universidad de león

Escuela de Ingenierías I.I.

Industrial, Informática y Aeroespacial

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

Detección de patrones para la identificación de ataques contra el heap de Windows

Autor: Carlos Domínguez Martínez

Tutor: Ricardo Julio Rodríguez Fernández

Co-Tutor: Camino Fernández Llamas

(Febrero, 2026)

UNIVERSIDAD DE LEÓN

Escuela de Ingenierías I.I.

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

ALUMNO: Carlos Domínguez Martínez

TUTOR: Ricardo Julio Rodríguez Fernández

CO-TUTOR: Camino Fernández Llamas

TÍTULO: Detección de patrones para la identificación de ataques contra el *heap* de Windows

CONVOCATORIA: Febrero, 2026

RESUMEN:

Los ataques basados en la corrupción de la memoria suponen uno de los principales problemas en la seguridad de las aplicaciones actuales. Sin embargo, muchas de las soluciones propuestas a día de hoy no se llegan a implementar por su gran sobrecarga en redimiento u otros problemas similares. Esto provoca que el análisis forense cobre un papel crucial en el estudio de estos ataques, especialmente en sistemas como Windows de los cuales se desconocen muchos detalles de implementación. De esta forma, el presente trabajo se centra en la búsqueda de patrones que permitan la detección de ataques clásicos contra el *heap* (*heap-overflow*, *use-after-free*, *double-free* y *heap-spraying*) de Windows. Para ello se realiza una serie de pruebas de concepto que incluyan estos ataques, así como el correspndiente análisis forense del *NT heap* de Windows mediante el *framework* Volatility 3 en combinación con el *plugin* HeapList. Finalmente, el trabajo concluye en la actualización del *plugin* mencionado gracias a una serie de heurísticas que detecten los patrones encontrados, logrando identificar con éxito los ataques *heap-overflow* y *heap-spraying*.

Palabras clave: *heap*, Windows, *overflow*, *spraying*, *use-after-free*, *double-free*, HeapList, patrones.

Firma del alumno:	**VºBº Tutor:**	**VºBº Co-Tutora:**

Abstract

Memory corruption–based attacks represent one of the main security issues in modern applications. However, many of the solutions proposed to date are not ultimately implemented due to their significant performance overhead or similar drawbacks. This highlights the crucial role that forensic analysis plays in the study of these attacks, especially in systems such as Windows, for which many implementation details remain unknown. Accordingly, this work focuses on identifying patterns that enable the detection of classic attacks against the Windows heap (heap-overflow, use-after-free, double-free, and heap-spraying). To this end, a series of proof-of-concept experiments including these attacks is conducted, together with the corresponding forensic analysis of the Windows NT heap using the Volatility 3 framework in combination with the HeapList plugin. Finally, this work concludes with an update to the aforementioned plugin through a set of heuristics that detect the identified patterns, successfully achieving this goal for heap-overflow and heap-spraying attacks.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Capítulo 1

Introducción

En el *software* moderno, la necesidad de manejar grandes cantidades de datos de tamaño variable hace que la memoria dinámica desempeñe un papel crucial, ya que permite una gestión flexible y eficiente de los recursos en tiempo de ejecución [25]. De esta forma, gran parte del tiempo de ejecución en las aplicaciones modernas se dedica a la gestión de memoria dinámica (reserva, liberación y acceso), siendo esta estructura determinante en el funcionamiento global de dichas aplicaciones [7].

Sin embargo, esta estructura de memoria no solo resulta atractiva para aquellos que desarrollan las aplicaciones, sino que resulta en un objetivo de interés para los ciberdelincuentes. Esto es debido a la volatilidad de la memoria RAM, la cual ofrece grandes ventajas para los atacantes que pretenden esquivar los sistemas de detección tradicionales [3].

Mientras los sistemas de seguridad avanzan en tareas de monitoreo continuo y análisis estático de artefactos en disco, los atacantes buscan en la memoria RAM una estructura que contribuya a que sus acciones pasen desapercibidas ante los controles tradicionales basados en ficheros [3]. De esta forma, la memoria dinámica se fija como un punto de interés donde inyectar código o realizar otras tareas dañinas con la seguridad de que el rastro de estas acciones desaparecerá al apagar o reiniciar el sistema [17].

Hoy en día las vulnerabilidades relativas a la corrupción de memoria se encuentran entre las más explotadas a nivel global, convirtiéndose en uno de los más peligrosos agujeros de seguridad en el *software* moderno [35]. Explotar este tipo de vulnerabilidades puede permitir que los atacantes manipulen el flujo de ejecución de un programa, llevando a la ejecución de código arbitrario, la escalada de privilegios o el filtrado de información, entre otros posibles ataques [16].

En este contexto, cobran especial importancia las aplicaciones de usuario, las cuales aprovechan estas asignaciones dinámicas para completar sus tareas. A diferencia del núcleo del sistema operativo, donde el acceso está más restringido, el *heap* de usuario es donde se procesa toda la información de entrada (interacción directa con datos externos no confiables), lo que lo convierte en el punto de entrada lógico para cualquier ciberdelincuente [34].

En especial, este problema se agrava en las aplicaciones que se ejecutan en máquinas Windows al tratarse del sistema operativo que más ataques recibe en su espacio de usuario como se muestra en la Figura 1.1 [2]. Además, este no solo se trata del sistema operativo más atacado, sino que se constituye como el sistema con mayor cuota de mercado (véase la Figura 1.2); lo cual lo convierte en un objetivo aún más atractivo para los atacantes [33].

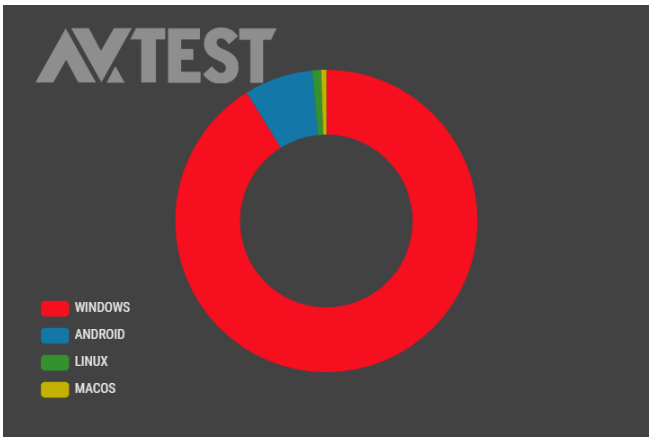

Figura 1.1: *Malware* recibido en función de los sistemas operativos más famosos.

Fuente: [2]

Debido a esta problemática, el análisis forense de los volcados de memoria RAM cobra especial importancia en la actualidad; proporcionando información crucial sobre

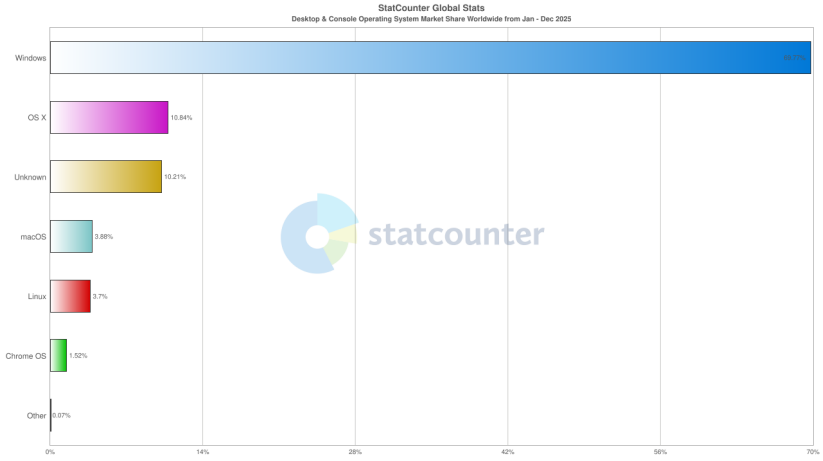

Figura 1.2: Cuota de mercado de los sistemas operativos de escritorio o consola. Fuente: [33]

posibles ataques recibidos, ya que la mayor parte de la información de las aplicaciones de usuario reside en estructuras como el *heap* [24]. Resulta así especialmente interesante el análisis del *heap* de Windows, al ser uno de los principales objetivos donde almacenar y ejecutar *malware* por parte de los ciberdelincuentes.

1.1. Motivación

Los ataques que corrompen la memoria dinámica son uno de los primordiales problemas en la seguridad de los sistemas informáticos actuales. Esto es producido por la falta de implementación de las principales soluciones propuestas para los ataques que aprovechan errores en la memoria, siendo la sobrecarga de rendimiento, los problemas de compatibilidad o la falta de exhaustividad en la protección la causa esencial que imposibilita la implementación de estas protecciones [34, 43].

De esta forma, evitar los ataques contra la memoria supone un reto aún no resuelto. En cambio, el análisis de la memoria RAM tras la ejecución de un ataque puede proporcionar información para determinar el origen y causas de este, así como ciertos detalles de gran valor para seguir avanzando en la investigación de contramedidas que ayuden a paliar estos problemas [24].

Sin embargo, hoy en día realizar un análisis forense sobre estructuras tan interesantes como el *heap* sigue resultando en una tarea compleja. Esta problemática se ve agravada en sistemas como Windows, los cuales presentan una estructura *heap* opaca y con mucha falta de documentación que dificulta las acciones forenses [49].

Es en este contexto donde entran en juego ciertas herramientas que tratan de resolver el problema de análisis forense de la memoria dinámica, facilitando un acceso que resulte confortable a las estructuras del *heap* a partir de un volcado de la memoria RAM. Una de estas herramientas, el *plugin* HeapList [41] de Volatility 3 [44], es la cual ha inspirado este trabajo con objeto de facilitar el análisis forense del *heap* de Windows.

1.2. Objetivo principal

El presente trabajo pone el foco en el análisis forense sobre el *heap* de Windows, contribuyendo así a la detección de ataques mediante volcados de la memoria RAM. Este análisis se ha pensado con un enfoque práctico que trata la ejecución de pruebas de concepto (POC por sus siglas en inglés) que simulan posibles *exploits* reales.

En concreto, se busca la ejecución de ataques clásicos sobre el *heap* (tales como *heap-overflow* [36], *use-after-free* [38], *double-free* [40] y *heap-spraying* [32]) con objeto de analizar la memoria dinámica en búsqueda de ciertos patrones que permitan la identificación de dichos ataques.

La idea principal del trabajo conlleva el uso de la herramienta de análisis, HeapList [41], con objeto de facilitar toda la información necesaria para localizar los patrones de identificación de los ataques estudiados en el *heap* de Windows. Además, de la actualización del *plugin* mencionado con los resultados obtenidos que permitan definir un patrón de identificación de los ataques mencionados.

1.3. Estructura del documento

El presente documento queda dividido en seis capítulos. Tras este primer capítulo introductorio, tiene lugar el Capítulo 2, el cual explica una serie de conceptos necesarios para entender el resto del trabajo. Por otro lado, en el Capítulo 3, se establece la metodología seguida en la elaboración de este trabajo, así como los detalles de implementación de los ataques estudiados y de la actualización del *plugin* HeapList. Tras esto se pasa al análisis de los resultados obtenidos tras estudiar los volcados de memoria resultantes de la ejecución de los ataques sobre el *heap*, lo cual se visualiza en el Capítulo 4. En el Capítulo 5 se analizan las tecnologías y herramientas que ayudan al estudio de la memoria dinámica en la detección de ataques, viendo donde encaja el trabajo elaborado. Finalmente, en el Capítulo 6, se presentan las conclusiones obtenidas tras este estudio junto a la enumeración de los posibles trabajos futuros.

