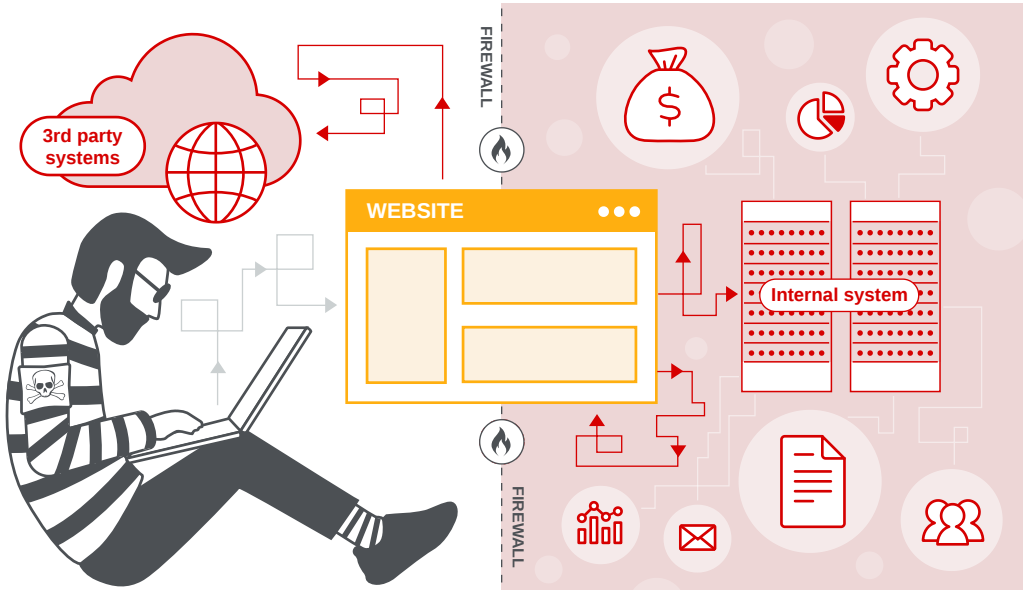
and exploit SSRF vulnerabilities.

What is SSRF?

Server-side request forgery is a web security vulnerability that allows an attacker to cause the server-side application to make request unintended location.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems. This could leak sensitive d as authorization credentials.



Labs

If you're familiar with the basic concepts behind SSRF vulnerabilities and want to practice exploiting them on some realistic, deliberate vulnerable targets, you can access labs in this topic from the link below.

- View all SSRF labs

What is the impact of SSRF attacks?

A successful SSRF attack can often result in unauthorized actions or access to data within the organization. This can be in the vulnera application, or on other back-end systems that the application can communicate with. In some situations, the SSRF vulnerability might attacker to perform arbitrary command execution.

An SSRF exploit that causes connections to external third-party systems might result in malicious onward attacks. These can appear t originate from the organization hosting the vulnerable application.

Common SSRF attacks

SSRF attacks often exploit trust relationships to escalate an attack from the vulnerable application and perform unauthorized actions. trust relationships might exist in relation to the server, or in relation to other back-end systems within the same organization.

SSRF attacks against the server

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

For example, imagine a shopping application that lets the user view whether an item is in stock in a particular store. To provide the sto information, the application must query various back-end REST APIs. It does this by passing the URL to the relevant back-end API en a front-end HTTP request. When a user views the stock status for an item, their browser makes the following request:

```
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%3D1
```

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

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

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

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

The server fetches the contents of the `/admin` URL and returns it to the user.

An attacker can visit the `/admin` URL, but the administrative functionality is normally only accessible to authenticated users. This me attacker won't see anything of interest. However, if the request to the `/admin` URL comes from the local machine, the normal access are bypassed. The application grants full access to the administrative functionality, because the request appears to originate from a tru location.

LAB APPRENTICE **Basic SSRF against the local server →**

Why do applications behave in this way, and implicitly trust requests that come from the local machine? This can arise for various reas

- The access control check might be implemented in a different component that sits in front of the application server. When a conn made back to the server, the check is bypassed.
- For disaster recovery purposes, the application might allow administrative access without logging in, to any user coming from the machine. This provides a way for an administrator to recover the system if they lose their credentials. This assumes that only a fu user would come directly from the server.
- The administrative interface might listen on a different port number to the main application, and might not be reachable directly by

These kind of trust relationships, where requests originating from the local machine are handled differently than ordinary requests, ofte SSRF into a critical vulnerability.

