TryHackMe SSRF

Saikat Karmakar | Aug 2 : 2021

SSRF

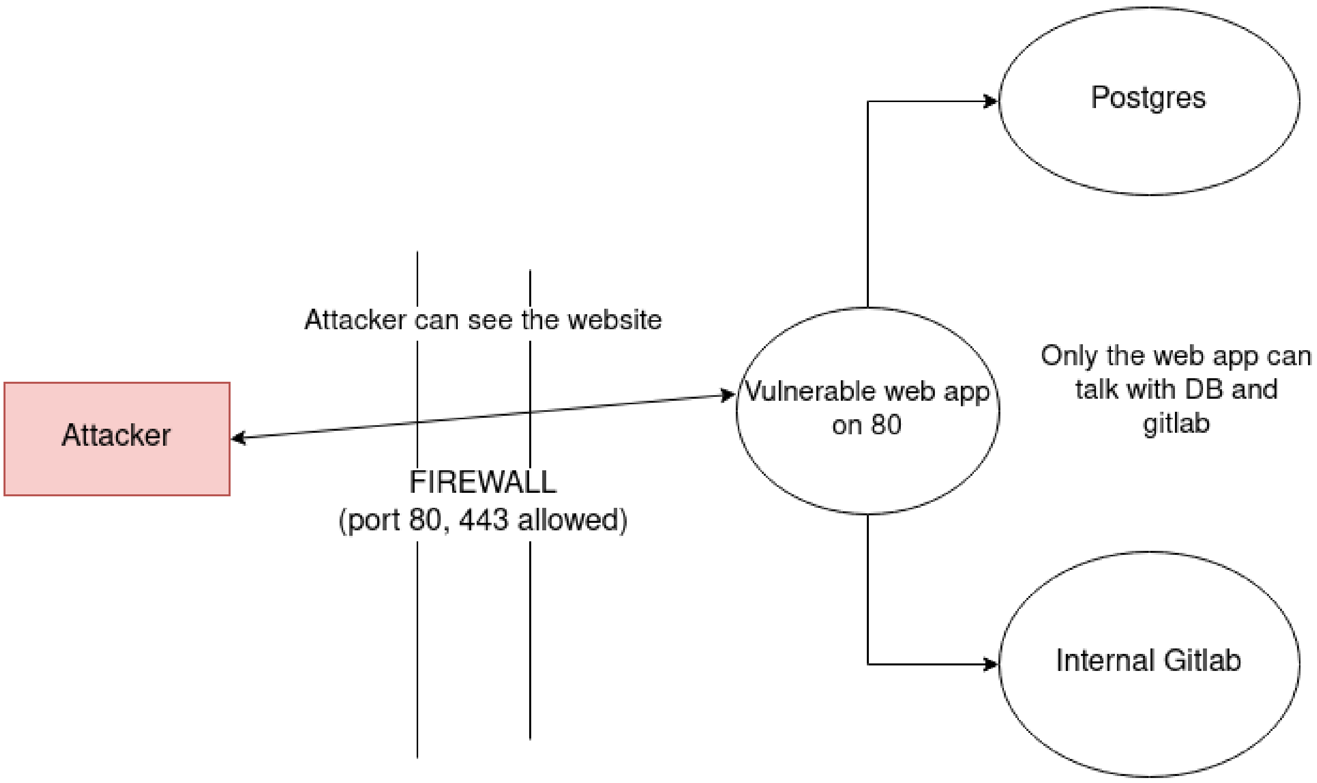
What is SSRF?

Overview

This room will focus on giving you a detailed understanding of what a server-side request forgery -- better known as SSRF -- vulnerability is, as well as how you can discover whether any website is vulnerable to any form of SSRF, and how an attacker can leverage SSRF to their own advantage.

What is SSRF?

In simpler terms, SSRF is a vulnerability in web applications whereby an attacker can make further HTTP requests through the server. An attacker can make use of this vulnerability to communicate with any internal services on the server's network which are generally protected by firewalls.


