Preventing Server-Side Request Forgery Attacks

Bahruz Jabiyev
Northeastern University
bahruz@ccs.neu.edu

Omid Mirzaei
Northeastern University
o.mirzaei@northeastern.edu

Amin Kharraz
Florida International University
ak@cs.fiu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

ABSTRACT

In today's web, it is not uncommon for web applications to take a complete URL as input from users. Usually, once the web application receives a URL, the server opens a connection to it. However, if the URL points to an internal service and the server still makes the connection, the server becomes vulnerable to Server-Side Request Forgery (SSRF) attacks. These attacks can be highly destructive when they exploit internal services. They are equally destructive and need much less effort to succeed if the server is hosted in a cloud environment. Therefore, with the growing use of cloud computing, the threat of SSRF attacks is becoming more serious.

In this paper, we present a novel defense approach to protect internal services from SSRF attacks. Our analysis of more than 60 SSRF vulnerability reports shows that developers' awareness about this vulnerability is generally limited. Therefore, coders usually have flaws in their defenses. Even when these defenses have no flaws, they are usually still affected by important security and functionality limitations. In this work, we develop a prototype based on the proposed approach by extending the functionality of a popular reverse proxy application and deploy a set of vulnerable web applications with that prototype. We demonstrate how SSRF attacks on these applications, with almost no loss of performance, are prevented.

ACM Reference Format:
Bahruz Jabiyev, Omid Mirzaei, Amin Kharraz, and Engin Kirda. 2021. Preventing Server-Side Request Forgery Attacks. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3442036

1 INTRODUCTION

Web applications have been always an attractive attack vector for adversaries. These applications are often open to the public-facing Internet and are commonly used to handle critical tasks and valuable data. Despite much attention to web applications in the security community, there has been little investigation into attacks on the trust relationship between web applications and critical services inside a target network. These attacks are often severe [35] and can render almost all of the contemporary security mechanisms ineffective.

In this work, we primarily focus on Server-Side Request Forgery (SSRF) attacks where the vulnerable web application redirects the attacker's requests to the internal network and exposes local services to the remote attacker, introducing different forms of risks [17]. The goal of this work is to reduce the risk of such crippling attacks by proposing a systematic framework that handles untrusted incoming requests without imposing a discernible performance impact or changing the underlying logic of web applications.

SSRF attacks can exploit internal services in different ways, from having them send spam emails [29, 36] to executing operating system commands remotely on servers that are running web applications [22, 44]. Specifically, if the vulnerable server (i.e., the server which runs the vulnerable web application) is hosted in a cloud environment, SSRF attacks take less effort for attackers and are usually more dangerous [23, 35]. One such attack happened less than a year ago against Capital One where more than 100 million consumer credit applications were stolen [35].

Mitigating SSRF vulnerabilities can be a non-trivial task. These issues can originate at different layers of the software stack (e.g., at the application level or a third-party library), making it very difficult to locate the vulnerable code. In fact, specific validations can take place for obvious forms of SSRF. However, it is possible to miss less obvious cases specifically due to transparency issues in third-party library usage in modern web applications.

In this paper, we begin by investigating the attack landscape and perform a manual analysis of 61 HackerOne SSRF vulnerability reports from 2014 to 2019. To choose these reports, we take the top 100 pages listed by Google from the total 163 HackerOne report pages which were indexed by the time of the search (i.e., September, 2019). However, only 61 of them contain enough information about how the attack was carried out; the rest has little information mostly due to limited disclosure by companies.

Our analysis reveals two main challenges for application developers. First, SSRF is an under-studied problem. Developers' awareness about the SSRF vulnerability is limited, and this lack of knowledge usually leads to incomplete defense mechanisms where adversaries' capabilities are underestimated. Second, it is not always obvious where the attack payload is delivered in (e.g., inserted in an uploaded file). Furthermore, our study shows that even when existing defense mechanisms are properly implemented by developers, they still suffer from important security and functionality limitations.

Two fundamental conditions underlie SSRF vulnerabilities: 1) based on a user input, the web application server makes a request,

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442036

and 2) the web application server has network access to internal services, commonly required for its normal functioning. In light of these conditions, we propose a defense approach which incorporates two main principles: 1) the web application server itself does not fetch any external resources and delegates this task to a helper server, and 2) the helper server has no network access to surrounding internal services and has the sole function of retrieving resources for the web application server.

We develop a prototype of our defense approach to evaluate its effectiveness. To do this, we extend the functionality of a popular reverse proxy application to support three main operations. First of all, it can recognize a URL in an incoming HTTP request. Second, it can modify the URL in a way so that it points to the helper server while preserving the original value of the URL. Finally, it can relay the modified request to the web application.

As a result, when the web application invokes the modified URL, a request containing the original URL of the resource is sent to the helper server. In response, the helper server fetches the resource at the original URL from the Internet and serves it back to the web application server. The helper server is given the minimum network access required to fetch resources from the Internet and is blocked from accessing internal services by firewall rules.

To evaluate our prototype, we use the OWASP Vulnerable Web Applications Directory (VWAD) Project [38]. This directory is a listing of web applications purposefully made vulnerable for educational means. We deploy them with the developed prototype and conduct experiments to examine our defense solution from three different angles. First, we assess how well it protects against SSRF attacks. Second, we investigate how much performance loss is imposed on web applications, and finally, we analyze if this defense solution can be evaded. We explain possible evasion methods for different implementation styles separately. We also discuss how these evasion methods can be addressed.

We hope that this work serves to raise awareness about the importance of defining reliable trust models in server-side security. We also hope our approach will prove useful to the web security community to tackle similar challenges that lie ahead in this dynamic landscape.

Contributions. In summary, we make the following main contributions:

- We systematically present different types of SSRF attacks along with their consequences. Also, we present defense mechanisms against these attacks and how attackers can potentially bypass them. We do all these by manually analyzing more than 60 HackerOne SSRF vulnerability reports.
- We introduce a novel SSRF defense approach which can be adopted in web applications.
- We develop a prototype of our method by extending the functionality of a popular reverse proxy application to validate the success of the proposed solution. We also share the extension code and other relevant artifacts to foster research in this area [10].

2 BACKGROUND

Our analysis of more than 60 SSRF reports on HackerOne [2] suggests that SSRF attacks are delivered in multiple ways, and they might have various consequences, depending on the the environment where the server is hosted.

