universidad de león

Departamento de Matemáticas

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

DESARROLLO DE UNA LIBRERÍA DE USUARIO PARA DETECCIÓN DE TOCTOU EN ENTORNOS LINUX

DEVELOPMENT OF A USER LIBRARY FOR TOCTOU DETECTION IN LINUX ENVIRONMENTS

Autor: Razvan Raducu

Tutor: Ricardo J. Rodríguez

(Septiembre, 2019)

UNIVERSIDAD DE LEÓN

Departamento de Matemáticas

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

ALUMNO: Razvan Raducu

TUTOR: Ricardo J. Rodríguez

TÍTULO: Desarrollo de una librería de usuario para detección de TOCTOU en entornos Linux

TITLE: Development of a user library for TOCTOU detection in Linux environments

CONVOCATORIA: Septiembre, 2019

RESUMEN:

Las condiciones de carrera pueden surgir cuando se realizan operaciones en el sistema de ficheros debido a que el sistema operativo no asegura la atomicidad de las operaciones de entrada/salida ni tampoco garantiza la consistencia del sistema de ficheros. Estas vulnerabilidades suponen una grave amenaza para la integridad, disponibilidad y confidencialidad de la información y de los sistemas, pues llegan a permitir a un atacante desde denegar el servicio a la aplicación hasta escalar privilegios en el sistema. Estas vulnerabilidades, conocidas como *Time Of Check to Time Of Use* (TOCTOU), suponen un auténtico reto dada su naturaleza no determinista. Su detección y prevención son difíciles y aún más su reproducción en análisis post-mortem debido a que su explotabilidad depende de factores externos como la carga del sistema o el estado de las variables de entorno. En este trabajo se presenta el estudio de la vulnerabilidad y el desarrollo de una librería de usuario para entornos Linux para detectarla y prevenirla llevando a cabo *hooking* de funciones. La librería trata de impedir la vulnerabilidad cambiando el comportamiento de las funciones marcadas como vulnerables para que comprueben la consistencia del sistema de ficheros. Se detallan, además, las limitaciones de la librería, así como su rendimiento mediante el uso de diferentes *benchmarks*.

Palabras clave: TOCTOU, TOCTTOU, Time Of Check to Time Of Use, Race Condition, Condición de Carrera, Librería, Sistema de Ficheros

Firma del alumno:

VºBº Tutor:

Digitally signed by RODRIGUEZ FERNANDEZ RICARDO JULIO - [redacted]

Date: 2019.09.02 15:26:00 +02'00'

Abstract

Race conditions may arise when performing operations with the file system because there is no guarantee of atomicity for I/O operations or consistency of the file system. These vulnerabilities are a true menace to integrity, availability and confidentiality of information systems since they may allow an attacker to scale privileges. These vulnerabilities, known as *Time Of Check to Time Of Use* (TOCTOU), are a true challenge given their non-deterministic nature. Detecting and preventing them is quite difficult and even more to reproduce them while performing post-mortem analysis, because their exploitability depends upon external factors such as system load or environment variables. This work presents a study of the vulnerability as well as the development of a user library for Linux to detect and prevent it through function hooking. The developed library aims at preventing the vulnerability by changing the vulnerable functions behavior to verify file system's object integrity. The limitations of the developed library are also discussed as well as its performance.

Agradecimientos

A mis padres, por todos sus sacrificios

A Gonzalo Esteban Costales, por su ayuda no remunerada

Y, por supuesto, al director de este trabajo, Ricardo J. Rodríguez, por su paciencia y constante ayuda.

GRACIAS.

Exploit code, not people.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

1. Introducción

Las condiciones de carrera se producen cuando procesos que se están ejecutando de manera simultánea realizan accesos no sincronizados (o mal sincronizados) a la memoria que comparten, o cuando diferentes hilos de ejecución comparten un espacio de direcciones común del que pueden leer y/o escribir de forma asíncrona. En ambos casos, el resultado de la ejecución depende del orden de estos accesos. Cuando se produce una situación de estas características, los programas se comportan de formas no previstas por los programadores. Desde el punto de vista de la seguridad esto es grave, pues estos comportamientos no controlados pueden resultar en vulnerabilidades que permitan a un atacante desde denegar el servicio a la aplicación hasta escalar privilegios en el sistema [1–3].

Existen diferentes tipos de condiciones de carrera. Uno de ellos es la vulnerabilidad conocida como *Time Of Check to Time Of Use* (TOCTOU), a veces también referenciada en la literatura como TOCTTOU o TOC/TOU. Las vulnerabilidades TOCTOU existen cuando las secuencias de operaciones de un determinado programa, proceso o hilo sobre el sistema de ficheros no se llevan a cabo de forma aislada. Esto es, estas operaciones no se ejecutan de manera totalmente independiente de otros procesos. Esto sucede cuando las aplicaciones (o en su defecto los programadores de las mismas) asumen que una determinada secuencia de operaciones sobre el sistema de ficheros es atómica o, por otra parte, que su programa es el único interactuando en ese momento con el sistema de ficheros [4].

El presente trabajo se centra en las vulnerabilidades de tipo TOCTOU dado que son una amenaza realmente importante para los sistemas de información y, además, llevan mucho tiempo existiendo. Tanto es así, que la primeras referencias acerca de esta vulnerabilidad datan de mediados de la década de los 70: en los años 1974 [5], 1976 [6] y 1978 [7], TOCTOU se definía como una subclase de la clase de vulnerabilidades relacionadas con la sincronización o el *timing*. A principios de milenio, entre 2000 y 2004, por ejemplo, en el CERT (*Computer Emergency Response Team*) de Estados Unidos se reportaron 20 incidencias relacionadas con TOCTOU de las que 11 permitían al atacante escalar privilegios de superusuario, es decir, acceso no autorizado como root.

Las vulnerabilidades se produjeron en gran variedad de aplicaciones y en múltiples sistemas operativos basados en Linux [8].

En el año 2005, en la Universidad de California, Berkeley, llevaron a cabo una auditoría del sistema operativo Red Hat 9 encontrando 41 vulnerabilidades de tipo TOCTOU [9] y, en el mismo año, aplicaciones de diversa índole fueron reportadas como víctimas de esta vulnerabilidad, como se aprecia en la Tabla 1.1. En la misma línea, entre los años 2008 y 2016 en la *Common Vulnerabilities and Exposures*[1] (CVE) se reportaron 377 vulnerabilidades de tipo condición de carrera, con tendencia creciente año tras año, y 389 vulnerabilidades relacionadas con la resolución de enlaces (*Link Following*), con tendencia estable año tras año [10]. Como se describe en la Sección 2.1 Condición de carrera, ambas están fuertemente ligadas al presente trabajo. Además, la *Common Weakness Enumeration*[2] (CWE) identifica diferentes vulnerabilidades relacionadas con TOCTOU. Estas vulnerabilidades se pueden ver en la Tabla 1.2.

Tabla 1.1. Aplicaciones víctimas de TOCTOU reportadas en 2005. (Fuente: [8, 11])

Dominio de aplicación	Nombre de aplicación
Aplicaciones empresariales	Apache, bzip2, gzip, getmail, Imp-webmail, procmail, openldap, openSSL, Kerberos, OpenOffice, StarOffice, CUPS, SAP, samba
Herramientas administrativas	at, diskcheck, GNU fileutils, logwatch, patchadd
Gestores de dispositivos	Esound, glint, pppd, Xinetd
Herramientas de desarrollo	make, perl, Rational ClearCase, KDE, BitKeeper, Cscope

A día de hoy la vulnerabilidad se sigue produciendo. Para comprobarlo basta con hacer una sencilla búsqueda en cualquier base de vulnerabilidades abierta como la NVD (*National Vulnerability Database*)[3]. Se puede apreciar que los incidentes reportados son de fecha reciente (julio 2019, concretamente) y que algunos de ellos están catalogados con severidad alta e incluso crítica.

[1] `https://cve.mitre.org/`

[2] `https://cwe.mitre.org/index.html`

[3] `https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=statistics&query=time+of+check+to+time+of+use&search_type=all`

Tabla 1.2. Listado de la CWE de vulnerabilidades relacionadas. (Fuente: CWE)

Vulnerabilidad	ID CWE
Resolución inapropiada de enlaces antes de acceso a ficheros (Seguimiento de enlaces)	CWE-59
Seguimiento de enlaces simbólicos en UNIX	CWE-61
Uso de enlaces (*hard links*) en UNIX	CWE-62
Ejecución concurrente haciendo uso de recursos compartidos con sincronización inapropiada (Condición de carrera)	CWE-362
Seguimiento de enlaces generando condiciones de carrera	CWE-363
Condición de carrera *Time-of-check Time-of-use* (TOCTOU)	CWE-367

Otra característica importante de las vulnerabilidades TOCTOU es que son transversales en cuanto al sistema operativo, es decir, que existen tanto en sistemas operativos basados en Linux como en Windows [12, 13]. Esto es debido a que estos sistemas operativos trabajan con sistemas de ficheros cuya consistencia es débil o, incluso, inexistente [14–16]. Asímismo, las vulnerabilidades de tipo condición de carrera pueden ser muy difíciles de detectar y eliminar en tiempo de compilación, con lo cual se considera de extrema utilidad desarrollar protecciones dinámicas (en tiempo de ejecución).

Este trabajo se centra en el sistema operativo Linux. Esta decisión no es arbitraria, pues con el paso del tiempo el uso de sistemas operativos basados en Linux ha aumentado notablemente. Tanto es así que a día de hoy, en agosto/septiembre 2019, el 70.6 % del top 10 millones de dominios web con más tráfico (según el ranking de Alexa[4]) utiliza algún sistema operativo basado en Unix, mientras que tan sólo el 29.4 % utiliza Windows [17]. Dentro de este 70.6 % que utilizan Unix, el 51.7 % hacen uso de Linux [18]. Linux, o cualquiera de sus distribuciones, es también a día de hoy el sistema operativo más usando en el mundo de la supercomputación [19].

El principal objetivo de este trabajo es estudiar la vulnerabilidad TOCTOU dentro del sistema de ficheros Linux para proponer una solución a esta problemática.

El resultado del trabajo es el desarrollo de una librería de usuario con alcance de sistema cuyo propósito es detectar y frustrar los intentos de explotación de TOCTOU.

[4] `https://www.alexa.com/siteinfo`

La librería recibe el nombre de libTTThwart y se ha publicado bajo licencia GNU GPL v3. El código fuente, la documentación y las instrucciones para utilizarla están disponibles en:

`https://github.com/RazviOverflow/libTTThwart`

La librería se ha evaluado mediante microbenchmarking así como con la *suite* CPU SPEC2006. El microbenchmarking consiste en evaluar el rendimiento de determinadas partes pequeñas de un software, como una llamada al sistema. CPU SPEC2006 es un estándar en la industria de software y permite medir el rendimiento de los sistemas informáticos en base a unas referencias predefinidas [20].

El presente documento se divide en 6 capítulos principales. El Capítulo 2 asienta las bases y el contexto de todo el trabajo llevado a cabo, explicando los conocimientos necesarios para comprender el resto del documento. El Capítulo 3 presenta la solución propuesta detallando sus características. El Capítulo 4 se centra en los experimentos llevados a cabo, así como la evaluación del impacto de la herramienta desarrollada en el sistema operativo. El Capítulo 5 hace un repaso del trabajo relacionado. Por último, el Capítulo 6 presenta las conclusiones del trabajo y las líneas futuras del mismo.

Adicionalmente, se cuenta con los siguientes anexos con el objetivo de complementar y aclarar todo lo expuesto a lo largo del documento. Los anexos son:

Anexo A Desglose del trabajo llevado a cabo, especificando las horas invertidas y el seguimiento del mismo.

Anexo B Detalle de la instalación de la librería desarrollada, libTTThwart, así como el acceso e interpretación de la documentación generada.

Anexo C Desglose de los resultados obtenidos con el benchmark SPEC CPU2006 y presentación de la estadística descriptiva de los mismos.

Anexo D Presentación del código fuente usado en las pruebas de microbenchmarking.

2. Conceptos previos

En el presente capítulo se detallan todos los conceptos necesarios para comprender el trabajo realizado. De cara a contextualizar el trabajo, es importante conocer qué es una condición de carrera y, concretamente, qué es TOCTOU y cómo se explota, qué es una librería y, por último, cómo funciona el sistema de ficheros en Linux.

2.1. Condición de carrera

Una condición de carrera puede definirse como: *"Comportamiento anómalo debido a la dependencia crítica del orden relativo de los eventos."* [21]. Las condiciones de carrera habitualmente involucran uno o más procesos accediendo a un recurso común, como una variable o un fichero, mientras este acceso múltiple no es controlado adecuadamente. Esto suele suceder en sistemas operativos que trabajan con planificación apropiativa [22–25] y multitarea apropiativa [26, 27], del inglés *preemptive scheduling* y *preemptive multitasking*. Es decir, sistemas operativos que ofrecen la posibilidad de que un proceso que se está ejecutando interrumpa su ejecución en favor de otro, lo que se conoce como un cambio de contexto. En los procesos de usuario la mayoría de condiciones de carrera se reducen al sistema de ficheros mientras que, por otra parte, en el núcleo del sistema operativo las condiciones de carrera pueden producirse en diversas zonas como en el código de gestión de memoria virtual [28].

Resumiendo, las condiciones de carrera se producen cuando dos eventos suceden y el segundo depende del resultado primero. Durante el intervalo de tiempo entre ambos eventos, se asume que el resultado del primero sigue siendo cierto al comenzar el segundo cuando, en realidad, un actor externo ha podido realizar acciones que invaliden esa suposición [15].

Precisamente el término "condición de carrera" hace referencia a la "carrera" que el atacante hace por invalidar dichas suposiciones antes de que se produzca el segundo evento. Para que se produzca una condición de carrera deben darse las siguientes propiedades:

- **Concurrencia**. Debe haber al menos dos programas, procesos, hilos o tareas ejecutándose de forma simultánea o concurrente.

- **Recurso común**. Los elementos que se están ejecutando simultánea o concurrentemente deben acceder a un recurso común, como una variable o un fichero.
- **Modificación**. El recurso común accedido debe ser modificado por al menos uno de los procesos. Si ninguno de los elementos que acceden al recurso alteran su estado, no se produce ninguna condición de carrera pues no importa el orden en el que se produzca el acceso ya que el resultado de la operación siempre será el mismo.

