Server-Side Request Forgery (SSRF) — Part 3: Other advanced techniques

Maxime Leblanc Follow
Sep 11, 2017 · 5 min read

Hello geeks; This is the third and final part of this series about SSRF attacks. Let's remember that in Part 1, we looked at the basic concepts behind such kinds of attacks. Then, in Part 2 we learned how to circumvent protections that a website could put in place by using some IPv4 magic tricks from Nicolas Grégoire. In this final part, I will explore with you two techniques I learned from Orange Tsai at the DEFCON 25. All the credit for the present technique goes to him and I of course thank him very much for sharing his work with the community 😀. I also remind to you that all the application code I will be using for the following demonstrations is publicly available on my SSRF Lab hosted on GitHub and this last part is still using a Docker-Compose script that should get all of it up and running as-is within it's own network.

So, let's use one more time our Super Popular App presented in Part 1 and Part 2, which has only one purpose: It's a test page for WebHook integration, and should be able only to call external network addresses:

Also note that this time, our WebHook code is written in Python Flask. We still have our other `secret.corp` internal server located at `10.0.0.3`. Remember my conclusion of Part 2, that could be rephrased as:

> *Do not try to parse URLs yourself; Instead use libraries that are made for this.*

So you do exactly that, and using the usual Python 2.7 libraries you implement a filter on the `hostname` part of the url that was submitted:

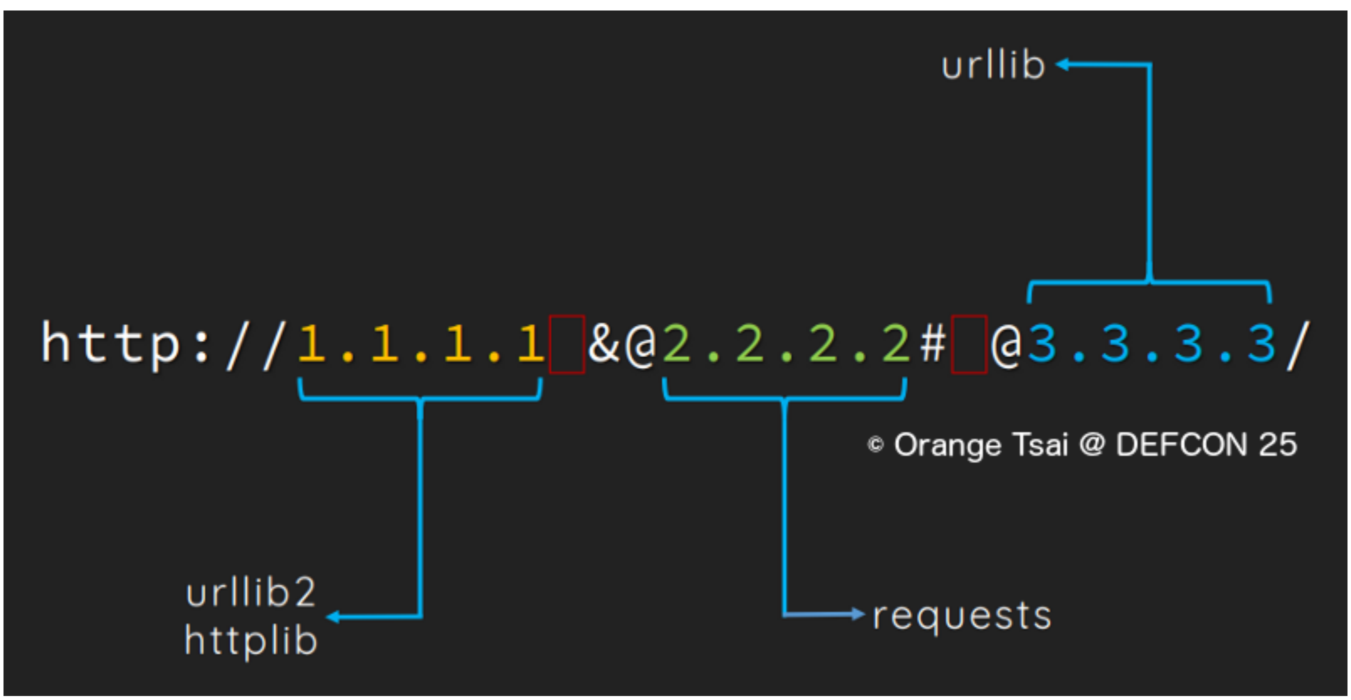
```
url=request.form['handler']
host = urlparse.urlparse(url).hostname
if host == 'secret.corp':
    return 'Restricted Area!'
else:
    return urllib.urlopen(url).read()
```

Let's try to access this famous server by trying `http://secret.corp`:

Well, the filter seems to do it's job! Now, just for science, let's try `http://google.com# @secret.corp` instead (notice the whitespace character in the middle of the URL):

Wait, what happened again? Our filter using a standard parsing library got bypassed and authorized a request it wasn't supposed to? Well, yes and no; The problem here is that we use one library to filter the `hostname` and another one to do the actual request, and they do not interpet whitespaces in URLs the same way! The `urlparse` library recognized the URL as being addressed to `google.com` while `urllib` thought it was for `secret.corp` and acted accordingly. So here it's the difference of interpretation between the two libraries that lead to this SSRF vulnerability.

But it gets worse… This is part of a slide from Orange Tsai's presentation:



Source: Orange Tsai @ DEFCON 25

As you can see, Python libraries all have their own ways of handling whitespaces in URLs, and if you don't stick with one way throughout your whole application, unexpected results might happen when dealing with those. Here is what says the official RFC3968:

> *The authority component is preceded by a double slash ("//") and is terminated by the next slash ("/"), question mark ("?"), or number sign ("#") character, or by the end of the URI*

From this definition, it is kind-of ambiguous which library behaves right; I would personally say that none correspond to this behaviour exactly.

Future work: Protocol injection

As a conclusion, I would like to introduce the notion of protocol injection: Remember in Part 1 the structure of a URL:

scheme://user:pass@host:port/path?query=value#fragment

How could-it be possible to bypass a fixed `scheme://` with only access to the HTTP(S) protocol? Meet the newline injection technique. Now, at the time of this writing, I have not implemented it as a webserver exploit in my SSRF Lab, but it will be available on this GitHub repository folder in a short time. Meanwhile, let's consider a user on our WebHook testing server. A mail server is located at `mail.corp` , using a default Ubuntu Postfix installation. Our WebHook server can only make HTTP requests. However, if we are able to inject newline characters (`\r\n`) as in the hostname and the library accepts it, it is possible to send valid commands to the SMTP server during the SSL handshake. This is what the TCP response from the mail server looks like:

```
Request:
url = https://mail.corp\r\nHELO web.corp\r\nMAIL FROM...:25/
Response:
SMTP: 502 5.5.2 Error: command not recognized <SSL Gibberish>
[illegible]
SMTP: 250 2.1.0 Ok
...
SMTP: 502 5.5.2 Error: command not recognized <SSL Gibberish>
```


The trick here is that during the SSL handshake, the hostname is used entirely, including with it's n[illegible] if they were accepted by the library making the [illegible]g i[illegible]s newlines as "end-of-command" signals so we can insert valid commands in between the SSL handshake traffic. I still have some testing to do in order to have a valid POC of this in my Lab, but my first Wireshark captures are promising 😀. Note here that the same trick should apply for any text-based protocol : I can imagine well using this in order to register to an internal SIP server, for example. Leave a comment if you have a suggestion of implementation for me and final thanks to Orange Tsai and Nicolas Grégoire for their work!

Happy Hacking!