2.1 Types of SSRF Attacks

SSRF attacks are divided into two types: in-band and out-of-band. In the following sections, we describe each category and their differences.

2.1.1 In-band. The SSRF attack is known as in-band when the attack payload is delivered in the same channel as HTTP messages between a client and a server. In other words, attack payloads are inserted in requests sent to the server. Listing 1 shows an example where an attack payload is sent in place of a URL that is supposed to be used to fetch a user's profile photo. This attack payload makes the server send `data` to a service running on port `123` on host `127.0.0.1` that is supposed to be unreachable to the attacker from outside of the firewall.

```
POST /profile/photo/upload HTTP/1.1
Host: foo.com
Content-Length: 37

image_url=http://127.0.0.1:123/data
```

Listing 1: The payload is delivered in the body of a request.

2.1.2 Out-of-band (OOB). The attack is called out-of-band when an attacker sends to a victim server a pointer to an attack payload. Once the pointer is referenced by the server, the attack payload gets delivered. In the example shown in Listing 2, when the web application processes the request (left), it finds a reference to an external file (right). Then, that external file is loaded and the attack payload is delivered. As a result, an internal service is exposed to an attacker-provided input.

This example also illustrates how an XXE (External XML Entity) vulnerability [20] can be used for an SSRF attack. XXE causes an SSRF attack when an XML parser substitutes an external entity in XML with its value and this value is a URL. In fact, XXE can cause both in-band and out-of-band SSRF attacks.

```
POST / HTTP/1.1
Host: foo.com

<?xml version="1.0"?>
<!DOCTYPE user [
  <!ENTITY % dtd SYSTEM
  'http://evil.com/e.dtd'>
  %dtd;
]>
<user>&send;</user>
```

```
<!--

Content of e.dtd
on evil.com

-->

<!ENTITY send SYSTEM
'http://127.0.0.1:123/data'
>
```

Listing 2: The request (left) contains a reference to an external file (right). The attack payload (right bottom) is delivered when the external file is loaded.

2.2 Consequences of SSRF attacks

Targeted victim servers can be hosted either in a cloud environment or within the physical confines of an enterprise. Each of these environments may bring about different consequences in SSRF attacks that we discuss below.

2.2.1 Cloud Environment. Web applications in the cloud run on instances (i.e., virtual servers). Every instance stores its metadata on a metadata server and makes it available through the metadata service. This service is required by scripts running on an instance to access its information, such as the IP address and hostname [15, 18]. In addition, the metadata service can be queried for the access credentials of an instance, and usually these queries require no authentication and authorization. As long as queries are made from within the instance, they will be served.

Therefore, when attackers discover that the vulnerable web application is running in the cloud, they take advantage of the SSRF vulnerability and forge requests from the application to the metadata service to steal the access credentials (of the instance hosting the application). Stolen access credentials let the attackers access services and resources under the permissions of those credentials. In fact, this is how the Capital One attacker obtained access to storage services, and from there to credit applications of more than 100 million customers [35]. In another attack [23], the attackers exploited an SSRF vulnerability and used the metadata service to steal the credentials of a Kubernetes (container orchestration) service and gain root access to all containers.

To address this issue, major cloud providers have introduced new versions to enforce specific headers in metadata service requests [8, 18]. However, as long as the previous versions are supported, the endpoints continue to be vulnerable as they were in the case of the Shopify attack [23].

2.2.2 On-premise Environment. When a web application is hosted in-house or within the physical confines of an enterprise, several services are usually running both on the application server itself and other servers in the internal network. Most of these services are usually not accessible from outside the firewall for security reasons. However, if the web application has an SSRF vulnerability, this will let an attacker forge requests from the application to all internal services that can be accessed from the application (or the server).

In order to exploit an internal service, the attacker should devise the payload in a way that the service can understand and process. This might be a challenge for SSRF attacks because they usually happen over an HTTP channel and not every service can speak HTTP. Even so, attackers are still able to succeed using different techniques for different types of services.

Some types of services can speak HTTP natively, and therefore, they do not require any specific technique. In fact, popular services (e.g., Elasticsearch, CouchDB) might come with a REST API that allows interacting with them in HTTP. Obviously, internal web applications are also of this type as they are served over HTTP. Moreover, as these web applications are only for internal use, they might lack the same level of protection as the ones that are open to public use.

Services that do not support the HTTP protocol can still accept HTTP inputs. This is because these services might not require inputs to be perfectly formed, and they can easily disregard irrelevant parts of inputs. Attackers take advantage of this by positioning an attack payload right after the HTTP part ends and have the server process the payload. In fact, this is how attackers exploited Memcached and Redis services to execute operating system commands remotely [21, 44].

3 EXISTING DEFENSE MECHANISMS

Our analyses of academic literature, security guidelines and vulnerability reports suggest that protection solutions and measures for SSRF attacks are limited to validation checks written by application developers in the application code. They often suffer from flaws, and even when they are flawless, they have important limitations.

3.1 Common Flaws and Bypasses

In this section, we discuss common mistakes developers usually make in defending against server-side request forgery attacks and how attackers can leverage these flaws to bypass the defense.

3.1.1 Incomplete Blacklisting. The simplest solution used by developers to defend against SSRF attacks is blacklisting. In this method, the host part of the user-provided URL is matched against a blacklist (see line 5 in Listing 3) to ensure that it does not point to an internal service. If a match is found, the request is rejected by the protection mechanism. This defense solution is bypassed by using different encoding schemes (e.g., decimal-encoded version of `127.0.0.1` is `2130706433`) and IPv6, as released publicly in multiple vulnerability reports [27, 28].

```
1 def is_allowed(url):
2   # url = 'http://127.0.0.1:123/data'
3   host = url.split('/')[2].split(':')[0]
4   # host = '127.0.0.1'
5   prefixes = ['192.168.', '172.', '10.', '127.', '0.',
    ↪  '169.254.']
6   for prefix in prefixes:
7     if host.startswith(prefix):
8       return False
9   return True
```

Listing 3: The host part is matched against a blacklist.