Capítulo 2

Conceptos previos

Este capítulo trata la definición de aquellos conceptos que resultan fundamentales para la correcta comprensión del trabajo realizado. Para ello, se comienza con la definición de *heap*, centrando la atención sobre el *heap* de Windows y el funcionamiento de sus estructuras internas. Tras esto, se comentan las herramientas que permiten el análisis forense de la memoria RAM y del *heap* de Windows.

2.1. Memoria dinámica: *heap*

El *heap* constituye una región de memoria basada en reservas dinámicas. A diferencia de la pila (o *stack* en inglés), que sigue una estructura LIFO con reservas de tamaño fijo, el *heap* permite una mayor flexibilidad en las reservas y liberaciones de memoria en tiempo de ejecución. De esta forma, la estructura *heap* resulta fundamental en los programas que desconocen el tamaño de los datos que almacenan en tiempo de compilación [1, 19].

Esta mayor flexibilidad ofrecida por el *heap* también se traduce en una mayor complejidad en su construcción, así como una mayor responsabilidad en su manejo por parte de los desarrolladores. Comúnmente, delegar en los desarrolladores parte del proceso de reserva y liberación de memoria se suele traducir en una puerta de entrada para

vulnerabilidades explotables por los atacantes, como se ve con los ataques analizados a lo largo de este trabajo [43].

2.2. El *heap* de Windows

Este trabajo se centra en el *heap* de Windows, el cual consiste en un sistema complejo del que se desconocen todos sus detalles. Sin embargo, el funcionamiento esencial de este *heap* viene definido por el *Heap Manager* de Windows. Este gestor es el encargado de facilitar las reservas y liberaciones de memoria dinámica, las cuales dependen de ciertas APIs internas de Windows.

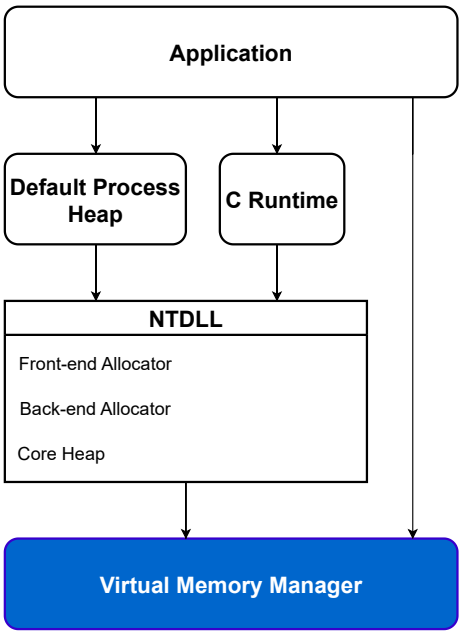

Figura 2.1: Esquemas de llamadas a APIs Windows para la gestión de memoria dinámica. Fuente: [23]

Mientras las APIs internas de Windows, como *Virtual Memory Manager*, permiten la reserva de grandes estructuras de memoria (normalmente a nivel de página), el *Heap Manager* proporciona una API más adecuada para la granularidad requerida por las aplicaciones. De esta forma, el *Heap Manager* funciona como una API que permite una eficiente gestión de la memoria dinámica usada por las aplicaciones [23].

Sin embargo, la mayoría de las aplicaciones no interactúan directamente con este gestor. En su lugar, el *Heap Manager* es invocado de forma indirecta a través de funciones del C/C++ *runtime*. Estas funciones son las empleadas directamente por las aplicaciones para realizar operaciones de reserva y liberación de memoria, tal y como se ilustra en la Figura 2.1 [23].

2.2.1. *Segment heap* y *NT heap*

En realidad, en Windows coexisten dos tipos de *heaps*: el *NT heap* y el *Segment heap*. El *Segment heap*, introducido a partir de Windows 10, está orientado principalmente a aplicaciones modernas y a determinados ejecutables del sistema, como aquellos pertenecientes a la Plataforma Universal de Windows (*Universal Windows Platform*, UWP). Por su parte, el *NT heap* representa el sistema de memoria dinámica más tradicional, el cual es usado por la mayoría de aplicaciones [48].

Al tratarse el *NT heap* como el gestor por defecto de las aplicaciones de escritorio, la mayoría de los ataques contra el *heap* de Windows documentados hasta la fecha se han centrado en esta implementación. En consecuencia, el presente trabajo se centra en este gestor.

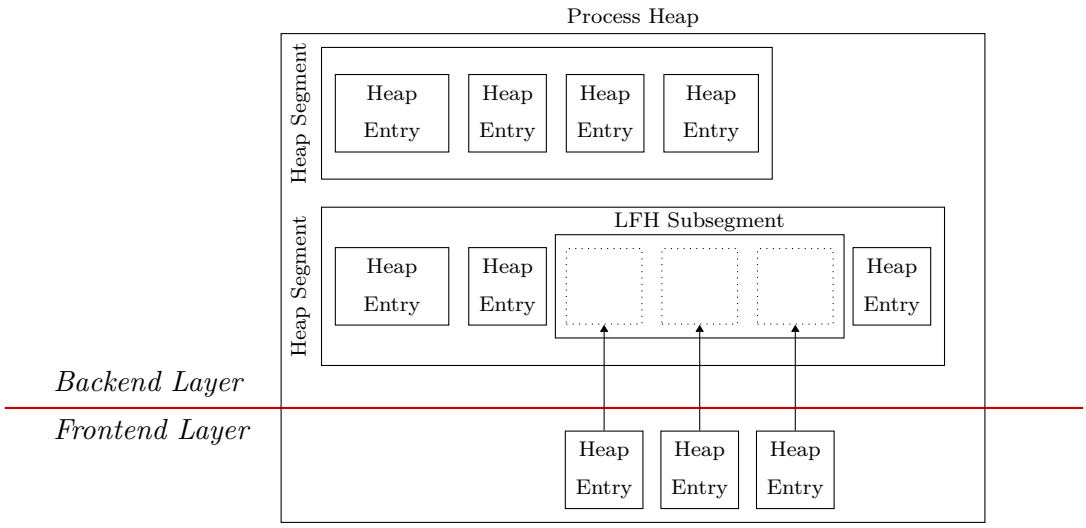

Figura 2.2: Arquitectura simplificada del *NT Heap* de Windows: representación gráfica de las capas *backend* y *frontend*. Fuente: [41]

Como se muestra en la Figura 2.2, la arquitectura del *NT heap* se divide principalmente en dos capas: la capa *backend* y la capa *frontend*. Estas capas cumplen una misión diferente entre sí dentro del *NT heap*, con objeto de proporcionar una mejor administración de los recursos en función de los requerimientos de memoria de las aplicaciones [23, 48].

2.2.2. *Backend*

Al intentar reservar memoria dinámica, el gestor del *heap* suele asignar dicha memoria en un *heap* que crea por defecto, el *DefaultHeap* (aunque permite la creación y uso de varios *heaps*). Dentro del *heap*, la principal capa que administra estas reservas es la capa *backend*. Esta capa se encarga de la administración de grandes bloques de memoria denominados *heap segments*, así como de porciones más pequeñas llamadas *heap entries* (equivalente a los *chunks* en otros *heaps*), como muestra la Figura 2.2 [21,23,48].

Los segmentos del *heap* normalmente están conformados por páginas virtuales que el gestor reserva con funciones de la API de *Virtual Memory Manager*, como `VirtualAlloc`. El gestor luego divide estos en porciones más pequeñas que ya pueden ser entregadas al usuario para cubrir las reservas de memoria dinámica solicitadas [21, 23, 48].

Estas porciones, conocidas como *heap entries*, son las que permiten una administración más eficiente y precisa del *heap*, como se mencionó anteriormente. De esta forma, la capa *backend* es la principal encargada de administrar la asignación y liberación de bloques de memoria. Además, debe encargarse de otras tareas como la solicitud de expansión del *heap* en caso de que el espacio reservado se torne insuficiente, o por contra la reducción de este en caso de no ser necesario tanto espacio [21, 23, 48].

Por tanto, el *backend* debe gestionar la reserva eficiente de la memoria dinámica, ocupándose también de organizar esta correctamente para evitar problemas de fragmentación, así como asegurarse de la correcta administración de las listas de *heap entries* vacías.

2.2.3. *Frontend*

Mientras el *backend* es la capa principalmente destinada a las tareas de reserva y liberación de memoria, el *frontend* se establece como una capa más bien destinada a labores de optimización. En concreto, tras un cierto número de reservas de igual tamaño, se activa esta capa, la cual crea un subsegmento dentro de un *heap segment* que se divide en *heap entries* de igual tamaño (véase la Figura 2.2) [21, 23, 48]. Estos subsegmentos dividen una porción de memoria en entradas (ocupadas y vacías) de igual tamaño, buscando evitar problemas de fragmentación en entornos con frecuentes reservas de pequeño e igual tamaño. Esta misión se efectúa con objeto de mejorar la eficiencia de la memoria, gracias a la única implementación válida en sistemas modernos del *frontend*, el *Low Fragmentation Heap* (LFH) [42].

El LFH supone una implementación más robusta del *frontend* que las versiones que le preceden, ya que no constituye un subsegmento determinista como hacían sus versiones predecesoras. Esto es posible gracias a medidas como la aleatorización de las *heap entries* adjudicadas en las reservas o la separación deliberada entre *heap entries* ocupados de un mismo subsegmento [42]. De esta manera, el *frontend* se fija como una capa que ayuda a una mejor gestión del espacio reservado, siendo complementaria al *backend*.

2.3. Herramientas de análisis forense

En el desarrollo del presente estudio se han usado ciertas herramientas para facilitar el análisis del *heap* de Windows, siendo Volatility 3 [45] la herramienta principalmente ejecutada para esta labor junto al *plugin* HeapList [41]. Por tanto, resulta de interés conocer los detalles de ambos.

2.3.1. Volatility 3

Volatility 3 se constituye como una de las principales y más extensamente utilizadas herramientas para el análisis forense de memoria volátil. Esta herramienta es un *framework* multiplataforma de código abierto, escrito en Python 3 [45].

Este *framework* se compone de una arquitectura modular basada en capas de memoria, tablas de símbolos y *plugins* extensibles, permitiendo a los usuarios crear sus propios *plugins* para resolver ciertas tareas aún no consideradas [45].

Los *plugins* de Volatility 3 implementan la lógica de análisis necesaria para recorrer estructuras internas del sistema operativo, como procesos o *heaps*, facilitando la extracción de artefactos relevantes para investigaciones forenses.

2.3.2. HeapList

HeapList es un *plugin* de Volatility 3 que se centra en el análisis de la memoria *NT heap* de Windows bajo el pretexto de una falta de herramientas que permitan un análisis satisfactorio de los *heaps* de Windows [41].

El funcionamiento del HeapList se basa en la extracción de todas las *heap entries* de un proceso a partir de un volcado de la memoria RAM. De esta forma, se busca una recreación precisa que permita una correcta navegación entre las capas *frontend* y *backend* que conforman la memoria dinámica de dicho proceso [41].

Capítulo 3

Metodología e implementación

A lo largo de este capítulo se comentan las estrategias seguidas para el cumplimiento del objetivo principal mencionado anteriormente. Asimismo, se comentan los detalles de desarrollo sobre los diferentes ataques propuestos y sobre la actualización resultante del *plugin* HeapList.

3.1. Metodología

En primer lugar, se menciona la metodología seguida para desarrollar una serie de POCs que representan ataques clásicos al *heap* que proporcionan los artefactos de análisis tras su ejecución. Asimismo se comenta la metodología seguida en la fase de análisis de la memoria volátil tras la ejecución de los ataques mencionados, así como del desarrollo de la extensión del *plugin* HeapList.

