

Detecting Server-Side Request Forgery (SSRF) Vulnerabilities In REST API Fuzz Testing

Susruthan Seran
Kristiania University of Applied
Sciences
Oslo, Norway

Guru Prasad Bhandari
Kristiania University of Applied
Sciences
Oslo, Norway

Andrea Arcuri
Kristiania University of Applied
Sciences
Oslo, Norway
Oslo Metropolitan University
Oslo, Norway

Abstract

The entangled internet reveals more vulnerable situations in modern interconnected software systems. Among these, Server-Side Request Forgery (SSRF) is one marked as a top ten vulnerability to look for in the OWASP Top 10 rankings. There has been a significant amount of work in academia to detect SSRF.

In this paper, we present our novel techniques for detecting SSRF in REST APIs, utilizing search-based techniques in a fully automated manner. Our work offers a unique language-agnostic approach, which can be adapted into other software fuzzers for both white-box and black-box contexts.

CCS Concepts

• **Software and its engineering** → Search-based software engineering; **Software verification and validation**.

Keywords

SSRF, SBSE, SBST, fuzzing

ACM Reference Format:

Susruthan Seran, Guru Prasad Bhandari, and Andrea Arcuri. 2026. Detecting Server-Side Request Forgery (SSRF) Vulnerabilities In REST API Fuzz Testing. In *19th Search-Based and Fuzz Testing Workshop (SBFT '26)*, April 12--18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3786155.3788581>

1 Introduction

The ever-evolving internet landscape has pushed software systems to become more interconnected, changing the architecture from monoliths to distributed microservices since the advent of the internet. This evolution introduced greater complexity, making it more vulnerable and prone to errors. In parallel, techniques to find such vulnerabilities are also evolving. Modern-day web applications have a significant number of identified weaknesses, including Server-Side Request Forgery (SSRF) [12], which can lead to unauthorized information retrieval or catastrophic failure of the software system.

The inclusion of SSRF as one of the top ten vulnerabilities in the OWASP Top 10 2021 is an indication of the gravity of the impact [14]. SSRF appears as well in the top 10 vulnerabilities

for APIs (OWASP 2023 [15]). For example, in a reported Common Vulnerability Exposure (CVE-2024-35451) [13] related to the LinkStack¹ shows that a simple SSRF could lead to other Common Weakness Enumeration (CWE) linked to Remote Code Execution (RCE) [10, 11]. Recognising the potential impact of SSRF, several attempts have been made to detect it in advance, both in academia and the industry.

In this paper, we present a novel technique for detecting standard SSRF using a fully automated search-based approach in both white-box and black-box contexts. We developed our techniques as an academic proof-of-concept. Therefore, we will focus on detecting *standard* and *blind* SSRFs, but leaving the *second-order* SSRF for future research. Furthermore, implementing a fuzzer is a major engineering endeavour; due to this, we extended an existing open-source, state-of-the-art fuzzer named EvoMASTER² rather than implementing a new fuzzer from scratch. Based on the existing literature, EvoMASTER performs best compared to other fuzzers for Web APIs [9, 16, 17, 20]. However, our techniques can be adapted to other existing REST API fuzzers, as they are not tailored to EvoMASTER.

Experiments are carried out on 4 artificial APIs and 2 real-world APIs with known SSRF vulnerabilities. In all cases, our novel techniques were able to automatically detect these SSRF vulnerabilities. Furthermore, our techniques are able to generate executable test cases in JUnit format that can reproduce the SSRF exploits, where a local (mock) server is automatically instantiated by the generated test cases themselves to show the malicious connections to it made by the tested APIs.

The remainder of this article is structured as follows. First, Section 2 provides background information and outlines the motivation for this research. Following that, Section 3 then presents a review of relevant literature. Section 4 details our novel techniques. Section 5 discusses the empirical study conducted to evaluate the effectiveness of our techniques. Section 6 addresses potential threats to the validity of this work. Finally, Section 7 concludes the paper and discusses directions for future research.

