonetworks.com/dns-rebinding/, accessed: June 28, 2023.

[12] "Dns rebinding based bypass," https://github.com/incredibleindishell/SSRF_Vulnerable_Lab, accessed: June 28, 2023.

[13] "Drupal," https://www.drupal.org/, accessed: June 28, 2023.

[14] "Fast and small flat file cms (5 files). built with php, json database," https://github.com/WonderCMS/wondercms, accessed: June 28, 2023.

[15] "A free and open source php website building cms system," https://github.com/Cherry-toto/jizhicms, accessed: June 28, 2023.

[16] "A free and open source php website building cms system," http://www.idreamsoft.com, accessed: June 28, 2023.

[17] "httpx is a the next generation http client. this package are vulnerable to server-side request forgery (ssrf)," https://security.snyk.io/vuln/SNYK-PYTHON-HTTPX-2805813, accessed: Oct 11, 2022.

[18] "Joomla," https://www.joomla.org/, accessed: June 28, 2023.

[19] "Kity minder," https://sourceforge.net/projects/kity-minder.mirror/, accessed: June 28, 2023.

[20] "A modern computer vision library," https://github.com/liuliu/ccv, accessed: June 28, 2023.

[21] "Monsta is your modern web interface for ftp," https://www.monstaftp.com/, accessed: June 28, 2023.

[22] "Multiple http redirects to bypass ssrf protections," https://infosecwriteups.com/multiple-http-redirects-to-bypass-ssrf-protections-45c894e5d41c, accessed: June 28, 2023.

[23] "National vulnerability database," https://nvd.nist.gov/vuln, accessed: June 28, 2023.

[24] "A new era of ssrf - exploiting url parser in trending programming languages!" https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf, accessed: Oct 11, 2022.

[25] "Openai api," https://platform.openai.com/docs/introduction, accessed: June 28, 2023.

[26] "Payloads all the things," https://github.com/swisskyrepo/PayloadsAllTheThings, accessed: June 28, 2023.

[27] "Php ssrf," https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/php-tricks-esp/php-ssrf, accessed: March 25, 2024.

[28] "phpbb," https://www.phpbb.com/, accessed: June 28, 2023.

[29] "Portswigger. burp suite." https://portswigger.net/burp, accessed: Aug 30, 2022.

[30] "Protecting against server side request forgery (ssrf)," https://neo4j.com/developer/kb/protecting-against-ssrf/, accessed: Oct 11, 2022.

[31] "rconfig is an advanced, modern and easy to use configuration management tool," https://www.rconfig.com/, accessed: June 28, 2023.

[32] "Server-side browsing considered harmful," https://www.agarri.fr/docs/AppSecEU15-Server_side_browsing_considered_harmful.pdf, accessed: June 28, 2023.

[33] "Server-side request forgery in .net," https://knowledge-base.securelag.com/vulnerabilities/server_side_request_forgery/server_side_request_forgery__net.html, accessed: Oct 11, 2022.

[34] "Server-side request forgery (ssrf)," https://portswigger.net/web-security/ssrf, accessed: Aug 29, 2023.

[35] "Server side request forgery (ssrf) in concrete5." https://security.snyk.io/vuln/SNYK-PHP-CONCRETE5CONCRETE5-72244, accessed: Oct 7, 2022.

[36] "Server-side request forgery (ssrf) vulnerability in updraftcentral dashboard," https://www.pluginvulnerabilities.com/2022/11/30/server-side-request-forgery-ssrf-vulnerability-in-updraftcentral-dashboard/, accessed: March 25, 2024.

[37] "Server-side request forgery (ssrf) vulnerable lab." https://github.com/incredibleindishell/SSRF_Vulnerable_Lab, accessed: Aug 30, 2022.

[38] "Ssrf to local file read." http://hassankhanyusufzai.com/SSRF-to-LFI/, accessed: Oct 7, 2022.

[39] "Ssrfire: An automated ssrf finder," https://github.com/ksharinarayanan/SSRFire, accessed: June 28, 2023.

[40] "Understanding the web vulnerability server-side request forgery," https://www.vaadata.com/blog/understanding-web-vulnerability-server-side-request-forgery-1/, accessed: March 25, 2024.

[41] "Usage statistics and market share of php for websites," https://w3techs.com/technologies/details/pl-php/all/all, accessed: June 28, 2023.