Las condiciones de carrera en accesos a sistemas de ficheros se producen cuando secuencias de operaciones sobre el sistema no se llevan a cabo de manera aislada. Es decir, cuando las aplicaciones asumen erróneamente que una secuencia de operaciones sobre el sistema de ficheros es atómica y el propio sistema de ficheros es, por otra parte, estático. Ambas suposiciones son totalmente erróneas y equivocaciones de esta índole llegan a permitir a atacantes una escalada de privilegios en el sistema afectado, entre otros ataques [4].

2.1.1. Vulnerabilidad TOCTOU

TOCTOU es un tipo de condición de carrera que sucede en sistemas de ficheros con semántica de sincronización débil. Esto es, sistemas operativos que no tienen mecanismos para asegurar que el sistema de ficheros permanece inalterable entre diferentes interacciones consecutivas con el mismo.

Las vulnerabilidades de tipo TOCTOU se producen en dos pasos [15]:

1. El programa vulnerable comprueba el estado de un fichero. Esta operación se define como *Time Of Check*.
2. El programa vulnerable opera sobre el fichero asumiendo que el resultado de la operación anterior ha permanecido invariante durante la ejecución. Este momento se denomina *Time Of Use*.

Es condición necesaria para que un ataque a una vulnerabilidad TOCTOU termine con éxito que la víctima (el programa vulnerable) se esté ejecutando con mayor nivel de privilegios que el atacante. Esto en sistemas operativos basados en Unix, como Linux, se traduce en que el atacante es un usuario con menos privilegios que el programa en

ejecución [8]. De esta forma, la vulnerabilidad sucede cuando el programa vulnerable interactúa con un objeto del sistema de ficheros cuyo propietario es el atacante. El premio de explotar con éxito la vulnerabilidad es conseguir escalar esos privilegios.

Una observación importante es que, además de que el programa vulnerable se ejecute con mayor nivel de privilegios, el atacante debe poder interactuar con el sistema de ficheros, pues es precisamente de la inconsistencia de este de donde nace la vulnerabilidad. Esto es, el atacante debe poder realizar operaciones sobre el fichero que la aplicación vulnerable está referenciando (por ejemplo, creación y borrado).

La ocurrencia de esta vulnerabilidad depende de llamadas concretas al sistema, en un orden concreto, así como de variables y condiciones de entorno [8, 11]. Todo esto hace que su reproducibilidad durante análisis post-mortem sea muy difícil.

Por otra parte, las vulnerabilidades de tipo TOCTOU son también más difíciles de explotar ya que, dada la naturaleza de las condiciones de carrera, el éxito del ataque dependerá de las acciones precisas y oportunas por parte de un atacante en un determinado momento de la ejecución, lo que se conoce como *ventana de vulnerabilidad*. Para tratar de *ganar la carrera* con la mayor probabilidad de éxito, los atacantes pueden emplear diversas técnicas como sobrecargar el sistema o fabricar entradas de datos que aumenten el tamaño de la ventana.

Otra dificultad añadida que tiene el estudio de estas vulnerabilidades es que su reproducción exacta es muy difícil de conseguir. Esto se debe a que, como se ha mencionado anteriormente, su explotación depende de elementos tales como la carga del sistema en el momento de la ejecución o las llamadas al sistema que puedan hacer el resto de procesos ejecutándose.

2.1.2. Explotación de vulnerabilidad TOCTOU

Para explotar una vulnerabilidad TOCTOU, el programa vulnerable deberá ser inducido a realizar alguna acción que tenga determinado provecho para el atacante a la hora de ejecutar la segunda llamada de la secuencia vulnerable.

Un elemento importante a tener en cuenta a la hora de la explotación de las vulnerabilidades TOCTOU es la ventana de vulnerabilidad. Tradicionalmente se ha considerado que la probabilidad de éxito del ataque es directamente proporcional al tamaño de la ventana de vulnerabilidad [14]. Sin embargo, el atacante puede fabricar entradas

al programa vulnerable de tal forma que, independientemente de su tamaño inicial, la ventana de vulnerabilidad crece hasta hacer posible la explotación.

Una de las formas más comunes de explotación de TOCTOU se lleva a cabo mediante el uso de enlaces simbólicos. Un enlace simbólico es una entrada en el sistema de ficheros que simplemente referencia a otro fichero o directorio. La vulnerabilidad se produce cuando un programa vulnerable hace uso de rutas relativas o absolutas para interactuar con un fichero o directorio. A lo largo de la ejecución puede que el fichero referenciado por dicha ruta se convierta en un enlace simbólico que apunte a otro objeto del sistema de ficheros. Dado que los procesos no se ejecutan atómicamente, un atacante puede aprovecharse de esta situación y alterar, durante la ventana de vulnerabilidad, el fichero o directorio originalmente consultado, convirtiéndolo en un enlace simbólico apuntando a otro objeto de mayor interés.

Supóngase un programa vulnerable que se ejecuta con privilegios elevados y que accede a un fichero cuya ruta es `/tmp/file.tmp`. El programa escribe en el fichero lo que el usuario le pasa como argumento. Antes de escribir, el programa usa la función `access()` para comprobar que el usuario que ha lanzado el proceso tiene permisos de escritura en el fichero. Si los tiene, utiliza la función `open()` para abrir el fichero y escribir en él. El atacante puede manipular el sistema de ficheros entre ambas llamadas y cambiar el objeto referenciado. Si con este cambio consigue que el fichero original se convierta en un enlace simbólico a un fichero protegido, el atacante podría escribir en él. Si, además, el atacante le pasa al programa un parámetro deliberadamente fabricado, este podría escalar privilegios. Este ejemplo se ilustra en la Tabla 2.1, donde, explotando esta vulnerabilidad TOCTOU, el atacante puede lograr escribir en el fichero que almacena los usuarios y contraseñas en un sistema Linux (fichero "/etc/passwd").

2.2. Librería

En informática, una librería es un conjunto de ficheros que definen implementaciones de funciones. Las librerías son extremadamente útiles pues permiten reutilizar funciones sin tener que escribir una y otra vez su implementación.

Una librería está formada por una interfaz y la implementación de dicha interfaz. Una librería puede hacer uso de las funciones propiamente definidas o puede hacer uso

Tabla 2.1. Ejemplo de exploit basado en enlace simbólico. (Fuente: adaptado de [4])

Programa vulnerable	Código atacante
	`./vulnerable "razvan::0:0:pwn3d:/home:/bin/bash"`
`access("/tmp/file.tmp",`	
`W_OK))`	
	`rm /tmp/file.tmp`
	`ln -s /etc/passwd /tmp/file.tmp`
`file = open("/tmp/file",`	
`O_APPEND)`	
`write(file, argv[1],`	
`...)`	

de funciones definidas en otras librerías como la librería estándar de C o la versión GNU de esta.

Existen dos tipos de librerías: las librerías estáticas y las librerías compartidas o dinámicas. Las librerías estáticas forman parte del fichero ejecutable final. Esto significa que el código de las librerías se inserta dentro del ejecutable final, haciendo que este sea totalmente independiente. Las librerías estáticas tienen extensión `.a` (*archive*) en sistemas Unix y `.lib` (*library*) en sistemas Windows. Las librerías compartidas, también llamadas librerías dinámicas, no se incluyen en el código del binario final. El compilador insertará en el binario final información acerca de estas librerías y de cómo usarlas, pero no el código en sí. Cuando se ejecuta el binario, el cargador/enlazador dinámico del sistema buscará las librerías compartidas necesarias y las cargará en memoria [29, 30]. Las librerías compartidas tienen extensión `.so` (*shared object*) en sistemas Unix y `.dll` (*dynamic link libraries*) en Windows.

En sistemas Linux, las librerías compartidas se pueden utilizar de dos formas:

- Enlazadas dinámicamente en tiempo de arranque de programa.
- Cargadas dinámicamente durante la ejecución.

Como se verá a lo largo del presente documento y, sobretodo, en el Capítulo 3 Librería de sistema libTTThwart, en el proyecto intervienen tanto el proceso de enlazado dinámico como el de carga dinámica.

Por otra parte, muchas veces se diferencia entre librerías de sistema y librerías de usuario. Las librerías de sistema son aquellas librerías que forman parte del núcleo del sistema opeartivo y permiten su funcionalidad. Las librerías de usuario son las que los diferentes usuarios del sistema operativo instalan para poder usar sus aplicaciones.

2.3. Sistema de ficheros de Linux

Es importante conocer cómo se comporta el sistema de ficheros de Linux a la hora de referenciar objetos ya que tiene un rol de gran importancia en la explotabilidad la vulnerabilidad así como en su detección y defensa. Recuérdese que las vulnerabilidades TOCTOU se producen porque el sistema de ficheros es inconsistente entre las operaciones.

2.3.1. Referencia de objetos

Linux provee a los usuarios y a las aplicaciones dos formas de referenciar objetos en el sistema de ficheros, cada una con semántica diferente [15, 31, 32].

La primera de ellas es la ruta del fichero, en inglés *path* o *pathname*. Conceptualmente, el sistema de ficheros de Linux es un árbol donde los nodos interiores son directorios y los nodos hoja son ficheros, dispositivos u otras entidades [31]. Hay dos formas de hacer uso de las rutas para referenciar objetos. La primera de ellas es mediante rutas absolutas, donde el primer nodo es el directorio raíz y se especifica la ruta desde este hasta el objeto que se desea referenciar. Las rutas absolutas se caracterizan por empezar con "/", el directorio raíz. La segunda forma consiste en usar rutas relativas. Las rutas relativas se denominan así porque son relativas al directorio desde el que se están invocando o desde el que se está ejecutando la aplicación que está usando la ruta. El primer nodo siempre es el directorio actual, al menos que se especifique lo contrario con patrones como ".." (directorio que contiene al directorio actual). Para interpretar una ruta, tanto absoluta como relativa, el núcleo del sistema tiene que recorrer nodo por nodo hasta alcanzar el nodo hoja.

La segunda forma de referenciar los objetos del sistema de ficheros en Linux consiste en el uso de descriptores de ficheros. Los descriptores de ficheros son números enteros que se asignan directamente a objetos del sistema de ficheros y son locales a cada proceso[5]. Es decir, cada proceso cuenta con su conjunto particular de descriptores de ficheros. Cuando un proceso solicita la asignación de un descriptor de fichero a un determinado objeto, invoca la llamada al sistema correspondiente pasándole la ruta del fichero. En ese momento, el sistema mapea esa ruta en una estructura interna y devuelve una referencia a ella, el descriptor de fichero. La Figura 2.1 refleja, de forma simplificada, este mapeo.

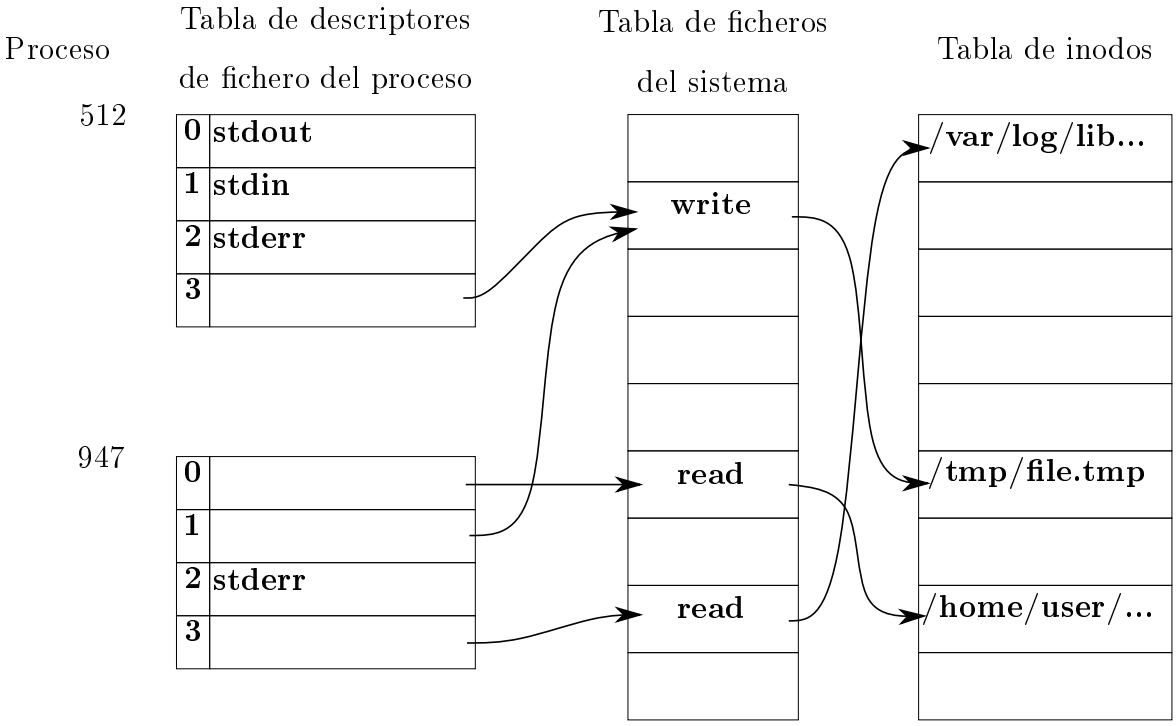

Figura 2.1. Tablas de descriptores de ficheros, ficheros abiertos e inodos. (Fuente: [31])

Nótese la diferencia entre ambas formas. Mientras que la referencia mediante rutas supone un conjunto de punteros indirectos al objeto y obliga al sistema a recorrer al menos un elemento del árbol, el uso de descriptores de ficheros supone un acceso directo, pues son punteros directos al objeto en cuestión.

[5]Hay que tener en cuenta que existen situaciones en las que un proceso hereda la tabla de descriptores de ficheros de otro proceso. como cuando se crean procesos hijos o se lanzan nuevos procesos durante la ejecución.

El uso de descriptores de ficheros proporciona al programa un acceso directo al objeto y, además, estos no pueden ser modificados por agentes externos. En cambio, el uso de rutas de ficheros obliga al núcleo a recorrer todos los elementos que las componen, corriendo así el riesgo de que actores externos modifiquen la asociación que existente el nodo de la ruta y el objeto del sistema de ficheros al que representa.