2 Background and Motivation

2.1 Server-Side Request Forgery (SSRF)

Figure 1 contains a code snippet taken from one of our real-world case studies with reported CVE, Lychee.³ This was part of a function that allows the user to import images from external



This work is licensed under a Creative Commons Attribution 4.0 International License. *SBFT '26, Rio de Janeiro, Brazil*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2387-2/2026/04

<https://doi.org/10.1145/3786155.3788581>

¹<https://linkstack.org/>

²<https://github.com/WebFuzzing/EvoMaster/tree/master>

³<https://lycheeorg.dev/>

```

1 // If the component parameter is specified, this
  function returns a string (or int in case of
  PHP_URL_PORT)
2 /** @var string $path */
3 $path = parse_url($url, PHP_URL_PATH);
4 $extension = '.' . pathinfo($path, PATHINFO_EXTENSION)
  ;
5
6 if ($extension !== '.') {
7   // Validate photo extension even when `create->
  add()` will do later.
8   // This prevents us from downloading unsupported
  files.
9   BaseMediaFile::
    assertIsSupportedOrAcceptedFileExtension(
      $extension);
10 }
11
12 // Download file
13 $downloaded_file = new DownloadedFile($url);
14

```

Figure 1: Code snippet taken from one our real-world case study, *Lychee*.

sources. The application checks whether the provided URL is an image using the file extension as the indicator in Line 9. However, during our manual inspection, we successfully downloaded a PHP script from our remote server masquerading as a JPG file through the function in Line 13, and the downloaded file is also publicly accessible. The application has a request rewrite rule set for *nginx*,⁴ which prevents further exploitation in this case. In addition, we were able to call other services running on the same machine and on the same network through this vulnerability in the application, but our ability to exploit it further was limited by the application’s functionality on the affected function.

In the following subsections, we provide a detailed discussion of the common SSRF attacks.

2.2 Standard SSRF Attacks

In typical Server-Side Request Forgery (SSRF) attacks, vulnerabilities in an application are exploited to make the application send requests to external servers within the same network, or to internal resources such as files or services located on the same host behind a firewall. This could lead to information retrieval from resources behind the firewall, or downloading a malicious file from an attacker-controlled server, or even, in some cases, executing malicious commands.

Accessing resources on the internet from user’s provided URLs can be part of the normal, expected behavior of the API. However, such URLs must be verified, e.g., to avoid making calls to local servers such as localhost and 127.0.0.1 (or host.docker.internal if the API is running inside Docker), and to avoid inadvertently enable local file access via file:// URL protocol. Failure to validate the provided URLs can lead to SSRF exploits.

⁴<https://nginx.org/>

2.3 Blind SSRF Attacks

In blind SSRF, an attacker manipulates a vulnerable application to initiate a request to an unauthorized location; however, the response from the remote server is not relayed to the attacker. As a result, blind SSRF is typically more difficult to detect and exploit than standard SSRF if one does not have direct access to the contacted server.

2.4 Second-order SSRF Attacks

Similar to blind SSRF mentioned in Section 2.3, second-order SSRF attacks are also harder to detect. In second-order SSRF attacks, the malicious URL is stored (e.g., in a database) when received, and not used immediately. Following calls to other endpoints, or running background jobs, might retrieve such malicious URL at a later time, and use it, leading to an SSRF vulnerability.

2.5 Uncommon Attack Surfaces

In most cases, identifying attack surfaces that may result in standard SSRF or other SSRF variants is straightforward, as these typically involve parameters that accept URLs as input. However, there are some unusual attack surfaces in the application that can be used for SSRF attacks.

For example, URLs embedded in data exchange formats such as XML can introduce vulnerabilities. Parsing XML documents may result in XML External Entity (XXE) attacks, which in turn can expose the application to SSRF.

Additionally, some applications process the HTTP Referer header for analytics purposes. Attackers can manipulate this header to exploit the application for SSRF.