3.1.2 DNS Pinning. To avoid the limitations of blacklisting, some developers rely on libraries (see lines 1 and 7 in Listing 4) to disallow a URL which has a private IP address as its host component. However, this solution is not resilient against DNS pinning. As many vulnerability reports show [36, 45], an attacker can use a hostname that resolves to a private IP (e.g., `http://localtest.me:123/data` where `localtest.me` resolves to `127.0.0.1`) to bypass the defense.

3.1.3 HTTP Redirection. As a response to DNS pinning, developers can check if the host component of a URL resolves to a private IP address by making a DNS query. However, this solution does not protect from HTTP redirection bypass. In this bypass method, an

```python
import ipaddress

def is_allowed(url):
  # url = 'http://127.0.0.1:123/data'
  host = url.split('/')[2].split(':')[0]
  # host = '127.0.0.1'
  return not ipaddress.ip_address(host).is_private
```

Listing 4: A library function is used to check if the host part is a private IP address.

attacker inputs a URL pointing to a page on a server she controls. As this server has a public IP address, the check shown in Listing 5 passes, and the page is visited. This page, in turn, redirects the vulnerable server to make a request to an internal service (see an example in Listing 6). We came across this bypass method in several vulnerability reports [29, 36].

```python
import socket
import ipaddress

def is_allowed(url):
  # url = 'http://127.0.0.1:123/data'
  host = url.split('/')[2].split(':')[0]
  # host = '127.0.0.1'
  ip = socket.gethostbyname(host)
  return not ipaddress.ip_address(ip).is_private
```

Listing 5: The check is performed after the DNS query.

```php
<?php
  header('Location: http://127.0.0.1:123/data');
?>
```

Listing 6: PHP code on an attacker-controlled server.

3.1.4 Time of Check to Time of Use (TOCTOU). To prevent HTTP redirection bypass, developers disable redirection in requests made from the application to the Internet. Nevertheless, the application might still be affected by a TOCTOU vulnerability. Two DNS requests are made commonly before an HTTP request is made to a target URL: 1) to check whether the host part of the URL points to a private IP address, and 2) to resolve the hostname of the URL to be able to start a connection to the target server. Attackers take advantage of this by providing a URL pointing to their own server and have their DNS server serve two different DNS responses: 1) a public IP address to pass the first check, and 2) a private IP address to achieve a request forgery to an internal service. At least one SSRF vulnerability report [24] documents such an attack.

3.2 Fundamental Pillars of Defense Mechanisms

Our analyses show that current defense mechanisms are constructed at the code level, and they are expected to meet at least these four requirements: 1) a DNS check is performed on the host part of an input URL to see whether or not an internal service is pointed to, 2) well-tested libraries are used to verify that a given IP address is not private, even in the presence of a disguise, 3) redirection in HTTP requests made from the application is disabled, and 4) the second DNS check is abandoned in favor of using the result of the first check to start the connection.

3.3 Important Limitations

While application developers could apply all these fundamental requirements to build a defense, it would still suffer from important security and functionality limitations.

3.3.1 Security Limitations. The discussed requirements that form the basis of current defense mechanisms do not provide complete protection from SSRF attacks. As they mainly consist of programmatic checks, it is assumed that the subject of these checks is given. However, this assumption is not valid when an attack payload is delivered in a way that is unexpected for a developer. In that case, the developer fails to receive it and to perform the required checks. This happens in at least two cases: 1) the payload is delivered in an unexpected part of a request (e.g., inserted in an uploaded file), and 2) the payload is delivered in a different channel (i.e., out-of-band SSRF). In fact, our analysis of vulnerability reports shows that these cases account for 20% of SSRF attacks documented in the reports [31, 41].

3.3.2 Functionality Limitations. To prevent the HTTP redirection bypass method, developers are forced to disable redirection in requests made to fetch resources. However, disabling redirection while fetching resources would render moved resources to be unreachable. Given the fact that resources are often moved either temporarily or permanently on the Internet, tension might arise between security and functionality.

4 PROPOSED APPROACH

The high-level architecture of the proposed defense solution against SSRF attacks is provided in Figure 1. In the first step, a reverse proxy, positioned in front of a web application server, examines all requests coming from clients before they reach a web application. When the reverse proxy finds a URL in the request body or in the request URI, it modifies the URL and relays the modified request to the web application server. Then, when the web application invokes the modified URL, a request is sent to a helper server with the original value of the URL contained in the request. An HTTP service (which we refer to as the "helper service" in the remainder of the paper) running on the helper server receives the original URL and fetches the resource hosted at that URL, and then passes it back to the web application. As the helper server has no access to services running in the internal network, all internal services are left untouched, even if the original value of the URL (i.e., the user-provided value) is manipulated. We provide more information about each step in the remainder of this section.

4.1 Request Modification

The reverse proxy, sitting in front of the web application server, is able to intercept all requests coming from clients and going to the