3.1.1. Adaptación de los ataques

Para el estudio de patrones que permitan la identificación de ataques en el *heap*, se ha comenzado por la selección de una serie de ataques clásicos que se centran en esta estructura:

- ***use-after-free:*** se trata de un ataque que busca aprovechar fallos de implementación en las aplicaciones que provocan el uso de un espacio de memoria dinámica ya liberada.
- ***double-free:*** un ataque basado en el aprovechamiento de la liberación de un espacio de memoria dinámica ya liberado, lo cual corrompe las listas de almacenamiento de porciones de memoria (*heap entries* o *chunks*) vacías.
- ***heap overflow:*** es un ataque que trata de acceder a un espacio de la memoria *heap* fuera de los límites del espacio reservado por el atacante.
- ***heap spraying:*** un ataque complementario que busca esparcir por la memoria dinámica un *payload* destinado a acciones maliciosas. Este *payload* está pensado para ser accedido mediante la modificación del flujo de control de la aplicación, modificación realizada por ataques como los anteriores.

Estos ataques son independientes de la plataforma sobre la cual se ejecutan, aunque sus detalles de implementación sí son dependientes de la plataforma concreta, en especial del tipo de gestor *heap* usado en esta. De esta forma, los detalles de implementación de estos ataques pueden ser diferentes en Windows que en otros sistemas operativos como Linux, por ejemplo.

Como se ha comentado anteriormente, este trabajo se centra en Windows, lo cual establece el primer paso como la implementación de pruebas de concepto sobre los ataques clásicos mencionados para su ejecución en Windows. En concreto, estos POCs están pensados para que su ejecución se detenga antes y después de ejecutarse la fase de ataque, permitiendo los volcados de memoria RAM para su posterior análisis en estos puntos clave.

3.1.2. Volcado y análisis de la memoria

Una vez desarrollados los POCs que definen las principales fuentes de análisis, hay que ejecutarlos y capturar la memoria RAM en los puntos clave establecidos. En concreto, se busca un volcado previo y posterior al propio ataque con objeto de comparar el estado del *heap* en busca de patrones que identifiquen la ejecución de dichos ataques.

Para esta fase se comienza con una compilación con la herramienta GCC [12] de todos los POCs, que posteriormente serán ejecutados. La captura de la memoria RAM se realiza con la herramienta WinPmem [29]. De esta forma, se genera un volcado de la totalidad de la memoria RAM en formato RAW (copia directa) para cada punto de captura establecido.

Los archivos RAW generados en el paso previo conforman los principales artefactos para el análisis forense de la memoria dinámica de cada POC, antes y después de la ejecución de cada ataque. Esto se debe a que dichos ficheros serán la entrada preestablecida por Volatility 3 para obtener un análisis completo del *heap* de cada uno de estos procesos, gracias al *plugin* HeapList.

Por tanto, el paso final de esta fase se configura como la ejecución de Volatility 3 junto al *plugin* HeapList por medio de un comando similar al mostrado en la Figura 3.1. Este comando permite obtener un resumen del análisis que sintetiza detalles como direcciones de memoria, flags o tamaños de cada *heap entry* capturado; junto a una serie de ficheros que guardan el contenido de cada *heap entry*.

3.1.3. Búsqueda de patrones y actualización del HeapList

Tras obtener todos los artefactos proporcionados por Volatility 3 y el HeapList, hay que buscar patrones que permitan identificar la ejecución de los ataques probados. El resultado del *plugin* HeapList proporciona una gran cantidad de *heap entries*, lo cual hace imposible una exhaustiva comparación entre todas las *heap entries* antes y después del ataque. Debido a esta problemática, se decidió previamente añadir a la salida de cada POC (salida en formato textual por consola) las direcciones de memoria *heap* que reserva explícitamente. Esto es necesario por la gran cantidad de entradas que hace el sistema para lanzar el proceso que representa a cada uno de los POCs, las cuales se asocian también al propio POC y dificultan la visión de aquellas entradas de interés para el análisis.

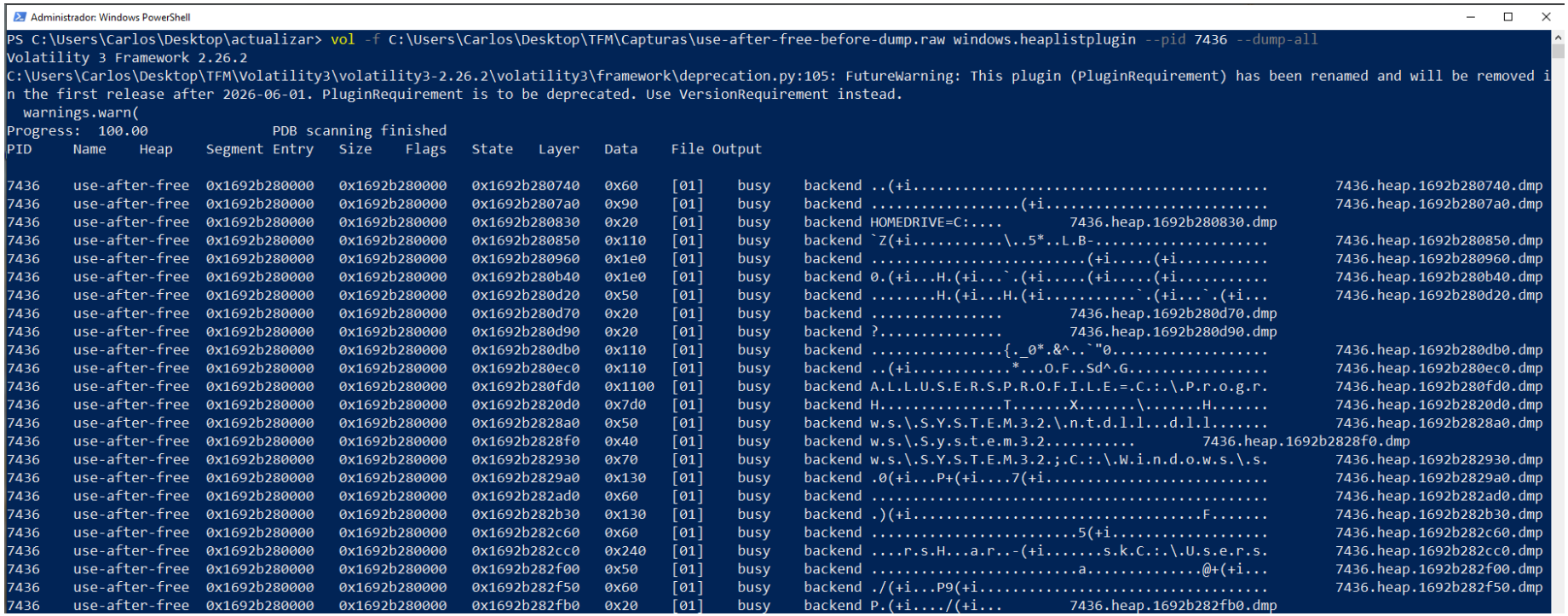

Figura 3.1: Ejemplo de ejecución del HeapList de Volatility 3. Fuente: propia

De esta forma, se observan en detalle los resultados proporcionados para cada ataque por HeapList en busca de datos susceptibles a identificar la ejecución del ataque. Además, para evitar falsos positivos, se cambian algunos valores de relleno que caracterizan ciertos ataques con objeto de comprobar si los patrones identificados son consecuencia de un valor concreto de relleno o resultan en un patrón general.

Finalmente, tras identificar estos patrones, se adapta el *plugin* con los detalles necesarios para señalar en su salida las *heap entries* susceptibles de presentar detalles propios de un ataque, según lo identificado mediante el análisis. Además, se marca el tipo de ataque que puedan contener esas *heap entries*.

3.2. Repositorio *software*

Para la mejor comprensión de las secciones que se detallan a continuación, se proporciona un enlace a un repositorio de GitHub[1] que contiene todos los códigos (ataques y *plugin* actualizado) explicados en este capítulo, así como otros ficheros que ayudaron a la construcción de los ataques elaborados.

3.3. Ataques analizados

En este apartado se detallan los ataques implementados en las POCs, los cuales se desarrollan en lenguaje C/C++ y se ejecutan principalmente en una máquina virtual Windows 10 Education. La elección de este sistema como entorno de pruebas principal se debe al refuerzo en la seguridad del *heap* que el sistema operativo sufre a partir de esta versión. Esto dificulta los ataques estudiados respecto a las versiones de Windows más antiguas debido a la pérdida de determinismo, que hace del *heap* una estructura más impredecible.

[1] `https://github.com/cdomim02/TFM-POC-Dynamic-MEM-Windows`

3.3.1. *Use-after-free* (UAF)

El primero de los ataques analizados es el *use-after-free*, el cual se basa en el uso de porciones de memoria (*heap entries*) que han dejado de estar reservadas con objeto de filtrar información o cambiar el flujo de ejecución de la aplicación [38, 47].

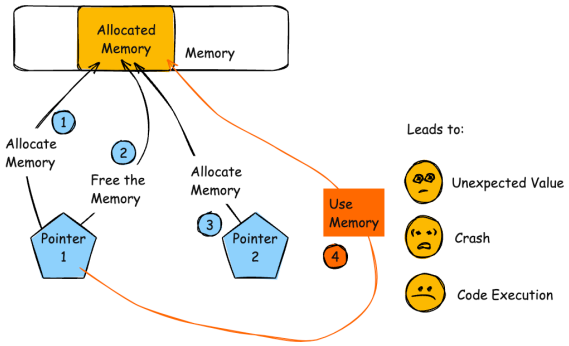

Figura 3.2: Esquema que representa el flujo de un UAF. Fuente: [38]

Debido al refuerzo de la seguridad a partir de Windows 10, este ataque pasa a ser inviable para labores de escritura en memoria (detección de escritura en espacios marcados como libres), que son requeridas en objetivos como la modificación del flujo de ejecución del programa. De esta forma, se realiza un POC que simula el filtrado de información de zonas de memoria con datos sensibles.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = malloc(sizeof(int));
    *p = 123;

    free(p);              // Primera liberacion
    //...

    // Nueva asignacion de igual tamanyo
    // Muy posiblemente ocupe el espacio liberado
    int *t = malloc(sizeof(int));
    *t = 333

    //...
    printf("%d\n", *p);  // UAF: lectura despues del free
                         // Posible valor devuelto: 333
```

```
19      return 0;
20  }
```

Código 3.1: Simplicación del POC relativo al UAF. Fuente: [38]

En mayor detalle, como se simplifica en el Código 3.1 y en la Figura 3.2, el POC desarrollado busca realizar una reserva de memoria dinámica (`malloc` en C/C++) que se rellena con un cierto valor (línea 6). Tras esto, se libera dicha porción de memoria (*free* en C/C++) en algún punto del programa. Una vez liberado el espacio de memoria se hace una nueva reserva de igual tamaño (línea 13), lo cual provoca que muy probablemente el gestor del *heap* proporcione el espacio recientemente liberado. Tras esto, se rellena el último espacio de memoria reservado con un nuevo valor (línea 14), el cual se muestra al intentar imprimir por pantalla el valor almacenado en el primer espacio reservado.

El POC desarrollado recibe una cadena de caracteres que posteriormente divide en dos partes de igual tamaño. Sin embargo, al recibir una cadena con un único carácter, y por tanto, no poder ser dividido, el código entra en una estructura de selección que conlleva un acceso mediante el puntero que referenciaba al primer espacio de memoria liberado.

Esto muestra la información almacenada en un espacio reservado posteriormente que ha reaprovechado el primero (apuntan a la misma dirección de memoria debido a la reutilización de la *heap entry*). Esto se puede ver en más detalle en la Figura 3.3, que muestra la salida del POC en el caso habitual (una cadena de más de un carácter que puede dividir en dos) y en el caso vulnerable (recibe un único carácter).

De esta forma, el POC busca ejemplarizar una mala codificación que puede ser forzada por los atacantes. Típicamente este problema se usa para sobreescribir la dirección de memoria de una función legítima para cambiar el flujo de control del programa o para aprovechar condiciones como la mostrada en el POC que permiten el filtrado de información de interés.