2.3.2. Metadatos de un objeto

Linux proporciona diferentes comandos, llamadas al sistema y utilidades para obtener información acerca de un objeto del sistema de ficheros. Un ejemplo es la función de C `stat()`, que forma parte del núcleo de Linux y es tanto un comando de consola como una llamada de la *Application Programming Interface* (API) de Linux. Versiones recientes del núcleo de Linux han introducido herramientas más completas, como la función `statx()`, pero su uso está limitado a las nuevas versiones del núcleo.

Cuando se invoca `stat` sobre un objeto del sistema de ficheros, este devuelve los metadatos que el sistema de ficheros registra acerca de ese objeto. El uso de la función `stat` de la API de Linux devuelve los datos en una estructura de tipo `stat`. En la Tabla 2.2 se especifican los datos que se obtienen así como una breve descripción de cada uno de ellos.

Tabla 2.2. Metadatos de un objeto del sistema de ficheros. Estructura `stat`.

(Fuente: [33])

Tipo	Metadato	Descripción
dev_t	st_dev	ID del dispositivo en el que está el objeto
ino_t	st_dino	Inodo del objeto
mode_t	st_mode	Tipo del objeto y sus permisos
nlink_t	st_nlink	Numero de enlaces (no simbólicos) que referencian al objeto
uid_t	st_uid	ID de usuario del propietario del objeto
gid_t	st_gid	ID de grupo del propietario del objeto
dev_t	st_rdev	ID de dispositivo (si es un objeto especial)
off_t	st_size	Tamaño total en bytes
blksize_t	st_blksize	Tamaño de bloque para E/S del sistema de ficheros
blkcnt_t	st_blocks	Número de bloques de 512B reservados

3. Librería de sistema libTTThwart

En el presente capítulo se describen todos los detalles de diseño e implementación de la solución, así como las limitaciones de la librería y algunas ideas descartadas.

3.1. Decisiones de diseño

El enfoque de esta solución está basado en la posibilidad que ofrece el cargador/enlazador dinámico de Linux de precargar librerías compartidas. Existen dos formas de hacerlo, o bien mediante la variable de entorno `LD_PRELOAD` o bien mediante el fichero de configuración `/etc/ld.so.preload`.

Ambas formas permiten precargar librerías compartidas especificando las rutas donde se encuentran. Si bien es cierto que parecen idénticas, no lo son. En primer lugar, la variable de entorno `LD_PRELOAD` tiene prioridad. Es decir, en caso de especificarse ambas, las librerías especificadas en `LD_PRELOAD` son precargadas antes. En segundo lugar, el alcance es diferente. `LD_PRELOAD` es una variable de entorno que sólo se aplica al proceso o procesos que tienen en cuenta esa variable, mientras que el fichero `/etc/ld.so.peload` tiene alcance de sistema, es decir, las librerías que se especifiquen serán precargadas para cualquier proceso que se ejecute. En tercer y último lugar, la variable de entorno `LD_PRELOAD` puede ser usada por cualquier usuario, lo cual podría suponer graves violaciones de seguridad, pues cualquier usuario podría sobreescribir funciones con código malicioso. Para evitar esto, `LD_PRELOAD` está sujeta a la siguiente restricción: si el *Real User ID* (RUID) del proceso a ejecutar es diferente de su *Effective User ID* (EUID), automáticamente `LD_PRELOAD` es ignorada. Esto es una imposición del sistema y por seguridad no se puede saltar. Por otra parte, el fichero `/etc/ld.so.preload` no sufre del tal restricción puesto que, al encontrarse en el directorio `/etc/`, se asume que si alguien tiene permisos de lectura/escritura en dicho directorio ya posee privilegios suficientes.

En la Figura 3.1a se muestra el uso de `LD_PRELOAD` y en la Figura 3.1b el de `/etc/ld.so.preload`. De esta forma, libTTThwart es una librería compartida que, haciendo uso de `/etc/ld.so.preload`, sobrescribe el comportamiento de las funciones

marcadas como vulnerables. El cambio de comportamiento es totalmente transparente para el programa original.

Se ha descartado `LD_PRELOAD` precisamente porque la limitación que tiene es incompatible con la vulnerabilidad. Recuérdese que TOCTOU se produce cuando el programa vulnerable se ejecuta con más privilegios que el atacante, es decir, el RUID es diferente al EUID.

```
$ LD_PRELOAD=/path/to/libTTThwart./vulnerable
```

(a) Librería especificada mediante `LD_PRELOAD`

```
$ echo "/path/to/libTTThwart" | sudo tee -a /etc/ld.so.preload
```
(*sólo ha de ejecutarse una vez*)

```
$./vulnerable
```

(b) Librería especificada mediante `/etc/ld.so.preload`

Figura 3.1. Ejemplos de ejecución del programa `vulnerable` y la librería libTTThwart usando (a) `LD_PRELOAD` y (b) `/etc/ld.so.preload`. (Fuente: propia)

La precarga de la librería libTTThwart la lleva a cabo el enlazador/cargador dinámico de Linux. El enlazador es un programa encargado de combinar y *enlazar* varios ficheros de tipo objeto en un único ejecutable, como el código principal del binario y las librerías estáticamente enlazadas. A un nivel más bajo, la función principal del enlazador es asociar los nombres simbólicos con direcciones de memoria, lo que se conoce como resolución de símbolos. Por otra parte, el cargador es un programa cuyo propósito es *cargar* en memoria los programas a ejecutar o las librerías dinámicamente enlazadas. A un nivel más bajo, el cargador es el encargado de reservar memoria suficiente para cargar las instrucciones y los datos de los programas, así como de inicializar la pila y otros registros [29, 34, 35].

Cuando un programa enlazado con librerías compartidas es ejecutado, el sistema operativo invoca el enlazador dinámico que escanea la lista de librerías enlazadas del binario y las busca en el sistema, habitualmente a partir de un fichero de configuración predefinido como `/etc/ld.so.preload`. Este proceso es conocido como **enlazado dinámico**. El proceso de enlazado dinámico es responsabilidad del sistema operativo.

Por otra parte, el enlazador dinámico proporciona a los programadores una interfaz para acceder a símbolos y poder cargar nuevas librerías en tiempo de ejecución. Esto es lo que se conoce como **carga dinámica** [30, 36].

A la hora de diseñar el sistema desarrollado se ha tenido en mente que este va a ser una librería que se interponga entre el programa y las llamadas a las funciones que hace. Aprovechando el funcionamiento del enlazador dinámico de Linux [37] y la posibilidad de llamar a las funciones originales [38], se ha planteado un diseño totalmente transparente para el programa original así como para los programadores. Es decir, los programadores no deberán preocuparse por el uso de funciones o librerías adicionales para poder hacer uso de libTTThwart; y únicamente serán los administradores de sistemas los encargados de especificar la librería en el fichero `/etc/ld.so.preload`.

El comportamiento de la librería cuando se ejecuta un binario se muestra en la Figura 3.2. En concreto, lo que sucede es lo siguiente:

- **Antes de la ejecución.** Al ejecutarse un binario en Linux, el enlazador dinámico consulta todas opciones especificadas en busca de librerías que precargar. Puesto que libTTThwart se especifica en `/etc/ld.so.preload`, su carga se produce antes que el resto. Una vez cargada, la librería se encarga de inyectar código para interceptar la llamada a ciertas funciones que se consideran como potencialmente inseguras.

- **Durante la ejecución.** El programa se ejecuta normalmente, pues el uso de libTTThwart es transparente. Antes de iniciar la ejecución del programa original, la librería se encarga de crear los directorios necesarios para los ficheros de logs así como los ficheros temporales. Los fichero de log se usan para registrar toda la actividad realizada por la librería. Los ficheros temporales se utilizan para crear enlaces a los objetos referenciados, en caso de que fuera necesario. Una vez hecho esto, cada vez que se llama a una función considerada parcialmente insegura, se ejecuta el código de libTTThwart que, a su vez, hace la llamada al código original.

- **Después de la ejecución.** Si a lo largo de la ejecución del programa no se han detectado intentos de explotar la vulnerabilidad, el programa finalizará como se espera. Por otra parte, si se detecta TOCTOU, el programa será abortado y

se le indicará al usuario un mensaje de advertencia, además de escribir los logs correspondientes.

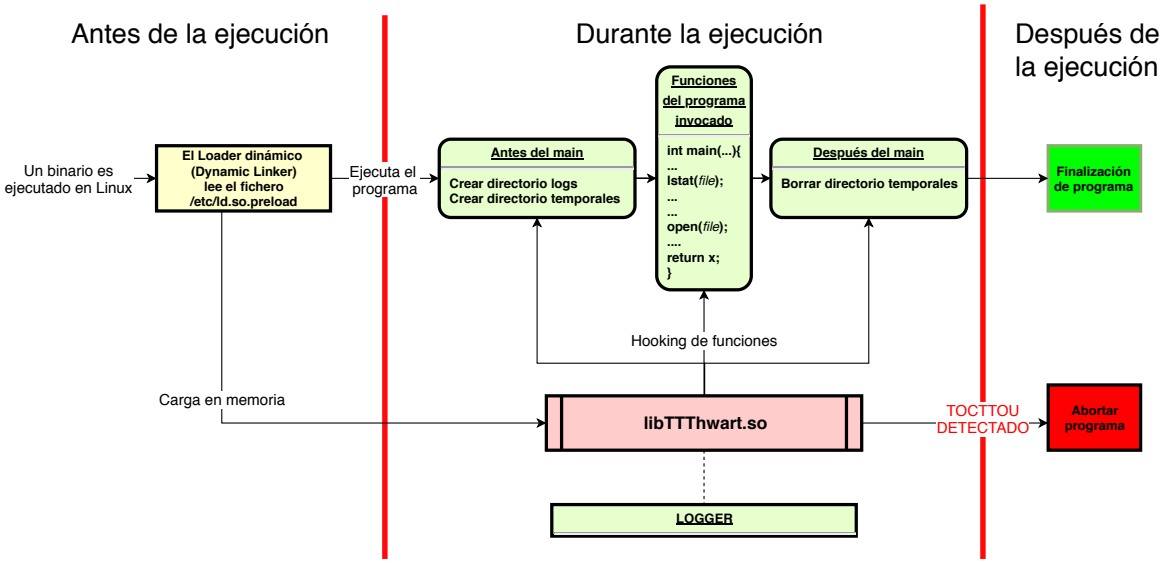

Figura 3.2. Diagrama de sistema de libTTThwart. (Fuente: propia)

Cuando se explota una vulnerabilidad TOCTOU mediante ataques basados en enlaces, algunos de los metadatos de los objetos del sistema de ficheros sufren modificaciones no autorizadas. Por tanto, para detectar precisamente esas modificaciones no autorizadas, la librería debe mantener en un registro la siguiente información acerca de los objetos utilizados por el programa:

- **Ruta.** Cuando se producen dos llamadas consecutivas referenciando un objeto mediante la misma ruta hay posibilidad de TOCTOU.
- **Inodo.** Mantener el inodo asociado a una determinada ruta es importante pues para explotar la vulnerabilidad es necesario modificar de alguna forma el objeto referenciado. Habitualmente, esta modificación consiste en borrar o cambiar de nombre del objeto original y crear otro con el mismo nombre. De esta forma, la ruta es la misma pero el objeto del sistema de ficheros que referencia es diferente. La creación de un nuevo objeto casi siempre implica un cambio de inodo. Haciendo uso de los inodos se obtiene un elemento exclusivo que hace referencia a un fichero concreto.

- **Identificación de dispositivo.** Es importante llevar registro acerca de la identificación del dispositivo en el que se encuentran los objetos referenciados. Como se ha mencionado anteriormente, un atacante podría manipular los objetos del sistema referenciados del tal forma que tengan la misma ruta y el mismo inodo, pero en dispositivos diferentes. De no ser por la identificación del dispositivo, no habría forma de diferenciar el cambio en el objeto.

- **Modo de fichero.** El modo de fichero es un metadato que permite obtener los permisos del objeto, así como el tipo de fichero. Este metadato informa, por ejemplo, si se trata de un directorio, de un fichero regular o de un fichero especial tipo socket. Cuando se modifica un fichero de tipo regular para convertirlo en un enlace simbólico, el cambio se verá reflejado en este metadato.

La propuesta para llevar a cabo este registro de metadatos se basa en el mantenimiento de una estructura de datos tipo vector dinámico en la que se llevarán a cabo operaciones de tipo *Create, Read, Update, Delete* (CRUD). La Figura 3.3 refleja el diseño de la clase `File Object Info`, que contiene los atributos referentes a cada objeto del sistema. Adicionalmente a los metadatos anteriormente mencionados, se mantiene registro de la ruta temporal del objeto. Como se menciona en la siguiente sección, esto sirve para mantener todos los ficheros temporales localizados. Por otra parte, la figura muestra el agregado `File Objects Info`, que será el encargado de mantener y gestionar la estructura de datos. En el momento de diseñar esta clase se ha tenido en cuenta la escalabilidad de la librería. Es por eso por lo que el primer parámetro que reciben los métodos de `File Objects Info` es siempre una referencia a la estructura de datos. Esto permite que en un futuro, si fuera necesario, coexistan varias estructuras de datos del mismo tipo y se use esta clase para gestionarlas y mantenerlas, a modo de interfaz.

Es importante, por otra parte, diferenciar los cambios legítimos y los cambios ilegítimos de los mencionados metadatos. Para conseguir esto, se ha sustituido el comportamiento de todas las funciones que permiten modificar un objeto del sistema de ficheros para que, además de realizar cambios en el sistema de ficheros, realicen esos mismos cambios en los elementos almacenados en la estructura dinámica. De esta forma, cuando un programa invoca `remove()` para borrar un fichero, se borrará también su información de la estructura.