[42] "Usage statistics of content management systems," https://w3techs.com/technologies/overview/content_management, accessed: June 28, 2023.

[43] "Webid support," https://webidsupport.4up.eu/, accessed: June 28, 2023.

[44] "Wordpress," https://wordpress.org/, accessed: June 28, 2023.

[45] "xmark - a php7 extension that can hook most functions/classes and parts of opcodes," https://github.com/fate0/xmark, accessed: June 11, 2023.

[46] "Z-blogphp," https://github.com/zblogcn/zblogphp, accessed: June 28, 2023.

[47] "Zoomeye," https://www.zoomeye.org/, accessed: June 28, 2023.

[48] "CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022.

[49] G. Agosta, A. Barenghi, A. Parata, and G. Pelosi, "Automated security analysis of dynamic web applications through symbolic code execution," in *2012 Ninth International Conference on Information Technology-New Generations*. IEEE, 2012, pp. 189–194.

[50] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "Navex: Precise and scalable exploit generation for dynamic web applications," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 377–392.

[51] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 261–272.

[52] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[53] J. Benoist, "Safecurl," https://github.com/j0k3r/safecurl, accessed: Feb 27, 2023.

[54] D. Ceara, L. Mounier, and M.-L. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 371–380, 2010.

[55] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *NDSS*, vol. 14. Citeseer, 2014, pp. 23–26.

[56] Y. Dai and R. Resig, "Firedrill: interactive {DNS} rebinding," in *7th {USENIX} Workshop on Offensive Technologies ({WOOT} 13)*, 2013.

[57] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 37–48. [Online]. Available: https://doi.org/10.1145/2557547.2557550

[58] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021, pp. 1125–1142.

[59] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Experience: Model-based, feedback-driven, grey-box web fuzzing with backrest," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[60] J. Huang, Y. Li, J. Zhang, and R. Dai, "Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 581–592.

[61] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006, pp. 6–pp.

[62] S. Lee, S. Wi, and S. Son, "Link: Black-box detection of cross-site scripting vulnerabilities using reinforcement learning," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 743–754.

[63] T. Lee, S. Wi, S. Lee, and S. Son, "Fuse: Finding file upload bugs via penetration testing." in *NDSS*, 2020.

[64] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," in *Proceedings of the Web Conference 2021*, 2021, pp. 2721–2732.

[65] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah, "V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2020.

[66] MITRE, "Cve project by mitre," https://cve.mitre.org/cve/search_cve_list.html, accessed: June 28, 2023.

[67] M. Musch, R. Kirchner, M. Boll, and M. Johns, "Server-side browsers: Exploring the web's hidden attack surface," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1168–1181.

[68] M. Nashaat, K. Ali, and J. Miller, "Detecting security vulnerabilities in object-oriented php programs," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 159–164.

[69] ne555, "Multiple http redirects to bypass ssrf protections," https://infosecwriteups.com/multiple-http-redirects-to-bypass-ssrf-protections-45c894e5d41c, accessed: Feb 27, 2023.

[70] "Openrasp testcases," https://github.com/baidu-security/openrasp-testcases, accessed: Oct 7, 2021.

[71] S. Park, D. Kim, S. Jana, and S. Son, "Fugio: Automatic exploit generation for php object injection vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 197–214.

[72] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow, "Uses and abuses of server-side requests," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 393–414.

[73] "Pikachu vulnerability testing platform," https://github.com/zhuifengshaonianhanlu/pikachu, accessed: Oct 7, 2021.

[74] W. Qianqian and L. Xiangjun, "Research and design on web application vulnerability scanning service," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*. IEEE, 2014, pp. 671–674.

[75] "Uniform resource identifier (uri): Generic syntax." https://www.rfc-editor.org/rfc/rfc3986, accessed: Oct 7, 2021.

[76] SirLeeroyJenkins, "Just gopher it: Escalating a blind ssrf to rce for $15k — yahoo mail," https://sirleeroyjenkins.medium.com/just-gopher-it-escalating-a-blind-ssrf-to-rce-for-15k-f5329a974530, accessed: Feb 27, 2023.

[77] S. Son and V. Shmatikov, "Saferphp: Finding semantic vulnerabilities in php applications," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011, pp. 1–13.

[78] "A10 Server Side Request Forgery (SSRF) - OWASP Top 10:2021," https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/, 2022.

