COMMON SSRF ATTACKS

SSRF attacks against the server itself

For example there is a request like below

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 request causes the server to make a request to the specified URL, retrieve the stock status, 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:

POST /product/stock HTTP/1.0

Content-Type: application/x-www-form-urlencoded

Content-Length: 118

stockApi=http://localhost/admin

Here, the server will fetch the contents of the /admin URL and return it to the user. Now of course, the attacker could just visit the /admin URL directly. But the administrative functionality is ordinarily accessible only to suitable authenticated users. So an attacker who simply visits the URL directly won't see anything of interest. However, when the request to the /admin URL comes from the local machine itself, the normal access controls are bypassed. The application grants full access to the administrative functionality, because the request appears to originate from a trusted location.

### 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://192.168.0.68/admin. Here, an attacker can exploit the SSRF vulnerability to access the administrative interface by submitting the following request:

POST /product/stock HTTP/1.0

Content-Type: application/x-www-form-urlencoded

Content-Length: 118

stockApi=http://192.168.0.68/admin

## **Circumventing common SSRF defenses**

It is common to see applications containing SSRF behavior together with defenses aimed at preventing malicious exploitation. Often, these 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 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. You can use spoofed.burpcollaborator.net for this purpose.

Obfuscating blocked strings using URL encoding or case variation.

**127.0.0.1 🡪 2130706433, 017700000001, 127.1**

IPv4 addresses can be represented in multiple ways. For example the default loopback IP can be one of:

* 127.0.0.1
* 0177.0.0.1
* 0x7f.0.0.1
* 127.0.1
* 127.1
* 2130706433
* 017700000001
* 0x7f000001

### 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.

The URL specification contains a number of features that are liable to be overlooked when implementing ad hoc parsing and validation of URLs:

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.

### Bypassing SSRF filters via open redirection

For example, suppose 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/admin

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

## **Blind SSRF vulnerabilities**

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

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

### SSRF via the Referer header –-> look again

### How to find and exploit blind SSRF vulnerabilities

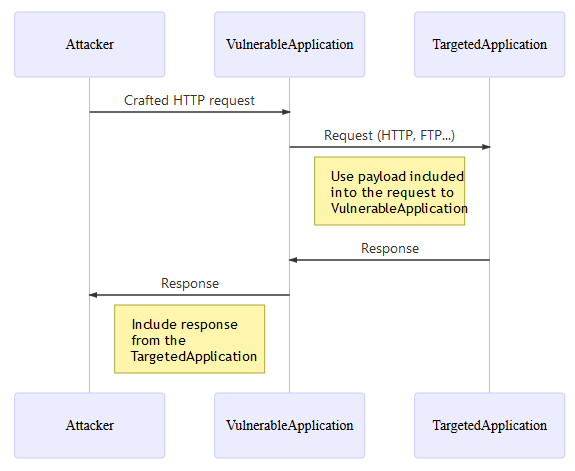
The most reliable way to detect blind SSRF vulnerabilities is using out-of-band (OAST) techniques. This involves attempting to trigger an HTTP request to an external system that you control, and monitoring for network interactions with that system.

The easiest and most effective way to use out-of-band techniques is using Burp Collaborator. You can use the Burp Collaborator client to generate unique domain names, send these in payloads to the application, and monitor for any interaction with those domains. If an incoming HTTP request is observed coming from the application, then it is vulnerable to SSRF.

Note

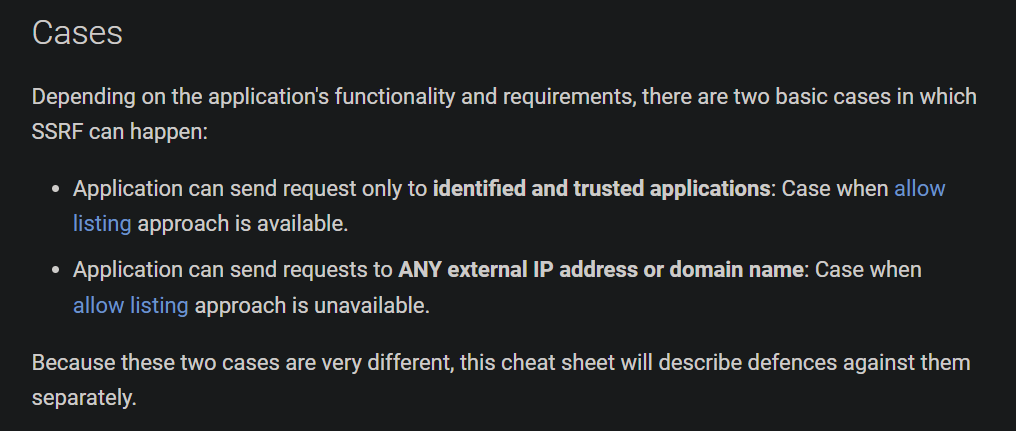
It is common when testing for SSRF vulnerabilities to observe a DNS look-up for the supplied Collaborator domain, but no subsequent HTTP request. This typically happens because the application attempted to make an HTTP request to the domain, which caused the initial DNS lookup, but the actual HTTP request was blocked by network-level filtering. It is relatively common for infrastructure to allow outbound DNS traffic, since this is needed for so many purposes, but block HTTP connections to unexpected destinations.