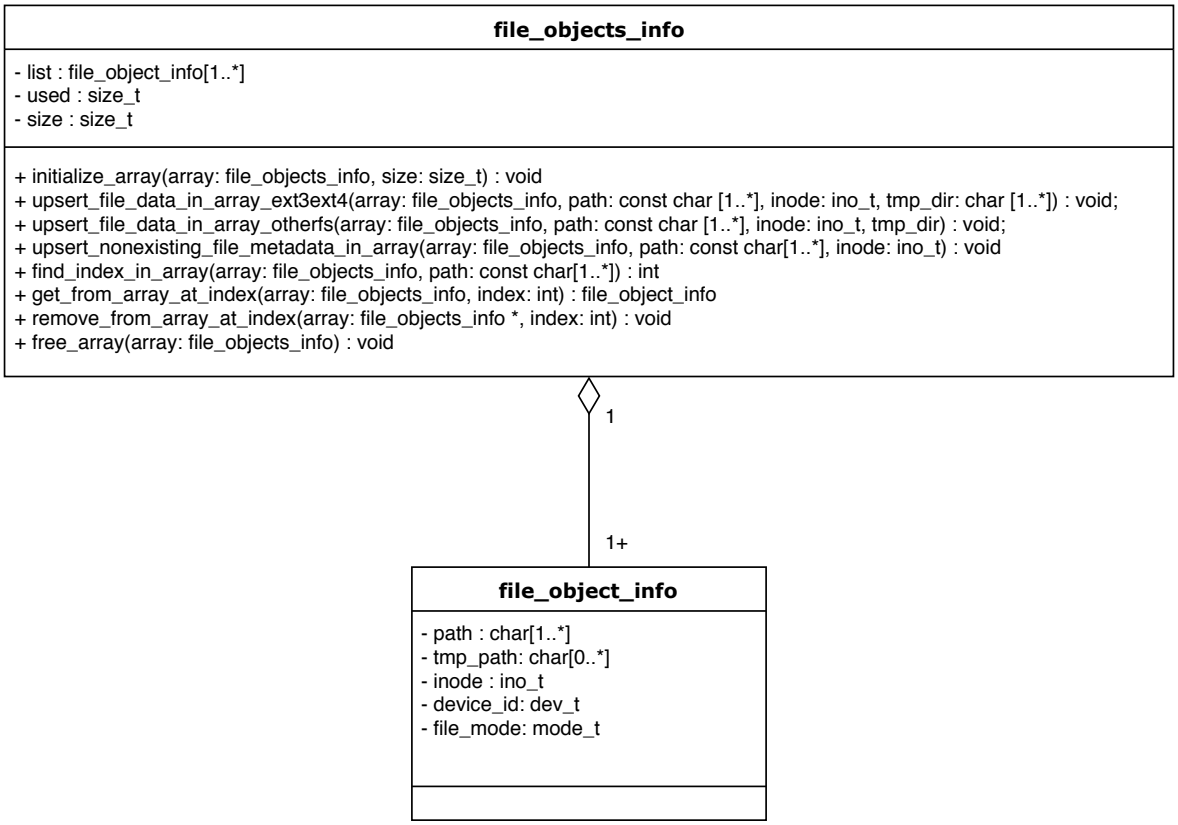

Figura 3.3. Diagrama de clases de `File Objects Info`. (Fuente: propia)

Otro de los elementos importantes que forman el sistema desarrollado es el `Logger`, ya que permite llevar el registro de todo lo que sucede durante la ejecución. El `Logger` es único por cada binario ejecutado y es accesible desde todos los puntos de la librería para plasmar en todo momento lo que está sucediendo. A la hora de diseñarlo, se ha contemplado su inicialización y su escritura por la salida estándar o bien en un determinado fichero. La Figura 3.4 muestra el diseño del `Logger`.

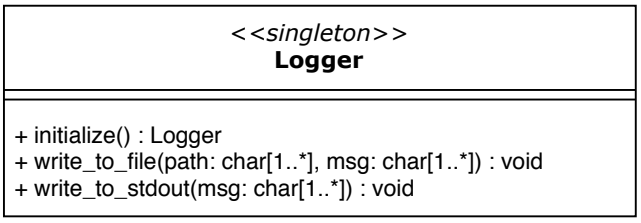

Figura 3.4. Diagrama de clases del `Logger`. (Fuente: propia)

3.2. Decisiones de implementación

La librería ha sido implementada íntegramente en C y ha sido desarrollada en Debian 9 (Stretch). El código ha sido compilado con GCC 6.3.0 20170516 (Debian 6.3.0-18+deb9u1). Para más información acerca del código, la instalación y las advertencias de uso de la librería, véase el Anexo B.

Como se ha mencionado anteriormente, la función principal de la librería es sobreescribir las funciones inseguras que posibilitan la explotabilidad de TOCTOU. Para llevar esto a cabo, se declaran nuevas funciones con la misma signatura que las originales y se añade la lógica necesaria para garantizar la protección frente a la explotación de TOCTOU. Este proceso se conoce como hooking de funciones. Así, pues, cuando se llama a una función hookeada se realizan las operaciones necesarias para obtener los metadatos mencionados, hacer las comprobaciones pertinentes y, si no hay indicios de TOCTOU, llamar a la función original.

Para obtener los metadatos de los objetos referenciados se consulta el sistema de ficheros a través de las funciones de la propia API. La Figura 3.5 refleja la obtención del inodo. Por otra parte, la llamada a la función original se consigue gracias al uso de `dlsym()` [38].

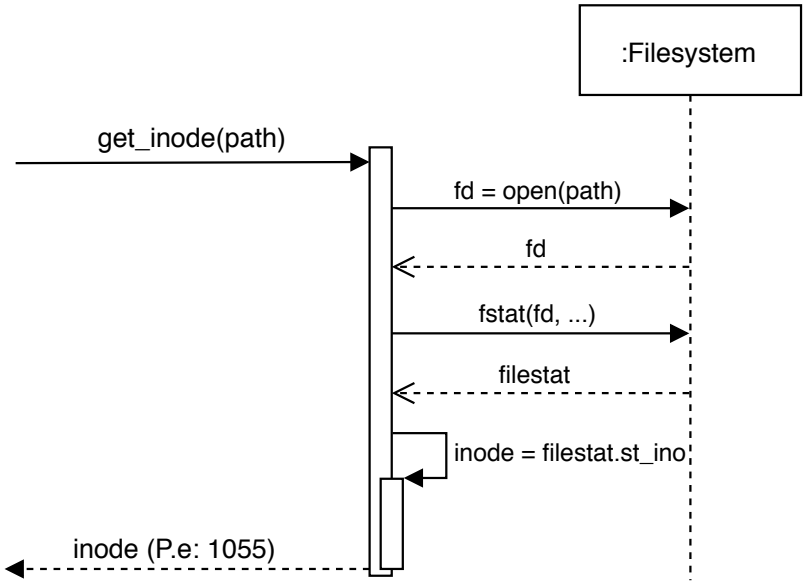

Figura 3.5. Interacción con el sistema de ficheros para la obtención de metadatos. (Fuente: propia)

A lo largo de toda la librería se invocan funciones que son propiamente hookeadas. Para evitar caer en redundancias cíclicas, se han declarado *wrappers* para cada función de la API de Linux que la librería utiliza y que a su vez hookea, encargados de simplemente hacer la llamada a la función original.

Nótese que libTTThwart hace uso, a su vez, de otras librerías. La más importante de ellas es la implementación estándar de C por parte de GNU, que contiene las funciones a las que libTTThwart hace hooking. La versión de GNU de la librería estándar de C es comúnmente llamada `GNU C Library` o, simplemente, `glibc` [39]. Su primera versión fue lanzada en 1987 y su versión más reciente, a fecha de escritura de este documento, data del 1 de agosto de 2019 [40]. Es una librería que contiene el código de las funciones principales de los sistemas GNU y Linux, y de muchos otros sistemas operativos cuyo núcleo está basado en Linux. La librería implementa las APIs fundamentales que permiten que el sistema operativo funcione como `open`, `read`, `wirte` y `malloc`, entre muchas otras.

Puesto que la vulnerabilidad se produce entre dos llamadas, es habitual hablar acerca de pares TOCTOU. Un par TOCTOU es una secuencia de dos funciones vulnerables. La primera de estas funciones es la que realiza una operación de comprobación, mientras que la segunda realiza una operación de uso. La librería no afronta el problema como una secuencia de llamadas vulnerables, sino simplemente estudia todas las llamadas de manera independiente. El conjunto de funciones que se han considerado en libTTThwart encuentra su base en el trabajo de Wei & Pu [11]. En él, los autores identifican todos los posibles pares TOCTOU en Linux. La identificación de pares TOCTOU hecha por los autores ha sido ampliada en el presente trabajo. La Tabla 3.1 muestra todas las funciones hookeadas por libTTThwart (se han destacado las ampliadas por este trabajo).

Las funciones se encuentran divididas en dos grandes grupos:

- **Funciones que establecen la condición.** Son aquellas funciones que leen el estado de un objeto del sistema de ficheros. Es decir, aquellas funciones que establecen una condición que más tarde en la ejecución se comprobará.
- **Funciones que establecen la condición y/u operan en base a esa condición.** Son todas las funciones que pueden tanto establecer la condición como

interactuar con el objeto del sistema de ficheros en base a una condición previamente establecida.

No se han detectado funciones cuyo riesgo radique en que exclusivamente operan en base a la condición. Esto es debido a que todas las funciones que interactúan con el sistema de ficheros lo hacen porque previamente se ha comprobado una condición o porque la propia interacción supone el establecimiento de la condición.

Tabla 3.1. Funciones hookeadas en libTTThwart. (Fuentes: [11] y propia)

Establecen condiciones	Establecen condiciones y/u operan en base a ellas			
stat	symlink	openat	chown	execlp
stat64	symlinkat	fopen	truncate	pathconf
lstat	link	fopen64	truncate64	mkdir
lstat64	linkat	freopen	utime	mkdirat
access	rename	popen	utimes	mount
unlink	renameat	mknod	execv	chdir
unlinkat	creat	mknodat	execve	
remove	creat64	mkfifo	execvp	
readlink	open	mkfifoat	execl	
readlinkat	open64	chmod	execle	

La lógica de la librería es diferente para las funciones que establecen condiciones que para las que establecen condiciones y/u operan en base a ellas. La Figura 3.6 resume mediante un diagrama UML de secuencia el comportamiento de la librería a la hora de hookear una función vulnerable. A continuación, se explican los pasos seguidos por la librería para cada tipo de función.

Cuando se llama a una función que establece condiciones, la librería realiza las siguientes acciones:

1. Se sanitiza y absolutiza la ruta recibida. El proceso de sanitización consiste en resolver todos los patrones como "." o "..". El proceso de absolutización consiste en obtener la ruta absoluta equivalente a la ruta recibida, en caso de que sea

relativa. De esta forma, libTTThwart trabaja con una única ruta por objeto, independientemente de la forma en la que el usuario la especifique.

2. Si el objeto que referencia la ruta existe en el sistema de ficheros, se obtienen los metadatos del objeto referenciado. Si la ruta referenciada ya existe en el vector dinámico de información de objetos, se comprueba si el inodo actual del objeto referenciado es diferente al del objeto almacenado en el vector. Si lo es, se actualiza el inodo del objeto del vector. Si la ruta referenciada no existe en el vector dinámico, se comprueba si el vector tiene espacio para un nuevo elemento. Si no lo tiene, se duplica su espacio actual. Se inserta en el vector un nuevo objeto con todos los metadatos obtenidos del objeto referenciado. Después, se continúa en el paso 4.

3. Si el objeto que referencia la ruta no existe en el sistema de ficheros, se comprueba si el vector dinámico tiene espacio para un nuevo elemento. Si no lo tiene, se duplica su espacio actual. Se inserta en el vector un nuevo objeto con la ruta especificada y el resto de metadatos inicializados a `NULL`. El inodo en este caso se establece con el valor 0.

4. Se llama a la función original.

Por otra parte, cuando se llama una función que establece condiciones u opera en base a ellas, el comportamiento es el siguiente:

1. Se sanitiza y absolutiza la ruta recibida, de manera similar al caso anterior.

2. Si el objeto que referencia la ruta existe en el sistema de ficheros, se obtienen los metadatos del objeto referenciado. Si la ruta referenciada no existe en el vector dinámico, se inserta. Si la ruta referenciada existe en el vector, se obtienen los metadatos del objeto referenciado y se comprueba su integridad. En el caso de que entre el objeto referenciado y el almacenado no coincidan la identificación de dipositivo, el modo de fichero o el inodo, se procede al paso 4. En caso contrario, se procede al paso 5.

3. Si el objeto referenciado no existe en el sistema de ficheros, se comprueba si el vector tiene espacio para un nuevo elemento. Si no lo tiene, se duplica su espacio

actual. Se inserta en el vector un nuevo objeto con la ruta especificada y el resto de metadatos inicializados a `NULL`. El inodo en este caso es 0. Se procede al paso 5.

