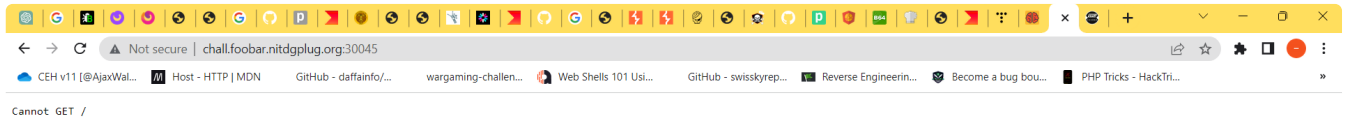


Foo-Bar CTF

Web-Inspect

Don't think, just push it to production....

When I opened this challenge for the first time, I thought the challenge was down

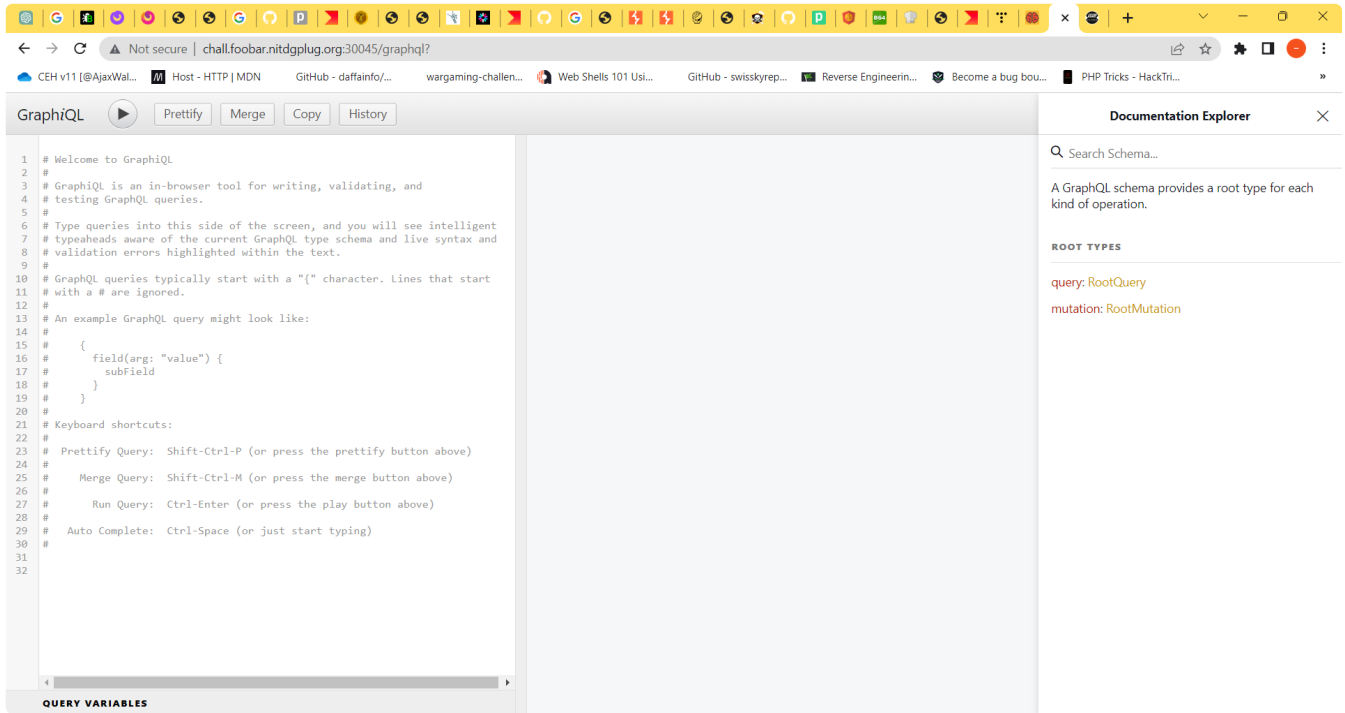


So I just left this one and started to try other challenges, but after a while, when I checked this challenge once again, some teams had solved it, but still the site looked the same. A new hint was also added to the challenge: "REST API IS BORING,SO I USED A MODERN ONE". From the hint, I figured that this challenge was something related to APIs. Basically An API is a set of defined rules that explain how computers or applications communicate with one another. API act as an intermediary between the web server and client and process data.