SSRF attacks against other back-end systems

In some cases, the application server is able to interact with back-end systems that are not directly reachable by users. These system have non-routable private IP addresses. The back-end systems are normally protected by the network topology, so they often have a v security posture. In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication anyone who is able to interact with the systems.

In the previous example, imagine there is an administrative interface at the back-end URL `https://192.168.0.68/admin`. An att submit the following request to exploit the SSRF vulnerability, and access the administrative interface:

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

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

LAB APPRENTICE
Basic SSRF against another back-end system →

Circumventing common SSRF defenses

It is common to see applications containing SSRF behavior together with defenses aimed at preventing malicious exploitation. Often, t defenses can be circumvented.

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 situ can often circumvent the filter using the following techniques:

- Use an alternative IP representation of `127.0.0.1`, such as `2130706433`, `017700000001`, or `127.1`.
- Register your own domain name that resolves to `127.0.0.1`. You can use `spoofed.burpcollaborator.net` for this purp
- Obfuscate blocked strings using URL encoding or case variation.
- Provide a URL that you control, which redirects to the target URL. Try using different redirect codes, as well as different protocols target URL. For example, switching from an `http:` to `https:` URL during the redirect has been shown to bypass some anti-S filters.

LAB PRACTITIONER
SSRF with blacklist-based input filter →

SSRF with whitelist-based input filters

Some applications only allow inputs that match, a whitelist of permitted values. The filter may look for a match at the beginning of the i contained within in it. You may be able to bypass this filter by exploiting inconsistencies in URL parsing.

The URL specification contains a number of features that are likely to be overlooked when URLs implement ad-hoc parsing and valida this method:

- You can embed credentials in a URL before the hostname, using the `@` character. For example:
  `https://expected-host:fakepassword@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 URL-encoded characters differently than the code that performs the back-end HTTP request. You can also try double-encoding c some servers recursively URL-decode the input they receive, which can lead to further discrepancies.
- You can use combinations of these techniques together.

LAB EXPERT

SSRF with whitelist-based input filter →

Read more

- A new era of SSRF

Bypassing SSRF filters via open redirection

It is sometimes possible to bypass filter-based defenses by exploiting an open redirection vulnerability.

In the previous example, imagine the user-submitted URL is strictly validated to prevent malicious exploitation of the SSRF behavior. H the application whose URLs are allowed contains an open redirection vulnerability. Provided the API used to make the back-end HTTF supports redirections, you can construct a URL that satisfies the filter and results in a redirected request to the desired back-end targe

For example, 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/
```

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

LAB PRACTITIONER

SSRF with filter bypass via open redirection vulnerability →

Blind SSRF vulnerabilities

Blind SSRF vulnerabilities occur if you can cause an application to issue a back-end HTTP request to a supplied URL, but the respons the back-end request is not returned in the application's front-end response.

Blind SSRF is harder to exploit but sometimes leads to full remote code execution on the server or other back-end components.

Read more

- Finding and exploiting blind SSRF vulnerabilities

Finding hidden attack surface for SSRF vulnerabilities

Many server-side request forgery vulnerabilities are easy to find, because the application's normal traffic involves request parameters containing full URLs. Other examples of SSRF 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 incorpo server-side into a full URL that is requested. If the value is readily recognized as a hostname or URL path, the potential attack surface obvious. However, exploitability as full SSRF might be limited because you do not control the entire URL that gets requested.

URLs within data formats

Some applications transmit data in formats with a specification that allows the inclusion of URLs that might get requested by the data p the format. An obvious example of this is the XML data format, which has been widely used in web applications to transmit structured d the client to the server. When an application accepts data in XML format and parses it, it might be vulnerable to XXE injection. It might vulnerable to SSRF via XXE. We'll cover this in more detail when we look at XXE injection vulnerabilities.

SSRF via the Referer header

Some applications use server-side analytics software to tracks visitors. This software often logs the Referer header in requests, so it c incoming links. Often the analytics software visits any third-party URLs that appear in the Referer header. This is typically done to ana contents of referring sites, including the anchor text that is used in the incoming links. As a result, the Referer header is often a useful surface for SSRF vulnerabilities.

See Blind SSRF vulnerabilities for examples of vulnerabilities involving the Referer header.

Read more

- Cracking the lens: Targeting auxiliary systems
- URL validation bypass cheat sheet