```
Seleccionar Windows PowerShell
PS C:\Users\Carlos\Desktop\TFM\Ataques> .\use-after-free.exe "Hola" # Caso habitual
Text: 0x000002970DF4D1C0
Part1: 0x000002970DF49A30
Part2: 0x000002970DF4D3D0
Presione una tecla para continuar . . .
Secret: 0x000002970DF4D1C0
Text division into two parts of equal size:
Part 1: Ho
Part 2: la
PS C:\Users\Carlos\Desktop\TFM\Ataques>
PS C:\Users\Carlos\Desktop\TFM\Ataques> .\use-after-free.exe "H" # Caso con UAF
Text: 0x000002C73F05D1C0
Part1: 0x000002C73F059A30
Part2: 0x000002C73F05D3D0
Presione una tecla para continuar . . .
Secret: 0x000002C73F05D1C0
Text:
Hello, that message is private, you shouldn't be able to see it!

Please, introduce some text with 2 or more letters as an argument
Presione una tecla para continuar . . .
PS C:\Users\Carlos\Desktop\TFM\Ataques>
```

Figura 3.3: Salida legítima e ilegítima del POC referente al ataque UAF. Fuente: propia

3.3.2. *Double-free*

El ataque conocido como *double-free* se basa principalmente en una mala codificación que conlleva a la liberación de un espacio de memoria dinámica previamente liberado. Esto ocasiona la corrupción de las estructuras internas del gestor del *heap* encargadas de almacenar las *heap entries* libres [40].

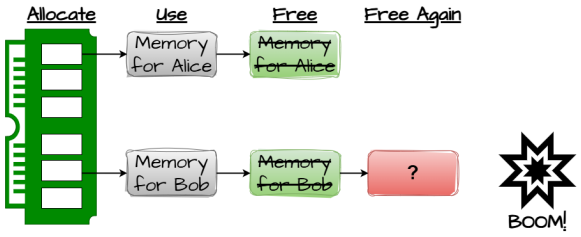

Figura 3.4: Esquema que representa el flujo de un *double-free*. Fuente: [40]

Dicha corrupción puede conllevar a la posterior adjudicación de este espacio de memoria recientemente liberado a varias peticiones de reserva a la vez, es decir, dos reservas posteriores pueden apuntar a este mismo espacio de memoria. A priori esto no debe ser posible, ya que las funciones de reserva de memoria (como `malloc` en C/C++ o `HeapAlloc` en la API de Windows) deben proporcionar un espacio libre [46].

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
```

```
5       char *a = malloc(50);
6       //...
7       free(a);  // Primera liberacion
8       //...
9       free(a);  // Segunda liberacion
10      //...
11
12      // Dos reservas que probablemente reasignen el espacio liberado en a
13      // apuntando asi ambas a la misma direccion de memoria
14      char* command = malloc(50);
15      char* username = malloc(50);
16
17      // Espacio donde el atacante puede introducir un comando malicioso
18      // aprovechando que apunta a la misma direccion que command
19      scanf("%512[^\n]", username);
20      // Ejecucion del comando que puede haber sido alterado
21      // desde el username al compartir heap entry
22      system(command);
23
24      return 0;
25  }
```

Código 3.2: Simplicación del POC relativo al *double-free*. Fuente: [13]

De esta forma, dichas funciones de reserva pueden llegar a proporcionar un espacio ya ocupado porque la *heap entry* liberada en dos ocasiones sigue perteneciendo a una estructura de gestión de espacios libres tras la primera reasignación (doble aparición en las estructuras debido a la doble liberación). Este comportamiento se puede ver simplificado, tanto en la Figura 3.4 como en el Código 3.2 [46].

En concreto, el POC desarrollado, busca aprovechar la doble liberación con objeto de hacer dos reservas posteriores de igual tamaño (líneas 14 y 15), las cuales provocan que dos hilos distintos compartan el mismo espacio de memoria, aunque no deba ser posible. Esto provoca que el atacante pueda llegar a visualizar (filtrado de información) o modificar (ejecución de código arbitrario) la información que almacena el hilo legítimo.

El POC está centrado en el filtrado de información por los mismos motivos que el UAF, ya que en ambos casos el propio sistema operativo detecta las escrituras en espacios liberados en tiempo de ejecución. Sin embargo, las actualizaciones del *NT heap*

a partir de Windows 10, no solo evitan la escritura en *heap entries* liberadas, sino que detectan la liberación de un espacio ya liberado.

De esta forma, tras varios intentos con distintas implementaciones similares, tanto en Windows 7, 8.1, como 10, se ha concluido que ninguna de estas versiones permite la ejecución de un ataque *double-free*. Por tanto, al comprobar que el sistema operativo ya se encarga de este problema, deja de tener sentido un análisis forense que permita detectarlo (ya es detectado en tiempo de ejecución). Es por este motivo, por el cual se decide descartar este ataque para el resto de fases del estudio.

3.3.3. *Heap overflow*

El ataque denominado *heap overflow* consiste en el acceso ilegítimo a un espacio de memoria dinámica (típicamente sobrepasando los límites de una *heap entry* para acceder a la siguiente) con objeto de visualizar (Figura 3.5) o modificar (Figura 3.6) la información almacenada en dicha porción de memoria [9, 36].

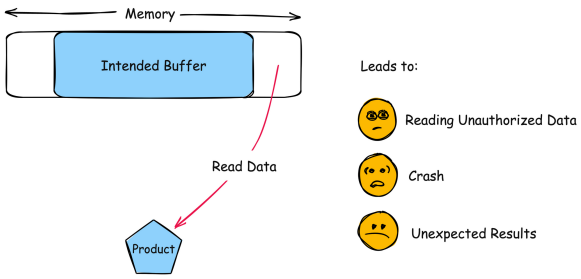

Figura 3.5: Esquema que representa el flujo de un *overflow* destinado al filtrado de información. Fuente: [37]

En concreto, este ataque se basa en exceder los límites de memoria proporcionados tras la reserva de un espacio de memoria dinámica, intentando acceder fuera de estos. Esta labor se suele realizar mediante la modificación de algún metadato, ya sea propio de una *heap entry* o de alguna otra estructura, lo cual permita el acceso a información de la *heap entry* o estructura adyacente [9, 36].

A partir de Windows 10, este tipo de ataques se complica significativamente, ya que en versiones anteriores bastaba con acceder fuera de los límites de la *heap entry*

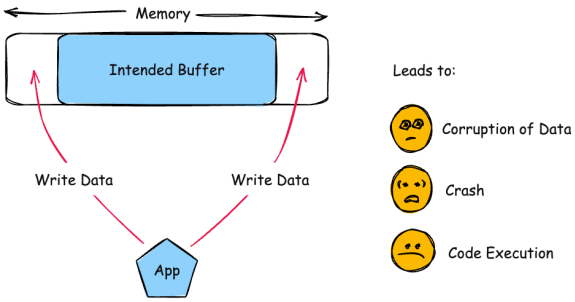

Figura 3.6: Esquema que representa el flujo de un *overflow* destinado a la modificación del contenido adyacente. Fuente: [39]

o estructura de datos almacenada en la memoria dinámica para llegar a acceder a los datos almacenados en la entrada adyacente [27].

Sin embargo, a partir de esta versión de Windows, el *NT heap* deja de ser determinista, haciendo que la ubicación de las *heap entry* objetivo de los ataques deje de ser predecible. Así, mientras que en las versiones previas a Windows 10 resultaba relativamente sencillo ubicar una porción de memoria adyacente al objetivo de los ataques; en las versiones modernas es necesario hacer una etapa de *grooming* o adecuación de la memoria [27].

De esta forma, el POC realizado se puede simplificar en una serie de pasos que comienzan por el denominado *grooming*. Esta fase consiste en hacer una serie de reservas que permitan configurar una estructura de *heap entries* predecible. En concreto, la capa del *NT heap* preferida para este proceso es el *backend*, debido a ser más predecible que el *frontend*, como ya se ha mencionado anteriormente.

El objetivo principal de esta fase se basa en conseguir una secuencia de reservas inferior a 18 (a partir de aquí se activa el LFH) alojadas en el *backend* con una separación de igual tamaño entre ellas (normalmente irán una seguida de otra en memoria). Sin embargo, la primera y la última reserva se descartan de este proceso por contener saltos mayores.

Una vez conseguida esta estructura, se necesitan al menos tres *heap entries* consecutivas con una estructura similar a la mostrada en la Figura 3.7. Donde, en una primera ocasión, todas están ocupadas y, tras una serie de reservas y reasignaciones se logren introducir dos estructuras controladas por el atacante, seguidas de una *heap*

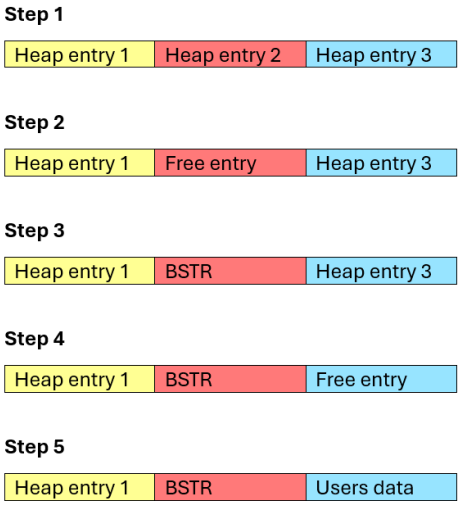

Figura 3.7: Esquema que representa el flujo de un *grooming* destinado a acomodar el *heap* en una estructura predecible. Fuente: propia

entry con datos de interés. En este punto es importante que en la *heap entry* localizada en el centro se ubique una estructura cuyo tamaño dependa de algún metadato propiamente almacenado en esta *heap entry*. Esto es lo que permite más adelante ejecutar el *heap overflow* y acceder a los datos de la siguiente *heap entry*. Finalmente, se libera la porción de memoria reservada posterior a la *heap entry* en la posición central. Esto se hace con el objetivo de dejar que ese espacio de memoria se rellene con objetos u otras estructuras que se desean filtrar.

Con esta distribución, se consigue una estructura predecible, donde se ubican datos susceptibles de ser filtrados tras la región (*heap entry*) de memoria reservada para el atacante. De esta forma, el siguiente paso consiste en desbordar la primera *heap entry* con objeto de sobreescribir los metadatos que definen el tamaño de la estructura almacenada en la *heap entry* central (esto ya se puede considerar un *heap overflow* destinado a la modificación de los datos de la entrada adyacente).

Con esto, se consigue que el tamaño de dicha estructura sea mayor que el tamaño de los datos que realmente almacena, llegando incluso a representar un tamaño que sobrepasa los límites de la *heap entry* que almacena dicha estructura. De esta forma, al usar la estructura (en este caso un BSTR, un tipo de cadena de caracteres), es posible acceder a la información almacenada en la siguiente *heap entry* (acceso de lectura principalmente, ya que para la escritura basta con exceder los límites de la *heap entry*

sin necesidad de estructuras como el BSTR), la cual almacena, en este POC, una serie de vectores con punteros a un método de un objeto.

Por tanto, el comportamiento legítimo del POC simula una aplicación donde se realizan llamadas a métodos de un objeto. En este contexto, el atacante busca obtener una lectura de memoria que revele la dirección de la función virtual asociada a dicho método; es decir, intenta filtrar la dirección del puntero correspondiente dentro de su tabla de funciones virtuales (*vtable*).

Esto permite conocer una dirección de interés que puede utilizarse posteriormente para alterar el flujo de control, sustituyendo el puntero al código legítimo de la función por la dirección donde el atacante haya inyectado su propio código. No obstante, en este POC solamente se muestra la dirección de la *vtable* para ejemplarizar las labores de filtrado de información de un *heap overflow*.

También se muestran otros datos relevantes como la dirección base donde se carga el ejecutable en memoria durante la ejecución. Esta dirección puede calcularse a partir del desplazamiento existente entre la dirección base y la *vtable* previamente mencionada, ya que permanece constante entre ejecuciones. Para ello, es necesario el uso de herramientas (como IDA o CFF Explorer) que permiten analizar los detalles del binario PE (ejecutable de Windows).