3 Related Work

There are several proposed methods in the academic literature with regard to automated vulnerability discovery [6–8, 18, 19]. Instead of discussing all related works on SSRF, we focus on automated SSRF detection for the Representational State Transfer (REST) Application Programming Interfaces (API). Other contexts, like for example frontend web applications, are not in scope of this paper. For example, the study [18] presented a technique to detect SSRF using dynamic tainting in a PHP web applications. However, the study focuses only on PHP applications having a web GUI frontend.

To the best of our knowledge, there are only two existing approaches in the research literature close to the work presented in this paper in terms of detecting SSRF for REST APIs [7, 19].

The work in [19] presents a general approach to test all different kinds of web APIs, and not just REST, stating it can detect SSRF vulnerabilities as well. But no information on how such checks are done, and how/if they are automated, is provided in [19]. As the tool is no longer available (referenced link⁵ on GitHub seems broken), it is not possible to verify what is done for SSRF by checking its source code.

Another study on detecting security vulnerabilities in REST APIs [7] presents an approach that can detect SSRF. However, such an approach is not fully automated, as it requires to incorporate

⁵https://github.com/apif-tool/APIF_tool_2024

an expert review step in the process to manually verify if the SSRF exploits were successful.

Existing work falls short in terms of a fully automated, language-agnostic approach for SSRF detection in REST APIs. In contrast, the work presented in this paper provides a fully automated solution, which is able to generate executable test cases that can reproduce and verify the presence of SSRF vulnerability (if any is present in the tested API).

4 Automated SSRF Detection

In this section, we discuss our novel techniques to detect SSRF for REST APIs using a fully automated approach. Our novel SSRF detection techniques are implemented in the search-based fuzzer EvoMASTER, but any other fuzzer could have been used. The detection process is divided into five stages: *fuzzing*, *selection*, *identification*, *execution* and *verification*.

In the *fuzzing* phase, the fuzzer is run as usual, aiming at generating a final, small test suite with high coverage (e.g., HTTP coverage and code coverage) and functional fault detection. In the fuzzing literature, this is typically run with search budgets such as 1 hour or 24 hours. The point here is that, typically, generating test cases for happy-day scenarios (e.g., returning HTTP status code in the 2xx family) is a complex task. There might be several constraints in the inputs that could lead to user errors (e.g., HTTP 400 status code). If the software code dealing with external service calls (which might be susceptible to SSRF exploits) is never executed because input validation fails, then there would be no possible effect when sending malicious SSRF payloads. Sending malicious SSRF payloads during the fuzzing session would likely be inefficient. Our approach is to first generate high coverage tests and, then, use those tests as starting templates to apply SSRF attacks.

In the *selection* phase, once the main fuzzing phase is over, we extract test cases from the final test suite that meet specific criteria. For each endpoint defined in the OpenAPI schema of the API, we extract one test case that has at least one HTTP call toward that endpoint (recall a test case can be composed of several HTTP calls), and that has as response a status code in the HTTP 2xx family. If more than one test case fits such criteria, we select the shortest in terms of the number of HTTP calls in it. In case of ties (i.e., same min length), we pick randomly. If for any reason, no test case is found with a 2xx, we look at test cases that return 400 (generic user error) and 422 (unprocessible content). Likely, those calls have fewer chances to lead to execute the code susceptible to SSRF, but it is not impossible (as we have seen in some preliminary study). However, likely there would be no much point in trying with other 4xx calls that result for example in 401 (not authenticated), 403 (not authorized) or 404 (not found).

After the test case selection, we proceed to the *identification* stage, where the selected test cases for each endpoint are used to identify calls with string input parameters that are likely to take a URL as a value. Trying SSRF payloads on every possible string inputs would be inefficient. So we use a strategy to concentrate only on the input parameters that are likely treated by the API as URLs. To determine if the input can be a URL, we use the parameter name and its description as provided in the SUT's OpenAPI specification.

We analysed the corpus from APIs Guru⁶ and selected input names that are indicated to be URLs. Using this corpus, we created regular expression (regex) patterns to identify potential inputs from the parameter name and description. In the next step, we use a deterministic approach, utilising the created regex patterns, to classify the input parameters.