4. TOCTOU DETECTADO. Se escribe en los logs, se informa al usuario y se aborta la ejecución programa.

5. Se llama a la función original.

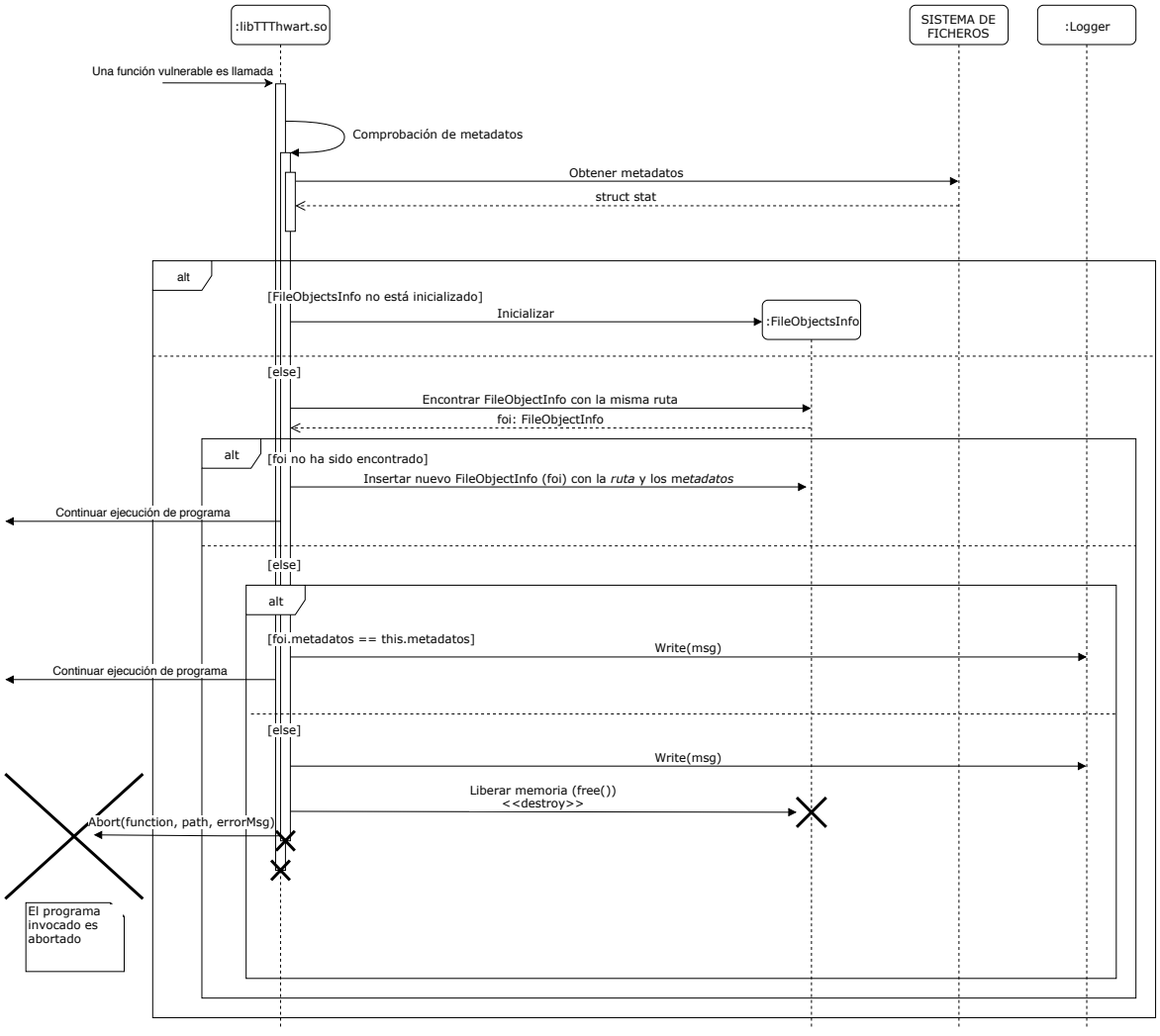

Figura 3.6. Diagrama UML de secuencia del hooking de funciones. (Fuente: propia)

La librería registra los metadatos incluso cuando se llama a una función vulnerable sobre un objeto que no existe. Esto se lleva a cabo por dos motivos: primero, de no hacerlo así se caería en el error de asumir que los programadores siempre escriben código robusto que considera los errores de todas las funciones; segundo, porque asumir eso implica que la librería, al igual que el código que intenta proteger, es vulnerable.

En el Código 3.1 se muestra un ejemplo de comprobación de errores indebida. El programador obtiene los metadatos de un fichero (línea 10) y, acto seguido, escribe en él (línea 15). El error está en que el programador asume que el fichero existe, ya que no comprueba el error devuelto por la función `stat()` (línea 10). Como es evidente, un atacante podría aprovecharse de esto pues las funciones `stat()` (línea 10) y `open()` (línea 12) forman una secuencia vulnerable a TOCTOU. Desde el punto de vista de libTTThwart, para defender al programa de la vulnerabilidad surgida a raíz de este error, es necesario registrar que se ha invocado una llamada vulnerable a un objeto que no existía. De esta forma, cuando se invocara la segunda llamada `open()`, la librería comprobaría los metadatos y detectaría que donde antes había un objeto inexistente ahora existe uno creado por un proceso externo.

La librería también hace uso de los inodos como parte identificativa de los objetos del sistema. Sin embargo, un inodo no siempre es único por objeto, ya que depende del sistema de ficheros subyacente. Se han llevado a cabo pruebas en algunos de los sistemas de ficheros más importantes del universo Unix [41] para determinar cuáles de ellos reusan inodos (cuando ya no hay referencias al objeto) y cuáles no.

Los resultados se muestran en la Tabla 3.2. Esto significa que no se puede asumir un cambio de inodo al borrar y volver a crear un fichero. Para paliar este problema, se comprueba el sistema de ficheros del objeto referenciado cada vez que se llama a una de las funciones hookeadas. Si el objeto referenciado se encuentra en un sistema de ficheros que reutiliza inodos se crea un enlace temporal al fichero referenciado. Esto se hace para garantizar que aunque el objeto original referenciado por la función fuese borrado, el inodo no estaría libre pues el enlace creado por libTTThwart seguiría apuntando a él. El nombre de los enlaces temporales es una secuencia de 25 caracteres aleatoria. Al finalizar el programa vulnerable, todos estos enlaces temporales se eliminan.

Otro elemento de la librería es el mecanismo de logging. Para el logging se hace uso de `Zlog`, una librería de logging que se ejecuta en memoria. Es decir, reserva un buffer en memoria y lo libera cada vez que se llena o cada vez que el programador lo indica. El código de `Zlog` está disponible en `https://github.com/zma/zlog`. Se han añadido ligeras mejoras al código original de `Zlog` como, por ejemplo, la posibilidad de escribir logs en `stderr`, la creación de dos niveles de log, `INFO` y `DEBUG`, o la modificación del formato de fecha para que sea más legible. Estas modificaciones han sido aceptadas

Tabla 3.2. Reutilización de inodos en diferentes sistemas de ficheros. (Fuente: propia)

Sistema de ficheros	Reutilización de inodos
BTRFS	No
EXT2	Sí
EXT3	Sí
EXT4	Sí
FAT16	No
FAT32	No
NTFS	No
HFS+	No
JFS	No
NILFS2	Sí
REISERFS	Sí
XFS	Sí
RAMFS	No
TMPFS	No

por el autor principal de la librería y se ven reflejadas en el repositorio principal como contribuciones al proyecto.

Por último, es importante mencionar que cuando se hookean las funciones, el código de la librería sólo se ejecuta en caso de que el `EUID` sea distinto del `RUID` o el `EGID` (*Effective Group ID*) sea distinto de `RGID` (*Real Group ID*). En caso contrario, simplemente se llama a la función original. Con esto se consigue no sobrecargar de forma innecesaria los programas cuando estos se ejecutan de tal forma que no podría producirse TOCTOU.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    struct stat statbuf;
    stat("temp.file", &statbuf);
    int fd;
    if((fd = open("temp.file", O_APPEND)) == -1){
        perror("Open error: ");
    } else {
        write_to_file_from_stdin();
    }
    return 0;
}
```

Código 3.1. Ejemplo de mala comprobación de errores. (Fuente: propia)

3.2.1. Estándares de codificación segura

Para complementar el desarrollo de la librería se ha hecho uso de SonarCloud, un analizador estático de código fuente basado en SonarQube [42]. SonarCloud es una plataforma de código abierto para la inspección continua de la calidad del código, centrada en detectar bugs, vulnerabilidades y otros fallos que puedan surgir en el desarrollo del código. Su funcionamiento es sencillo: interpreta el código fuente de todos los ficheros y, en base al lenguaje de programación y a normas predefinidas por la comunidad para ese lenguaje, detecta los diferentes problemas que puede haber. Además, proporciona diferentes métricas como número de líneas de código totales, número de comentarios, número de declaraciones, etc.

Según el análisis de SonarCloud, actualmente libTTThwart no tiene vulnerabilidades en el código, tiene 1.3 % de código repetido repartido en dos bloques y un total de 4684 líneas. Los resultados completos de los análisis realizados por SonarCloud son

públicos y se pueden acceder a través del siguiente enlace: `https://sonarcloud.io/dashboard?id=RazviOverflow_libTTThwart`.

3.3. Limitaciones

La librería está sujeta a ciertas limitaciones. Algunas de estas limitaciones se pueden superar avanzando en el desarrollo de libTTThwart, mientras que otras son intrínsecas a la naturaleza no atómica de la API de Linux. Para superar estas últimas, haría falta un cambio total en el núcleo de Linux.

Esta última limitación es la más importante a la que está sometida libTTThwart. Esto es, la propia librería también puede ser víctima de condiciones de carrera. En el presente trabajo se ha identificado el momento crítico de la librería donde esta puede ser víctima, a su vez, de TOCTOU. Los momentos críticos de la librería son:

- **Entre la invocación de la función por parte de cualquier programa y la obtención o comprobación de los metadatos por parte de libTTThwart**. Esto podría llegar a suponer que en la estructura de datos se inserten directamente los metadatos del objeto del sistema de ficheros ya manipulado, lo cual implicaría que cuando se invoque a la siguiente llamada que referencia a dicho objeto, la librería considere como válidos los metadatos del objeto manipulado.

- **Entre la finalización del registro o comprobación de metadatos por parte de libTTThwart y la llamada a la función original por parte del programa**. De igual manera, el sistema de ficheros puede manipularse entre estos momentos indicados.

En la Figura 3.7 se ha plasmado el orden de eventos para que uno de estos ataques se produzca contra libTTThwart, además de dibujar las franjas entre eventos donde la librería es vulnerable. Las ventanas de vulnerabilidad identificadas son:

- **pre-check**. Surge entre el inicio de la llamada vulnerable y la obtención de los metadatos por parte de libTTThwart. Durante este intervalo de tiempo el sistema de ficheros podría ser alterado.

- **post-check**. Existe entre la obtención de los metadatos por parte de libTTThwart y la llamada a la función original. Si se altera el sistema de ficheros en este breve

intervalo de tiempo, la función original se invocará sobre el objeto manipulado, aunque en el vector dinámico de datos la información sea correcta.

- **pre-use**. Sucede entre la llamada a la función original y la obtención de los metadatos del objeto referenciado.
- **post-use**. Se da entre la comprobación de los metadatos y la llamada original.

Para tratar de paliar este problema se han pensando diferentes soluciones. Una de ellas consistía en eliminar el bit SetUID (+s) del binario cuando se detectase TOCTOU, de esta forma la vulnerabilidad ya no podría existir. El bit SetUID es un permiso en sistemas Unix que permite que los usuarios ejecuten el binario con los permisos de su propietario. Esta solución se descartó pues es demasiado drástica e intrusiva ya que afecta a todos los usuarios del sistema. Podría darse el caso en que un programa tuviera que ser necesariamente ejecutado con el bit SetUID activado y, en ese caso, la librería impediría la correcta ejecución. Otra solución que se ha considerado consiste en hacer uso de software adicional que complemente la funcionalidad de libTTThwart como podría ser Graylog o Elastic Stack. Esta clase de aplicaciones permiten, entre muchas otras cosas, llevar a cabo monitorización en tiempo real de diferentes logs, lo cual ayudaría a los administradores de sistemas a detectar en tiempo real cualquier intento de TOCTOU. Sin embargo, supone depender de aplicaciones de terceros.

Otra limitación de la librería tiene que ver con la creación de nuevos procesos mediante el uso de funciones como `fork()` y `exec()`. Cuando se invoca `fork()` para crear un nuevo proceso, el sistema duplica la imagen del proceso actual, con todos los datos que haya en ese momento. Esto implica que la estructura de datos que lleva el registro de los metadatos es copiada en el nuevo proceso. Si este nuevo proceso invoca una de las funciones hookeadas por libTTThwart y, por lo tanto, modifica la estructura, esas modificaciones sólo se reflejarán en su copia de la estructura de datos, no en la del proceso original. Esto supone que modificaciones en los objetos del sistema de ficheros legítimos podrían no tratarse como tal. De igual manera, cuando se crea un nuevo proceso con alguna de las funciones de la familia `exec()`, este reemplaza el proceso actual en memoria. Es decir, lo sustituye totalmente con nuevos datos y nuevas instrucciones. Como es evidente, es imposible controlar las llamadas vulnerables de esta forma ya que todos los datos registrados hasta ese momento serían borrados.

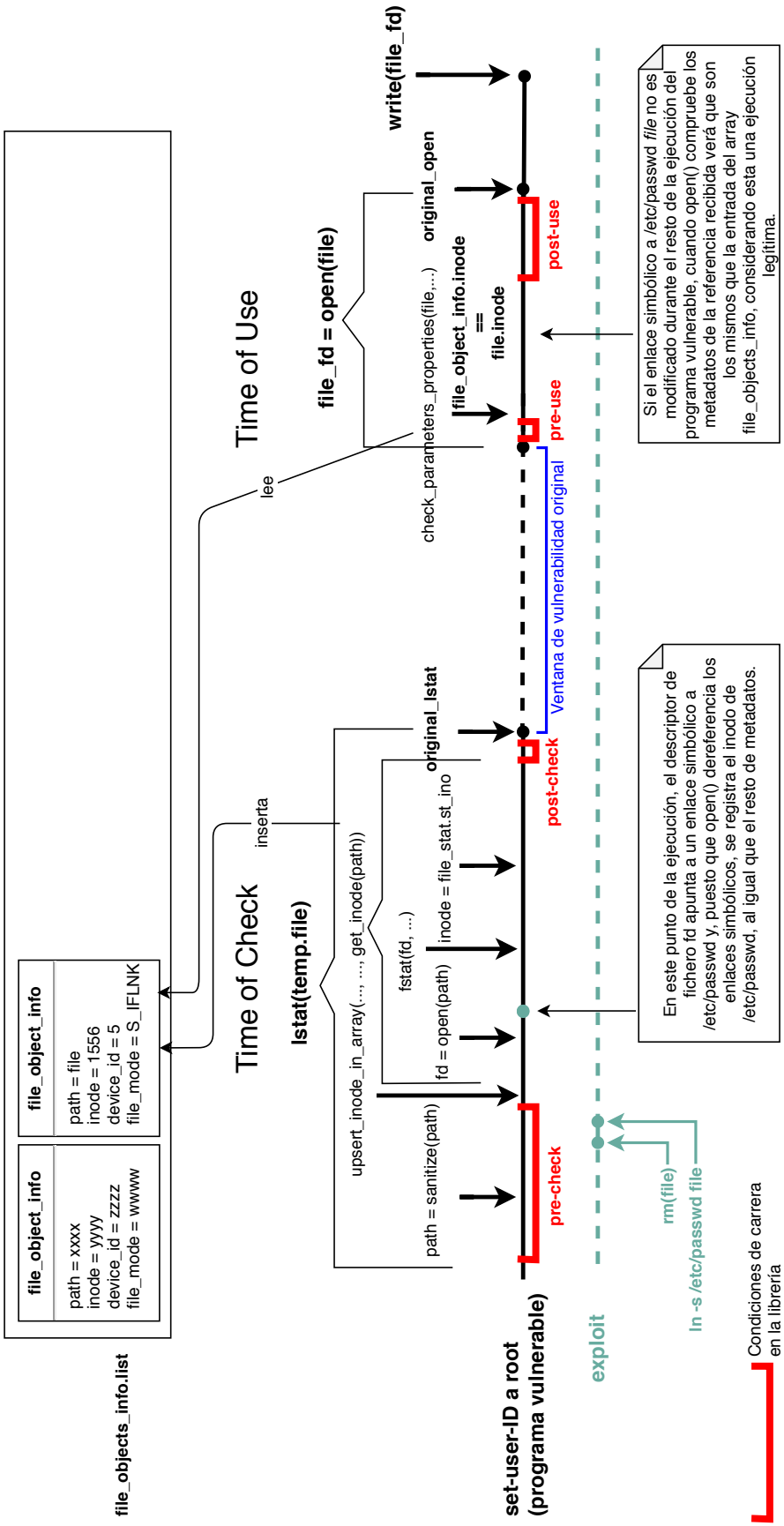

Figura 3.7. Condiciones de carrera intrínsecas. (Fuente: propia)

Por último, otra de las limitaciones a las que está sometida libTTThwart, como es evidente, es el uso de librerías dinámicamente enlazadas. La librería libTTThwart sólo entra en funcionamiento cuando las librerías han de cargarse con el cargador/enlazador dinámico y, por consiguiente, se produce la lectura del fichero `/etc/ld.so.preload`. Si el programa vulnerable tiene las librerías enlazadas estáticamente, el uso de libTTThwart no será posible.

3.4. Enfoque inicial

En las fases iniciales del proyecto, se estudió un planteamiento diferente para la detección de la vulnerabilidad TOCTOU mediante análisis estático de código fuente. Para este análisis se pretendía hacer uso de las Redes de Petri (RdP), un formalismo matemático de grafos bipartitos para la descripción formal de sistemas que se caracterizan por concurrencia, sincronización, exclusión mutua y conflicto. Las RdP permiten representar las propiedades estáticas y dinámicas de un sistema real [43].

La solución propuesta consistía en hacer uso de las RdP para representar la vulnerabilidad y obtener, a partir de esa representación, algunos rasgos característicos de la misma. Una vez conseguida la caracterización de la vulnerabilidad se podría determinar si la representación se correspondía con código realmente vulnerable en base a sus propiedades y la mencionada caracterización.

La aplicación de las RdP en este contexto no se ha podido llevar a cabo debido a los problemas que surgen al realizar el análisis e interpretación del código fuente, como por ejemplo el problema de aliasing de punteros (la existencia de varias referencias a la misma zona de memoria) [44–46]. A pesar de considerar diferentes estructuras de representación de código fuente como AST *(Abstract Syntax Tree)* o CFG *(Control Flow Graph)* y probar diferentes herramientas que permiten obtener estas representaciones, elementos imprescindibles para la detección de TOCTOU como los parámetros en las llamadas a funciones no se conseguían extraer satisfactoriamente. Esto suponía una carga de trabajo excesiva en el estudio y mejora de aplicaciones para la obtención de una representación óptima, quitando tiempo al estudio de la vulnerabilidad mediante RdP.

4. Experimentos y Evaluación

En el presente capítulo se detallan los experimentos llevados a cabo para comprobar el funcionamiento de la librería y las pruebas de rendimiento para determinar el impacto de la misma.

4.1. Prueba de concepto

Para comprobar el comportamiento de la librería se han llevado a cabo diferentes pruebas de concepto con código vulnerable. Se han probado en dos escenarios diferentes: nativo y con libTTThwart activada en el sistema. El resultado ideal es explotar la vulnerabilidad con éxito en nativo y fallar con la librería.

El Código 4.1 es vulnerable al par TOCTOU <`lstat`, `open`>. El momento del *check* se produce con la llamada a `lstat()` (línea 13) y el momento del *use* sucede con la llamada a `open()` (línea 22). El programa escribe el texto recogido por la entrada estándar en un fichero pasado como argumento (línea 27). La recogida de datos se realiza por entrada estándar entre ambas llamadas vulnerables (línea 20). Este código de prueba de concepto no es muy diferente a trozos de código que se puede encontrar en aplicaciones reales.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <string.h>
6  #include <unistd.h>
7
8  int process_filename(char *filename)
9  {
10     struct stat aux_stat;
11     char buffer[1024];
12
13     if((lstat(filename, &aux_stat) == 0) && !S_ISLNK(aux_stat.st_mode))
14     {
15         printf("[+] Opening file %s...\n", filename);
16         fflush(stdout);
```

```
17          int fd, nb;
18
19          printf("Input to be appended: ");
20          fgets(buffer, sizeof(buffer), stdin);
21
22          if((fd = open(filename, O_RDWR | O_APPEND)) == -1){
23              printf("[!] Error while trying to open %s.\n", filename);
24              return -1;
25          }
26
27          nb = write(fd, buffer, strlen(buffer));
28          printf("[+] Done! %d bytes written to %s\n", nb, filename);
29          return 0;
30      }else
31          printf("[-] ERROR: %s is a symlink or does not exist. Exiting...\
    n", filename);
32
33      return 1;
34  }
35
36  int main(int argc, char * argv[])
37  {
38      if(argc != 2){
39          fprintf(stderr, "usage: %s filename\n", argv[0]);
40          exit(1);
41      }
42
43      return process_filename(argv[1]);
44  }
```

