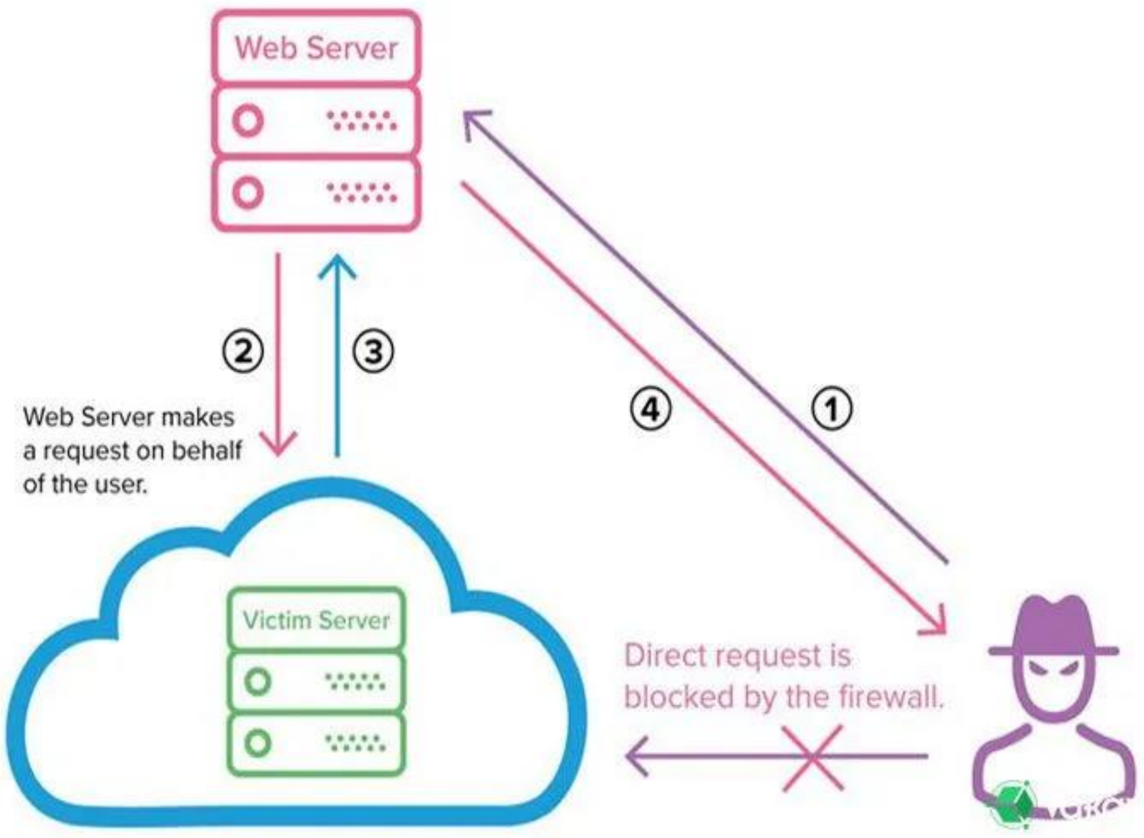
Server Side Request Forgery

I. What is SSRF ?

Server Side Request Forgery is a web vulnerability that lets attackers make requests from a backend server vulnerable to internal or external systems.



Web Server
2
3
4
1
Web Server makes a request on behalf of the user.
Victim Server
Direct request is blocked by the firewall.

II. What is the impact of SSRF attack ?

If the user-supplied URL is processed and the back-end response is not sanitised then the attack can lead to several impacts like:

1. **Port scanning**: A user can scan the port of a particular website through the vulnerable web application which is processing the user’s URL.
2. **Denial-of-Service (DoS)** attack
3. Attacking internal or external web applications.
4. Reading local web server files using the file:/// protocol handler.

5. In some cases, a successful SSRF attack can even lead to **Remote Code Execution (RCE).**

III. Type of SSRF and How to Exploit it ?

1. Basic SSRF

This is the type of SSRF in which the victim server returns data to the hacker. When they perform an SSRF attack, a hacker is sending a request to a victim server. Basic SSRF is when the victim server sends back some data after the crafted request is made. A hacker would use Basic SSRF when they want to get data from the server or want to access unauthorized features.

1.1 SSRF attacks against the server itself

In an SSRF attack against the server itself, the attacker induces the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. This will typically involve supplying a URL with a hostname like 127.0.0.1 (a reserved IP address that points to the loopback adapter) or localhost (a commonly used name for the same adapter).