Postgres
Attacker can see the website
Attacker
Vulnerable web app on 80
Only the web app can talk with DB and gitlab
FIREWALL (port 80, 443 allowed)
Internal Gitlab

Now if you focus on the above diagram, In a normal case the attacker would only be able to visit the website and see the website data. The server running the website is allowed to communicate to the internal GitLab or Postgres database, but the user may not, because the firewall in the middle only allows access to ports 80 (HTTP) and 443 (HTTPS).However, SSRF would give an attacker the power to make a connection to Postgres and see its data by first connecting to the website server, and then using that to connect to the database. Postgres would think that the website is requesting something from the database, but in reality, it's the attacker making use of an SSRF vulnerability in website to get the data. The process would usually be something like this: an attacker finds an SSRF vulnerability on a website. The firewall allows all requests to the website. The attacker then exploits the SSRF vulnerability by forcing the webserver to request data from the database, which it then returns to the attacker. Because the request is coming from the webserver, rather than directly from the attacker, the firewall allows this to pass.

Cause of the vulnerability

The main cause of the vulnerability is (as it often is) blindly trusting input from a user. In the case of an SSRF vulnerability, a user would be asked to input a URL (or maybe an IP address). The web application would use that to make a request. SSRF comes about when the input hasn't been properly checked or filtered.

Let's look at some vulnerable code:

Example 1: PHP

Assume there is an application that takes the URL for an image, which the web page then displays for you. The vulnerable SSRF code would look like this:

```php
<?php
    if (isset($_GET['url']))
    {
        $url = $_GET['url'];
        $image = fopen($url, 'rb');
        header("Content-Type: image/png");
        fpassthru($image);
    }
?>
```
language-php

This is simple PHP code which checks if there is information sent in a 'url' parameter then, without performing any kind of check on it, the code simply makes a request to the user-submitted URL. Attackers essentially have full control of the URL and can make arbitrary GET requests to any website on the Internet through the server -- as well as accessing resources on the server itself.

Example 2: Python

```python
from flask import Flask, request, render_template, redirect
import requests

app = Flask(__name__)

@app.route("/")

def start():
    url = request.args.get("id")
    r = requests.head(url, timeout=2.000)
    return render_template("index.html", result = r.content)

if __name__ == "__main__":
    app.run(host = '0.0.0.0')
```
language-python

The above example shows a very small flask application which does the same thing:

1. It takes the value of the "url" parameter.

2. Then it makes a request to the given URL and shows the content of that URL to the user.

Again we see that there is no sanitisation or any kind of check performed on the user input. This is why you should always try as many different payloads as you can when testing an application.

Speaking of payloads, in the next task, we'll see what kind of payloads are used in SSRF.

Now we are going to learn what kind of payloads are used to exploit an SSRF vulnerability. To understand this we are going to use a live demonstration. Connect to `http://10.10.112.120:5000`