Código 4.1. Código vulnerable a TOCTOU. (Fuente: propia)

Supóngase que este programa vulnerable es un programa que pertenece a root pero es ejecutable por todo el mundo (como lo son la mayoría de utilidades de Linux) y que, además, tiene el bit SetUID (+s) activado. En el mismo directorio existen dos ficheros, uno perteneciente al atacante y otro a root. El atacante, como es evidente, no tiene permisos para escribir en el fichero de root. En este caso la explotación manual es sen-

Figura 4.1. TOCTOU explotado. (Fuente: propia)

cilla y un atacante podría aprovecharse para escribir en el fichero que pertenece a root. La Tabla 4.1 muestra cómo un atacante se puede aprovechar de esta vulnerabilidad.

Tabla 4.1. Explotación manual de código vulnerable. (Fuente: [4, 15])

Programa vulnerable	Atacante Terminal 1	Atacante Terminal 2
	`./vulnerable user.file`	
`lstat(user.file, ...)`		
`fgets(buffer, ...)`		
		`rm user.file`
		`ln -s r00t.file user.file`
	User wrote in r00t	
`open(user.file, ...)`		
`write(...)`		

El fichero perteneciente a root y sobre el cual el atacante no tiene permisos ha sido modificado gracias a las acciones realizadas en paralelo a la ejecución (en otra terminal). El fichero `user.file` ha sido borrado y se ha creado un enlace simbólico a `r00t.file` con el mismo nombre (Atacante Terminal 2). Puesto que las funciones `lstat()` y `open()` no se ejecutan de forma atómica, el programa ha sido engañado gracias a la inconsistencia del sistema de ficheros. La Figura 4.1 muestra el contenido del fichero tras el ataque.

Si se trata de ejecutar exactamente el mismo ataque partiendo del mismo escenario que en el caso anterior pero con libTTThwart activada, aparece un mensaje de error informando de la posible detección de una TOCTOU y se termina el proceso vulnerable, como se muestra en la Figura 4.2.

Así pues, la explotación ha sido frustrada ya que la librería ha detectado cambios en los metadatos del objeto del sistema de ficheros realizados por otro proceso ajeno. A la hora de abortar y detectar la vulnerabilidad la librería imprime por la salida estándar

```
Razvan:~/Desktop/TFM_TOCTTOU/ld.so.preload/POC$ ./vulnerable user.file
Input to be appended: User wrote in r00t
[+] Opening file user.file...
[+][!] WARNING! TOCTTOU DETECTED!. [!][+]
[#] PROGRAM vulnerable ABORTED [#]
[#] Check logs for more info [#]
[!] LOGIFLE: /home/alumno/libTTThwart/vulnerable/7204_2019-08-18_12:21:09.log [!]
```

Figura 4.2. TOCTOU frustrado. (Fuente: propia)

```
1   [12:21:35.571301s] [+][!] WARNING! TOCTTOU DETECTED! [+][!]
2   Inode of </absolute/path/to/user.file> has changed since it was previously
    invoked. Threat detected when invoking <open> function. Inode was <4266>
    and now it is <19881>.
3   [#] PROGRAM vulnerable ABORTED [#]
```

Figura 4.3. Entrada de log de libTTThwart. (Fuente: propia)

la información pertinente así como la ruta al fichero de log que se corresponde con la ejecución abortada. En los logs se escribe la hora en la que se ha producido el ataque (la fecha se indica en el nombre del fichero), así como la ruta del fichero referenciado que ha sido modificado sin permiso y el nombre del proceso que ha sido abortado. Un ejemplo de entrada de log se muestra en la Figura 4.3.

Como se detalla en la Sección 3.3 Limitaciones, podría darse el caso en el que la vulnerabilidad se explote con la librería activa si el atacante es capaz de ganar las condiciones de carrera pertinentes y modifica los metadatos de tal forma que la librería los considere correctos.

4.2. Evaluación del rendimiento

A la hora de evaluar el impacto que libTTThwart puede tener en el rendimiento del sistema se han realizado dos pruebas. La primera de ellas, conocida como microbenchmarking, consiste en medir el tiempo que tardan en ejecutarse ciertas instrucciones de un programa. La segunda prueba realizada ha consistido en medir el rendimiento general del procesador con un *suite* de benchmarking estandarizado, SPEC CPU. Todas las pruebas se han realizado sobre los mismos componentes y en el mismo entorno:

- **Sistema operativo**: Debian 10 (Buster) 64-bit.
- **Procesador**: Intel i5-8250U (4 núcleos físicos, 8 hilos) de frecuencia base 1.60GHZ.
- **Memoria RAM**: 8GB DDR4 2400MHz en formato SODIMM.

- **Disco duro**: 256GB SSD conectado vía interfaz NVMe (velocidad de escritura y lectura ∼1.8GiB/s).

4.2.1. Microbenchmark

Los microbenchmarks se utilizan para comprobar el rendimiento de ciertas instrucciones de un programa. Para realizar las pruebas de microbenchmarking de libTTThwart se ha tomado como referencia el microbenchmark realizado en [47] con ligeras modificaciones. Se han definido 4 test de microbenchmarking: *access*, *open/close*, *largo* y *muy largo*. El código utilizado para hacer estas pruebas se detalla en el Anexo D.

Para realizar las pruebas cada microbenchmark se ha lanzado 50000 veces con y sin libTTThwart en diferentes sistemas de ficheros. Los sistemas de ficheros probados han sido BTRFS, EXT4, RAMFS y TMPFS dada su importancia en universo Unix [41]. A diferencia de BTRFS y EXT4, los sistemas de ficheros TMPFS y RAMFS son volátiles (es decir, residen en memoria).

Los resultados del microbenchmarking muestran que, inevitablemente, la librería tiene impacto en el tiempo de ejecución de las instrucciones de los programas. Este impacto varía en función del sistema de ficheros y de la longitud del test. Cuando se ha realizado el microbenchmark en nativo, el sistema de ficheros más eficiente ha sido TMPFS por parte de los volátiles y BTRFS por parte de los no volátiles. Por otra parte, cuando se han realizado las pruebas con la librería activada, el sistema de ficheros volátil más eficiente ha sido, de nuevo, TMPFS y el no volátil más eficiente ha sido EXT4. En base al cálculo del tiempo total, el sistema de ficheros que menos ha sufrido el impacto de la librería ha sido EXT4 (1.89x) y el que más ha visto su rendimiento afectado ha sido TMPFS (3.66x). Los resultados del proceso de microbenchmarking se pueden apreciar en la Tabla 4.2.

4.2.2. SPEC CPU 2006

Para evaluar de forma más genérica el impacto de la librería en el rendimiento del sistema, se ha utilizado un benchmark estándar que mide el rendimiento del procesador. Este benchmark, llamado SPEC CPU2006 [20], ha sido desarrollado por SPEC con el propósito de proporcionar a la industria una referencia para medir el rendimiento de sistemas informáticos. En el presente trabajo se ha ejecutado el benchmark 8 veces, 4

Tabla 4.2. Resultados microbenchmarking. (Fuente: propia)

	Sistema de ficheros			
Benchmark	BTRFS		EXT4	
	Nativo (s)	Librería (s)	Nativo (s)	Librería (s)
access	0.342172	2.52359 (7.37x)	0.352382	2.22854 (6.32x)
open/close	0.422826	2.53462 (5.99x)	0.448327	2.42796 (5.42x)
largo	1.77522	4.05748 (2.28x)	1.68429	3.55952 (2.11x)
muy_largo	5.37345	8.47677 (1.58x)	6.23881	8.30093 (1.33x)
Total	7.913668	17.60246 (2.22x)	8.723809	16.51695 (1.89x)

Benchmark	TMPFS		RAMFS	
	Nativo (s)	Librería (s)	Nativo (s)	Librería (s)
access	0.331838	2.16216 (6.51x)	0.334351	2.15347 (6.44x)
open/close	0.401499	2.4144 (6.01x)	0.434642	2.39326 (5.5x)
largo	0.723228	2.80886 (3.88x)	0.966573	3.00171 (3.1x)
muy_largo	1.56037	3.66102 (2.35x)	1.71061	3.82698 (2.24x)
Total	3.016935	11.04644 (3.66x)	3.446176	11.37542 (3.3x)

Tabla 4.3. Resultados benchmark SPEC CPU2006. (Fuente: propia)

	Nativo				**Librería**			
Ejecución #	1	2	3	4	1	2	3	4
Puntuación	29.9	30.3	29.8	29.5	30.5	29.9	30.1	30.01
Tiempo (s)	13585	13454	13729	13748	14021	14279	14099	13875
Puntuación media	29.875				30.15 (1.01x)			
Tiempo medio (s)	13628.25				14068.5 (1.03x)			
Tiempo total (s)	54513				56274 (1.03x)			

sin librería y 4 con librería, calculando posteriormente la media. Todas las ejecuciones se han realizado en el sistema de ficheros BTRFS y con el hardware anteriormente mencionado. Los resultados de ejecutar el benchmark son públicamente accesibles a través del siguiente enlace: `https://github.com/RazviOverflow/libTTThwart/tree/master/SPEC%20CPU%202006%20Results`.