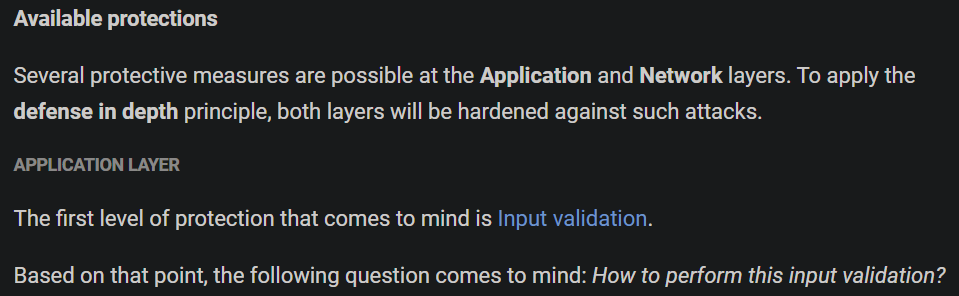
Overview of SSRF common flow



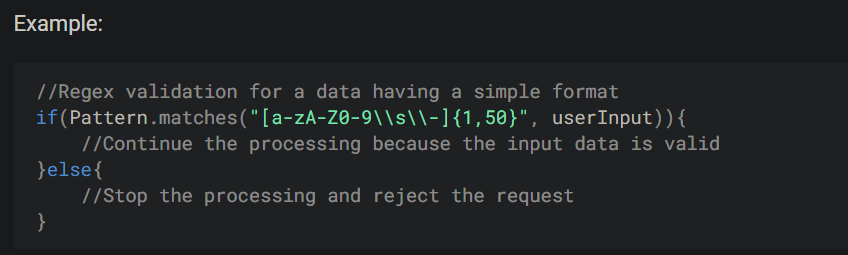
**IMPORTANT**

* If the application is vulnerable to [XML eXternal Entity (XXE) injection](https://portswigger.net/web-security/xxe) then it can be exploited to perform a [SSRF attack](https://portswigger.net/web-security/xxe#exploiting-xxe-to-perform-ssrf-attacks), take a look at the [XXE cheat sheet](https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html) to learn how to prevent the exposure to XXE.





String validations can be perform by using regex



**IP address**[**¶**](https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html#ip-address)

In the context of SSRF, there are 2 possible validations to perform:

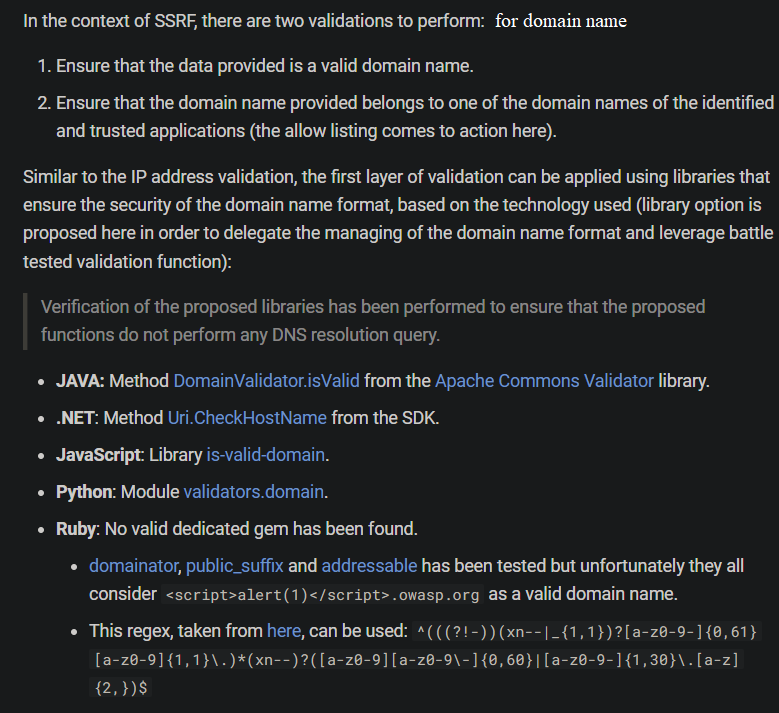
1. Ensure that the data provided is a valid IP V4 or V6 address.
2. Ensure that the IP address provided belongs to one of the IP addresses of the identified and trusted applications.

The first layer of validation can be applied using libraries that ensure the security of the IP address format, based on the technology used (library option is proposed here to delegate the managing of the IP address format and leverage battle-tested validation function):

Verification of the proposed libraries has been performed regarding the exposure to bypasses (Hex, Octal, Dword, URL and Mixed encoding) described in this [article](https://medium.com/@vickieli/bypassing-ssrf-protection-e111ae70727b).