Now I have to find the correct API to access this site, so I started to search about REST API's alternatives, Modern API's,...

Luckily, I found a comparison between REST API vs. GraphQL in one of my Google search results.

So I quickly opened the following URL: `chall.foobar.nitdgplug.org:30045/graphql`.



Okay, now that I've confirmed that the site uses the GraphQL API, but where is the flag?

I was clueless here, but luckily I found a writeup related to this GraphQL:

<https://itsfading.github.io/posts/Hackerone-GraphQL-CTF-Writeup/>

The first thing to do with this GraphQL API is to understand how it is structured. We need to provide an introspection query to get the structure and the available resources.

Introspection is the ability to query which resources are available in the current API schema.

```
query IntrospectionQuery {
  __schema {
    queryType { name }
    mutationType { name }
    types {
      ...FullType
    }
    directives {
      name
      description
      locations
      args {
        ...InputValue
      }
    }
  }
}

fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name

    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
    deprecationReason
  }
  possibleTypes {
    ...TypeRef
  }
}

fragment InputValue on __InputValue {
  name
```

```

description
type { ...TypeRef }
defaultValue
}
fragment TypeRef on __Type {
  kind
  name
  ofType {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
            }
          }
        }
      }
    }
  }
}

```

After passing this query, we get the rootquery, columns, and their datatype in the form of the query itself, like this:

The screenshot shows the GraphQL Playground interface. On the left, a schema is defined with fragments for `__InputValue` and `__Type`. The `__Type` fragment is used in a query to introspect the schema. The output on the right is a JSON object representing the schema, including root types like `RootQuery` and `RootMutation`, and object types like `Event` and `Booking`.

```

47  ...TypeRef
48  }
49  }
50  fragment InputValue on __InputValue {
51    name
52    description
53    type { ...TypeRef }
54    defaultValue
55  }
56  fragment TypeRef on __Type {
57    kind
58    name
59    ofType {
60      kind
61      name
62      ofType {
63        kind
64        name
65        ofType {
66          kind
67          name
68          ofType {
69            kind
70            name
71            ofType {
72              kind
73              name
74              ofType {
75                kind
76                name
77                ofType {
78                  kind
79                  name
80                }
81              }
82            }
83          }
84        }
85      }
86    }
87  }

```

```

{
  "data": {
    "__schema": {
      "queryType": {
        "name": "RootQuery"
      },
      "mutationType": {
        "name": "RootMutation"
      },
      "types": [
        {
          "kind": "OBJECT",
          "name": "Booking",
          "description": null,
          "fields": [
            {
              "name": "_id",
              "description": null,
              "args": [],
              "type": {
                "kind": "NON_NULL",
                "name": null,
                "ofType": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ]
        },
        {
          "name": "event",
          "description": null,
          "args": [],
          "type": {
            "kind": "NON_NULL",
            "name": null,
            "ofType": {
              "kind": "OBJECT",
              "name": "Event",
              "ofType": null
            }
          }
        }
      ]
    }
  }
}