In the *execution* phase, we filter all the selected test cases that have at least one identified input parameter as a potential URL. We create clones/duplicates of the test cases for each identified URL parameter. Cloning is done to avoid any negative impact on the existing search results (i.e., the final output test suite). In the cloned test cases, we select only the parameters identified as potentially being URLs. For each of these cloned tests, we update the value of the identified URL parameter with a SSRF payload. The SSRF payload is a HTTP callback URL on the same localhost. Hereafter, we compute the fitness of this cloned and modified test case by executing it. In the next phase, if a HTTP call has parameters marked as potential URLs and could potentially lead to an SSRF fault, it will also be flagged as finding an SSRF fault.

In the *verification* phase, when the cloned tests with SSRF payloads are executed, to validate the vulnerable endpoints, we need to automatically determine if the SSRF exploit was successful. In our approach, we run a dynamically configurable mock HTTP server. Instead of writing our own mock HTTP server, we employed WireMock⁷ for this purpose. We created a dynamically configurable HTTPCallbackVerifier, where the SSRF payloads point to it. Once a test case with a SSRF payload is executed, if any call was made toward the WireMock instance, then the SSRF exploit was successful. If so, a SSRF fault is found, and these cloned/modified tests are added to the final test suite generated as output of the fuzzer.

Upon completion of this process, test suites are generated (e.g., in JUnit format) with WireMock instantiated and a self-contained HTTPCallbackVerifier to automatically validate the exploitability of SSRF.

Furthermore, this approach might enable us to further expand this functionality, allowing us to test for *chained* attacks through SSRF. We leave this for our future work, and we briefly discussed this possibility under Section 7.

An important clarification here is that our approach is for *security testing* of REST APIs, and not for *hacking*. To automatically verify that a SSRF was successful, and to generate executable test cases (e.g., in JUnit format) that can flag such issue and be able to reproduce it, the fuzzer has to run on the same network of the tested API. This is not a problem when the user of the fuzzer is a developer of the API (e.g., running the fuzzer in CI [3]) or a QA engineer inside an enterprise tasked to create test cases for the APIs developed there [4]. However, our approach would not directly work when hacking an external API on the internet, without direct access to its local network. On the one hand, a hacker who does an SSRF attack would then need to find a way to detect if it was successful and a way to exploit it. On the other hand, a security tester just needs to discover that an endpoint is vulnerable to SSRF, so that such a security fault can be fixed before a way to exploit it maliciously is found.

⁶<https://apis.guru/>

⁷<https://wiremock.org/>

```

1 httpCallbackVerifier55528.resetAll()
2 httpCallbackVerifier55528.stubFor(
3   get("/EM_SSRF_0")
4     .withMetadata(Metadata.metadata().attr("ssrf",
5       "POST:/api/fetch"))
6     .atPriority(1)
7     .willReturn(
8       aResponse()
9         .withStatus(200)
10        .withBody("SSRF")
11    )
12
13 // Verifying that there are no requests made to
14   HttpCallbackVerifier before test execution.
15 assertFalse(httpCallbackVerifier55528
16   .allServeEvents
17   .filter { it.wasMatched && it.stubMapping.metadata
18     != null }
19   .any { it.stubMapping.metadata.getString("ssrf")
20     == "POST:/api/fetch" }
21 )
22

```

Figure 2: Code snippet of setting up the HttpCallbackVerifier and the mock server taken from the generated test suite for the synthetic case study, *Base*.

5 Empirical Study

To evaluate our novel techniques presented in this paper, we conducted an empirical study to answer the following research questions:

RQ1: Can our novel techniques automatically detect SSRF security faults in synthetic APIs?

RQ2: Can our novel techniques automatically detect SSRF security faults in reported vulnerabilities from real-world APIs?

To answer our two research questions, we carried out two different sets of experiments, one per research question, as shown in Table 1 and Table 2.

