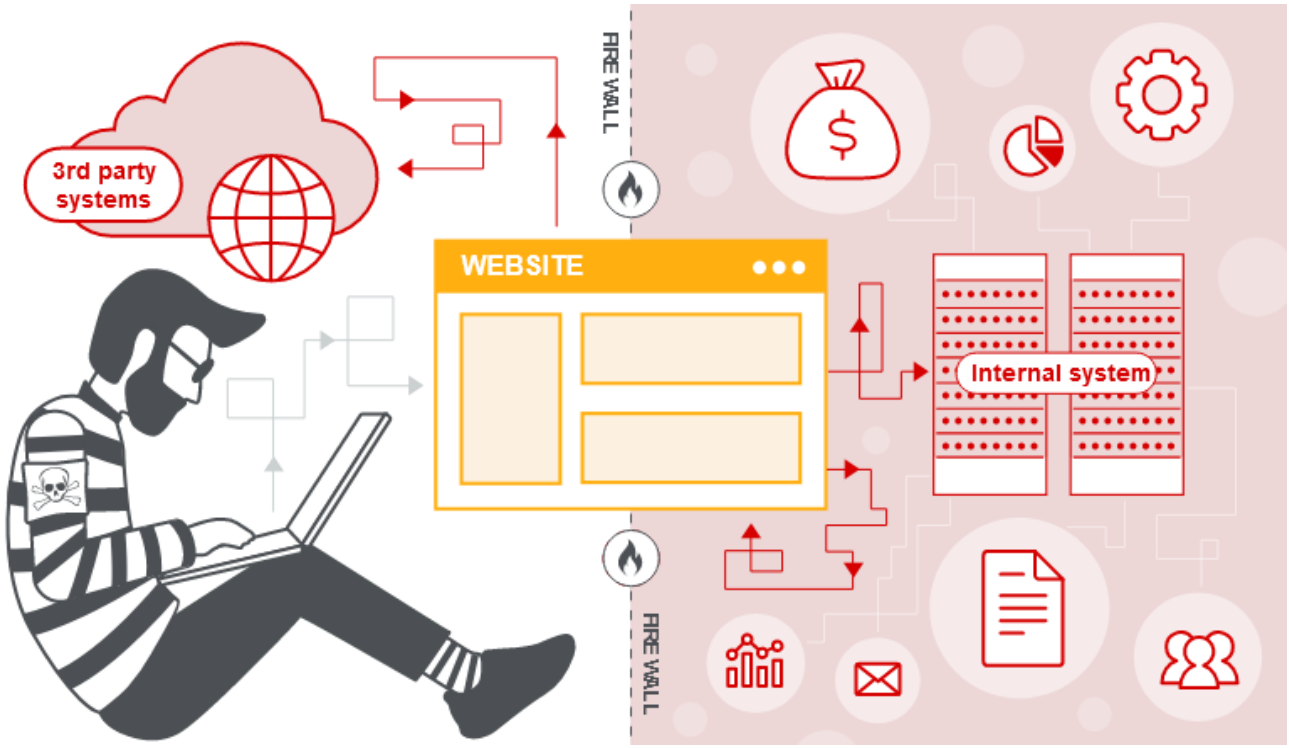
Server-Side Request Forgery

15/08/2022

Aug 14, 2022

12 min read
SSRF Web-Notes

Server-side request forgery (SSRF) is a vulnerability that lets an attacker send requests on behalf of a server. During an SSRF, attackers **forge the request** signatures of the vulnerable server, allowing them to assume a privileged position on a network, bypass firewall controls, and gain access to internal services.



References

SSRF (Server Side Request Forgery)

AllAboutBugBounty/Server Side Request Forgery.md at master · daffainfo/AllAboutBugBounty

bugbounty-cheatsheet/ssrf.md at master · EdOverflow/bugbounty-cheatsheet

HowToHunt/SSRF at master · KathanP19/HowToHunt

Bug_Bounty_writeups/README.md at main · alexbieber/Bug_Bounty_writeups

Server Side Request Forgery (SSRF)

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

GitHub - vavkamil/awesome-bugbounty-tools: A curated list of various bug bounty tools

Mind-Maps/SSRF - Hackerscroll at master · imran-parray/Mind-Maps

Web Application Penetration Testing Notes

pentest-guide/Server-Side-Request-Forgery at master · Voorivex/pentest-guide

SSRF

GitHub - 0xL1mb0/Web-CTF-Cheatsheet: Web CTF CheatSheet

Server Side Request Forgery [SSRF] - Pastebin.com

Mechanisms

It's accrues when the attacker sends a server request as a trusted one to the target's network.