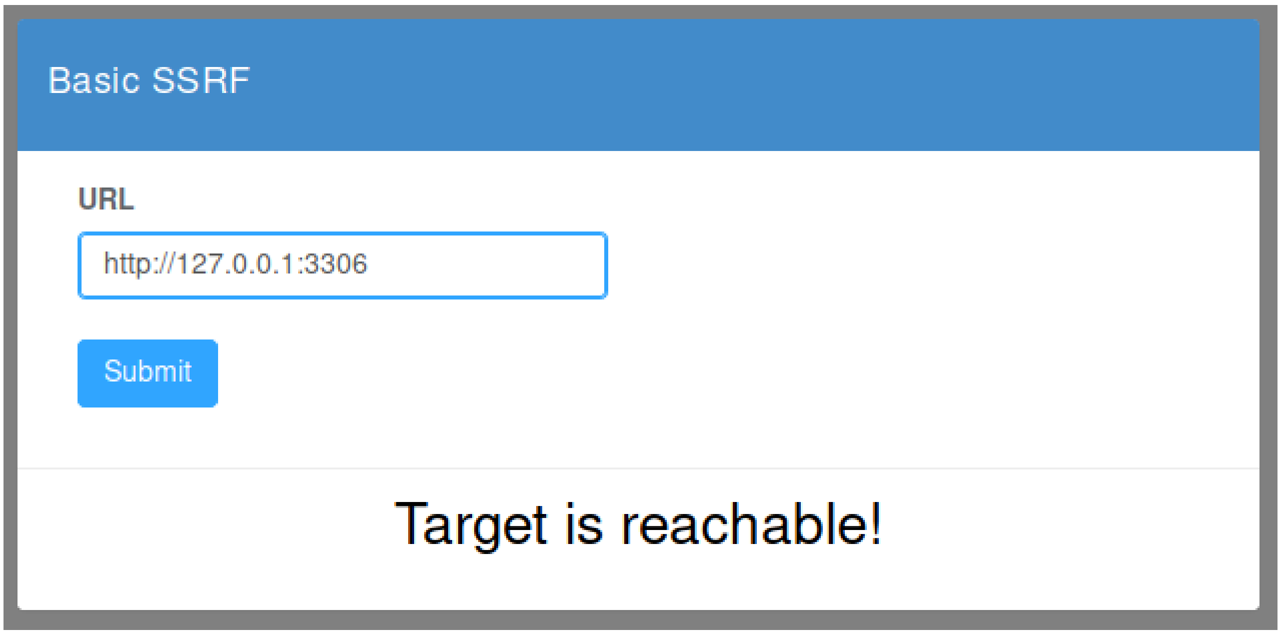
The page gives us the option to enter a url, which it will fetch for us.

Basic Payloads

Let's start with the basic payload. This payload might give you the hint that there is an SSRF vulnerability, and give you a hint as to which payloads which you should try next.

Initially, start by searching for the localhost IP (127.0.0.1) with any port to see if the port is running a service. Say you wanted to check if the server has a hidden database, you might search for `http://127.0.0.1:3306`, 3306 is the port for MySQL DB so if there is a database running, you will likely get a positive response.

If we try this payload on our VM we will see the following output:



Target is reachable!

This shows that the port 3306 is open.

In a similar manner, we could also have used "localhost" or "0.0.0.0" in place of 127.0.0.1