* **JAVA:** Method [InetAddressValidator.isValid](https://commons.apache.org/proper/commons-validator/apidocs/org/apache/commons/validator/routines/InetAddressValidator.html#isValid(java.lang.String)) from the [Apache Commons Validator](https://commons.apache.org/proper/commons-validator/) library.
  + **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.
* **.NET**: Method [IPAddress.TryParse](https://docs.microsoft.com/en-us/dotnet/api/system.net.ipaddress.tryparse?view=netframework-4.8) from the SDK.
  + **It is exposed** to bypass using Hex, Octal, Dword and Mixed encoding but **NOT** the URL encoding.
  + As allow listing is used here, any bypass tentative will be blocked during the comparison against the allowed list of IP addresses.
* **JavaScript**: Library [ip-address](https://www.npmjs.com/package/ip-address).
  + **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.
* **Python**: Module [ipaddress](https://docs.python.org/3/library/ipaddress.html) from the SDK.
  + **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.
* **Ruby**: Class [IPAddr](https://ruby-doc.org/stdlib-2.0.0/libdoc/ipaddr/rdoc/IPAddr.html) from the SDK.
  + **It is NOT exposed** to bypass using Hex, Octal, Dword, URL and Mixed encoding.

After ensuring the validity of the incoming IP address, the second layer of validation is applied. An allow list is created after determining all the IP addresses (v4 and v6 to avoid bypasses) of the identified and trusted applications. The valid IP is cross-checked with that list to ensure its communication with the internal application (string strict comparison with case sensitive).





After ensuring the validity of the incoming domain name, the second layer of validation is applied:

1. Build an allow list with all the domain names of every identified and trusted applications.
2. Verify that the domain name received is part of this allow list (string strict comparison with case sensitive).

To address that issue, the following action must be taken in addition of the validation on the domain name:

1. Ensure that the domains that are part of your organization are resolved by your internal DNS server first in the chains of DNS resolvers.
2. Monitor the domains allow list in order to detect when any of them resolves to a/an:
3. Local IP address (V4 + V6).
4. Internal IP of your organization (expected to be in private IP ranges) for the domain that are not part of your organization.

The following Python3 script can be used, as a starting point, for the monitoring mentioned above:

# Dependencies: pip install ipaddress dnspython  
import ipaddress  
import dns.resolver  
  
# Configure the allow list to check  
DOMAINS\_ALLOWLIST = ["owasp.org", "labslinux"]  
  
# Configure the DNS resolver to use for all DNS queries  
DNS\_RESOLVER = dns.resolver.Resolver()  
DNS\_RESOLVER.nameservers = ["1.1.1.1"]  
  
def verify\_dns\_records(domain, records, type):  
 *"""  
 Verify if one of the DNS records resolve to a non public IP address.  
 Return a boolean indicating if any error has been detected.  
 """* error\_detected = False  
 if records is not None:  
 for record in records:  
 value = record.to\_text().strip()  
 try:  
 ip = ipaddress.ip\_address(value)  
 # See https://docs.python.org/3/library/ipaddress.html#ipaddress.IPv4Address.is\_global  
 if not ip.is\_global:  
 print("[!] DNS record type '%s' for domain name '%s' resolve to  
 a non public IP address '%s'!" % (type, domain, value))  
 error\_detected = True  
 except ValueError:  
 error\_detected = True  
 print("[!] '%s' is not valid IP address!" % value)  
 return error\_detected  
  
def check():  
 *"""  
 Perform the check of the allow list of domains.  
 Return a boolean indicating if any error has been detected.  
 """* error\_detected = False  
 for domain in DOMAINS\_ALLOWLIST:  
 # Get the IPs of the current domain  
 # See https://en.wikipedia.org/wiki/List\_of\_DNS\_record\_types  
 try:  
 # A = IPv4 address record  
 ip\_v4\_records = DNS\_RESOLVER.query(domain, "A")  
 except Exception as e:  
 ip\_v4\_records = None  
 print("[i] Cannot get A record for domain '%s': %s\n" % (domain,e))  
 try:  
 # AAAA = IPv6 address record  
 ip\_v6\_records = DNS\_RESOLVER.query(domain, "AAAA")  
 except Exception as e:  
 ip\_v6\_records = None  
 print("[i] Cannot get AAAA record for domain '%s': %s\n" % (domain,e))  
 # Verify the IPs obtained  
 if verify\_dns\_records(domain, ip\_v4\_records, "A")  
 or verify\_dns\_records(domain, ip\_v6\_records, "AAAA"):  
 error\_detected = True  
 return error\_detected  
  
if \_\_name\_\_== "\_\_main\_\_":  
 if check():  
 exit(1)  
 else:  
 exit(0)

<https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html#case-2-application-can-send-requests-to-any-external-ip-address-or-domain-name>