Imagine a public-facing web server on example.com's network named public.example.com. This server hosts a proxy service,

located at public.example.com/proxy, that fetches the web page specified in the URL parameter and displays it back to the user.

For example, when the user accesses the following URL, the web application would display the google.com home page: https://public.example.com/proxy?url=https://google.com

Now let's say admin.example.com is an internal server on the network hosting an **admin panel**. To ensure that only employees can access the panel, administrators set up access controls to keep it from being reached via the internet. Only machines with a **valid internal IP**, like an employee workstation, can access the panel. Now, what if a regular user accesses the following URL?

https://public.example.com/proxy?url=https://admin.example.com the web application would display the admin panel to the user, because the request to the admin panel is coming from the web server, public.example.com, a **trusted machine** on the network.

> Through SSRF, servers accept unauthorized requests that firewall controls would normally block

By forging requests from trusted servers, an attacker can **pivot** into an organization's internal network and conduct all kinds of malicious activities.

SSRF vulnerabilities have two types: regular SSRF and blind SSRF.

The only difference is that in a **blind SSRF**, the attacker **does not receive feedback** from the server via an HTTP response or an error message.

Let's say that on public.example.com another functionality allows users to send requests via its web server. But this endpoint d**oes not return the resulting** page to the user. If attackers can send requests to the internal network, the endpoint suffers from a blind SSRF vulnerability:

https://public.example.com/send_request?url=https://admin.example.com/delete_user?user=1

Prevention

For example, when you post a link on Twitter, Twitter fetches an image from that external site to create a thumbnail. If the server doesn't stop users from accessing internal resources using the same mechanisms, SSRF vulnerabilities occur. Let's look at another example. Say a page on public.example.com allows users to upload a profile photo by retrieving it from a URL via this POST request:

```
POST /upload_profile_from_url
Host: public.example.com
```

```
(POST request body)
user_id=1234&url=https://www.attacker.com/profile.jpeg
```

But if the server does not make a distinction between internal and external resources, an attacker could just as easily request a local file stored on the server or any other file on the network.

```
POST /upload_profile_from_url
Host: public.example.com
(POST request body)
user_id=1234&url=https://localhost/passwords.txt
```

Two main types of protection against SSRFs exist **blocklists** and **allowlists**. Blocklists are lists of banned addresses. The server will block a request if it contains a **blocklisted** address as input. On the other hand, when a site implements **allowlist** protection, the server allows only requests that have URLs found in a predetermined list and rejects all other requests. Some servers also protect against SSRFs by requiring special headers or secret tokens in internal requests.

Hunting for SSRFs

The best way to discover SSRF vulnerabilities is through a review of the **application's source code**, in which you check if the application validates all user-provided URLs.

Step 1: Spot Features Prone to SSRFs

SSRFs occur in features that require **visiting and fetching external resources**. These include **webhooks**, **file uploads**, **document** and **image processors**, link **expansions** or **thumbnails**, and **proxy services**.

And pay attention to potential SSRF entry points that are less obvious, like **URLs embedded** in **files** that are processed by the application (XML files and PDF files can often be used to trigger SSRFs), hidden **API endpoints** that accept URLs as input, and input that gets inserted into HTML tags.

And in the event that one action from an application needs to trigger an action on another application, **webhooks** are a way of notifying the system to kick-start another process. For example, if a company wants to send a welcome email to every user who follows its social media account, it can use a **webhook to connect the two applications**.

For example, Slack allows application owners to set up a webhook via its app configuration page https://api.slack.com/apps/

Under the Event Subscriptions heading, you can specify a URL at which Slack will notify you when special events happen. The Request URL field of these webhook services is often **vulnerable to SSRF**.

Potential SSRF Endpoints

Add a new webhook:

```
POST /webhook
Host: public.example.com
(POST request body)
url=https://www.attacker.com
```

File upload via URL:

```
POST /upload_profile_from_url
Host: public.example.com
(POST request body)
user_id=1234&url=https://www.attacker.com/profile.jpeg
```

Proxy service: https://public.example.com/proxy?url=https://google.com

.

.