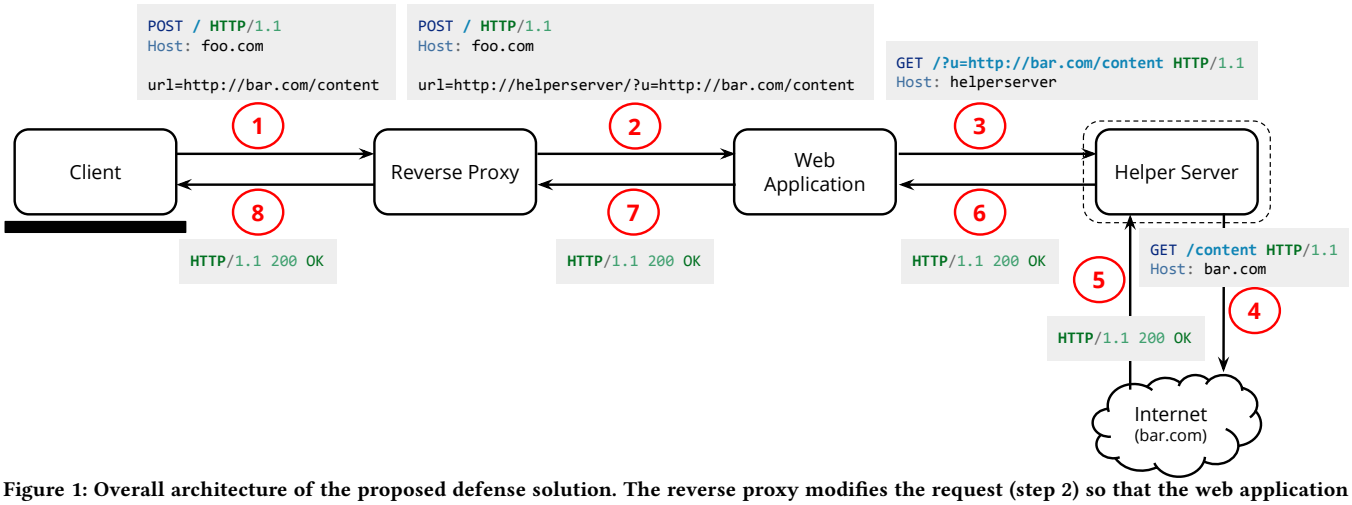

Figure 1: Overall architecture of the proposed defense solution. The reverse proxy modifies the request (step 2) so that the web application itself does not directly fetch resources from the Internet. Instead, it delegates the function of fetching resources to helper server (step 4).

web application. In addition, it has the ability to inspect and modify the requests.

4.1.1 Inspection and Search. An incoming request (step 1 in Figure 1) is intercepted and examined to determine whether or not any URL information is being sent. More specifically, the request body and URI are searched for a URL pattern. In Section 6, we discuss how successful this pattern matching is in the recognition of attack payloads.

4.1.2 Modification and Retransmission. Once a URL is found either in the request body or the request URI, it is modified. In Figure 1, the request sent in Step 2 has the modified version of the URL sent in the request sent in Step 1. However, when the URL is modified, the original value is retained. This is to make sure that the helper service will be able to fetch (step 4 in Figure 1) the resource hosted at the initial value of the URL and pass it back to the web application server.

The URL modification ensures that the initial (and possibly manipulated) value of the URL is handled by the helper server. Therefore, if the initial value of the URL is malicious, only the helper server is exposed, and therefore, the web application server is kept safe. Once the modification is done, the request is retransmitted (step 2 in Figure 1) to the web application server. As a result, only the modified URLs reach the application.

4.1.3 Page and Parameter Exclusion. URLs transmitted in requests are not always used for fetching resources. For example, it can determine the page a client will be redirected to. We see another example in "contact us" pages, where website URL information is sometimes asked. Therefore, modifying URLs can affect the functionality of the application in these cases. These cases can be handled in two different ways: 1) excluding pages and parameters from the modification on the reverse proxy and 2) discarding the prepended part in the application code.

However, these exclusions (especially for pages) might open a hole in the defense. Therefore, we recommend converting URLs back to their initial states in the application code by trimming the prepended part. For instance, when a URL is to be returned to a user, the conversion can be done right before returning it, to make sure that this URL did not cause any harm in case it was invoked earlier.

4.2 Resource Fetching

The web application server delegates the process of fetching resources from the Internet to a helper server. The helper server is an isolated server with no access to internal services. When it fetches resources, it keeps track of content types to avoid potential processing errors.

4.2.1 Isolation. The helper server needs minimal access in the internal network to perform its function. Every time a resource is fetched, the helper server needs to make outgoing connections only to the web application server on a temporarily opened port. As that port is open only during the connection and does not belong to any service, it does not pose any threat.

Except for the web application server, the helper server is blocked from accessing other internal IP addresses by firewall rules. In addition, the list of all services running on the application server is obtained and any access to them is blocked.

4.2.2 Connecting to the Internet. While the helper server is isolated from internal services, the helper service running on it is capable of connecting to servers on the Internet and fetching resources from them. This service receives the URL of a resource in the request from the web application server (step 3 in Figure 1) and opens a connection to that URL to retrieve the hosted resource. Once the resource is retrieved (step 5 in Figure 1), it is passed back to the web application server (step 6 in Figure 1), and later, to the client (steps 7 and 8 in Figure 1) from the application server.

As opposed to current defense mechanisms, our approach does not require disabling the HTTP redirection when fetching resources from the Internet. This is because the helper server is isolated, and therefore, even if it is redirected to an internal service, that service remains unreachable.

4.2.3 Tracking Content Types. When the helper service fetches resources for the web application, content type (e.g., text, image) has equal importance as the content itself. Depending on the content type, the way content is processed by the web application or the way it is rendered by client browsers (if it is passed on further to clients) might vary. Therefore, the helper service should also fetch the content type information of the resource.

5 PROTOTYPE IMPLEMENTATION

In our implementation, we extend the functionality of a popular reverse proxy application to be able to read and write HTTP requests sent by clients. Also, we deploy a server that has the ability to fetch a resource from the Internet at a given URL and is configured to have no access to internal services.

5.1 Extending Reverse Proxy

We choose NGINX [11] and extend it to support the operations of searching and modifying requests. We also enable developers to except pages and parameters from those operations.

5.1.1 Extending NGINX with Lua. NGINX is a high-performance web server and reverse proxy. We add the "ngx_http_lua_module" module [9] to NGINX. This module allows writing Lua code to extend the functionality of NGINX. Using this, we write an extension to process incoming requests. We make our extension code publicly available [10] to help other researchers benefit from this work. We also share the HackerOne reports which were analyzed for this paper.

5.1.2 Searching and Modifying. The extension code searches for a URL pattern in the request body and URI. The pattern consists of a sequence of characters beginning with a single colon and followed by a double slash (i.e. "://"). Different versions of this pattern are also searched, which we detail later in Section 6 when we discuss defense evasion techniques. If the pattern is matched, it is assumed that the matching text is a separator between a URL scheme and a hostname, and thus, a URL is found. We refer to these characters as "scheme separator" in the rest of this paper.

Once a URL is found, the extension code modifies it by prepending the address of the helper service. This prepending essentially passes the URL value as an argument to the helper service so that it can fetch the corresponding resource from the Internet (see the body of the request in Step 2 of Figure 1).

5.1.3 Exclusions. Our prototype supports modification exclusions both at the page and parameter levels. A page-based exclusion excepts all requests sent to that page from modification. Similarly, a parameter-based exclusion makes the value of that parameter exempt.

We specify the list of pages and parameters to be excluded, in a space-separated format, in an NGINX configuration file. This list is used as a reference from within the extension code to enforce necessary exceptions.