```

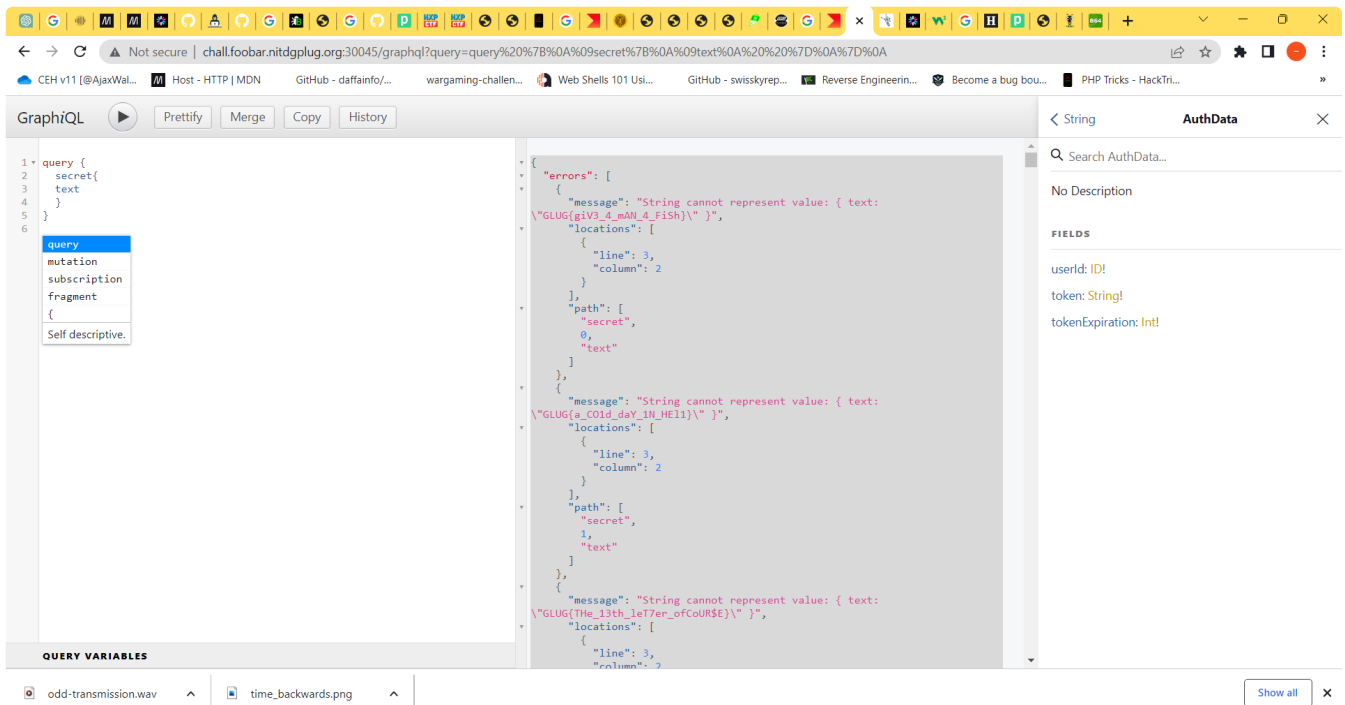
Difficult to read, right? Don't worry, we've got this amazing site <https://apis.guru/graphql-voyager/> which transforms this output query into an interactive graph.

The screenshot shows the GraphQL Voyager interface. On the left, there's a 'Type List' with a search bar and a list of types: `RootQuery`, `AuthData`, `Booking`, `Event`, `Flag`, and `User`. The main area displays a type graph where nodes represent types and edges represent fields. The `RootQuery` node has fields `secret` (type `Flag!`), `events` (type `Event!`), `bookings` (type `Booking!`), and `login` (type `AuthData!`). The `Event` node has fields `_id` (type `ID!`), `title` (type `String!`), `description` (type `String!`), `price` (type `Float!`), and `date` (type `String!`). The `Booking` node has fields `_id` (type `ID!`), `event` (type `Event!`), `user` (type `User!`), `createdAt` (type `String!`), and `updatedAt` (type `String!`). The `AuthData` node has fields `userId` (type `ID!`), `token` (type `String!`), and `tokenExpiration` (type `Int!`). The `User` node has fields `_id` (type `ID!`), `username` (type `String!`), `password` (type `String!`), and `createdEvents` (type `Event!`). The `Flag` node has a field `text` (type `String!`). At the bottom, there are filters for 'Sort by Alphabet', 'Skip Relay', 'Skip deprecated', and 'Show leaf fields'. A 'RESET' button is also present.

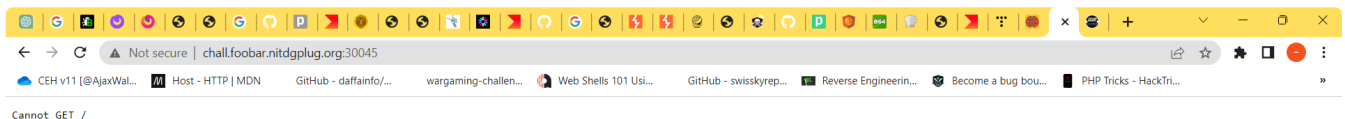
We got this beautiful graph upon pasting our output of introspection query, from this graph we can find the Flag column .