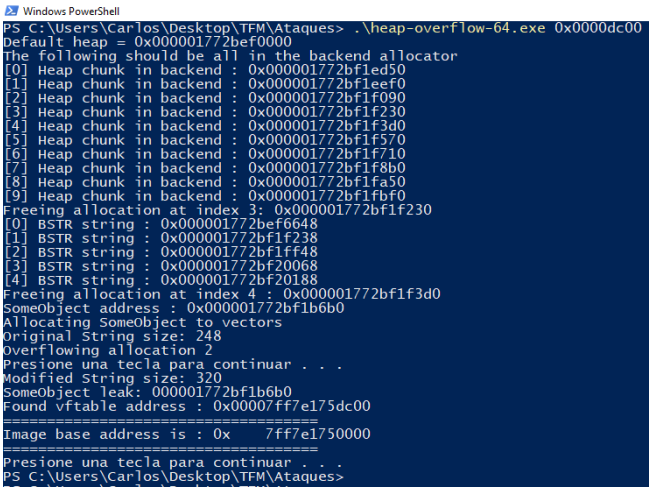

Figura 3.8: Salida tras realizar el ataque relativo al POC del *heap overflow*, el cual muestra los resultados de dicho ataque. Fuente: propia

De esta forma, se ha descrito un POC capaz de ejecutar un *heap overflow* centrado principalmente en el filtrado de información, aunque también sobrescribe datos de una *heap entry* adyacente como parte complementaria de este ataque. El resultado de este ataque se puede ver tras su ejecución en la salida estándar por consola mostrada en la Figura 3.8, donde se aprecian los detalles de la obtención de la dirección base y de la dirección de la función virtual en la *vtable*.

3.3.4. *Heap spraying*

Finalmente, el *heap spraying* representa un ataque complementario, ideal para inyectar código malicioso dentro de la memoria dinámica. Este ataque completa la labor de otros ataques vistos, como el *use-after-free* o el *heap overflow* cuando buscan un objetivo de ejecución de código arbitrario [15, 32]. El *heap spraying* basa su funcionamiento en el llenado masivo del *heap* con un *payload* (código máquina almacenado en memoria) malicioso. Además, este *payload* suele ir precedido de NOPs *sleds*, es decir, de operaciones sin efecto que buscan rellenar el resto del *heap entry* donde reside el *payload* [15, 32].

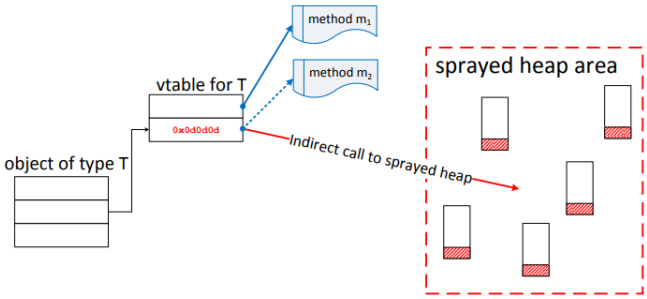

Figura 3.9: Esquema que representa el flujo de un *heap spraying* al alterar la dirección de una función legítima, para que esta caiga dentro de una *heap entry* cubierta con NOPs *sleds* y un *payload*. Fuente: [28]

Este proceso se realiza con la intención de maximizar la probabilidad de que una dirección corrupta (de una función o método principalmente) lleve a la ejecución de este *payload*; lo cual es provocado porque dicha dirección que sobrescribe a la legítima caiga sobre una NOP *sled* (véase la Figura 3.9) [15].

De esta forma, el POC desarrollado se basa en completar el ataque visto en el POC relativo al *heap overflow*, permitiendo no solo el filtrado de información, sino la ejecución de código arbitrario. Para ello se empieza con la misma etapa de *grooming* y alteración de la estructura BSTR vistas en el POC anterior, con intención de preparar el *heap overflow* final. Tras esto se diseña un *payload* que, en este caso, busque abrir la aplicación *calc.exe* (calculadora de Windows) para ejemplarizar las posibles acciones de un atacante. Una vez definido dicho *payload* se pasa a esparcirlo por toda la memoria, completando así la fase de *heap spraying*. En concreto, la etapa de *heap spraying* se desarrolla mediante la reserva de muchas *heap entries*, las cuales se rellenan con las típicas NOPs *sleds* seguidas del *payload* que abre la calculadora.

Tras esto, se completa el ataque *heap overflow* modificando la dirección de uno de los métodos perteneciente al objeto reservado que se referencia en los vectores que cubren la *heap entry* posterior al BSTR (mismo funcionamiento básico que el POC anterior, cambiando la lectura por la modificación). De esta forma, al ejecutar dicho método desde la referencia alterada, este abre la calculadora de Windows, como se muestra en la Figura 3.10.

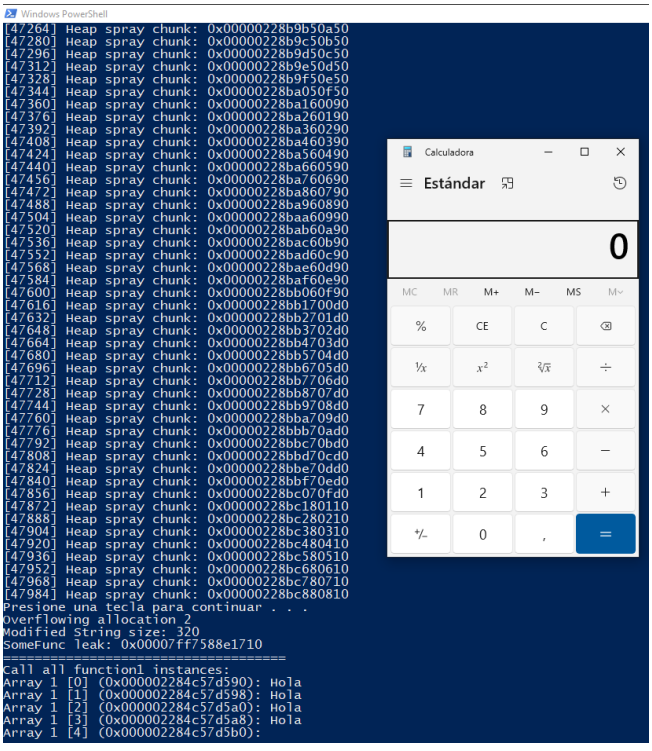

Figura 3.10: Salida tras realizar el ataque relativo al POC del *heap spraying*, el cual muestra la ejecución de *calc.exe* resultante de dicho ataque. Fuente: propia

Con todo esto se obtiene un POC capaz de simular un ataque *heap overflow* con objeto de ejecutar código arbitrario, el cual es posible gracias a un ataque *heap spraying* complementario que rellene la suficiente cantidad de memoria para que la dirección legítima alterada caiga sobre una NOP *sled*.

3.4. Desarrollo de extensión del HeapList

El desarrollo de esta actualización se ha dividido principalmente en dos fases, empezando por una fase destinada a la búsqueda de aquellos patrones que se incluyen en la segunda fase dentro del *plugin* para la detección de la ejecución de los ataques descritos anteriormente.

3.4.1. Búsqueda de patrones

Para la detección de patrones se analizan los artefactos resultantes de la ejecución y volcado de memoria de las POCs desde dentro del propio *plugin*, es decir, mediante la modificación temporal de este para que imprima los detalles que sean de interés para dicho análisis. En concreto, se busca el código referente a las pistas dadas por el resumen original y se imprimen o buscan condiciones que puedan identificar o dar más información sobre dichas pistas hasta encontrar el patrón que identifique el problema inequívocamente.

Tabla 3.1: Resumen de los patrones localizados tras la ejecución de los distintos ataques objeto de estudio. Fuente: propia

Ataque	Objetivo	Patrón detectado
Use-after-free	Filtrado de información	Ninguno
Heap overflow	Filtrado y modificación de información	Alteración de metadatos
Heap spraying	Ejecución de código arbitrario	Presencia de código en memoria, gran acumulación de NOP sleds

En el resumen de la Tabla 3.1 se pueden observar los patrones identificados tras finalizar esta fase. En esta se aprecia cómo no se ha podido localizar ningún patrón que sea capaz de identificar la ejecución de un UAF. Esto se debe a la falta de alteración del espacio de memoria devuelto al usuario o de sus metadatos. De esta forma, la única manera para detectar este problema se basa en el análisis de las estructuras internas del gestor del *heap* (como las listas de *heap entries* liberadas) al momento de volver a usar dicho espacio liberado. Sin embargo, el análisis de estas estructuras no es posible a través del *plugin* HeapList.

Por otro lado, se ve cómo escribir fuera de los límites de la *heap entry* controlada por el atacante provoca la sobreescritura de los metadatos de la siguiente *heap entry*, los cuales se encuentran en su cabecera. Esto ocasiona la corrupción de algunos datos de interés como las *flags* o el tamaño de la entrada, generando combinaciones incongruentes entre sí.

Finalmente se visualiza la presencia de código máquina (intrucciones en ensamblador) dentro de alguna *heap entry* como indicativo de la presencia de un *heap spraying*. En concreto, se destaca la presencia de una gran cantidad de NOP *sleds*.

3.4.2. Detección de los patrones

Una vez se ubican los patrones que permiten identificar la ejecución de un ataque, se incluye este conocimiento dentro del propio *plugin*. Para ello se construye una heurística de detección por cada uno de los patrones vistos en la sección anterior:

- **Detección del *heap overflow*:**

  Para detectar la sobreescritura de metadatos debida a un *heap overflow* se establecen una serie de condiciones que buscan detectar las incongruencias mencionadas sobre las *flags*, resumidas en la Tabla 3.2. Por otro lado, se establece una condición que comprueba que ninguna *heap entry* comienza en una dirección incoherente, la cual no coincida con el final de la entrada anterior.

- **Detección del *heap spraying*:**

  En primer lugar, para detectar una entrada susceptible de contener *heap spraying*

se utiliza la herramienta Capstone de Python [5] para recuperar posibles instrucciones en ensamblador almacenadas en las *heap entries*. Una vez desensamblado el contenido de cada *heap entry*, se aplica el cálculo de la entropía de las instrucciones recuperadas (en caso de que se hayan recuperado las suficientes). Al encontrarse las *heap entries* afectadas por el *heap spraying* repletas de una gran cantidad de NOP *sleds*, la variabilidad de las instrucciones será baja, debiéndose la poca variabilidad localizada al *payload* (supone un bajo porcentaje del contenido respecto a las instrucciones NOP). De esta forma, una baja entropía supone la posible presencia de algún *payload* en la *heap entry* analizada.

Sin embargo, esto no es suficiente para evitar falsos positivos como los relacionados con una entropía nula debidos a *heap entries* completamente cubiertas de algún dato identificable como instrucción (*heap entries* a cero, por ejemplo). Por tanto, se aplica una segunda comprobación más robusta, la cual se basa en analizar el porcentaje de instrucciones detectadas que pueden conformar una operación NOP (en la Tabla 3.3 se detallan las instrucciones NOP consideradas). Así, si más de la mayoría del código detectado en una *heap entry* se identifica como una instrucción NOP, se considera que dicha entrada es susceptible de contener un *payload*.

Finalmente, lo detectado mediante estas heurísticas se resume en la salida de Volatility 3 en combinación del *plugin* HeapList mediante dos nuevas columnas. La columna ***Detected Attack***, la cual indica el ataque encontrado en caso de que lo haya; y la columna ***Attack Details***, que especifica la condición o condiciones por las cuales se ha llegado a detectar dicho ataque. Todo esto solo se muestra si se activa la opción `--detect-attacks` al ejecutar el *plugin* (véase la Figura 3.11), ya que la detección de ataques se trata de una extensión del comportamiento original que se puede decidir activar o no.