5.2 Deploying Helper Server

We use container technology to deploy a helper server in the internal network. Also, the helper service is started on the server to fetch resources from the Internet. We use firewall rules to isolate the helper server.

5.2.1 Server Image. We build a container image for the helper server from a Dockerfile. This file uses the official Ubuntu image as the base image. It also includes instructions for installing packages needed to start the helper service and add firewall rules. We also release this Dockerfile in the same repository [10].

5.2.2 Helper Service. The helper service is implemented with a few lines of PHP code and it runs on an NGINX server. This code takes a URL as an argument and makes an HTTP request to that URL and returns the response. To keep track of content types, the "Content-Type" header is used while fetching from the URL.

5.2.3 Firewall Rules. We rely on the UFW tool [2] to write and enable firewall rules on the helper server. Using UFW, access to private network ranges (`10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`), including link-local addresses (`169.254.0.0/16`) is blocked. Additionally, the loopback interface is disabled to prevent access to localhost (i.e. `127.0.0.1`). Access to the web application server IP is explicitly allowed, while all TCP and UDP ports opened by services are configured to be unreachable.

6 EVALUATION

The proposed SSRF defense solution aims at mitigating the impact of SSRF attacks by automatically adjusting and sandboxing URLs sent in requests. The solution relies on the ability of recognizing URLs as a primary step. Therefore, we start this section with the evaluation of our URL recognition module. We then present the overall ability of our solution in protecting against SSRF attacks. This is followed by a discussion on the effects of our solution on the performance of web applications. We wrap up this section with the discussion of methods attackers can leverage to evade our defense and how they can be addressed.

6.1 Recognizing URLs

Not all web client libraries parse a given URL in the same way. For example, some of the libraries might require the presence of a scheme name (e.g., "http://") in a given URL, while others might already assume it. This difference makes the accurate recognition of URLs a challenging task. Therefore, we analyze several libraries across various popular server-side languages and frameworks to get insight into how they parse the given URLs, and we discuss possible effects of parsing differences on our defense solution.

6.1.1 Collecting Libraries. To examine the parsing process, we compile a list of popular web client libraries (see Table 1). For each server-side programming language, there are usually multiple standard and third-party web client libraries. Normally, standard libraries offered by languages themselves are used most widely. However, in some languages, a third-party library might have the same popularity due to its ease of use and speed. In such cases, we also add those libraries to our list.

6.1.2 False Negatives. To recognize URLs sent in HTTP requests, we use the scheme separator pattern (i.e., "://"). If a server-side library accepts URLs without a scheme, such URLs will pass unnoticed. In that case, the defense solution will fail to prevent the

Library/Function	Language	URL Patterns		
		http://a.bc/	//a.bc/	a.bc/
HttpWebRequest	C#	✓		
HttpClient	C#	✓		
net/http	Go	✓		
java.net	Java	✓		
http	JS (Node.js)	✓		
request	JS (Node.js)	✓		
libwww-perl	Perl	✓		
file_get_contents()	PHP	✓		
cURL	PHP	✓		✓
urllib	Python	✓		
requests	Python	✓		
net/http	Ruby	✓	✓	

Table 1: Most libraries require URLs to start with a scheme name.

attack. However, our analysis of web client libraries suggests that accepting URLs without the scheme part is unusual. In fact, as Table 1 shows, only the "cURL" library of PHP and built-in http client library of Ruby accept URLs without a scheme name, while the rest of them, require this part to be present in the URL.

If a web application relies on the "cURL" library of PHP or "http/net" of Ruby to make HTTP requests, an attacker can easily bypass our protection by omitting the scheme part of URLs. However, this issue can be easily fixed by adding a check in the application code to fail URLs which do not start with a scheme name.

6.1.3 False Positives. Confusing non-URL elements in a request with URLs and modifying them might have unintended consequences. We assume that the pattern we use (i.e., "://") is almost unique to URLs. To verify this assumption, we do text analysis on a hundred arbitrary `README.md` files from Github as they usually have a good combination of technical and non-technical texts. We observe that the scheme separator pattern exclusively belongs to URLs, therefore, it can hardly be found in non-URL elements of a request. As a result, non-URL elements will hardly be confused as URLs.

6.2 Preventing Attacks

To demonstrate how our proposed defense solution prevents SSRF attacks, we deploy it with web applications from OWASP Vulnerable Web Applications Directory (VWAD) [38]. We evaluate how effectively in-band and out-of-band SSRF attacks are prevented.

6.2.1 Compiling List of Applications. Not all applications listed in the OWASP VWAD are vulnerable to SSRF attacks. Usually, known vulnerabilities are listed for each application. Thus, we choose those applications which have Server-Side Request Forgery (SSRF) vulnerabilities. We also include applications with XXE (External XML Entity) vulnerabilities in our list as XXE vulnerabilities usually allow SSRF attacks.

Not all applications in the compiled list are suitable for our experiments. Some of them are restrictive (i.e., do not allow attacks), and there are few others we cannot install (e.g., the installation interface does not work), and thus, cannot deploy. Applications listed in Table 2 are those we could deploy successfully with our defense solution. We use ID numbers shown in the table to refer to these applications in the remainder of this section.

ID	Name
1	Vulnerable Java Web Application [5]
2	NodeGoat [14]
3	OWASP Juice Shop [12]
4	Magical Code Injection Rainbow [1]
5	Mutillidae [13]
6	Xtreme Vulnerable Web Application [6]

Table 2: Names and ID numbers of sample applications, are listed.

These applications become vulnerable to SSRF attacks in different ways. Application #3 accepts a URL to set the profile photo of a user, while application #2 retrieves stock data from a user-specified source. Application #6 receives an image URL from a user to download and show it to the user. The rest process XML inputs and become vulnerable.

For each application, we follow the same procedure to prepare it for experiments. We first start each application on our local machine and provide its address to our extended reverse proxy to route all incoming HTTP traffic to that address. Simultaneously, we start the helper server by running a container created from its image.

6.2.2 Preventing In-band Attacks. In-band attacks make up a much larger portion of overall SSRF attacks. In fact, our analysis of 61 HackerOne vulnerability reports shows that the attack is in-band in nine out of ten cases.