[79] G. Stivala and G. Pellegrino, "Deceptive previews: A study of the link preview trustworthiness in social platforms," in *NDSS*, 2020.

[80] F. Sun, L. Xu, and Z. Su, "Detecting logic vulnerabilities in e-commerce applications." in *NDSS*, 2014.

[81] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2022, pp. 116–133.

[82] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "webfuzz: Grey-box fuzzing for web applications," in *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer, 2021, pp. 152–172.

[83] "W3techs. 2022. usage statistics of php for websites." https://w3techs.com/technologies/details/pl-php, accessed: Oct 7, 2021.

[84] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.

[85] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 32–41.

[86] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *USENIX Security Symposium*, vol. 15, 2006, pp. 179–192.

[87] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014, pp. 590–604.

[88] M. Y. Yudi Zhao, Yuan Zhang, "Remote code execution from ssti in the sandbox:automatically detecting and exploiting template escape bugs," in *USENIX Security'23*, 2023.

Appendix A.

A.1. A prompt for test case generation

We manually review the code snippets generated by ChatGPT to ensure their accuracy. Our evaluation reveals that ChatGPT achieves an 89% success rate in generating test cases.

Test Case Generation Prompt

You are a world-class PHP developer with an eagle eye for unintended bugs and edge cases. You don't need to offer corrections or explanations, just generate the precise code requested.
A code example that uses the `file_get_contents` function to obtain a resource specified by a URL is:

```
<?php
$url=$_GET['url'];
$resourceContent =
              file_get_contents($url);
foreach ($resourceContent as $line) {
    echo $line;
}
```

Now, I need you to generate a code that uses the #{FuncName} function to obtain the resource specified by a URL. Suppose the function has information about the return value; output this information. Otherwise, nothing needs to be output. Don't explain the code or say extra words. Generate the code block itself.

Model Output

```
<?php
$url = $_GET['url'];
$handle = fopen($url, "r");
if ($handle) {
    while (($line = fgets($handle))
                              !== false){
        echo $line;
    }
    fclose($handle);
}
```

A.2. An SSRF vulnerability in WordPress.

```
<methodCall>
   <methodName>pingback.ping</methodName>
  <params>
    <param>
      <value><string>http://evil.tld</string></value>
    </param>
    <param>
      <value><string>http://blog.tld/?p=1</string></
       value>
    </param>
  </params>
</methodCall>
```

Listing 2: An XML data that can trigger SSRF vulnerability in WordPress

Due to an insufficient comprehension of WordPress code, tools such as BurpSuite, SSRFmap, and SSRFuzz are incapable of constructing XML data in the requisite format (details in Listing 2), nor do they facilitate the generation of "`pingback.ping`" calls. This limitation precludes them from triggering and consequently identifying the SSRF vulnerability in this case.

A.3. An SSRF vulnerability in WonderCMS.

```
<?php
// index.php
public function addCustomThemePluginRepository(): void
  {
    ...
    $url = rtrim(trim($_POST['pluginThemeUrl']), '/');
    ...
    switch (true) {
      case strpos($url, 'https://github.com/') ===
     false && strpos($url, 'https://gitlab.com/') ===
     false:
        $errorMessage = 'Invalid repository URL. Only
     GitHub and GitLab are supported.';
    ...
    $this->cacheSingleCacheThemePluginData($url, $type
     );
    ...
}
...
//index.php
public function getFileFromRepo(string $file, string
     $repo = self::WCMS_REPO): string
  {
    $repo = str_replace('https://github.com/', 'https
     ://raw.githubusercontent.com/', $repo);
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_URL, $repo . $file);
    $content = curl_exec($ch);
    ...
  }
```

Listing 3: An SSRF vulnerability in WonderCMS. The code is simplified for demonstration purposes.

A.4. An SSRF vulnerability in WeBid.

```
<?PHP
// getthumb.php
...
$fromfile = (isset($_GET['fromfile'])) ? $_GET['
     fromfile'] : '';
...
// control parameters and file existence
if (!isset($_GET['fromfile']) || $fromfile == '') {
    ...
} elseif (!file_exists($_GET['fromfile']) && !fopen(
     $_GET['fromfile'], 'r')) {
    ...
}
...
```

Listing 4: An SSRF vulnerability in WeBid. The code is simplified for demonstration purposes.

A.5. Payloads and Operations

TABLE 3: List of probe payloads and the SSRF oracle of each payload.