Tabla 3.2: Condiciones para la detección de *heap overflows* en una *heap entry* a partir de incompatibilidades con sus *flags*. Fuente: propia

Flag afectada	**Incompatibilidad**	**Motivación**
Patrón de relleno	Activo en producción	La *flag* relativa al patrón de relleno solo es posible en entornos de depuración, como la activación de GFlags [22].
Heap entry liberada	Reserva mediante `VirtualAlloc`	Las reservadas mediante `VirtualAlloc` (páginas de memoria) no se almacenan en listas de *heap entries* liberadas, se gestionan aparte.
	Cabecera extra	Al liberar una *heap entry* se deja de tener en cuenta las cabeceras extra al no ser de interés para nuevas reservas que puedan no contar con estas cabeceras.
Última entrada del segmento	No concuerda con los metadatos del segmento	Los metadatos del segmento que indican la dirección de la última *heap entry* deben coincidir con la *flag* de dicha entrada que indica lo mismo.

Tabla 3.3: Intrucciones NOP consideradas para la detección del *heap spraying*. Fuente: propia

NOP	**Detalles**
Clásicos (`0x90`, `0x0F1F00`, etc.)	Instrucciones con código de operación NOP.
`xchg reg, reg`	NOP camuflada en intercambiar el contenido de un registro consigo mismo.
`mov reg, reg`	NOP camuflada en mover el contenido de un registro hacia el mismo registro.
`lea reg, [reg+0]`	NOP camuflada en cargar la misma dirección ya almacenada en un registro.
`lea reg, [reg]`	NOP camuflada en cargar la misma dirección ya almacenada en un registro.
`add/sub reg, 0`	NOP camuflada en una operación aritmética válida.

Administrador: Windows PowerShell

```
PS C:\Users\Carlos\Desktop\tests> vol -f C:\Users\Carlos\Desktop\TFM\Capturas\use-after-free-before-dump.raw windows.heaplistplugin --pid 7436 --dump-all --detect-attacks
Volatility 3 Framework 2.26.2
C:\Users\Carlos\Desktop\TFM\Volatility3\volatility3-2.26.2\volatility3\framework\deprecation.py:105: FutureWarning: This plugin (PluginRequirement) has been renamed and will be removed in the first release after 2026-06
-01. PluginRequirement is to be deprecated. Use VersionRequirement instead.
  warnings.warn(
Progress:  100.00               PDB scanning finished
PID     Name    Heap    Segment Entry   Size    Flags   State   Layer   Data    File Output     Detected Attack Attack Details

7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280740   0x60    [01]    busy    backend ..(+i.....................................       7436.heap.1692b280740.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2807a0   0x90    [01]    busy    backend .................(+i.....................       7436.heap.1692b2807a0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280830   0x20    [01]    busy    backend HOMEDRIVE=C:....        7436.heap.1692b280830.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280850   0x110   [01]    busy    backend `Z(+i.....\..5*..L.B-.....................       7436.heap.1692b280850.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280960   0x1e0   [01]    busy    backend .........................(+i.....(+i..........       7436.heap.1692b280960.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280b40   0x1e0   [01]    busy    backend 0.(+i...H.(+i...`.(+i.....(+i.....(+i..........       7436.heap.1692b280b40.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280d20   0x50    [01]    busy    backend ........H.(+i...H.(+i...........`.(+i...`.(+i...       7436.heap.1692b280d20.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280d70   0x20    [01]    busy    backend ...............         7436.heap.1692b280d70.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280d90   0x20    [01]    busy    backend ?...............        7436.heap.1692b280d90.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280db0   0x110   [01]    busy    backend ................{._0*.&^..``0.....................       7436.heap.1692b280db0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280ec0   0x110   [01]    busy    backend ..(+i...........*...O.F..Sd^.G.................       7436.heap.1692b280ec0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b280fd0   0x1100  [01]    busy    backend A.L.L.U.S.E.R.S.P.R.O.F.I.L.E.=.C.:.\.P.r.o.g.r.       7436.heap.1692b280fd0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2820d0   0x7d0   [01]    busy    backend H..............T.......X.......\.......H.......       7436.heap.1692b2820d0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2828a0   0x50    [01]    busy    backend w.s.\.S.Y.S.T.E.M.3.2.\.n.t.d.l.l...d.l.l.......       7436.heap.1692b2828a0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2828f0   0x40    [01]    busy    backend w.s.\.S.y.s.t.e.m.3.2...........        7436.heap.1692b2828f0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282930   0x70    [01]    busy    backend w.s.\.S.Y.S.T.E.M.3.2.;.C.:.\.W.i.n.d.o.w.s.\.s.       7436.heap.1692b282930.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2829a0   0x130   [01]    busy    backend .0(+i...P+(+i....7(+i...........................       7436.heap.1692b2829a0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282ad0   0x60    [01]    busy    backend ..............................................       7436.heap.1692b282ad0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282b30   0x130   [01]    busy    backend .)(+i.....................................F.......       7436.heap.1692b282b30.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282c60   0x60    [01]    busy    backend .........................5(+i...................       7436.heap.1692b282c60.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282cc0   0x240   [01]    busy    backend ....r.s.H...a.r...-(+i.......s.k.C.:.\.U.s.e.r.s.       7436.heap.1692b282cc0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282f00   0x50    [01]    busy    backend .........................a.............@+(+i...       7436.heap.1692b282f00.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282f50   0x60    [01]    busy    backend ./(+i...P9(+i.................................       7436.heap.1692b282f50.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282fb0   0x20    [01]    busy    backend P.(+i..../(+i...        7436.heap.1692b282fb0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b282fd0   0x50    [01]    busy    backend w.s.\.S.y.s.t.e.m.3.2.\.u.c.r.t.b.a.s.e...d.l.l.       7436.heap.1692b282fd0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b283020   0xa0    [00]    free    backend ........IZ......`.............................       7436.heap.1692b283020.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2830c0   0x130   [01]    busy    backend .6(+i....)(+i....\(+i....7(+i.....d.....te.....       7436.heap.1692b2830c0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2831f0   0x60    [01]    busy    backend .........................5(+i...................       7436.heap.1692b2831f0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b283250   0x50    [01]    busy    backend w.s.\.S.y.s.t.e.m.3.2.\.K.E.R.N.E.L.3.2...D.L.L.       7436.heap.1692b283250.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2832a0   0x1e0   [01]    busy    backend 0R(+i...HR(+i...`R(+i....8(+i....R(+i..........       7436.heap.1692b2832a0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b283480   0x100   [01]    busy    backend .........................4(+i....4(+i..........       7436.heap.1692b283480.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b283580   0x30    [01]    busy    backend .5(+i...p,(+i...        7436.heap.1692b283580.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2835b0   0x30    [01]    busy    backend .5(+i....2(+i...        7436.heap.1692b2835b0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b2835e0   0x20    [01]    busy    backend .[(+i....6(+i...        7436.heap.1692b2835e0.dmp       False   Undetected Attack
7436    use-after-free  0x1692b280000   0x1692b280000   0x1692b283600   0x30    [01]    busy    backend ee.exe..........        7436.heap.1692b283600.dmp       False   Undetected Attack
```

Figura 3.11: Salida obtenida tras la ejecución del HeapList tras su actualización. Fuente: propia

Capítulo 4

Análisis de resultados

Durante este capítulo se comentan los resultados obtenidos tras finalizar la actualización del *plugin* HeapList gracias a los patrones detectados, así como las pruebas que validan los resultados esperados. Además se comentan los detalles del entorno sobre el cual se han realizado todas las pruebas necesarias.

4.1. Entorno de pruebas

El entorno de trabajo consta de una máquina virtual Microsoft Windows 10 Education 10.0.19045.0 y compilación 19045.6456, lanzada desde VirtualBox 7.2.4 r170995 con 8GB de RAM y 6 núcleos de procesador. Todo esto ejecutado sobre una máquina real con Microsoft Windows 11, procesador Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz y 16 GB de RAM. Por otro lado, para el análisis de la memoria se utiliza WinPmem versión 2.0.1 y Volatility 3 en su versión 2.26.2, junto a Python 3.13.9; así como GCC en versión 15.2.0 y la consola PowerShell con versión 5.1.19041.6456.

4.2. Detección de ataques

En primer lugar, en la Tabla 4.1 se observan los posibles mensajes resultantes para las nuevas columnas añadidas en la ejecución del *plugin* HeapList tras su actualización.

De esta forma, se puede dar un caso más no considerado directamente en la tabla al detectar a la vez un *overflow* y un *spraying* en una misma *heap entry*, aunque esto pueda resultar muy poco común. Esto se muestra con los resultados de ambos separados por punto y coma en la columna ***Detected Attack***, así como sucede con los detalles (de un mismo ataque o de ambos) de la columna ***Attack Details***. Además, se visualiza la falta de detección de ataques al ejecutar un UAF, ya que este no se ha podido detectar con los datos obtenibles con HeapList, como ya se ha comentado.

Tabla 4.1: Resumen de los posibles mensajes tras ejecutar la versión actualizada del HeapList. Fuente: propia

Ataque ejecutado	***Detected Attack***	***Attack Details***
Ninguno	*False*	*Undetected Attack*
Use-after-free	*False*	*Undetected Attack*
Heap overflow	*Overflow*	*fill pattern flag in production* *free entry mark as internal* *free entry with extra header* *non-last entry marked as last* *previous entry size jump is incorrect*
Heap spraying	*Spraying*	*N NOPs of T total asm instructions*

4.3. Pruebas de funcionamiento

Para comprobar la correcta emisión de los mensajes mencionados en la sección anterior, se ejecuta el *plugin* actualizado con la detección de ataques activa contra cada uno de los POCs elaborados durante este trabajo. Nuevamente se busca la ejecución previa y posterior al ataque, lo cual permite comprobar la posible presencia de falsos positivos (antes del ataque) y la presencia de los propios ataques. Además, se añade una prueba con una aplicación real, Telegram, la cual sirve para analizar un volcado complejo en búsqueda de posibles falsos positivos.

Tabla 4.2: Conteo de ataques detectados tras probar la actualización del *plugin*. Fuente: propia

POC	Momento del volcado	Entradas con *overflow* detectadas	Entradas con *overflow* reales	Entradas con *spraying* detectadas	Entradas con *spraying* reales
use-after-free	Previo al ataque	0	0	0	0
	Posterior al ataque	0	0	0	0
heap overflow	Previo al ataque	0	0	0	0
	Posterior al ataque	1	1	0	0
heap spraying	Previo al ataque	0	0	0	0
	Posterior al ataque	1	1	48000	48000

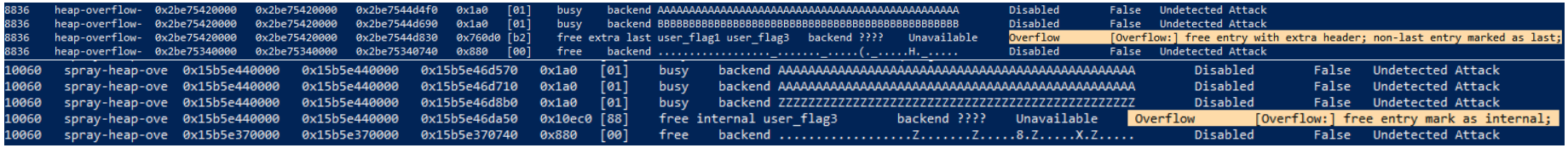

Figura 4.1: Ejemplos de detección de un *heap overflow* tras la actualización del *plugin*. Fuente: propia

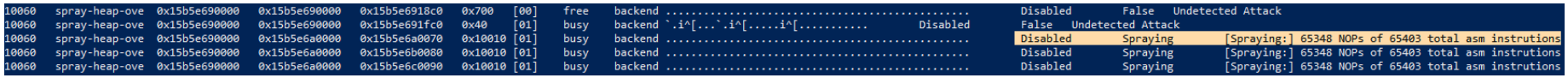

Figura 4.2: Ejemplo de detección de un *heap spraying* tras la actualización del *plugin*. Fuente: propia

En la Tabla 4.2 se puede apreciar la detección de los ataques relativos al *heap overflow* (Figura 4.1) y al *heap spraying* (Figura 4.2) tras la ejecución de sus respectivos POCs. Además, se aprecia cómo en el *heap spraying* no solo se detectan todas las entradas que se habían reservado con este propósito, sino que también se detecta el *overflow* al igual que sucede tras ejecutar el POC relativo a este ataque. Sin embargo, se aprecia la ausencia de estas detecciones en los POCs antes de que se ejecuten dichos ataques, lo cual confirma la ausencia de falsos positivos.