To print the Flag we need to write a simple GraphQL query like this

```
query { secret{ text } }
```



when I opened this challenge first time , I thought the challenge was down



So I just left this one and started to try other challenges but after a while when I checked this challenge once again some teams have solved this but still the site looked same.

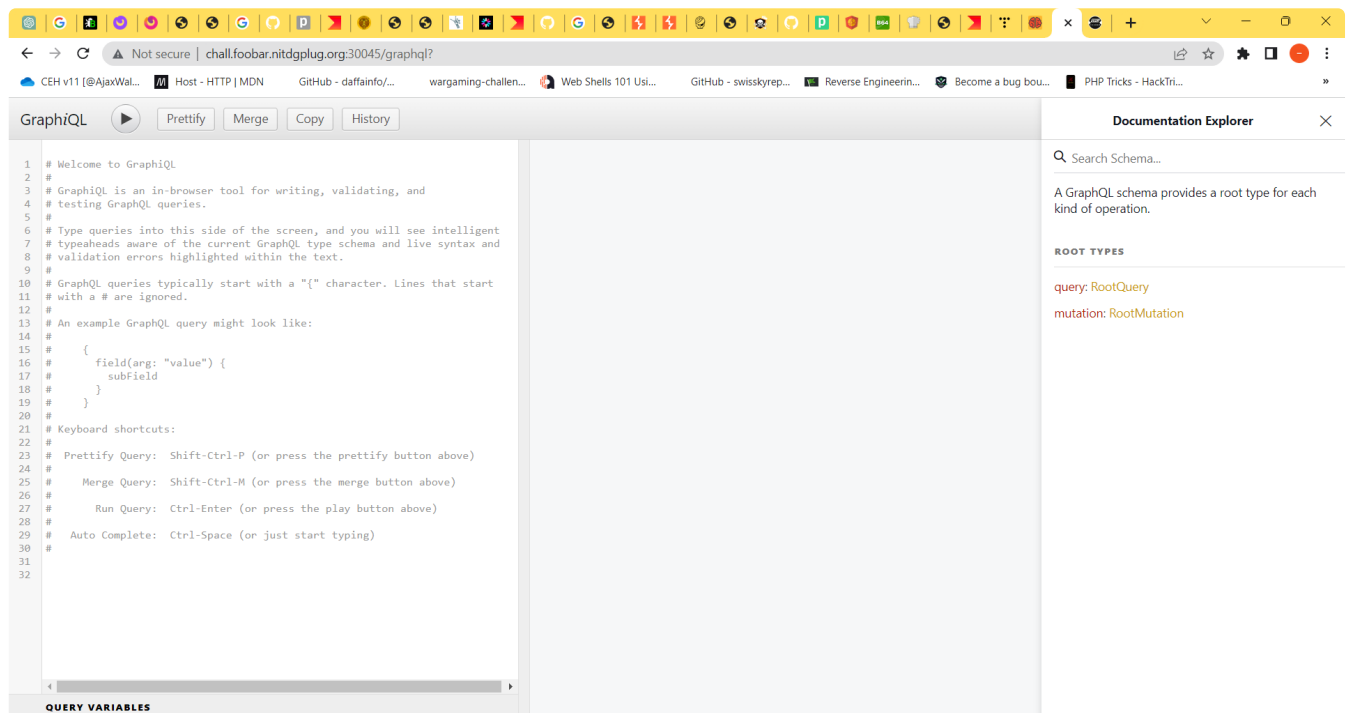
A new hint was also added to the challenge "REST API IS BORING , SO I USED A MODERN ONE".

From the hint I figured that this challenge is something related to API's. Basically An API is a set of defined rules that explain how computers or applications communicate with one another. API act as an intermediate between web server and client and process data.