Step 2: Provide Potentially Vulnerable Endpoints with Internal URLs

Depending on the network configuration, you might need to try several addresses before you find the ones in use by the network. Here are some common ones reserved for the private network: `localhost`, `127.0.0.1`, `0.0.0.0`, `192.168.0.1`, and `10.0.0.1`.

To illustrate, this request tests the webhook functionality:

```
POST /webhook
Host: public.example.com
(POST request body)
url=https://192.168.0.1
```

This request tests the file upload functionality:

```
POST /upload_profile_from_url
Host: public.example.com
(POST request body)
user_id=1234&url=https://192.168.0.1
```

And this request tests the proxy service: https://public.example.com/proxy?url=https://192.168.0.1

Step 3: Check the Results

For example, does the response contain service banners or the content of internal pages? A service banner is the name and version of the software running on the machine. Check for this by sending a request like this:

```
POST /upload_profile_from_url
Host: public.example.com
(POST request body)
user_id=1234&url=127.0.0.1:22
```

Port 22 is the default port for the **Secure Shell Protocol** (SSH). This request tells the application that the URL of our profile picture is located at 127.0.0.1:22, or port 22 of the current machine. This way, we can **trick the server into visiting its own port 22** and returning information about itself. Then look for text like this in the response:

```
Error: cannot upload image: SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.4
```

If you find a message like this, you can be sure that an **SSRF vulnerability exists** on this endpoint, since you were able to gather information about the localhost.

Common SSRF attacks

SSRF attacks against the server itself

For example, consider a shopping application that lets the user view whether an item is **in stock** in a particular store. To provide the stock information, the application must **query various back-end REST APIs**, dependent on the product and store in question. The function is implemented by **passing the URL to the relevant back-end API**

endpoint via a front-end HTTP request. So when a user views the stock status for an item, their browser makes a request like this:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118
stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%
```

In this situation, an attacker can modify the request to specify a URL local to the server itself. For example:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://localhost/admin
```

Here, the server will fetch the contents of the `/admin`URL and return it to the user. The application grants full access to the administrative functionality, because the request appears to originate from a **trusted location**.

Lab: Basic SSRF against the local server Web Security Academy

Why do applications behave in this way?

- When a connection is made back to the server itself, the check is bypassed.
- For disaster recovery purposes, the application might allow administrative access without logging in The assumption here is that only a fully trusted user would be coming directly from the server itself.
- The administrative interface might be listening on a different port number than the main application, and so might not be reachable directly by users.

SSRF attacks against other back-end systems

Another type of trust relationship that often arises with server-side request forgery is where the application server is able to interact with other back-end systems that are not directly reachable by users. These systems often have non-routable private IP addresses. In the preceding example, suppose there is an administrative interface at the back-end URL `https://192.168.0.68/admin` Here, an attacker can exploit the SSRF vulnerability to access the administrative interface by submitting the following request:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://192.168.0.68/admin
```

Lab: Basic SSRF against another back-end system Web Security Academy

Circumventing common SSRF defenses

Bypass Allowlists

Some applications only allow input that matches, begins with, or contains, a whitelist of permitted values. In this situation, you can sometimes circumvent the filter by exploiting inconsistencies in URL parsing.

The URL specification contains a number of features that are liable to be overlooked when implementing ad hoc parsing and validation of URLs:

- You can **embed credentials** in a URL before the hostname, using the `@` character. For example: `https://expected-host@evil-host`
- You can use the `#` character to indicate a URL fragment. For example: `https://evil-host#expected-host`
- You can leverage the **DNS naming hierarchy** to place required input into a fully-qualified DNS name that you control. For example: `https://expected-host.evil-host`
- You can **URL-encode characters** to confuse the URL-parsing code. This is particularly useful if the code that implements the filter handles URL-encoded characters differently than the code that performs the back-end HTTP request.
- You can use combinations of these techniques together.

Lab: SSRF with whitelist-based input filter Web Security Academy

Bypassing SSRF filters via open redirection

For example, suppose the application contains an open redirection vulnerability in which the following URL:

`/product/nextProduct?currentProductId=6&path=http://evil-user.net`

returns a redirection to:

`http://evil-user.net`

You can leverage the open redirection vulnerability to bypass the URL filter, and exploit the SSRF vulnerability as follows:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://weliketoshop.net/product/nextProduct?
currentProductId=6&path=http://192.168.0.68/admin
```

This SSRF exploit works because the application first validates that the supplied `stockAPI` URL is on an allowed domain, which it is. The application then requests the supplied URL, which triggers the open redirection. It follows the redirection, and makes a request to the internal URL of the attacker's choosing.