OP	Probe Payload	SSRF oracle
O1	file:///flag	Accessing flag files
O2	dict://127.0.0.1:8181/info	Monitoring Server receives the request
O3	sftp://127.0.0.1:8181/	Monitoring Server receives the request
O4	ldap://127.0.0.1:8181/	Monitoring Server receives the request
O5	tftp://127.0.0.1:8181/	Monitoring Server receives the request
O6	gopher://127.0.0.1:8181/_POST%20/flag%20HTTP/1.1%0D%0A	Monitoring Server receives the request
O7	ssh://127.0.0.1:22/	Accessing the port monitored by Monitoring Server
O8	http://127.0.0.1:8181/	Monitoring Server receives the request
O9	https://127.0.0.1:8181/	Monitoring Server receives the request
O10	expect://127.0.0.1:8181/	Monitoring Server receives the request
O11	ogg://127.0.0.1:8181/	Monitoring Server receives the request
O12	ftp://127.0.0.1:8181/	Accessing the port monitored by Monitoring Server
O13	php://filter/resource=http://127.0.0.1:8181/	Monitoring Server receives the request
O14	compress.bzip2://http://127.0.0.1:8181/archive.gz	Monitoring Server receives the request
O15	compress.zlib://http://127.0.0.1:8181/myarchive.gz	Monitoring Server receives the request
O16	zip:///flag#bar	Accessing flag files
O17	rar://%2Fflag#file.txt	Accessing flag files
O18	phar:///flag.phar	Accessing flag.phar files
O19	glob://flag_dir	Accessing flag_dir directory
O20	/flag	Accessing flag files
O21	IP:PORT	Accessing the port monitored by Monitoring Server

TABLE 4: List of filling operations for each URL component.

OP	Description	Component
F1	Filling in the protocols supported by PHP (e.g. http, ftp, etc.)	Scheme
F2	Filling in 127.0.0.1	Userinfo,Host/Domain
F3	Filling in the domain name or IP address of the server that has the 302 redirection function	Userinfo, Host/Domain
F4	Filling in the IP address or domain name of the HTTP OOB server	Userinfo, Host/Domain
F5	Fill in the subdomain of the DNS OOB server, where the third-level domain name is randomly generated characteristic characters.	Userinfo,Host/Domain
F6	Filling in the IP address or domain name of the Port monitor.	Userinfo,Host/Domain
F7	Filling in the domain name or IP address extracted from the source code.	Userinfo,Host/Domain
F8	Filling in the Top 10 ports (21, 22, 25, 80, 443, 1433, 5432, 3306, 8080, 6379).	Port
F9	Filling in the randomly generated feature string.	Path
F10	Filling in the file path ”/etc/passwd”	Path
F11	Filling in the file path ”/flag” (A file that is being monitored for access status)	Path

TABLE 5: List of mutation operations for each initial payload.

OP	Description	Explanation
M1	Replacing 127.0.0.1 with localhost	a presentation format for IP
M2	Replacing 127.0.0.1 with 0.0.0.0	a presentation format for IP
M3	Replacing 127.0.0.1 with [::] or [::1]	IPv6 addresses
M4	Replacing 127.0.0.1 with [0:0:0:0:0:ffff:127.0.0.1]	IPv6 addresses
M5	Replacing 127.0.0.1 with 0000::1	IPv6 addresses
M6	Replacing 127.0.0.1 with 127.127.127.127	a presentation format for IP
M7	Replacing 127.0.0.1 with 127.0.1.3	a presentation format for IP
M8	Replacing 127.0.0.1 with 127.0.0.0	a presentation format for IP
M9	Replacing 127.0.0.1 with 0	shortened IPs
M10	Replacing 127.0.0.1 with 127.1	shortened IPs
M11	Replacing 127.0.0.1 with 127.0.1	shortened IPs
M12	Converting IP address to dotless decimal	a presentation format for IP
M13	Converting IP address to dotted octal	a presentation format for IP
M14	Converting IP address to dotted octal with padding	a presentation format for IP
M15	Converting IP address to dotted hexadecimal	a presentation format for IP
M16	Converting IP address to dotless hexadecimal	a presentation format for IP
M17	Convert only parts of the IP address to decimal/octal/hexadecimal	a presentation format for IP
M18	IPv4-compatible IPv6 address	IPv6 addresses
M19	IPv4-mapped IPv6 addressess	IPv6 addresses
M20	Inserting %09 into random position	URL encoding for tab
M21	Inserting %0A into random position	URL encoding for linefeed
M22	Inserting %0D into random position	URL encoding for creturn
M23	Inserting %3A into random position	URL encoding for ":"
M24	Inserting %3B into random position	URL encoding for ";"
M25	Converting characters to enclosed alphanumerics	PHP will parse enclosed alphanumerics into numbers
M26	Single encode the characters in the Path component randomly	Bypassing Deny Lists with PHP Decoding Features
M27	Double encode the characters in the Path component randomly	Bypassing Deny Lists with PHP Decoding Features
M28	Replacing scheme with 0://	Bypassing URL validation of the filter_var() function
M29	Inserting feature characters and request object combinations at random positions between URL components.	Bypass access control policies
M30	insert File Suffix in Path Component	Bypassing validation of URL access to file extensions
M31	Splicing ".xip.io" after the normal domain name or IP	xip.io is a magic domain name that provides wildcard DNS for any IP address.