Now I have to find the correct API to access this site, so started to search about REST API's alternatives, Modern API's ...

Luckily I founded a comparison between REST API vs GraphQL , in one of my google search results.

So quickly opened the following URL: chall.foobar.nitdgpug.org:30045/graphql



Now I confirmed that the site using GraphQL API ,okay but where is the flag?

I was clueless here but luckily I found a writeup related to this GraphQL <https://itsfading.github.io/posts/Hackerone-GraphQL-CTF-Writeup/>

First step to do on this GraphQL API is to understand how it is structured , we need to provide introspection query to get the structure and the available resources.

Introspection is the ability to query which resources are available in the current API schema.


```
query IntrospectionQuery {
  __schema {
    queryType { name }
    mutationType { name }
    types {
      ...FullType
    }
    directives {
      name
      description
      locations
      args {
        ...InputValue
      }
    }
  }
}

fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
    deprecationReason
  }
  possibleTypes {
    ...TypeRef
  }
}

fragment InputValue on __InputValue {
  name
```

```

    description
    typeRef {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
            kind
            name
            ofType {
              kind
              name
              ofType {
                kind
                name
              }
            }
          }
        }
      }
    }
  }
}

```

After passing this query , we get the rootquery ,coloumns and their datatype in the form of query itself like this,

```

47  ...TypeRef
48  }
49  }
50  fragment InputValue on __InputValue {
51    name
52    description
53    type { ...TypeRef }
54    defaultValue
55  }
56  fragment TypeRef on __Type {
57    kind
58    name
59    ofType {
60      kind
61      name
62      ofType {
63        kind
64        name
65        ofType {
66          kind
67          name
68          ofType {
69            kind
70            name
71            ofType {
72              kind
73              name
74              ofType {
75                kind
76                name
77                ofType {
78                  kind
79                  name
80                }
81              }
82            }
83          }
84        }
85      }
86    }
87  }
88  }
89  }
90  }
91  }
92  }
93  }
94  }
95  }
96  }
97  }
98  }
99  }
100 }

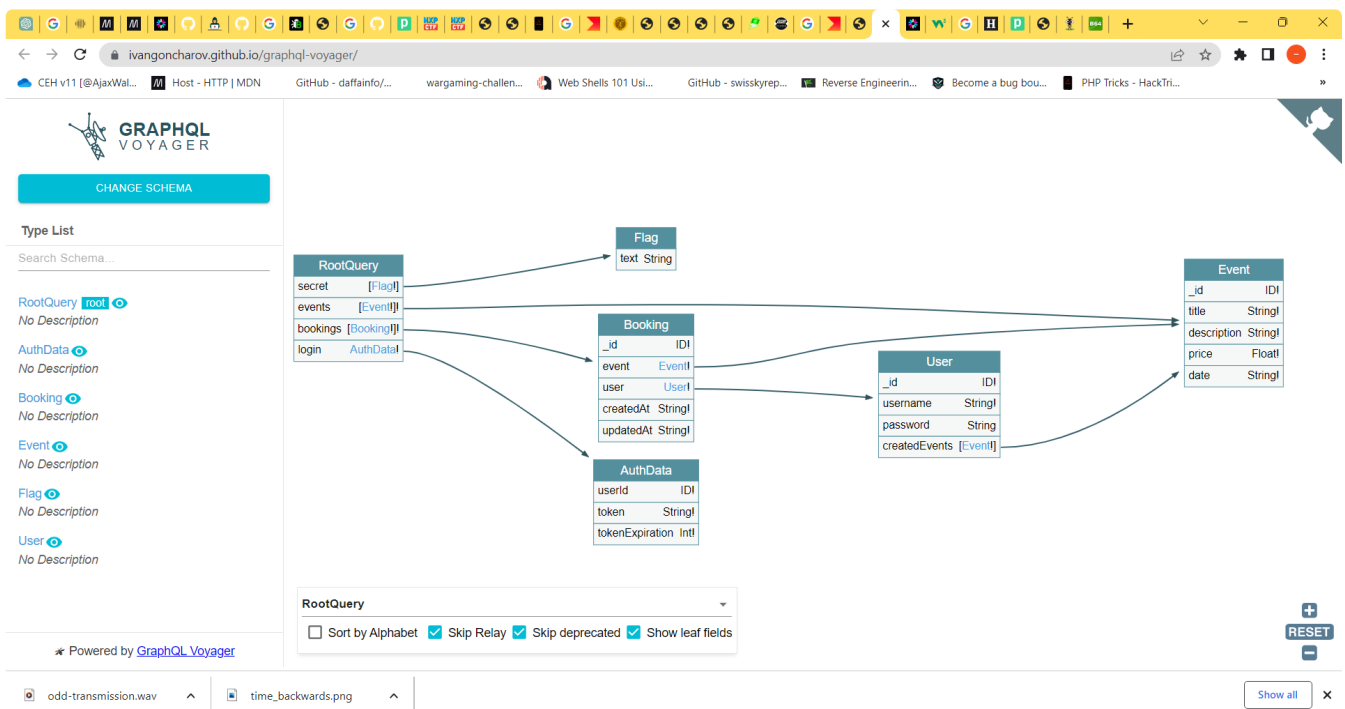
```

```

{
  "data": {
    "__schema": {
      "queryType": {
        "name": "RootQuery"
      },
      "mutationType": {
        "name": "RootMutation"
      },
      "types": [
        {
          "kind": "OBJECT",
          "name": "Booking",
          "description": null,
          "fields": [
            {
              "name": "_id",
              "description": null,
              "args": [],
              "type": {
                "kind": "NON_NULL",
                "name": null,
                "ofType": {
                  "kind": "SCALAR",
                  "name": "ID",
                  "ofType": null
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ]
        },
        {
          "name": "event",
          "description": null,
          "args": [],
          "type": {
            "kind": "NON_NULL",
            "name": null,
            "ofType": {
              "kind": "OBJECT",
              "name": "Event",
              "ofType": null
            }
          }
        }
      ]
    }
  }
}