Con el propósito de conseguir resultados reales entre cada ejecución del benchmark se ha reiniciado el sistema para eliminar la cache. En la Tabla 4.3 se pueden ver los resultados obtenidos.

SPEC CPU2006 mide el rendimiento de la CPU en cuanto a potencia de cálculo con números enteros y reales, mientras que libTTThwart intercepta e incide en las operaciones E/S con el sistema de ficheros. Esto quiere decir que estando libTTThwart activa en el sistema, las operaciones aritméticas no se verán afectadas. Esto se aprecia en la puntuación media obtenida. Mientras que las ejecuciones nativas tienen una puntuación media de 29.875, con libTTThwart se obtiene una puntuación media de 30.15, un 0.9 % más. Esto no significa que el hecho de activar libTTThwart en el sistema hace que la CPU rinda mejor, sino que en los resultados finales intervienen diferentes factores que no se pueden controlar.

Adicionalmente se han medido los tiempos de cada ejecución y se aprecia que, de media, el benchmark es un 3.23 % más lento cuando libTTThwart está activada.

Un desglose más detallado de los resultados de SPEC CPU2006 se puede consultar en el Anexo C.

5. Trabajo relacionado

La vulnerabilidad TOCTOU es una amenaza para la seguridad cuyas primeras referencias se remontan a hace más de 40 años [5–7]. Aún a día de hoy sigue siendo un grave problema pues dada su naturaleza no determinista, no existe una solución definitiva. Sin embargo, a lo largo de la historia se ha invertido mucho esfuerzo en tratar de paliar sus efectos. En el presente capítulo se hace un repaso a las propuestas cuyo objetivo era detectar e impedir la vulnerabilidad TOCTOU. Estas propuestas se han dividido en dos categorías, modelos teóricos y modelos con implementación.

5.1. Modelos teóricos

Ko & Redmond presentaron en el año 2002 una metodología para la detección de intrusiones basada en el concepto de no interferencia cuyo propósito es detectar los ataques de condición de carrera [48]. Esta metodología consiste en monitorizar las llamadas al sistema en tiempo de ejecución y comprobar si la propiedad conmutativa se mantiene. Esto es, las operaciones del programa vulnerable y con más privilegios deben tener el mismo resultado independientemente del orden de eventos respecto a las operaciones del programa atacante y con menos privilegios.

En 2004, Dean & Hu presentaron un mecanismo de defensa contra el par TOCTOU <`access`, `open`> llamado $k-$Race [49]. La defensa que propusieron consise en añadir múltiples pares <`access`, `open`> al par original, lo que ellos llamaron rondas de refuerzo. La idea detrás de $k-$Race es que para conseguir explotar la vulnerabilidad, el atacante debe ganar todas las condiciones de carrera. El hecho de tener que ganar k condiciones de carrera hace que la probabilidad de éxito del ataque se reduzca drásticamente.

En el año 2005, Borisov et al. presentaron el llamado ataque laberinto (*Maze Attack*) [50] con el que se consigue explotar $k-$Race. Este ataque consiste, de forma resumida, en crear múltiples cadenas de 16 directorios donde el último nodo de una cadena apunta al primero de la siguiente. Este ataque es satisfactorio cuando los directorios de las cadenas no se encuentran en la cache del sistema. Al no estar en la cache, se obliga al sistema a realizar accesos a disco para comprobar el contenido de

los mismos. Estas operaciones E/S son costosas y permiten ganar tiempo. Este tipo de ataque fue considerado más tarde como una forma genérica de explotar TOCTOU.

Para contrarrestar los ataques laberinto presentados por Borisov et al., en 2008 una nueva forma de $k-$Race fue presentada por Tsafrir et al. [16]. En su trabajo ellos la denominan *column-oriented* $k-$Race. La diferencia con la original es que en lugar de recorrer toda la ruta por cada invocación de par TOCTOU (por filas), la nueva forma de $k-$Race recorre la ruta elemento por elemento, iterando cada elemento específico k veces.

Más tarde, en 2009, Cai et al. presentaron una técnica para derrotar tanto el trabajo de Tsafrir et al., como el de Tsyrklevich & Yee. Esta técnica consistía en utilizar lo que los autores llaman ataques de complejidad algorítmica a la resolución de nombres. Este ataque se basa en el comportamiento del núcleo del sistema operativo cada vez que debe obtener el inodo de una determinada ruta. El núcleo siempre busca el nombre en su cache interna. Si no lo encuentra, accede a disco para obtener la información. Cualquier llamada a una función que tome como argumento una ruta se comporta de la misma forma. Los autores determinan que si consiguen hacer que la búsqueda en cache sea lenta, las llamadas serán lentas [51].

En 2012 Yang et al presentaron un estudio exhaustivo acerca de los ataques de concurrencia y las implicaciones de seguridad que conllevan los errores en la concurrencia [52]. Concluyeron que el tamaño de la ventana de vulnerabilidad afecta gravemente la explotabilidad de la vulnerabilidad, los errores de concurrencia en las API hace que sean particularmente propensas a ataques de concurrencia y los ataques de concurrencia son mucho más que TOCTOU. De hecho, para explotar una vulnerabilidad TOCTOU hace falta que los procesos se ejecuten de manera concurrente pero estos pueden ser secuenciales, es decir, realizar todas las tareas de forma lineal en un sólo hilo de ejecución.

5.2. Modelos con implementación

Los primeros estudios exhaustivos acerca de la vulnerabilidad se llevaron a cabo en la década de los 90 gracias a Bishop & Dilger [14, 15]. En estos trabajos se propuso una herramienta de análisis estático de código fuente que detectaba pares TOCTOU

gracias al reconocimiento de patrones. El analizador estático interpreta código fuente escrito en C y construye grafos de control de dependencias así como grafos del flujo de datos. Posteriormente, una persona debe inspeccionar estos resultados y decidir si realmente la vulnerabilidad podría producirse.

En 2001, Cowan et al. presentaron RaceGuard [53], una mejora del núcleo de Linux que detecta e impide intentos de explotación de condiciones de carrera en la creación de ficheros temporales. Esto se lleva a cabo detectando cambios en el entorno entre el momento en el que el programa comprueba la existencia del fichero y el momento en el que intenta crearlo.

En el año 2003, Tsyrklevich & Yee [4] estudiaron las condiciones de carrera en los accesos al sistema de ficheros tratando de entender bajo qué circunstancias podían ocurrir. El resultado de su trabajo fue el desarrollo de un módulo para el núcleo del sistema operativo OpenBSD. Este módulo es el encargado de interceptar diferentes llamadas al sistema, principalmente llamadas al sistema de ficheros que reciben una ruta como parámetro. La idea de su sistema es que, en base a unas políticas previamente definidas, no puede haber determinadas operaciones sobre el mismo inodo cuando no pertenecen al mismo proceso. Por ejemplo, no puede haber una llamada a `access()` seguida de otra llamada a `unlink()` desde otro proceso. Lo que el módulo haría en este caso sería suspender el segundo proceso hasta que el primero termine, evitando de esta forma que la llamada a `unlink()` tenga algún impacto sobre el primer proceso.

También en 2003, Goyal et al. presentaron un enfoque unificado para detectar periodos de vulnerabilidad en las asociaciones que el sistema operativo hace entre el objeto del sistema de ficheros y su referencia en sistemas Unix [54]. La solución propuesta informa al administrador del sistema si el programa es vulnerable analizando la traza de las ejecuciones concurrentes del programa vulnerable y el programa atacante. Su propuesta analiza el conjunto de objetos del sistema de ficheros compartidos entre la aplicación vulnerable y la atacante, así como las acciones llevadas a cabo por ambos programas. Los autores definen en el trabajo un conjunto de reglas en las cuales se basa el algoritmo de detección.

Park et al. presentaron en 2004 un mecanismo basado en un punto común por el que pasan todas las aplicaciones a la hora de realizar llamadas vulnerables [55]. El sistema propuesto, llamado RPS (*Race-attack Prevention System*), consiste en extender

el comportamiento del monitor de referencias a nivel de núcleo. Puesto que el monitor de referencias analiza cada una de las llamadas realizadas, es capaz de detectar condiciones de carrera. Esto se lleva a cabo implementando un módulo para el núcleo del sistema opeartivo que intercepta diferentes llamadas al sistema y realiza comprobaciones sobre el inodo.

Wei & Pu [8] en 2005 crearon el llamado modelo CUU que define pares de llamadas vulnerables. En base a este modelo definieron un *framework* de detección de vulnerabilidades TOCTOU en aplicaciones sensibles, es decir, aquellas que se ejecutan con permisos de root. El framework desarrollado se encarga de monitorizar dinámicamente las llamadas al sistema realizadas por estos programas. Gracias a este framework confirmaron vulnerabilidades en diferentes aplicaciones software populares como *rmp, emacs* y *vi*. Los mismos autores presentaron en 2006 un mecanismo de defensa dirigido por eventos y basado en su anterior trabajo. Este mecanismo de defensa, llamado EDGI, consiste en modificaciones modulares del núcleo de Linux para que, en caso de detectarse una posible TOCTOU, el segundo proceso fuera paralizado en mitad de las dos llamadas vulnerables [56]. Un año más tarde, los mismos autores estudiaron el comportamiento de la vulnerabilidad en sistemas de un sólo procesador y en sistemas de varios. El resultado de su trabajo fue un modelo probabilístico para predecir el índice de éxito de un ataque a TOCTOU en ambos tipos de sistemas [57]. Finalmente, en el año 2010, los mismos autores presentaron un modelo para la vulnerabilidad TOCTOU que enumera todos los posibles pares TOCTOU en sistemas UNIX y Linux [11]. Este modelo, llamado STEM, está basado en todos sus anteriores trabajos y ha sido la base de pares TOCTOU vulnerables considerada en este trabajo.

En 2005 Lhee & Chapin [2] presentaron un modelo de detección de TOCTOU en tiempo de ejecución. Su modelo se basa en la creación de una librería compartida que hookea las funciones vulnerables y, mediante la invocación de un módulo del núcleo creado específicamente para ello, se comprueba la integridad de las operaciones.

Uppuluri et al. propusieron en 2005 un modelo basado en políticas que, monitorizando las operaciones sobre el sistema de ficheros, previene la explotación de la ventana de vulnerabilidad entre cualesquiera operaciones sobre el sistema de ficheros consecutivas [58]. Las políticas definidas son expresadas como patrones que se aplican a secuencias de llamadas al sistema que interactúan con el sistema de ficheros y sus argu-

mentos. Se definen dos tipos de políticas: las genéricas, que capturan la conmutabilidad entre las operaciones sobre el sistema de ficheros y las específicas a las aplicaciones.

En el año 2009 cuando Spillane et al. presentaron una API para las operaciones en el sistema de ficheros que implementase el modelo transaccional [59]. El modelo presentado se basa en el modelo transaccional de las bases de datos. En este modelo, cuando un proceso lee o escribe un fichero o directorio, lo bloquea para que nadie más pueda hacerlo. La implementación de la nueva API se llevó a cabo dentro del núcleo del sistema.

Payer & Gross crearon una herramienta llamada DynaRace en 2012 con el propósito de proteger a las aplicaciones de TOCTOU en el sistema de ficheros [47]. Esta herramienta intercepta las llamadas al sistema de ficheros mediante instrumentación dinámica gracias al uso del framework de reescritura de binarios libdetox. DynaRace mantiene información acerca de todos los objetos del sistema de ficheros accedidos y comprueba dinámicamente si puede existir TOCTOU. DynaRace, en el momento de la publicación, sólo consideraba las llamadas `stat`, `access`, `open`, `creat`, `chmod`, y `close`. Estas llamadas inseguras son sustituidas por implementaciones propias.

En el mismo año, Vijayakumar et al. desarrollaron una herramienta llamada Sting [60] cuyo propósito era la detección de vulnerabilidades en la resolución de nombres. Se basaba en el análisis dinámico del proceso de resolución para detectar cuando un atacante tiene permisos de escritura a un directorio compartido con el programa vulnerable. Esta herramienta ha sido desarrollada como un prototipo para la versión 3.2.0 del núcleo de Linux.

Adicionalmente a todas estas soluciones se han desarrollado otras más genericas. Porter et al. propusieron TxOS, una variante a Linux que implementa transacciones [61]. Las transacciones permiten al programador hacer cambios en los recursos del sistema con la garantía de atomicidad, consistencia, aislado y durabilidad. TxOS añade dos llamadas al sistema que permiten especificar regiones de código que deben ser ejecutadas con el sistema en modo transacción. Uno de los parches propuestos por el Openwall, el proyecto de Solar Designer [62], restringe las operaciones sobre ficheros de forma que explotar las condiciones de carrera sea más difícil. Por ejemplo, no se permite crear enlaces a ficheros a los que no se tenga acceso de lectura y escritura.

Otras soluciones como los *wrappers* a las llamadas del sistema fueron estudiadas por diferentes trabajos [63, 64] y llegaron a la misma conclusión, que si bien es cierto que para algunas vulnerabilidades es una solución adecuada, para las condiciones de carrera no, pues son igualmente vulnerables. Los *wrappers* a las llamadas del sistema son un mecanismo que añade una capa intermediaria entre las aplicación vulnerable y el sistema operativo. Este mecanismo es también llamado interposición de llamadas al sistema e intercepta cada una de las llamadas al sistema realizando comprobaciones en base a los parámetros recibidos y el estado de la aplicación.

Por último, es importante mencionar que diversos analizadores estáticos de propósito general trataron de enfrentar este problema como [9, 65–68], entre muchos otros.

El trabajo presentado en este proyecto se asemeja al desarrollado por Lhee & Chapin [2] en 2005. Lhee & Chapin plantearon la detección de condiciones de carrera en operaciones con ficheros a través de un modelo en tiempo de ejecución. Para llevar esto a cabo, propusieron un modelo que comprobase la integridad de la relación entre el objeto del sistema de ficheros y su referencia en las secciones críticas. Esto se consigue mediante el mantenimiento de una tabla formada por elementos del tipo <`ruta`, `inodo`>. Cuando se produce una operación de tipo *check*, se crea una entrada en la tabla. Cuando se produce una operación de tipo *use*, se comprueba su integridad y se informa en caso de TOCTOU.