5.1 Synthetic APIs

To answer **RQ1**, the first set of SUTs consists of three synthetic REST APIs, developed by us, as listed in Table 1. This set includes REST APIs that encompass three distinct SSRF scenarios. The *Base* application addresses standard server-side request forgery in the HTTP request body parameter. This application receives a URL and issues a request to retrieve remote data. Similar to *Base*, the *Query* application also performs similar actions while the vulnerable parameter is in the HTTP request query. In contrast to the other two applications, the *Header* application does not return any information in response to the attacker's request. Instead, it uses the value in the HTTP Referer header to make a call to the remote location, simulating crawling for analytics.

In all three cases, by employing our novel techniques, we were able to produce tests that detect the fault and provide evidence of exploitation.

Figure 2 presents the beginning of a test selected from the generated test suite for the synthetic case study *Base*. As the first steps, HttpCallbackVerifier will be reset to ensure the verifier is clear

```

1 // Fault202. Server-Side Request Forgery (SSRF).
2   sensorUrl.
3   given().accept("/*/*")
4     .header("x-EMextraHeader123", "")
5     .contentType("application/json")
6     .body(" { " +
7       "  \"sensorUrl\": \"http://localhost:55528/
8       EM_SSRF_0\" " +
9       " } ")
10    .post("${baseUrlOfSut}/api/fetch")
11    .then()
12    .statusCode(200)
13    .assertThat()
14    .contentType("text/plain")
15    .body(containsString("OK"))
16

```

Figure 3: Code snippet demonstrates the call made to the vulnerable endpoint in *Base*.

```

1 // Verifying that the request is successfully made to
2   HttpCallbackVerifier after test execution.
3   assertTrue(httpCallbackVerifier55528
4     .allServeEvents
5     .filter { it.wasMatched && it.stubMapping.metadata
6       != null }
7     .any { it.stubMapping.metadata.getString("ssrf")
8       == "POST:/api/fetch" }
9   )
10

```

Figure 4: The final assertion to ensure the exploitability of SSRF in the vulnerable endpoint in *Base*.

in terms of served requests before the actual call made to the SUT. After that, in Line 14, an assertion is performed to make sure that there are no requests made by the SUT and served related to the RestCallAction. Further down on the execution, the stub for the request payload will be configured in the mock server with a unique value in the URL path, as in Line 2. This configuration enables identification of the specific request associated with the vulnerable endpoint when the application interacts with the mock server.

In Figure 3 and in Line 9, the test suite initiates a call to the vulnerable endpoint and verifies the response in subsequent steps.

Finally, as shown in Figure 4 in Line 2, an assertion is performed to ensure that the request has been made to the HttpCallbackVerifier, thereby proving the existence of SSRF and exploitability.

In addition to the listed synthetic APIs in Table 1, we developed another open-source case study to test our novel techniques in black-box settings. *Clerk*⁸ is an open-source REST API being developed to demonstrate software vulnerabilities in a real-world context, that can be used for testing and education purposes. We used this open-source API (having 12 endpoints) to study whether our techniques would work in more complex scenarios. Figure 5 contains the generated test for *Clerk* in black-box, illustrating the whole test execution flow. We were able to detect SSRF in the endpoint to fetch the product image from a remote location. Apart from the reported steps in the previous examples, as in Figure 5,

⁸<https://github.com/seran/clerk>

Table 1: Descriptive statistics of the employed synthetic SUTs. For each SUT, #Endpoints represents the number of declared HTTP endpoints in those APIs. #Path represents the source code path in the E2E test folder of our extension of EvoMASTER.²