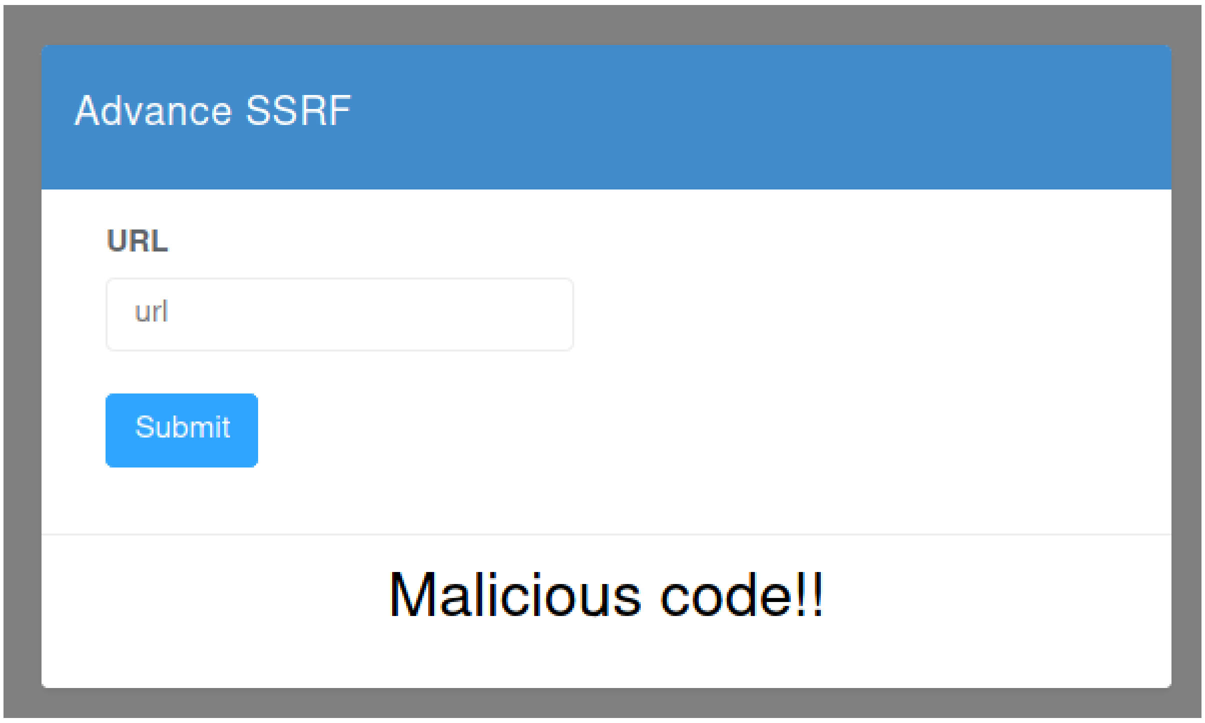
Advanced payloads

Now it's very possible that some sort of sanitization will be being applied to the input, so the system might detect strings like "localhost" or "127.0.0.1" and stop the request. That said, it's possible to try and bypass those kinds of restrictions.

The very first way is to try the IPv6 version of the localhost i.e [::]. So the payload from before would look like this `http://[::]:3306`.

The basic page we've already worked with doesn't have any filter, so to try these advanced payloads head to `http://10.10.112.120:5000/advanced`. The page is very similar to the previous one but, you'll notice that if you try the basic payload it detects it as "Malicious code".

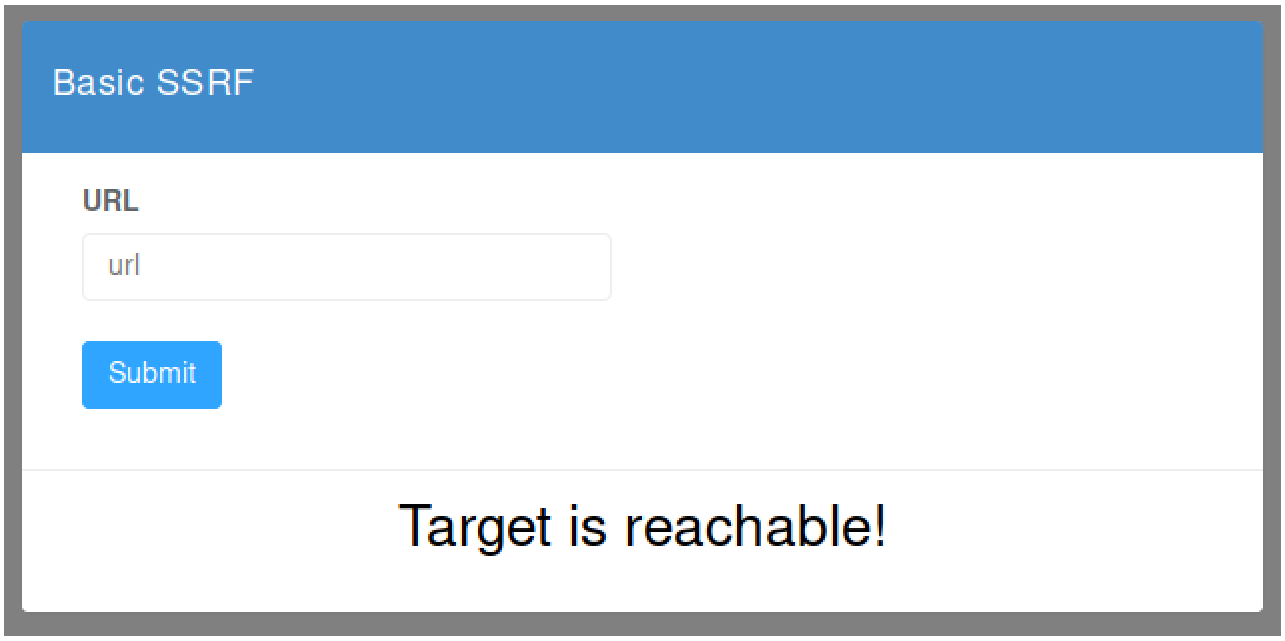
Try it for yourself: try to enter the old payloads such as "`http://127.0.0.1:3306`" or `http://localhost:3306` -- you'll get something like:



Malicious code!!