Lab: SSRF with filter bypass via open redirection vulnerability Web Security Academy

Using IPv6 Addresses

Sometimes the SSRF protection mechanisms a site has implemented for IPv4 might not have been implemented for IPv6. That means you can try to submit IPv6 addresses that point to the local network. For example, the IPv6 address ::1 points to the localhost, and fc00:: is the first address on the private network. For more information about how IPv6 works, and about other reserved IPv6 addresses, visit Wikipedia: https://en.wikipedia.org/wiki/IPv6_address.

Tricking the Server with DNS

Modify the A/AAAA record of a domain you control and make it point to the internal addresses on the victim's network. You can check the current A/AAAA records of your domain by running these commands:

```
nslookup DOMAIN
nslookup DOMAIN -type=AAAA
```

You can usually configure the DNS records of your domain name by using your domain registrar or web-hosting service's settings page. For instance, I use Namecheap as my domain service. In Namecheap, you can configure

your DNS records by going to your account and choosing Domain List Manage Domain Advanced DNS Add New Record. Create a custom mapping of hostname to IP address and make your domain resolve to 127.0.0.1. You can do this by creating a new A record for your domain that points to 127.0.0.1. Then you can ask the target server to send a request to your server, like: https://public.example.com/proxy?url=https://attacker.com Now when the target server requests your domain, it will think your domain is located at 127.0.0.1 and request data from that address.

Blind SSRF vulnerabilities

What is blind SSRF?

Blind SSRF vulnerabilities arise when an application can be induced to issue a back-end HTTP request to a supplied URL, but the response from the back-end request is not returned in the application's front-end response.

What is the impact of blind SSRF vulnerabilities?

The impact of blind SSRF vulnerabilities is often lower than fully informed SSRF vulnerabilities because of their one-way nature. They cannot be trivially exploited to retrieve sensitive data from back-end systems, although in some situations they can be exploited to achieve full remote code execution.

How to find and exploit blind SSRF vulnerabilities

The most reliable way to detect blind SSRF vulnerabilities is using out-of-band (OAST) techniques. This involves attempting to trigger an HTTP request to an external system that you control, and monitoring for network interactions with that system.

The easiest and most effective way to use out-of-band techniques is using Burp Collaborator. You can use the Burp Collaborator client to generate unique domain names, send these in payloads to the application, and monitor for any interaction with those domains. If an incoming HTTP request is observed coming from the application, then it is vulnerable to SSRF.

Lab: Blind SSRF with out-of-band detection Web Security Academy

Simply identifying a blind SSRF that can trigger out-of-band HTTP requests doesn't in itself provide a route to exploitability. Since you cannot view the response from the back-end request, the behavior can't be used to explore content on systems that the application server can reach. However, it can still be leveraged to probe for other vulnerabilities on the server itself or on other back-end systems. You can blindly sweep the internal IP address space, sending payloads designed to detect well-known vulnerabilities. If those payloads also employ blind out-of-band techniques, then you might uncover a critical vulnerability on an unpatched internal server.

Lab: Blind SSRF with Shellshock exploitation Web Security Academy

Another avenue for exploiting blind SSRF vulnerabilities is to induce the application to connect to a system under the attacker's control, and return malicious responses to the HTTP client that makes the connection. If you can exploit a serious client-side vulnerability in the server's HTTP implementation, you might be able to achieve remote code execution within the application infrastructure.

Finding Your First SSRF!

Let's review the steps you can take to find your first SSRF:

1. Spot the features prone to SSRFs and take notes for future reference.
2. Set up a callback listener to detect blind SSRFs by using an online service, Netcat, or Burp's Collaborator feature.

3. Provide the potentially vulnerable endpoints with common internal addresses or the address of your callback listener.
4. Check if the server responds with information that confirms the SSRF. Or, in the case of a blind SSRF, check your server logs for requests from the target server.
5. In the case of a blind SSRF, check if the server behavior differs when you request different hosts or ports.
6. If SSRF protection is implemented, try to bypass it by using the strategies discussed .
7. Pick a tactic to escalate the SSRF.
8. Draft your first SSRF report!