Por otro lado, en caso de utilizar la función `VirtualAlloc` en lugar de `HeapAlloc` a la hora de crear las *heap entries* que contengan el *spraying*, los resultados de la Tabla 4.2 cambian significativamente para el POC relativo a este ataque. De esta forma, el *heap spraying* dejaría de ser detectado (aunque el *overflow* seguiría siendo detectado), ya que al usar `VirtualAlloc` se pide el espacio de memoria directamente al sistema operativo, en lugar de pedírselo al gestor del *heap* y, por tanto, esas *heap entries* no son visibles desde el *plugin* HeapList.

Finalmente, tras varias ejecuciones del *plugin* contra volcados de memoria relativos a la ejecución de Telegram, se observa la total ausencia de falsos positivos. Con todo esto se confirma el correcto comportamiento de la funcionalidad de detección de posibles ataques contra el *heap* añadida al *plugin* HeapList.

4.4. Volcados de memoria y resúmenes de ejecución

Para finalizar este capítulo, se ofrece un enlace a Google Drive[1] con todos los volcados de memoria esenciales para la realización del presente trabajo, así como los resumenes ofrecidos tras la ejecución del *plugin* HeapList (antes y después de su actualización) que permiten comprender los resultados analizados.

[1] `https://drive.google.com/drive/folders/1ugiqo0p9t97qBQf_Sfp5SSwdvm-T4fHL?usp=sharing`

Capítulo 5

Estado del arte

A lo largo de este capítulo se presenta un resumen de la literatura relacionada con los ataques al *heap*, así como una serie de medidas de seguridad o herramientas de análisis que permiten detectar este tipo de problemas.

En primer lugar, en [18] se comenta cómo los ataques clásicos contra el *heap*, tales como *use-after-free*, *double-free* y *heap overflow*; siguen siendo ampliamente utilizados por los atacantes. Sin embargo, en la actualidad la mayoría de estos ataques se tornan más complejos, usando técnicas previas al propio ataque que requieren de la manipulación del gestor de memoria. Algunos ejemplos son la reutilización controlada de bloques liberados, la creación de solapamientos entre regiones de memoria o la corrupción dirigida de estructuras internas del *allocator*.

Por su lado, los autores de [20] refuerzan la idea centrada en que las mitigaciones actuales han conllevado al aumento de la complejidad de los ataques clásicos, requiriendo pasos como el *heap grooming*, *Heap Feng Shui* o *heap layout manipulation*. Este tipo de acciones intermedias se basan principalmente en la adecuación de la memoria que permita la explotación de los ataques clásicos. En concreto, los autores se enfocan en la necesidad de realizar reservas y liberaciones previas a la explotación real de las vulnerabilidades para conseguir una estructura de memoria predecible que permita tal explotación. En [10] continúa el refuerzo de esta teoría, viendo un nuevo enfoque sobre el ataque *double-free*, el cual permite su ejecución a pesar de las mitigaciones actuales (como *Safe-Linking* en las versiones recientes de `glibc`), gracias a una fase de ade-

cuación de memoria que permite la creación de bloques solapados. De esta forma, se demuestra que los atacantes suelen ir un paso por delante, aunque sus nuevas implementaciones siguen dejando huellas que permiten implementar nuevas defensas para cubrir estos avances en los ataques.

Otros artículos como [11] amplían esta tesitura sobre la necesidad por parte de los atacantes de reinventarse, siendo necesario, incluso, en ataques complementarios como el *heap spraying*. De esta forma, sus autores comentan cómo el *heap spraying* clásico ha dado paso a técnicas más sofisticadas integradas con motores JIT y entornos como `WebAssembly`, dando lugar a variantes como el JIT *spraying*. Estas estrategias suelen combinarse con fases de *grooming* y con vulnerabilidades tradicionales como UAF u *overflows*, conformando cadenas de explotación híbridas; las cuales permiten aprovechar comportamientos específicos del compilador JIT y del *runtime* para generar código o datos controlados en memoria.

Por otro lado, estudios como [31] reflejan las técnicas y herramientas existentes a día de hoy para detectar este tipo de ataques (misma misión que la actualización del *plugin* HeapList realizada). En concreto, sus autores hablan sobre `AddressSanitizer`, una de las herramientas más extendidas para la detección práctica de *heap overflows* y *use-after-free*. Esta herramienta demuestra que muchos errores del *heap* pueden identificarse eficazmente mediante instrumentación en tiempo de ejecución. Sin embargo, en muchas ocasiones, la aplicación de estos sistemas de detección no es viable por la gran sobrecarga en ejecución que conllevan.

Otras técnicas como la *probabilistic memory safety*, cuya evolución es comentada en [30], refuerzan esta situación. De esta forma, la técnica basada en ejecutar múltiples réplicas del programa con disposiciones de memoria diferentes, de modo que un mismo ataque provoque divergencias observables entre instancias; supone una gran sobrecarga tanto en memoria como en ejecución.

A partir de esta situación, surgen otras herramientas y técnicas de detección de ataques de corrupción de memoria que sean viables en tiempo de ejecución. `Xmon` [8] se conforma como una de las primeras, centrándose en identificar ataques reales en navegadores mediante la monitorización de comportamientos característicos de corrupción

de memoria. Para lograr esta labor se basan en indicadores dinámicos, como patrones de *heap spraying*, modificaciones sospechosas de estructuras internas o secuencias anómalas de asignación previas al desencadenamiento del fallo.

Otra de estas herramientas puede ser `TAILCHECK` [14], la cual se conforma como un esquema ligero de detección de *heap overflows* que aprovecha la protección de páginas y el etiquetado de punteros. En concreto, se basa en la creación de objetos sombra asociados al objeto real denominados `TailObject`. Estos se crean en una página protegida distanciada del objeto real, de forma que dicha distancia se almacena en los bits superiores no utilizados de la dirección del objeto real. De esta forma, `TAILCHECK` realiza un acceso de bajo sobrecoste al objeto sombra por cada acceso normal, el cual sirve para comprobar si se ha ejecutado un *heap overflow*.

También existen otras herramientas que se basan en nuevas tendencias como la inteligencia artificial. `Glyph` [26] es una de ellas, proponiendo un sistema de detección de *heap spraying* basado en aprendizaje automático. Este sistema se centra en extraer características estadísticas del contenido del *heap*, tales como entropía, repetitividad parcial y distribución de valores; llegando a detectar el *heap spraying* incluso cuando el atacante añade variabilidad (aleatorización de valores de relleno o NOP *sleds*). Este artículo representa una herramienta basada en tecnologías en auge, el cual ofrece dos versiones de la herramienta, una más ligera y algo menos precisa y una más robusta, pero pesada.

Por contraparte, no solo existen medidas *software*, sino que también surgen algunas medidas de detección basadas en *hardware*. Entre ellas se encuentra la *Memory Tagging Extension* introducida por ARM [4], la cual asocia etiquetas tanto a punteros como a regiones de memoria y verifica su coherencia en cada acceso. Debido a su implementación *hardware*, este sistema permite la detección de ataques como el *use-after-free* o los *overflows* con un bajo sobrecoste de ejecución.

Finalmente, se aprecia cómo todas estas herramientas y técnicas de detección no son suficientes para pararle los pies a los atacantes, que más ingeniosos suelen resultar al encontrar siempre la forma de sobrepasar este tipo de defensas. Esto es reflejado incluso por Microsoft [6], que comenta que el endurecimiento de algunos procesos del gestor

del *heap* puede contribuir a frenar ataques como UAFs o *heap overflows*, pero esto no evita que los atacantes encuentren nuevas técnicas o vulnerabilidades que puedan aprovechar.

Es en este punto donde entran herramientas como Volatility 3 y el *plugin* HeapList [41], permitiendo el análisis de la memoria dinámica en busca de patrones que ayuden a crear o actualizar herramientas de detección. Además, la actualización propuesta en el presente trabajo sobre el *plugin* HeapList conforma una herramienta de análisis forense que permite la detección de ciertos ataques en volcados de memoria RAM postmortem.

Capítulo 6

Conclusiones y trabajos futuros

El presente trabajo logra completar su objetivo principal encontrando patrones entre las *heap entries* de un proceso donde se han ejecutado ataques de tipo *heap overflow* o *heap spraying*. Sin embargo, esto no es posible para el ataque *use-after-free* al no alterar las estructuras o metadatos visibles con el *plugin* HeapList. Además, el ataque restante estudiado, el *double-free*, se descarta para el análisis forense realizado durante este trabajo debido a la detección en ejecución ya realizada por el sistema operativo.

Como demuestran los resultados, la actualización del *plugin* HeapList en base a estos patrones detectados se ha resuelto con éxito al conseguir una nueva versión capaz de detectar los ataques *heap overflow* y *heap spraying*. De esta forma, al activar la opción de detección de ataques se logra la identificación de los problemas mencionados en los POCs construidos para el análisis de estos ataques. Sin embargo, no se llegan a detectar falsos positivos, ni en los POCs elaborados, ni en un volcado de memoria referente a una aplicación más completa, Telegram.

Tras finalizar este trabajo se detectan una serie de trabajos futuros que pueden mejorar los resultados conseguidos. En primer lugar, se contempla la inclusión de nuevas pruebas, pricipalmente basadas en *heap spraying* con técnicas de evasión de la baja variabilidad presentada en las instrucciones NOP. Esto pondría a prueba y mejoraría la heurística basada en la entropia, la cual es utilizada para la detección de este tipo de problemas.

Por otro lado, se considera de interés la extensión de este trabajo a Windows 11, elaborando las adaptaciones necesarias para la ejecución tanto de los ataques, como del *plugin* en esta versión de Windows. Por último, se propone la inclusión de nuevos POCs que incluyan ataques más modernos al análisis realizado, así como la respectiva actualización en consecuencia del *plugin*. En esta expansión se pueden considerar ataques que conlleven condiciones o situaciones más específicas, los cuales se salgan de los clásicos estudiados en este trabajo.