SUT	#Endpoints	#Path
<i>Base</i>	1	com/foo/rest/examples/spring/openapi/v3/security/ssrf/base
<i>Header</i>	1	com/foo/rest/examples/spring/openapi/v3/security/ssrf/header
<i>Query</i>	1	com/foo/rest/examples/spring/openapi/v3/security/ssrf/query

```

1 /**
2 * Calls:
3 * (422) POST:/api/product/fetch/image
4 * Found 1 potential fault of type-code 202
5 */
6 @Test @Timeout(60)
7 fun test_7_postOnImageVulnerableToSSRF() {
8
9     assertNotNull(httpCallbackVerifier59917.isRunning)
10
11     httpCallbackVerifier59917.resetAll()
12     httpCallbackVerifier59917.stubFor(
13         get("/EM_SSRF_2")
14             .withMetadata(Metadata.metadata().attr("ssrf", "POST:/api/product/fetch/image"))
15             .atPriority(1)
16             .willReturn(
17                 aResponse()
18                     .withStatus(200)
19                     .withBody("SSRF")
20             )
21     )
22
23     // Verifying that there are no requests made to HttpCallbackVerifier before test execution.
24     assertFalse(httpCallbackVerifier59917
25         .allServeEvents
26         .filter { it.wasMatched && it.stubMapping.metadata != null }
27         .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/product/fetch/image" }
28     )
29
30     // Fault202. Server-Side Request Forgery (SSRF). url.
31     given().accept("*/*")
32         .contentType("application/json")
33         .body(" { " +
34             "  \"url\": \"http://host.docker.internal:59917/EM_SSRF_2\", " +
35             "  \"productId\": 257 " +
36             " } ")
37         .post("${baseUrlOfSut}/api/product/fetch/image")
38         .then()
39         .statusCode(422)
40         .assertThat()
41         .contentType("text/plain")
42         .body(containsString("Unable to fetch."))
43
44     // Verifying that the request is successfully made to HttpCallbackVerifier after test execution.
45     assertTrue(httpCallbackVerifier59917
46         .allServeEvents
47         .filter { it.wasMatched && it.stubMapping.metadata != null }
48         .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/product/fetch/image" }
49     )
50 }

```

Figure 5: Code snippet demonstrates the full flow of the text execution of the vulnerable endpoint in Clerk, tested under black-box settings.

Line 4, we report additional information such as the fault category reported in the generated tests.

RQ1: Our approach demonstrates the capability of automatically detecting SSRF faults for synthetic examples.

5.2 Real-World APIs

To answer **RQ2**, we decided to evaluate our approach using real-world applications. Therefore, our second set of SUTs consists of two APIs that have reported CVEs, as selected from the CVE database.⁹

Microcks is a platform for turning your API and microservices assets - *OpenAPI specs*, *AsyncAPI specs*, *gRPC protobuf*, *GraphQL schema*, *Postman collections*, *SoapUI projects* - into live mocks in seconds.¹⁰ Note that, on the original reporting of the vulnerability in the CVE database, the version was wrongly reported (i.e., 1.17.1 instead of 1.7.1).

Lychee is a free, open-source photo-management PHP tool that runs on your server or webpage.¹¹

Given the stochastic behaviour of search algorithms, we repeated each experimental setting 10 times under a consistent termination criterion of 1 hour, in accordance with established guidelines for evaluating randomised techniques in software engineering research [1]. In this context, *EvoMASTER* was run in black-box mode (i.e., as white-box mode only works for JVM APIs, and *Lychee* is written in PHP). Test cases with SSRF exploits were successfully generated for both case studies. For *Lychee*,¹¹ SSRF was detected and exploited in two out of ten runs. Similarly, for *Microcks*,¹⁰ SSRF was detected and exploited in six out of ten runs.

Figure 6 contains a part of the test taken from the generated test suites for *Lychee* with SSRF detection. This tests the functionality shown in Figure 1 in Section 2. This endpoint accepts multiple URLs as an array, resulting in repeated values. At the end of the assertion, the SUT returns HTTP 422 because the application expects an image on the given URL. Additionally, the album identifier is required in the request for the application to process the images further. If the provided album identifier is invalid, the application returns an HTTP 422 before processing the URLs. On the contrary, if the album identifier is null, the application tries to download the images. In this example, since the request made in the test shown in Figure 6, the provided URL does not have an image, and the album identifier is null, the application returns HTTP 422.

In the end, as shown in Figure 7, we were able to validate the detected SSRF in this endpoint. Because the application has a validation rule, if there is an album identifier in the request, the SSRF detection process exhibits a low success rate.