As expected, this is because there are SSRF checks in place, but still it's possible to bypass them. It's also possible that if you try to access `http://[::]:3306`, you might get "target not reachable". When this happens it can be the result of how the framework is handling input in the application. If it was a PHP application then this would work, but flask/Django might interpret these payloads differently. If you fail with that payload, try removing the brackets (i.e try `http://:::3306`): remember that the third colon indicates the separation between port and IP.

So if we enter: `http://:::3306` we see something like:



Target is reachable!

It is possible that the IPv6 payload may also be detected. In that case what we usually do is to encode our IP: either into a decimal format or a hexadecimal format.

The IP "`127.0.0.1`" can be replaced with its Decimal and Hexadecimal counterparts to bypass the restrictions. The decimal version of the localhost IP would be "2130706433" and the Hexadecimal version would be "0x7f000001".

Note - There is a script to do this conversion process, you can find it here

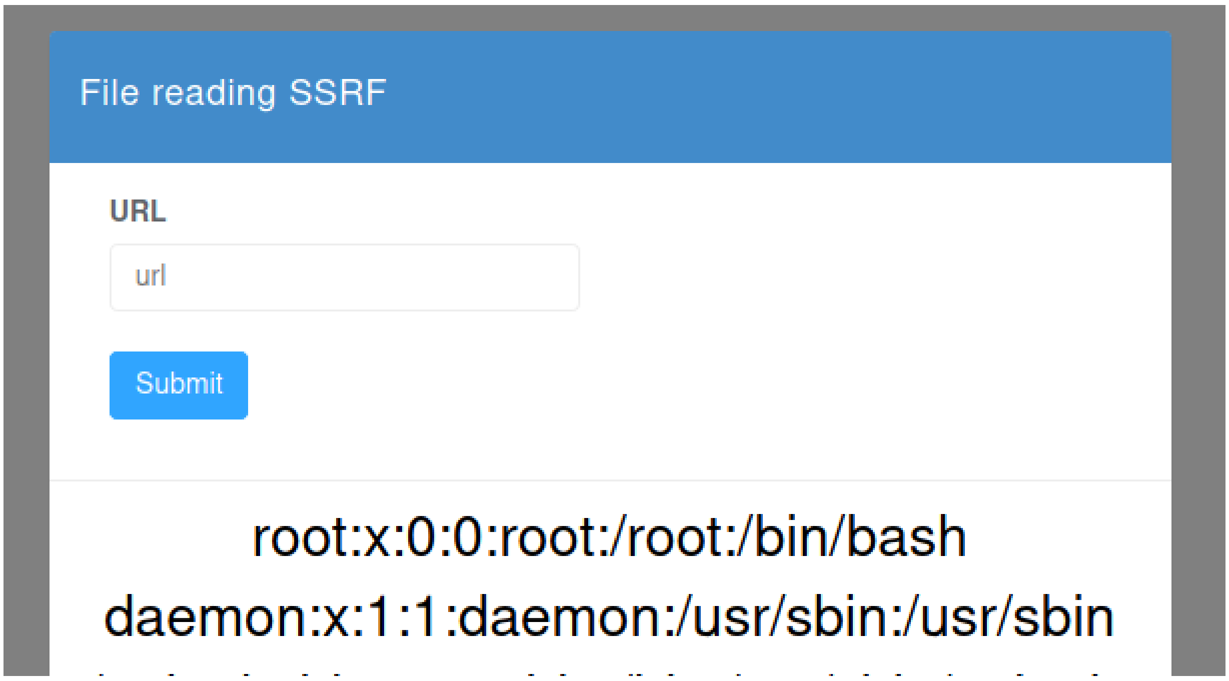
Reading files

Port scanning isn't the only thing that we do with SSRF -- we can also read files from the server, but only if we use the proper schema (i.e for a HTTP request we would start the URL with "http://" -- in a similar manner if we start the URL with "file://" it would then try to read the files from the server itself).

