Another vision for SSRF

Another vision for SSRF - @phor3nsic_br, gccybermonks.com.

- Published: date not stated
- Original: <https://gccybermonks.com/posts/ssrfvision/>
- Preserved from: https://gccybermonks.com/posts/ssrfvision/ (stored) on 2026-08-09
- Capture timestamp: 20260511140211
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Another vision for SSRF

by @phor3nsic_br

[image: image]

Summary

For a long time, I tested SSRF failures to search for services and ports from the internal network and use the information to obtain interesting data or reach a RCE. But in the last few days, I came across situations where I didn't have an internal scenario, I had a good flaw but its impact would be low.

Until a great idea came up, I would like to share it with you!

What is SSRF

As much as I believe that more than half of the readers who are seeing this article already know what SSRF *(Server Side Request Forgery)* is, I leave the description and reference to make it easier for everyone to understand:

"The target application may have functionality for importing data from a URL, publishing data to a URL or otherwise reading data from a URL that can be tampered with. The attacker modifies the calls to this functionality by supplying a completely different URL or by manipulating how URLs are built (path traversal etc.)."

Owasp

Common Vision

When reading all the writings about SSRF, we have a line of reasoning to be understood, perhaps because this is the most talked-about by all, where an attacker manipulates the server-side request to gain access to internal data.

However, companies knowing the possible problem, segregate these hosts, so that the attacker cannot get anything from the internal network, limiting access to ports and implementing a series of restrictions where SSRF would not be accepted by bug bounty programs, or by the client.

Okay, is SSRF limited to just that? Is there nothing else we can do?

Another Vision

SSRF To Account Takeover

Before continuing, during this writing the channel another cool view on SSRF+Phishing that can be seen on the @gregxsunday channel, Bug Bounty Reports Explained, but still, we will deal with another point of view!

The concept I bring is to use the server-side, to reach the client-side, and we will still get a high criticality, because the final goal is to reach the user and their data, through SSRF in a sub/domain.

To reach the client-side, we need our SSRF to reflect the headers used by the client this is common in SSRF that will serve as a proxy so the attacker can define a proxy in the application.

By doing this we have something similar to a subdomain takeover, as we can take over all the content that transits to that subdomain. This already very impactful in the case of a domain but in the case of an out of scope subdomain, or a dev environment, it may not have as much effect, as there are not many direct requests for regions like this and the company can inform that this is not under their control, reducing the impact of the vulnerability.

We move on to another functionality. We need authentication/session cookies for applications on the same domain as the SSRF.

Example:

If the SSRF is on ssrf.dev.example.com, we need to find an application with authentication on *.example.com that has cookies set as follows:

```
Set-cookie: session=SECRET_SESSION, domain=.example.com, HttpOnly
```

Note that the cookie is being set so that it can move freely between all subdomains of *.example.com.

Most of us know that if a cookie has the HttpOnly flag it cannot be obtained through a XSS flaw, if you don't know read:

HttpOnly

So, even if we have a XSS in any subdomain of example.com, we can't capture the cookies, but we can still reuse them with fetch read more here

In case Set-Cookie is deploying SameSite this way:

```
Set-cookie: session=SECRET_SESSION, domain=.example.com, HttpOnly, SameSite=Lax
```

We could neither get the cookies nor reuse them with fetch through a XSS due to the SameSite. implementation.

Ok, Ok, Ok...

We cannot obtain or reuse cookies through a XSS, but we can steal these cookies through a SSRF!

[image: image]

The request carries cookies, and they can be obtained through logs from the server that's monitoring the requests.

The attacker must determine that the SSRF directs requests to its server and then makes its content available through the vulnerable domain. By sending the server URL to the victim, the victim somehow accesses the URL. Soon attacker will get the session cookie through your logs!

[image: image]

Simple and a little weird, right? To make it easier, we'll follow with an example lab.

Lab

When accessing the app.localhost.io we receive a cookie with the HttpOnly flag and `domain=.localhost.io` :

[image: image]

If we try to get that cookie through a XSS, we won't even see it due to HttpOnly:

[image: image]

However, if we use a SSRF described above, our cookies are sent to the destination server.

[image: image]

If we redirect the request to our server, we can view the requests and get the victim's cookies:

[image: image]

Conclusion

I could let this SSRF go by not being able to access what I wanted to see, but when we stop following a list of thoughts, we can think differently.

The concept of hacking is talked about a lot: "think outside of the box", however, we stop thinking in a small box and think in bigger boxes, the truth is, don't fall into any box, think above implanted limits, the beauty can be on the other side of the edges.