Feature characters include (`"."`,`"!"`, `"?"`,`" "`,`"/"`,`"#"`,`";"`,`":"`;`"@"`,`"&"`, `"%ff@"`,`"%ff."`,`"%25253F@"`,`"%253F@"`,`"%3F@"`,`"@"`,`":80@"`,`"%20@"`,`"%09@"`,`"%252F@"`)
Request Object in Table includes the IP and domain name of the 302 redirection server, the IP and domain name extracted from the source code, the server IP address of HTTP Out Of Band (OOB), the server domain name of DNS OOB, and "127.0.0.1".

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper introduces and evaluates SSRFuzz, a tool to automatically detect SSRF vulnerabilities in PHP applications. The approach consists of three phases: a comprehensive detection of PHP SSRF sink functions using a semi-automated strategy supported by LLMs; dynamic taint tracking based on PHP code instrumentation to identify data flows from user inputs to SSRF sinks; mutation-based fuzzing to discover SSRF vulnerabilities. SSRFuzz is evaluated on 27 real-world PHP applications and compared against existing tools (Burp Suite and SSRFmap). The tool identified 28 vulnerabilities (25 are novel), obtaining 16 CVEs.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability

B.3. Reasons for Acceptance

1) The paper presents an effective tool to detect SSRF vulnerabilities in PHP applications. Based on this work, future research can be conducted to improve the performance of the tool (e.g., reduce false negatives by supporting additional payloads), detect other classes of vulnerabilities, and target additional programming languages.
2) SSRFuzz identifies multiple impactful SSRF vulnerabilities in popular PHP applications, including Joomla, Drupal, and phpBB, resulting in 16 CVEs.

B.4. Noteworthy Concerns

1) The novelty of the reported SSRF sinks is not well justified. Non-academic research has already identified many of the sinks the authors claim to be novel. Furthermore, the methodology is similar to existing work in Web application fuzzing and does not compare with previous research on SSRF detection (Pellegrino et al., Uses and Abuses of Server-Side Requests. RAID 2016).
2) The evaluation dataset comprises 27 applications, which is relatively small to draw general conclusions about the effectiveness and generalizability of SSRFuzz. It is unclear whether the application developers confirmed all reported vulnerabilities and the paper does not precisely characterize the discovered issues.
3) False negatives (FNs) and false positives (FPs) are not thoroughly discussed. For instance, monitored requests (e.g., DNS requests) may be the effect of legitimate functionality rather than revealing the presence of SSRF vulnerabilities

Appendix C. Response to the Meta-Review

We sincerely thank the reviewers for their valuable feedback. In response to the noteworthy concern:

1) We have open-sourced the identified SSRF sinks on Github to justify their novelty. We have added a comparison with Pellegrino's work in the related work to clarify the differences.
2) In our methodology, we utilize white-box testing approaches that necessitate the local deployment of the target web applications, leading to fewer applications being tested compared to black-box testing methods. Among the 27 web applications we tested, 17 are popular and complex, including well-known platforms such as WordPress and Drupal. Additionally, we tested 10 other applications previously evaluated in earlier studies, which could demonstrate the effectiveness and generalizability of SSRFuzz.
3) For false negatives, SSRFuzz can identify 3 out of 4 known vulnerabilities, missing 1 vulnerability. For false positives, we have manually reviewed all the vulnerabilities and found all vulnerabilities can be exploitable, with no false positives.