Lista de referencias

[1] ACTE Technologies: Difference Between Stack and Heap – Memory Management (2025), https://www.acte.in/difference-between-stack-and-heap

[2] AV-TEST Institute: AV-ATLAS – Malware & PUA Statistics (2026), https://portal.av-atlas.org/malware?s=eb7e441930ede3460e580911a33a0f5157bf9d5d

[3] Baker, K.: Fileless Malware Explained. Fileless Malware (2024), https://www.crowdstrike.com/en-us/cybersecurity-101/malware/fileless-malware

[4] Bannister, S.: Memory Tagging Extension: Enhancing memory safety through architecture. Arm, Aug **5** (2019), https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety

[5] Capstone Engine Project: Capstone–Python Bindings Documentation (2026), https://www.capstone-engine.org/lang_python.html

[6] Center, M.S.R.: Software Defense: mitigating heap corruption vulnerabilities (2013), https://www.microsoft.com/en-us/msrc/blog/2013/10/software-defense-mitigating-heap-corruption-vulnerabilities

[7] Chang, J.M., Srisa-an, W., Lo, C.T.D.: DMMU: Dynamic Memory Management Unit. Dynamic Memory Management https://www.ece.iastate.edu/~morris/papers/dmmu.pdf

[8] Chennette, S., Joseph, M.: Detecting Web Browser Heap Corruption Attacks. presentation, Black Hat Europe (2007), https://blackhat.com/presentations/

bh-usa-07/Chenette_and_Joseph/Presentation/bh-usa-07-chenette_and_joseph.pdf

[9] CTF101: Heap Exploitation (2026), https://ctf101.org/binary-exploitation/heap-exploitation/

[10] Du, S., Li, Z., Li, J.: Heap Vulnerability Exploitation Techniques on the Linux Platform: An Overview. In: 2024 9th International Conference on Signal and Image Processing (ICSIP). pp. 371–382. IEEE (2024), https://ieeexplore.ieee.org/abstract/document/10671522/

[11] Ducasse, Q., Cotret, P., Lagadec, L.: War on JITs: Software-Based Attacks and Hybrid Defenses for JIT Compilers-A Comprehensive Survey. ACM Computing Surveys **57**(9), 1–36 (2025), https://dl.acm.org/doi/abs/10.1145/3731598

[12] Free Software Foundation, Inc.: GNU Compiler Collection (GCC) (2026), https://gcc.gnu.org/

[13] Gingele, T.: double free vulnerability poc. https://github.com/B1TC0R3/double-free-vulnerability-poc (2021)

[14] Gopal, A.U.S., Soori, R., Ferdman, M., Lee, D.: TAILCHECK: A lightweight heap overflow detection mechanism with page protection and tagged pointers. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). pp. 535–552. USENIX Association, Boston, MA (Jul 2023), https://www.usenix.org/conference/osdi23/presentation/gopal

[15] corelanc0d3r (traducido por Ivinson): Heap Spraying Desmitificado. https://elhacker.info/manuales/Exploit/Creacion%20de%20Exploits%2011%20Heap%20Spraying%20Desmitidficado%20por%20corelanc0d3r%20traducido%20por%20Ivinson.pdf (2026)

[16] Jain, A.: Heap Exploitation: Understanding Memory Corruption Attacks. Memory Corruption (2025), https://www.akshayjain.blog/post/heap-exploitation-understanding-memory-corruption-attacks

[17] Kara, I.: Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges. Expert Systems with Applications

214, 119133 (2023). https://doi.org/https://doi.org/10.1016/j.eswa.2022.119133, https://www.sciencedirect.com/science/article/pii/S0957417422021510

[18] Karimi, A.: A Survey of Heap-Exploitation Techniques (2021), https://www.researchgate.net/profile/Alireza-Karimi-31/publication/369594354_A_survey_of_heap-exploitation_techniques/links/6423d78392cfd54f84388e5b/A-survey-of-heap-exploitation-techniques.pdf

[19] Lenovo: Heap Memory Explained (2026), https://www.lenovo.com/us/en/glossary/heap

[20] Li, R., Zhang, B., Chen, J., Lin, W., Feng, C., Tang, C.: Towards Automatic and Precise Heap Layout Manipulation for General-Purpose Programs. In: NDSS (2023), https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s232_paper.pdf

[21] Liu, Z.E.: Advanced Heap Manipulation in Windows 8. Black Hat Europe (2013), https://media.blackhat.com/eu-13/briefings/Liu/bh-eu-13-liu-advanced-heap-WP.pdf

[22] Microsoft: GFlags, https://learn.microsoft.com/es-es/windows-hardware/drivers/debugger/gflags

[23] Nicula, Ş., Zota, R.D.: Analysis of Heap Manager for Windows 7 & 10 from an Exploitation Perspective. International Journal of Economics, Commerce and Management **IX**(5), 213–223 (2021), https://ijecm.co.uk/wp-content/uploads/2021/05/9514.pdf

[24] Nyholm, H., Monteith, K., Lyles, S., Gallegos, M., DeSantis, M., Donaldson, J., Taylor, C.: The evolution of volatile memory forensics. Journal of Cybersecurity and Privacy **2**(3), 556–572 (2022), https://www.researchgate.net/publication/362147648_The_Evolution_of_Volatile_Memory_Forensics

[25] Patil, N.V., Irabashetti, P.S.: Dynamic memory allocation: role in memory management. International Journal of Current Engineering and Tech-

nology **4**(2), 531–535 (2014), https://www.researchgate.net/publication/265166374_Dynamic_Memory_Allocation_Role_in_Memory_Management

[26] Pierazzi, F., Cristalli, S., Bruschi, D., Colajanni, M., Marchetti, M., Lanzi, A.: Glyph: Efficient ML-based detection of heap spraying attacks. IEEE Transactions on Information Forensics and Security **16**, 740–755 (2020), https://ieeexplore.ieee.org/abstract/document/9171343/

[27] Rapid7: Heap Overflow Exploitation on Windows 10 Explained. https://www.rapid7.com/blog/post/2019/06/12/heap-overflow-exploitation-on-windows-10-explained/ (2019), explicación técnica de explotación de heap overflow en Windows 10; consultado en 2026

[28] Ratanaworabhan, P., Livshits, V.B., Zorn, B.G.: NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In: USENIX security symposium. pp. 169–186 (2009), https://www.usenix.org/legacy/event/sec09/tech/full_papers/ratanaworabhan.pdf

[29] Rekall Forensic and Incident Response Framework: Persistent Memory (pmem) Tool — Rekall Documentation (2026), https://rekall.readthedocs.io/en/gh-pages/Tools/pmem.html

[30] Rösti, A., Voulimeneas, A., Franz, M.: The astonishing evolution of probabilistic memory safety: From basic heap-data attack detection toward fully survivable multivariant execution. IEEE Security & Privacy **22**(4), 66–75 (2024), https://ieeexplore.ieee.org/abstract/document/10557576/

[31] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: {AddressSanitizer}: A fast address sanity checker. In: 2012 USENIX annual technical conference (USENIX ATC 12). pp. 309–318 (2012), https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[32] Soares, L.R.: Understanding Heap Spraying Attacks (2020), https://medium.com/@luishrsoares/understanding-heap-spraying-attacks-eb1577419410, medium blog post

[33] StatCounter Global Stats: Desktop and Console Operating System Market Share Worldwide (2025) (2026), https://gs.statcounter.com/os-market-share/desktop-console/worldwide/#monthly-202501-202512-bar

[34] Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. Memory Corruption pp. 48–62 (2013), https://ieeexplore.ieee.org/abstract/document/6547101

[35] The MITRE Corporation: 2025 CWE Top 25 Most Dangerous Software Weaknesses (2025), https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

[36] The MITRE Corporation: CWE-122: Heap-based Buffer Overflow (2025), https://cwe.mitre.org/data/definitions/122.html

[37] The MITRE Corporation: CWE-126: Buffer Over-read (2025), https://cwe.mitre.org/data/definitions/126.html

[38] The MITRE Corporation: CWE-416: Use After Free (2025), https://cwe.mitre.org/data/definitions/416.html

[39] The MITRE Corporation: CWE-787: Out-of-bounds Write (2025), https://cwe.mitre.org/data/definitions/787.html

[40] The MITRE Corporation: CWE-415: Double Free (2025), https://cwe.mitre.org/data/definitions/415.html

[41] Uroz, D., Díaz-Campo Pinilla, A., Rodríguez, R.J.: Structural Analysis of the Windows NT Heap for Memory Forensics. Forensic Science International: Digital Investigation **51**, 301726 (2026)

[42] Valasek, C.: Understanding the low fragmentation heap. Black Hat USA **11** (2010), https://www.illmatics.com/Understanding_the_LFH.pdf

[43] van der Veen, V., dutt Sharma, N., Cavallaro, L., Bos, H.J.: Memory Errors: The Past, the Present, and the Future. Memory Errors **7462**, 86–106 (2012), https://link.springer.com/chapter/10.1007/978-3-642-33338-5_5

[44] Volatility Foundation: Volatility 3 Documentation (2025), https://volatility3.readthedocs.io/en/stable/

[45] Volatility Foundation: The Volatility Framework (2026), https://volatilityfoundation.org/the-volatility-framework/

[46] White Knight Labs: Understanding Double Free in Windows Kernel Drivers (2025), https://whiteknightlabs.com/2025/06/10/understanding-double-free-in-windows-kernel-drivers/

[47] White Knight Labs: Understanding Use After Free (UAF) in Windows Kernel Drivers (2025), https://whiteknightlabs.com/2025/06/03/understanding-use-after-free-uaf-in-windows-kernel-drivers/

[48] Yason, M.V.: Windows 10 segment heap internals. Blackhat USA (2016), https://blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals-wp.pdf

[49] ZHAI, J., CHEN, P., XU, X., YANG, H.: The memory forensic research oriented to segment heap in Windows 10 system. Xibei Gongye Daxue Xuebao/Journal of Northwestern Polytechnical University **39**, 1139–1149 (10 2021), https://www.researchgate.net/publication/357024125_The_memory_forensic_research_oriented_to_segment_heap_in_Windows_10_system

Anexo A

Distribucción temporal de tareas

Durante este anexo se detalla la distribucción temporal de las distintas etapas que han compuesto la elaboración del presente trabajo. Esto se realiza mediante un diagrama de Gantt (Figura A.1) y una tabla (Tabla A.1) que detalla las tareas cubiertas en cada etapa, así como una aproximación de las horas empleadas.

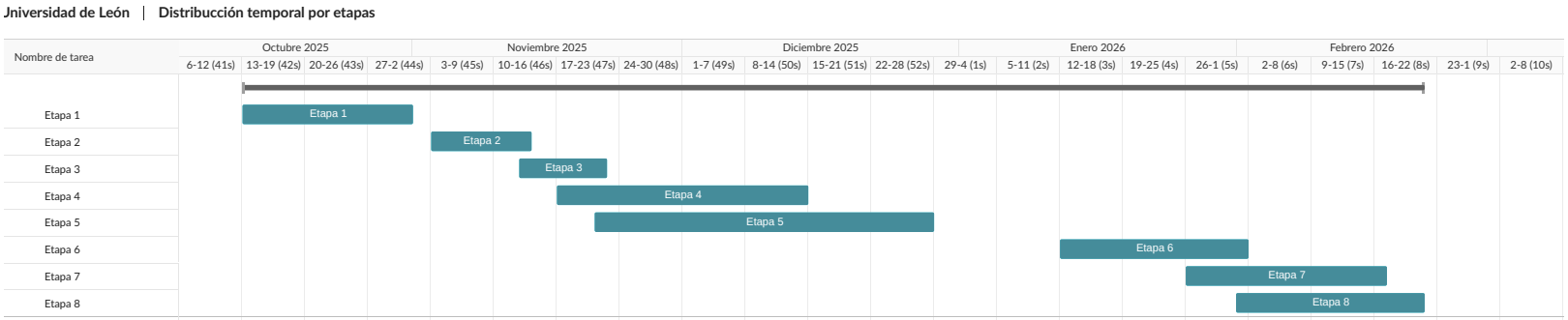

Figura A.1: Diagrama de Gantt. Fuente: propia

Tabla A.1: Distribución de horas por etapa. Fuente: propia

Etapas	Descripción	Horas
Etapa 1	Lectura y comprensión de la documentación relacionada con la detección de ataques de corrupción de memoria y el HeapList.	70
Etapa 2	Construcción del entorno de pruebas.	30
Etapa 3	Comprensión y prueba de las herramientas para el análisis forense.	40
Etapa 4	Búsqueda de información y compresión de los ataques contra el *heap* de Windows.	100
Etapa 5	Implementación de las POCs que ejecutan los ataques contra el *heap*.	250
Etapa 6	Volcados de memoria y análisis en búsqueda de patrones que identifiquen los ataques elaborados en las POCs.	50
Etapa 7	Actualización del *plugin* HeapList en consecuencia a los patrones detectados.	35
Etapa 8	Escritura de la presente memoria.	75
	Total de horas empleadas	**615**