Experimental results show that the proposed solution can prevent all in-band SSRF attacks. Only one of them requires minimal developer collaboration. This is because the web client library, Node.js Needle, used by the application, accepts URLs without a scheme name. The developer's minimal assistance of adding a check to fail URLs without a scheme name lets our defense solution be fully effective. Attacks on the rest of the applications are successfully prevented in an automated way.

This solution is independent of technologies used on a subject system. In fact, as seen in Table 3, applications used in our experiments use different languages and frameworks. Nevertheless, our defense solution works with all of them.

Attack payloads are negated regardless of where they are placed in the request. Current defense mechanisms usually assume that developers know where an attack payload might reside in a request (i.e., usually some POST or GET parameter). Therefore, they usually fail when an attack payload is delivered in an unexpected part of a request. Our defense solution automatically deals with all possible places an attack payload might be inserted. In fact, as Table 3 shows, attacks on selected applications have their payloads inserted in three different ways: 1) as a parameter value, 2) within a value of a parameter, and 3) within a request body (i.e., outside a parameter).

Thanks to the search pattern used by our defense solution, attacks involving URLs with a non-http scheme (e.g., gopher, dict) are also prevented although no example of such cases did exist in our experiments. While some of these schemes can enable more complicated attacks, they are supported by very few web client libraries.

ID	Technology	Payload Place	Automated	Assisted
1	Java	within body	✓	
2	Node.js	parameter		✓
3	Node.js	parameter	✓	
4	PHP	within parameter	✓	
5	PHP	within parameter	✓	
6	PHP	parameter	✓	

Table 3: While one application needs a minimal developer collaboration, in all other applications all in-band attacks are prevented in an automated manner.

6.2.3 Preventing Out-of-band Attacks. Our analysis of SSRF vulnerability reports suggests that only one out of ten SSRF attacks are of this type. Our defense solution cannot prevent out-of-band attacks in its current state as in this type of attack the payload is not transferred between an attacker and a victim application. Instead, the application retrieves it from an external location specified by the attacker.

Nevertheless, we believe that our defense approach can also be applied to out-of-band attacks by deploying a system in between the application and the Internet to modify URLs contained in downloaded content. However, in this scenario, URLs will be much more common, and therefore, various challenges might arise in ensuring that the application works as before.

6.3 Affecting Application Performance

The proposed defense solution modifies some incoming requests and uses a helper server to fetch resources for an application, and thus, the functionality and speed of the application are both affected.

6.3.1 Affecting Functionality. As the solution modifies an incoming request only when its URI or body contains a URL, the majority of requests pass through the reverse proxy unchanged. As a result, most of the application's functionality remains unaffected. For example, static files (e.g., CSS, JavaScript) are loaded in the same way as before, and thus, nothing changes in the appearance.

Functionality effects come into play when a URL is sent in a request. For instance, when a user submits a URL to upload her profile photo, this HTTP request (sent after the "submit" button is clicked) is modified and the photo is retrieved through a helper server.

To see how the functionality of selected applications is affected when they are deployed with our defense solution, we manually test the pages of all applications against their expected output.

We observe different behavior only in two pages. One of them does a client-side redirection based on a URL input. We solve this by adding the name of that parameter to the list of excluded parameters. The other refuses to download an image over SSL with the self-signed SSL certificate installed on the helper server. This can be fixed with a legitimate SSL certificate.

6.3.2 Affecting Speed. Our solution affects the speed of web applications, because it comes with two extra operations: 1) processing requests for search and modification, and 2) fetching resources over the helper server. The former applies to all requests coming to the application, whereas the latter happens only when the application needs to fetch a resource from a user-provided URL.

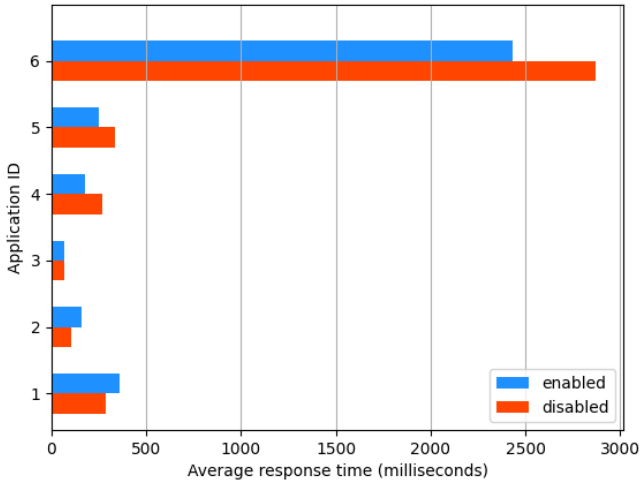

Figure 2: For each application, the average response time of the URL submitting request is recorded both when the protection is enabled and disabled.

To process HTTP requests, our prototype benefits from a module that embeds the Lua Just-In-Time (LuaJIT) interpreter into NGINX reverse proxy. Lua in itself has a good reputation for speed, and LuaJIT makes it even faster [32]. In addition, our request processing consists mainly of only two types of string searches (i.e., one in the URI and another in the body).

Fetching resources through the helper server increases the Round-Trip Time (RTT) of the request because the helper server is an intermediate node between a web application server and the Internet. However, since the helper server is deployed in the internal network, only two more network hops (one going and another coming back) are added. In addition, the helper server comes into play only when the web application needs to fetch a resource from the Internet.

To measure the extent of the slowdown, we use a tool called Burp Suite [7]. The "Intruder" part of this tool allows running a request many times automatically and provides the "Response completed" time for each request. In fact, this is the amount of time taken for the response to complete after the request is sent [16].

We specifically choose the request which involves URL submission from each application, because URL submitting requests face more slowdown than others. To be more specific, URL submitting requests are slowed down by the combination of both request processing and fetching over the helper server, whereas other requests are only affected by request processing overhead. Therefore, our choice of URL submitting requests for speed measurements lets us see the extent of the slowdown in the worst-case scenario.

Then, we run each (URL submitting) request 100 times with the same parameters, both with the protection enabled and disabled. To minimize the effects of outlying values, we calculate the 5% trimmed mean (after sorting values, 5% lowest and 5% highest values are trimmed and 90% remains to be used in the calculation).

The results given in Figure 2 show that our solution does not significantly reduce the speed of applications. In fact, little increase is observed in the response time in applications #1, #2, and #3. Interestingly, for the rest of the applications (#4, #5, #6) the response time

has decreased when our protection is enabled. We speculate that this happens because the helper server is dedicated to downloading contents, whereas the web application server is much busier, and therefore, might be a little slower to download data.