So, for example, a simple SSRF file reading payload would be `file:///etc/passwd`, to read the /etc/passwd file on a Linux machine.

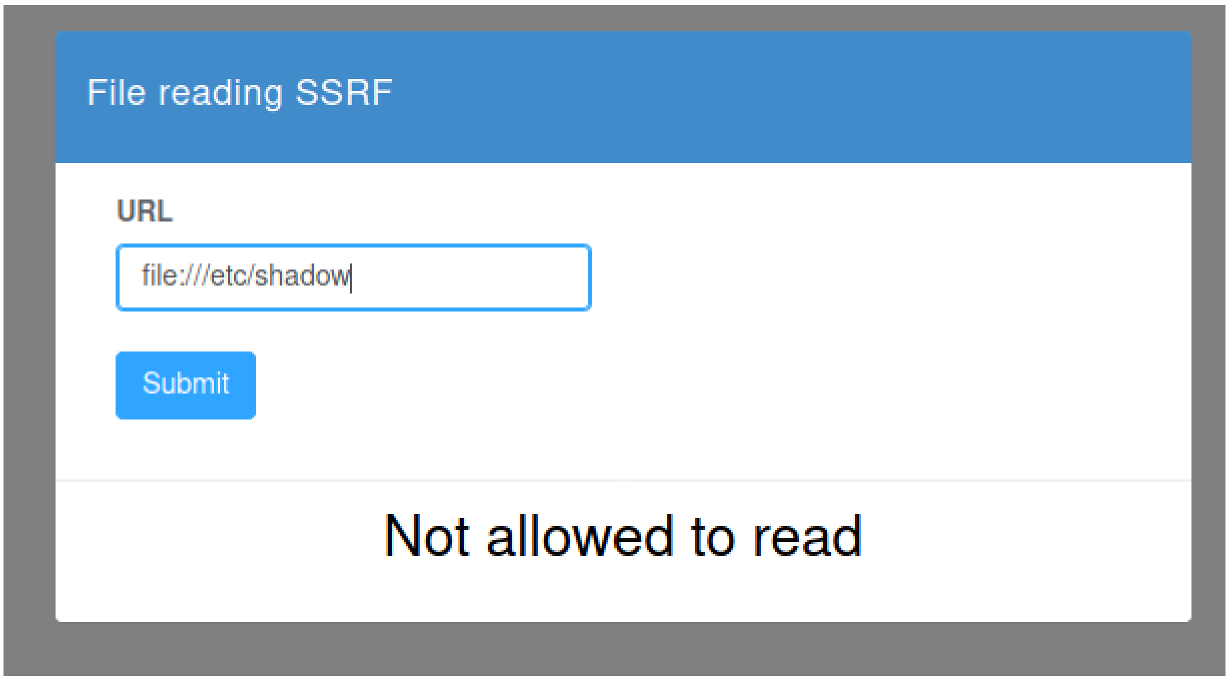
To see this happen for yourself, go to `http://10.10.112.120:5000/filessrf`, you'll get another very similar form as with the advanced and basic SSRF tutorials -- but one difference that you'll notice is that you can enter "file://" in this one.

So if now we try to read the file we'll be able to see the contents:



```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin
```

Remember is that it's unlikely that we'll be able to read files from any higher privileged user such as "root".



file:///etc/shadow

Not allowed to read

There are a lot of other types of payload that can be used (such as URL encoding, double URL encoding, using schemes like dict, etc). You can see some of these payloads here