Suppose a web has the below request:

GET https://blog-vul.com/id?content=dashboard.html

This causes the server to make a request to the specified URL, retrieve the page content and return this to the user.

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

GET https://blog-vul.com/id?content=http://localhost/admin

The loopback interface can be used to access the content that is accessible only for the host. This states that we have access to the host, which implies we also have access to the admin page.

1.2 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. Since the back-end systems are normally protected by the network topology, they often have a weaker security posture. In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication by anyone who is able to interact with the systems.

In the preceding example, suppose there is an administrative interface at the back-end URL https://192.168.49.53/admin. Here, an attacker can exploit the SSRF vulnerability to access the administrative interface by submitting the following request:

GET https://blog-vul.com/id?content=https://192.168.49.53/admin

Bypass SSRF defenses

Some web applications have mechanism to protect against SSRF attack. But, these defenses can be circumvented.

1.3 SSRF with blacklist-based input filters

Some applications block input containing hostnames like 127.0.0.1 and localhost, or sensitive URLs like /admin. In this situation, you can often circumvent the filter using various techniques:

- Using an alternative IP representation of 127.0.0.1, such as 2130706433, 017700000001, or 127.1.
- Registering your own domain name that resolves to 127.0.0.1.
- Obfuscating blocked strings using URL encoding or case variation

1.4 SSRF with whitelist-based input filters

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.

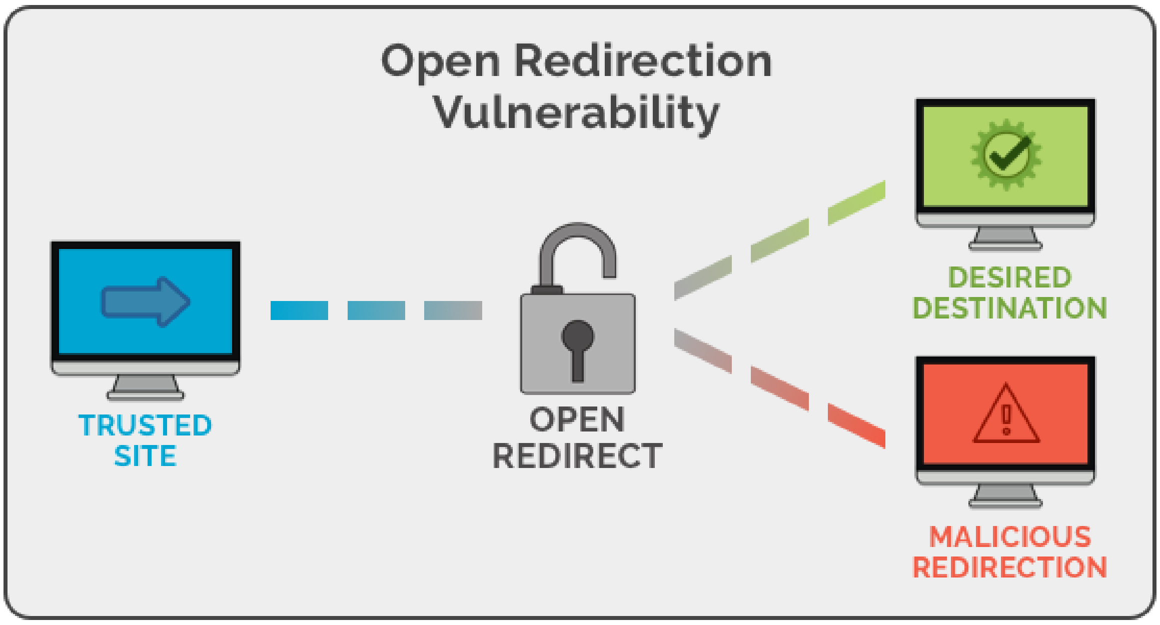
- You can embed credentials in a URL before the hostname, using the **@** character. For example: https://expected-host@evil-host.
- You can use the # character to indicate a URL fragment. For example: https://evil-host#expected-host.
- You can leverage the DNS naming hierarchy to place required input 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.