6.4 Potential Evasion

An important part of our defense solution is URL recognition that is accomplished by pattern matching. Also, our approach can be implemented in essentially two different ways: 1) on an intermediate node sitting between clients and an application, and 2) in the application code. Therefore, in what follows, we first discuss how our URL recognition system is affected by the broad category of URL obfuscation, and then, we explain how evasion methods can affect our defense system depending on the way it is implemented.

6.4.1 URL Obfuscation. URL obfuscation techniques have proven to be effective in deceiving defense systems and users [40]. Attackers might use URL shortening services to hide malicious URLs, or they might abuse URL redirects on a legitimate website to take users (or possibly systems) to malicious websites. The latter is also known as an "Unvalidated Redirects and Forwards" vulnerability [19].

However, these techniques pose no threat to our defense solution since all URLs are adjusted without distinction, and as a result, they are handled by the helper server. Therefore, even if manipulated URLs are obfuscated, they cannot inflict any harm to internal services on the target network.

6.4.2 Intermediate Deployment. When our defense solution is deployed on an intermediate system between clients and the web application server, it might be bypassed in multiple ways. We discuss three categories of such bypass methods.

The first category manipulates the process of URL encoding. URL encoding is needed to avoid confusion in the interpretation of requests by servers. In the context of an HTTP request, URL encoding is used in two different ways: 1) a request URI is usually URL-encoded, because it might contain a reserved character (e.g. "/", "?") [43], and 2) when a form data is submitted, the default encoding type is "application/x-www-form-urlencoded" and it uses URL encoding to encode reserved characters [26]. Therefore, to correctly interpret requests, servers usually decode a URI and form data (if it is the default type) in HTTP requests they receive.

Attackers can take advantage of URL encoding to bypass the search pattern (i.e., "://") our defense solution uses. For example, when attackers create their payloads, they can use "http%3a%2f/" instead of "http://", as they are treated the same by a server when decoding happens. To avoid this complexity brought by URL encoding, defenders can decode the incoming request before they start searching for URLs in the request.

However, decoding incoming requests opens a door to double encoding bypass methods. Attackers will encode the payload twice to bypass the defense. In this case, the intermediate defense system decodes only once and fails to notice the pattern, and the server decodes one more time to process the payload. Therefore, in our prototype, we do not decode incoming requests. Instead, we search for all encoded combinations of the "://" pattern, even though it is computationally more expensive.

The second category of bypass methods involves character encodings (e.g. utf-8, ibm037). Character encoding information is usually sent in the "charset" parameter of the "Content-Type" request header to let the server know how bytes should be translated into text [3, 4]. It has been shown that multiple encodings might be supported by servers depending on the technologies used on the server side [25]. Attackers can manipulate this to encode their payloads in multiple ways and to evade pattern matching. Therefore, in our prototype, we check the existence of the "charset" parameter in requests and require it to be "utf-8" if it exists.

The final category is HTTP desynchronization attacks which involves HTTP request smuggling. These attacks usually take advantage of different interpretations of request headers by intermediate and back-end systems. Attackers might use these attacks to bypass our defense solution. They can craft malicious requests which are totally missed by the intermediate system and hit the back-end system [34]. These attacks can be prevented by a cooperation between intermediate and back-end systems aiming to ensure that request headers and their values are interpreted the same way.

6.4.3 Code-level Deployment. Our defense technique can also be deployed as part of an application, at the code level. We are aware of only one bypass method for this fashion of defense implementation. This involves omitting the scheme part of URLs by attackers. If the application accepts URLs without a scheme, the searching mechanism would miss those URLs, and therefore, they pass unmodified and might inflict harm if they are malicious. However, this issue can easily be solved by minimal developer assistance where the developer adds some code to refuse to open a connection to URLs which do not contain a scheme.

As opposed to deployment on an intermediate system, code-level deployment has the advantage of having access to the state of a request as the application code sees it. This, in turn, allows the defense technique to be more accurate and be safe from disguised deliveries. Therefore, even though this type of deployment might not bring the same speed benefits of intermediate deployment, it leaves much less room for evasions.

7 RELATED WORK

Our review of the academic literature suggests that no defense solution has been proposed against SSRF attacks. To date, the SSRF problem has not received much attention from the academic community. Several studies have been done to show the goals of these attacks and their prevalence.

A study done by Pellegrino et al. [39] shows security implications of server-side requests in general. They describe different attack scenarios that can happen when a server-side request is abused. In fact, they show that server-side requests can be exploited to evade browser security mechanisms, to start denial-of-service attacks, to carry out network reconnaissance and to send arbitrary data to non-HTTP network services.

Some studies [33, 42] analyse XML parsers in regard to how well they are protected against XML vulnerabilities (which can eventually lead to SSRF attacks) with their default configurations. Späth et al. [42] find that many XML libraries in different languages (e.g., Java, Python, C#) are vulnerable to SSRF attacks by default. In another study [33], they analyze XML parsing libraries which

are widely used by projects on Github and Google Code. They find that more than half of them have the XXE vulnerability when they have a default configuration.

The Open ID Connect protocol which is used by single sign-on systems has also been shown to allow SSRF attacks [30, 37]. Mainka et al. [37] show that a service provider can be exposed to SSRF attacks if the identity provider is malicious. Not only service providers, but identity providers are also shown to be vulnerable [30].

8 CONCLUSION

In this paper, we present an automated approach for defending against SSRF attacks. We build a prototype for this approach to evaluate its ability to prevent attacks. Furthermore, we examine and report the effects of this solution on web applications. Our experimental results suggest that SSRF attack payloads can be automatically negated without significant effects on the application performance. In addition, they are negated regardless of where they are inserted and delivered in a request. Therefore, this approach can form a basis for defense systems that will be designed against the growing threat of SSRF attacks.

9 ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for reading the paper carefully and making helpful comments. This work was supported by National Science Foundation under grant CNS-1703454.