Meanwhile, in *Microcks*,¹⁰ similar to *Lychee*, we ran experiments 10 times under a consistent termination criterion of 1 hour. The vulnerable endpoint takes a URL as one parameter, and if the parameter is valid, the application processes the request; otherwise, it responds with HTTP 500. Since we do not consider HTTP 500 during the selection stage for SSRF detection, this created an impact on the success rate. Compared with *Lychee*, *Microcks* performed better in our experiments due to the less complexity in the affected endpoint.

Overall, our approach effectively identified SSRF in both real-world case studies. However, in contrast to the experiments on synthetic APIs, SSRF vulnerabilities were not detected in all the experiment repetitions on real-world APIs. Real-world APIs might

have complex input and state validation. Even when using state-of-the-art fuzzers such as *EvoMASTER*, it does not mean that the right input data can be generated reliably each time.

RQ2: *Our approach demonstrates the capability of automatically detecting SSRF faults for real-world applications which have reported CVEs for SSRF.*

6 Threats To Validity

Internal validity. The experiments were conducted primarily using a software tool, from which the results were derived. Software faults are inevitable and may introduce threats to internal validity. Moreover, during the implementation, the software tool was tested extensively at various levels (i.e., unit and end-to-end tests provided by *EvoMASTER* [5]). However, it cannot be guaranteed that the implementation is entirely free of faults. Our implementation is built on top of the open-source fuzzer *EvoMASTER* [5]. The extended tool used in this study and the example APIs are accessible as open-source code on GitHub.² Anyone can review them.

External validity. The experiments were conducted on three synthetic example APIs, one artificial open-source didactic project, and against two real-world APIs with previously disclosed CVEs for SSRF. The three synthetic APIs are modeled on real-world examples adapted from some reported Common Vulnerabilities Exposures. Although common scenarios from real-world applications were considered, it is likely that some edge cases remain unaddressed. Further investigation is required to address additional cases, which are identified as future work.

7 Conclusions

In this paper, we have provided novel techniques to detect SSRF using a fully automated approach that is adaptable in both white-box and black-box fuzzing strategies. Our techniques have been implemented as an extension to the fuzzer *EvoMASTER* [2]. Experiments on 6 REST APIs demonstrate the viability of our language-agnostic novel techniques.

There are several avenues for further enhancement of our techniques in future work. The second-order SSRF mentioned in Section 2 detection is a complex issue that varies depending on the application. Furthermore, building on this approach of using a dynamic input, such as the use of a configurable *HttpCallbackVerifier*, there is research potential to create novel techniques to identify other vulnerability classes that involve external connections. Compared with using a static payload, we can expand the search to look for other cases where SSRF could lead to other vulnerabilities, such as SQL injection, command injection, and code injection. For example, in reported CVE (CVE-2024-35451) [13] for *LinkStack*¹ leads to a Command Injection [10] through SSRF.

Acknowledgments

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

⁹<https://www.cve.org>

¹⁰<https://microcks.io>

¹¹<https://lycheeorg.dev>

Table 2: Descriptive statistics of the employed SUTs. For each SUT, where Version represents the affected version of the software, where CVE represents the issued CVE number. Finally, Endpoint represents the affected REST endpoint.

SUT	Version	CVE	Endpoint
Microcks	1.7.1	CVE-2023-48910	/artifact/download
Lychee	6.6.13	CVE-2025-53018	/api/v2/Photo::fromUrl

```

1 // Fault202. Server-Side Request Forgery (SSRF). urls_item.
2 given().accept("application/json")
3 .header("Authorization", "IoDq6KQsD5atHyBi7sFlbQ==") // Fixed Headers
4 .header("X-Requested-With", "XMLHttpRequest") // Fixed Headers
5 .header("Content-Type", "application/json") // Fixed Headers
6 .contentType("application/json")
7 .body(" { " +
8     "  \"album_id\": \"\", " +
9     "  \"urls\": [ " +
10    "    \"http://host.docker.internal:46213/EM_SSRF_0\" " +
11    "  ] " +
12    " } ")
13 .post("${baseUrlOfSut}/api/v2/Photo::fromUrl")
14 .then()
15 .statusCode(422)
16 .assertThat()
17 .contentType("application/json")
18 .body("message", containsString("A photo could not be imported"))
19 .body("exception", containsString("MassImportException"))
20