```

Difficult to read right?, don't worry we got this amazing site <https://apis.guru/graphql-voyager/> which transforms this output query into an interactive graph .



We got this beautiful graph upon pasting our output of introspection query, from this graph we can find the Flag column .

To print the Flag we need to write a simple GraphQL query like this

```
query { secret{ text } }
```

The screenshot shows a web browser with a GraphQL IDE interface. The URL is `chall.foobar.nitdglug.org:30045/graphql?query=query%20%7B%0A%09secret%7B%0A%09text%0A%20%7D%0A%7D%0A`. The IDE has a left sidebar with a query editor and a right sidebar with an 'AuthData' panel.

Query Editor:

```
1 query {
2   secret{
3     text
4   }
5 }
6
```

AuthData Panel:

- Search AuthData...
- No Description
- FIELDS
- userId: ID!
- token: String!
- tokenExpiration: Int!

JSON Response:

```
{
  "errors": [
    {
      "message": "String cannot represent value: { text: \\\"GLUG{g1v3_4_m4n_4_F1sh}\\\" }",
      "locations": [
        {
          "line": 3,
          "column": 2
        }
      ],
      "path": [
        "secret",
        0,
        "text"
      ]
    },
    {
      "message": "String cannot represent value: { text: \\\"GLUG{a_C0ld_d4V_1N_H311}\\\" }",
      "locations": [
        {
          "line": 3,
          "column": 2
        }
      ],
      "path": [
        "secret",
        1,
        "text"
      ]
    },
    {
      "message": "String cannot represent value: { text: \\\"GLUG{Th3_13th_l3t7er_ofCoUR$E}\\\" }",
      "locations": [
        {
          "line": 3,
          "column": 2
        }
      ],
      "path": [
        "secret",
        2,
        "text"
      ]
    }
  ]
}
```

QUERY VARIABLES:

- odd-transmission.wav
- time_backwards.png

Show all

I got like 74 flags which all of them are in correct format GLUG{}. So I wrote a python script sorted the flags and tried to submit the flags via intruder using burpsuite and Got the correct flag GLUG{1nsp3c7_1n_gr4phq6}. After the CTF, I learnt the intended solution, all flags are in text format but the actual flag is in flag format.