1.5 Bypassing SSRF filters via open redirection

Suppose the user-submitted URL is strictly validated to prevent malicious exploitation of the SSRF behavior. However, the application whose URLs are allowed contains an open redirection vulnerability. Provided the API used to make the back-end HTTP request supports redirections, you can construct a URL that satisfies the filter such as:

https://blog-vuln.herokuapp.com/login?next= http://192.168.0.68/admin

The filter will only validates the main domain blog-vul, so we was break filter then leverange Open redirect vulnerability to redirect to target .

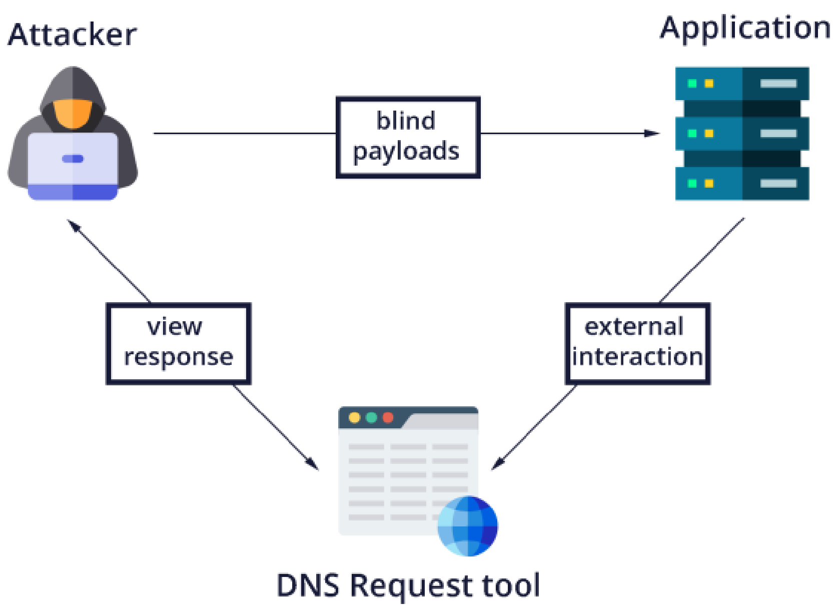


2. Blind SSRF

As the name indicates, in this type of SSRF, hackers don’t get data back from the server. You see this commonly when the request is just to trigger some action on the victim server without returning anything back to the requesting server. Hackers use this type of SSRF when they want to make some changes using the victim server. Because they needn’t see results for the activity.

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.

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.



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.

3. How to detect SSRF vulnerabilities

Many server-side request forgery vulnerabilities are relatively easy to spot, because the application's normal traffic involves request parameters containing full URLs. But it also have some case of SSRF vulnerabilities are harder to locate.

Partial URLs in requests

Sometimes, an application places only a hostname or part of a URL path into request parameters. The value submitted is then incorporated server-side into a full URL that is requested.

URLs within data formats

Some applications transmit data in formats whose specification allows the inclusion of URLs that might get requested by the data parser for the format. An obvious example of this is the XML data format, which has been widely used in web applications to transmit structured data from the client to the server. When an application accepts data in XML format and parses it, it might be vulnerable to XXE injection, and in turn be vulnerable to SSRF via XXE

SSRF via the Referer header

Some applications employ server-side analytics software that tracks visitorsand often the analytics software will actually visit any third-party URL that appears in the Referer header. As a result, the Referer header often represents fruitful attack surface for SSRF vulnerabilities.

IV. How To Prevent An SSRF Attack

- Use a **whitelist** of approved domains and protocols through which remote resources can be acquired by the web server.
- **User input** should always be sanitised or validated.
- **Restrict request protocols**: If only HTTP or HTTPS are used by your application to make requests, allow only these URL schemas. If URL schemas like file:///, dict://, ftp:// and gopher:// are disabled, the attacker won't be able to use the web application to make dangerous requests using these URL schemas.

- Services like **Memcached, Redis, Elasticsearch**, and **MongoDB** do not need authentication by default. Server Side Request Forgery vulnerabilities may be used by an attacker to access any of these services without any authentication. Therefore it is best to allow authentication wherever possible, even for services on the local network to ensure security for the web application.

Reference

OWASP: https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

Web Security Academy: https://portswigger.net/web-security/ssrf

Beagle Security : https://beaglesecurity.com/blog/article/server-side-request-forgery-attack.html

Blog:

https://blog.sqreen.com/ssrf-explained/

https://blog.intigriti.com/hackademy/server-side-request-forgery-ssrf/

https://portswigger.net/research/top-10-web-hacking-techniques-of-2017#1