```

Figure 6: Code snippet illustrates the HTTP request made to the vulnerable endpoint in Lychee.

```

1 // Verifying that the request is successfully made to
  HttpCallbackVerifier after test execution.
2 assertTrue(httpCallbackVerifier46213
3 .allServeEvents
4 .filter { it.wasMatched && it.stubMapping.metadata
  != null }
5 .any { it.stubMapping.metadata.getString("ssrf")
  == "POST:/api/v2/Photo::fromUrl" }
6 )
7

```

Figure 7: Code snippet which verifies the successful exploitation of SSRF in Lychee.

References

- [1] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. 24, 3 (2014), 219--250.
- [2] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6, 57 (2021), 2153.
- [3] Andrea Arcuri, Philip Garrett, Juan Pablo Galeotti, and Man Zhang. 2025. Widening The Adoption of Web API Fuzzing: Docker, GitHub Action and Python Support for EvoMaster. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 1084--1088.
- [4] A. Arcuri, A. Poth, and O. Rrjolli. 2025. Introducing Black-Box Fuzz Testing for REST APIs in Industry: Challenges and Solutions.
- [5] Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and Susruthan Seran. 2023. Building an open-source system test generation tool: lessons learned and empirical analyses with EvoMaster. *Software Quality Journal* (2023), 1--44.
- [6] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang. 2023. {NAUTILUS}: Automated {RESTful}{API} Vulnerability Detection. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5593--5609.
- [7] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue. 2024. Vulnerability-oriented testing for restful apis. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 739--755.
- [8] Sadeeq Jan, Annibale Panichella, Andrea Arcuri, and Lionel Briand. 2019. Search-based multi-vulnerability testing of XML injections in web applications. *Empirical Software Engineering* 24 (2019), 3696--3729.
- [9] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289--301. doi:10.1145/3533767.3534401
- [10] MITRE. 2006. *CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')*. MITRE Corporation. <https://cwe.mitre.org/data/definitions/77.html> CWE.
- [11] MITRE. 2006. *CWE-94: Improper Control of Generation of Code ('Code Injection')*. MITRE Corporation. <https://cwe.mitre.org/data/definitions/94.html> CWE.
- [12] MITRE. 2013. *CWE-918: Server-Side Request Forgery (SSRF)*. MITRE Corporation. <https://cwe.mitre.org/data/definitions/918.html> CWE.
- [13] NVD. 2024. *CVE-2024-35451*. National Vulnerability Database. <https://nvd.nist.gov/vuln/detail/CVE-2024-35451>
- [14] OWASP. 2021. *OWASP Top 10 - 2021*. <https://owasp.org/Top10/2021/>
- [15] OWASP. 2023. *About OWASP*. Retrieved February 6, 2024 from <https://owasp.org/API-Security/editions/2023/en/0x01-about-owasp/>
- [16] Omur Sahin, Man Zhang, and Andrea Arcuri. 2025. WFC/WFD: Web Fuzzing Commons, Dataset and Guidelines to Support Experimentation in REST API Fuzzing. arXiv:2509.01612 [cs.SE] <https://arxiv.org/abs/2509.01612>
- [17] Hassan Sartaj, Shaikat Ali, and Julie Marie Gjöby. 2025. Rest api testing in devops: A study on an evolving healthcare iot application. *ACM Transactions on Software Engineering and Methodology* (2025).
- [18] Enze Wang, Jianjun Chen, Wei Xie, Chuhuan Wang, Yifei Gao, Zhenhua Wang, Haixin Duan, Yang Liu, and Baosheng Wang. 2024. Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 239--257.

- [19] Yu Wang and Yue Xu. 2024. Beyond REST: Introducing APIF for Comprehensive API Vulnerability Fuzzing. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*. 435--449.
- [20] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology* (may 2023). doi:10.1145/3597205