REFERENCES

[1] 2016. The Magical Code Injection Rainbow! https://github.com/SpiderLabs/MCIR. Accessed: 2020-05-11.
[2] 2017. The UFW firewall configuration tool. https://help.ubuntu.com/community/UFW. Accessed: 2020-06-10.
[3] 2019. Accept-Charset. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Charset. Accessed: 2020-05-26.
[4] 2019. Character encoding. https://developer.mozilla.org/en-US/docs/Glossary/character_encoding. Accessed: 2020-05-26.
[5] 2019. Vulnerable Java based Web Application. https://github.com/CSPF-Founder/JavaVulnerableLab. Accessed: 2020-05-11.
[6] 2019. XVWA is a badly coded web application written in PHP/MySQL that helps security enthusiasts to learn application security. https://github.com/s4n7h0/xvwa. Accessed: 2020-05-11.
[7] 2020. The Burp Suite family. https://portswigger.net/burp. Accessed: 2020-06-10.
[8] 2020. Configuring the instance metadata service. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/configuring-instance-metadata-service.html. Accessed: 2020-08-29.
[9] 2020. Embed the Power of Lua into NGINX HTTP servers. https://github.com/openresty/lua-nginx-module. Accessed: 2020-05-08.
[10] 2020. Implementation Materials. https://github.com/bahruzjabiyev/prevent-ssrf/. Accessed: 2020-12-16.
[11] 2020. The NGINX reverse proxy. https://www.nginx.com. Accessed: 2020-06-10.
[12] 2020. OWASP Juice Shop: Probably the most modern and sophisticated insecure web application. https://github.com/bkimminich/juice-shop. Accessed: 2020-05-11.
[13] 2020. OWASP Mutillidae II is a free, open source, deliberately vulnerable web-application. https://github.com/webpwnized/mutillidae. Accessed: 2020-05-11.
[14] 2020. The OWASP NodeGoat project. https://github.com/OWASP/NodeGoat. Accessed: 2020-05-11.
[15] 2020. Retrieving instance metadata. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval.html. Accessed: 2020-04-30.
[16] 2020. Running attacks. https://portswigger.net/burp/documentation/desktop/tools/intruder/attacks. Accessed: 2020-05-25.
[17] 2020. Server Side Request Forgery. https://owasp.org/www-community/attacks/Server_Side_Request_Forgery. Accessed: 2020-05-18.
[18] 2020. Storing and retrieving instance metadata. https://cloud.google.com/compute/docs/storing-retrieving-metadata. Accessed: 2020-04-30.
[19] 2020. Unvalidated Redirects and Forwards Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html. Accessed: 2020-06-12.
[20] 2020. XML External Entity (XXE) Processing. https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing. Accessed: 2020-05-19.
[21] Jobert Abma. 2017. Evaluating Ruby code by injecting Rescue job on the system_hook_push queue through web hook. https://hackerone.com/reports/299473
[22] Peter Adkins. 2017. Pivoting from blind SSRF to RCE with HashiCorp Consul. https://www.kernelpicnic.net/2017/05/29/Pivoting-from-blind-SSRF-to-RCE-with-Hashicorp-Consul.html
[23] Andre Baptista. 2018. SSRF in Exchange leads to ROOT access in all instances. https://hackerone.com/reports/341876
[24] Alex Chapman. 2019. GitLab::UrlBlocker validation bypass leading to full Server Side Request Forgery. https://hackerone.com/reports/541169
[25] Soroush Dalili. 2017. Request encoding to bypass web application firewalls. https://www.nccgroup.com/uk/about-us/newsroom-and-events/blogs/2017/august/request-encoding-to-bypass-web-application-firewalls/
[26] Ian Jacobs Dave Raggett, Arnaud Le Hors. 1999. HTML 4.01 Specification. https://www.w3.org/TR/html401/interact/forms.html
[27] Ed. 2017. SSRF vulnerability in gitlab.com via project import. https://hackerone.com/reports/215105
[28] Elb. 2019. Bypass of the SSRF protection in Event Subscriptions parameter. https://hackerone.com/reports/386292
[29] Eugene Farfel. 2016. SSRF in https://imgur.com/vidgif/url. https://hackerone.com/reports/115748
[30] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 189–202.
[31] floyd. 2017. SVG Server Side Request Forgery (SSRF). https://hackerone.com/reports/223203
[32] Michael Gogins. 2013. Writing Csound Opcodes in Lua. In *Ways Ahead: Proceedings of the First International Csound Conference*. Cambridge Scholars Publishing, 32.
[33] Sadeeq Jan, Cu D. Nguyen, and Lionel Briand. 2015. Known XML Vulnerabilities Are Still a Threat to Popular Parsers and Open Source Systems. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, Vancouver, BC.
[34] James Kettle. 2019. HTTP Desync Attacks: Request Smuggling Reborn. https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn
[35] Brian Krebs. 2019. Capital One Data Theft Impacts 106M People. https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack/
[36] Corben Leo. 2018. Sending Emails from DNSDumpster - Server-Side Request Forgery to Internal SMTP Access. https://hackerone.com/reports/392859
[37] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. 2017. SoK: Single Sign-On Security — An Evaluation of OpenID Connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 251–266.
[38] OWASP. 2020. OWASP Vulnerable Web Applications Directory. https://owasp.org/www-project-vulnerable-web-applications-directory/
[39] Giancarlo Pellegrino, Onur Catakoglu, Davide Balzarotti, and Christian Rossow. 2016. Uses and Abuses of Server-Side Requests. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses*.
[40] Raymond Pompon. 2017. URL Obfuscation—Still a Phisher's Phriend. https://www.f5.com/labs/articles/threat-intelligence/url-obfuscationstill-a-phishers-phriend
[41] Slim Shady. 2016. SSRF and local file read in video to gif converter. https://hackerone.com/reports/115857
[42] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. 2016. SoK: XML Parser Vulnerabilities. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/woot16/workshop-program/presentation/spath
[43] L. Masinter T. Berners-Lee, R. Fielding. 2005. Uniform Resource Identifier (URI): Generic Syntax. https://tools.ietf.org/html/rfc3986
[44] Cheng-Da Tsai. 2017. How I Chained 4 vulnerabilities on GitHub Enterprise, From SSRF Execution Chain to RCE! http://blog.orange.tw/2017/07/how-i-chained-4-vulnerabilities-on.html
[45] ylujion. 2016. Blind SSRF on synthetics.newrelic.com. https://hackerone.com/reports/141304