La implementación que Lhee & Chapin llevaron a cabo consiste en la creación de una librería compartida para interceptar las llamadas vulnerables y de un módulo del núcleo en el que se hacían las comprobaciones acerca de la integridad. En el caso de libTTThwart, se trata de una librería compartida, sin necesidad de módulos del núcleo. Es importante mencionar que Lhee & Chapin no tuvieron en cuenta las condiciones de carrera que se pueden producir por el simple hecho de incorporar dos elementos en su modelo de detección. Es decir, entre las operaciones que hace su librería compartida y el módulo del núcleo, el sistema de ficheros puede ser alterado. Lo mismo sucede entre que el módulo del núcleo termina de hacer las comprobaciones e invoca finalmente a la llamada al sistema correspondiente.

Otra diferencia muy importante es la cantidad de funciones hookeadas. En su trabajo, Lhee & Chapin hookean explícitamente 20 funciones vulnerables y aseguran que

indirectamente están hookeando otras 9 más. La librería libTTThwart actualmente hookea de manera explícita 46 funciones diferentes.

Otra carencia que se ha detectado en el trabajo de Lhee & Chapin es la poca información acerca de la implementación real. Si bien es cierto que los algoritmos están claros así como la idea de qué es lo que hace (o debería hacer) el sistema propuesto, hay escasa información acerca de la implementación. En el presente trabajo se ha detallado en gran nivel la implementación de libTTThwart y, gracias a ello, se ha podido comprobar que los fallos de seguridad siguen existiendo pues son intrínsecos a la API de Linux, lo cual está a un nivel mucho más bajo y requiere cambios en el propio núcleo del sistema operativo.

Por último, los autores asumen la unicidad del inodo. Como se ha visto a lo largo del presente proyecto, esto es totalmente dependiente del sistema de ficheros. Otros autores como Park et al. [55] o Payer & Gross [47], por mencionar algunos, también incluyen en sus trabajos la detección de cambios de inodo sin considerar que no es una característica fiable.

A diferencia del resto de soluciones mencionadas en este capítulo, la instalación y funcionamiento de libTTThwart no depende de módulos para determinadas versiones del núcleo del sistema operativo, de *frameworks* o software adicional o de la definición de políticas de funcionamiento, pues para hacer uso de libTTThwart basta con compilar la librería y escribir su ruta en el fichero `/etc/ld.so.preload`. Respecto a la cantidad de funciones vulnerables, libTTThwart es la solución que identifica el mayor número de funciones que generan TOCTOU. A la hora de detectar la vulnerabilidad, libTTThwart registra los cambios en el inodo, la identificación del dispositivo y el modo del fichero del objeto del sistema de ficheros. El resto de soluciones que registran cambios en los metadatos, a excepción de DynaRace [47], basan su protección en sólo algunos de esos metadatos. Otro factor que tiene en cuenta libTTThwart, y que ninguna otra solución lo considera, es el sistema de ficheros donde se encuentra el objeto referenciado. Esto es importante ya que es lo que determina la reutilización de los inodos y su consideración como elemento identificativo exclusivo. Finalmente, libTTThwart comprueba si en el momento de la ejecución el programa se está ejecutando con más permisos que el usuario, evitando así hookear las funciones cuando la vulnerabilidad no puede producirse.

6. Conclusiones y trabajo futuro

En el presente trabajo se ha llevado a cabo un estudio acerca de las vulnerabilidades de tipo condición de carrera que se producen en los sistemas de ficheros, también conocidas como TOCTOU. En base a este estudio se ha desarrollado una librería de usuario para combatir la vulnerabilidad. Esta librería está implementada íntegramente en C y sólo precisa de la librería `GLIBC` y del enlazador/cargador dinámico de GNU, `ld.so`. En este trabajo se ha ampliado, además, el listado de funciones vulnerables a TOCTOU. Adicionalmente se han presentado pruebas de concepto y la evaluación del rendimiento del sistema cuando la librería desarollada está activada. También se han presentado las limitaciones de la librería así como un estudio de las vulnerabilidades que la librería, a su vez, sufre.

La vulnerabilidad TOCTOU, dada su naturaleza no determinista y sutileza, es realmente difícil de detectar y de reproducir. Su completa eliminación es aún a día de hoy un reto. Si bien es cierto que a lo largo de la historia se han propuesto muchos mecanismos de defensa y de detección temprana, estos se enfocaban en el problema parcialmente o han sido derrotados en posteriores trabajos. Las soluciones que existen actualmente pueden ayudar a paliar la vulnerabilidad pero no alcanzan su impedimento. Para eliminar completamente la vulnerabilidad es necesario un cambio en el núcleo del sistema operativo. Lo ideal sería modificar todas las funciones que interactúan con el sistema de ficheros para que lo hagan a través de descriptores y no de rutas. La creación de nuevas APIs para Linux o librerías con diferentes implementaciones de las funciones no es suficiente pues, por debajo, las llamadas al sistema serían las mismas. Otra posible solución al problema del ataque basado en enlaces consistiría en modificar el sistema de ficheros para impedir que un usuario pueda crear enlaces a ficheros sobre los cuales no tienen ningún permiso.

Dado el estado actual de la defensa de esta vulnerabilidad, gran parte de la responsabilidad para prevenirla recae sobre los programadores. Esto es, que la vulnerabilidad se elimine en las fases más tempranas del desarrollo de la aplicación, escribiendo código que no sea vulnerable. Si bien es cierto que esto es posible, es extremadamente difícil. Escribir código seguro es una ardua tarea, laboriosa y propensa a errores y, no siendo

esto suficiente, todo depende de la experiencia de los programadores. A día de hoy, la mayoría del código fuente no se escribe pensando en la seguridad, sino en alcanzar las fechas límite, añadir funcionalidad a toda costa y en satisfacer las diversas necesidades que el mercado impone. El desarrollo software hoy en día es un negocio y como tal, su principal objetivo es conseguir beneficios ahorrando costes en los tiempos de producción, en la contratación de personal con menor experiencia o incluso en la contratación de menos personal, quedando la seguridad queda relegada a un segundo o tercer plano.

Como líneas futuras al presente trabajo se proponen diversas mejores que completarán la funcionalidad de libTTThwart.

En primer lugar, la superación del límite de un único proceso se puede conseguir con un cambio en la implementación de la estructura de datos, haciendo uso de memoria compartida entre procesos y así hacer que la protección abarque también llamadas a la familia `exec()` y `fork()`. Esto se puede conseguir mediante el uso de funciones como `mmap()` o `shmget()` que permiten inicializar segmentos de memoria compartidos, entre otras.

Es importante mejorar el rendimiento de la librería. Como se describe en la Sección 4.2 Evaluación del rendimiento, la librería tiene un impacto notable en el rendimiento. Para conseguir una reducción en el impacto se propone mejorar el algoritmo de búsqueda de las entradas de la estructura de datos. Actualmente el proceso de búsqueda es lineal. Conseguir minimizar el impacto de la protección contra TOCTOU en el rendimiento conllevará que su aceptación sea mayor y, por supuesto, que su uso sea más transparente.

Otro aspecto que mejorar de cara al futuro consiste en la protección de la estructura de datos mediante mecanismos de sincronización y exclusión mutua. Esto permitiría, además, que la librería protegiese aplicaciones que hacen uso de hilos sin ningún problema.

Finalmente, otra de las líneas futuras considerada consiste en ampliar las funciones hookeadas por parte de la librería, contemplando las nuevas versiones de Linux.

Glosario

AST ***(Abstract Syntax Tree)*** El AST, en español *árbol de sintaxis abstracta*, es una representación en forma de árbol jerárquico de la estructura del código fuente de un programa. Se denomina abstracto porque omite muchos detalles del código fuente, pues se centra en reglas más que en elementos particulares como paréntesis, puntos u otros.

benchmark Un benchmark es un estándar o conjunto de estándares que se establecen para evaluar y medir el rendimiento o la calidad de un sistema o elemento software. El microbenchmarking, por otra parte, consiste en evaluar una parte pequeña de un sistema o elemento software, como el tiempo que `write()` tarda en escribir en disco.

CFG ***(Control Flow Graph)*** El CFG, en español *grafo de control de flujo*, es una representación en forma de grafo dirigido de todos los caminos que un programa podría llegar a tomar durante su ejecución.

distribución Una distribución de Linux, abreviada habitualmente como distro, es un sistema operativo creado a partir del núcleo de Linux. Sólo a modo de ejemplo, Debian y Fedora son dos de las distros más famosas. Ubuntu es una distro basada en Debian.

enlace En los sistemas operativos de tipo Unix, un enlace, del inglés *link*, es un objeto en el sistema de ficheros que referencia, apunta o clona a otro objeto del sistema. Existen dos tipos de enlaces: los enlaces simbólicos (*symbolic link*) o los enlaces duros (*hard link*). Los enlaces simbólicos son conocidos como symlink y los enlaces duros simplemente como enlaces.

hilo El hilo es la unidad de ejecución que manejan los procesos. Cuando un proceso lanza alguna tarea, esta se ejecuta en el hilo principal o se crean hilos adicionales. Un proceso puede tener múltiples hilos pero un hilo no puede pertenecer a más de un proceso.

hooking El hooking de funciones consiste en capturar las llamadas a las funciones que se desee y poder sustituir su comportamiento original.

proceso Un proceso es un programa en ejecución. Múltiples procesos pueden asociarse al mismo programa, ejecutándolo múltiples veces, por ejemplo. Un proceso puede ejecutar uno o más hilos.

programa Un programa es un conjunto de instrucciones. Un programa reside en disco, no en la memoria principal. Mientras que un programa es un elemento pasivo, pues es una colección de instrucciones, cuando se ejecuta pasa a la memoria principal y se lanza en procesos, que son elementos activos.

root Root es como se conoce comúnmente al usuario que tiene privilegios o permisos elevados. En sistemas operativos se diferencia entre usuarios normales y el usuario root. Root es un término empleado en el mundo de los sistemas operativos basados en Unix, como Linux. En función del sistema operativo recibe otros nombres como administrador, admin, superusuario, etc.

Bibliografía

[1] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 1, pp. 74–88, 1992.

[2] K. S. Lhee and S. J. Chapin, "Detection of file-based race conditions," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 105–119, 2005.

[3] T. Farah, R. Shelim, M. Zaman, M. Hassan, and D. Alam, "Study of race condition: A privilege escalation vulnerability," *WMSCI 2017 - 21st World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings*, vol. 2, no. 1, pp. 22–26, 2017.

[4] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses," *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, p. 17, 2003.

[5] W. S. McPhee, "Operating system integrity in OS/VS2," *IBM Systems Journal*, vol. 13, no. 3, pp. 230–252, 1974.

[6] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security analysis and enhancements of computer operating systems," Institute for Computer Sciences and Technology. National Bureau of Standards. Washington, D.C., Tech. Rep., 1976.

[7] R. Bisbey and D. Hollingsworth, "Protection analysis project final report," *ISI/RR-78-13, DTIC AD A*, vol. 56816, 1978.

[8] J. Wei and C. Pu, "TOCTTOU Vulnerabilities in UNIX-Style File Systems : An Anatomical Study," *Security*, no. December, pp. 1–13, 2005.

[9] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu, "Model checking an entire linux distribution for security violations," in *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005, pp. 10–pp.

[10] D. R. Kuhn, M. S. Raunak, and R. Kacker, "An Analysis of Vulnerability Trends, 2008-2016," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2017, pp. 587–588.

[11] J. Wei and C. Pu, "Modeling and preventing TOCTTOU vulnerabilities in Unix-style file systems," *Computers and Security*, vol. 29, no. 8, pp. 815–830, nov 2010.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[13] C. Mulliner and B. Michéle, "Read It Twice! A Mass-Storage-Based TOCTTOU Attack," in *WOOT*, 2012, pp. 105–112.

[14] M. Bishop, "Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux," Tech. Rep., 1995.

[15] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, no. 2, pp. 131–152, 1996.

[16] D. Tsafrir, T. Hertz, D. A. Wagner, and D. Da Silva, "Portably Solving File TOCTTOU Races with Hardness Amplification," in *FAST*, vol. 8, 2008, pp. 1–18.

[17] W3techs, "Usage Statistics and Market Share of Operating Systems for Websites, August 2019," 2019. [Online]. Available: https://w3techs.com/technologies/overview/operating_system/all

[18] ——, "Usage Statistics and Market Share of Unix for Websites, August 2019," 2019. [Online]. Available: https://w3techs.com/technologies/details/os-unix/all/all

[19] TOP500, "Development over Time | TOP500 Supercomputer Sites," 2019. [Online]. Available: https://www.top500.org/statistics/overtime/

[20] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[21] D. A. Wheeler, "Secure programming for Linux and Unix HOWTO," 1999.

[22] T. Gonzalez and S. Sahni, "Preemptive scheduling of uniform processor systems," *Journal of the ACM (JACM)*, vol. 25, no. 1, pp. 92–101, 1978.

[23] E. L. Lawler and J. Labetoulle, "On preemptive scheduling of unrelated parallel processors by linear programming," *Journal of the ACM (JACM)*, vol. 25, no. 4, pp. 612–619, 1978.

[24] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," in *Annals of discrete mathematics*. Elsevier, 1979, vol. 5, pp. 287–326.

[25] S. Khanna, M. Sebree, and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0," in *in Proceedings of the USENIX Winter Conference*. USENIX Association, 1992, pp. 375–390.

[26] L. Bianco, J. Blazewicz, P. Dell'Olmo, and M. Drozdowski, "Preemptive multiprocessor task scheduling with release times and time windows," *Annals of Operations Research*, vol. 70, pp. 43–55, 1997.

[27] C. J. Fidge, "Real-time schedulability tests for preemptive multitasking," *Real-Time Systems*, vol. 14, no. 1, pp. 61–93, 1998.

[28] P. Mateti, "Race Condition Exploits," 2012.

[29] D. M. Beazley, B. D. Ward, and I. R. Cooke, "The inside story on shared libraries and dynamic loading," *Computing in Science & Engineering*, vol. 3, no. 5, pp. 90–97, 2001.

[30] D. A. Wheeler, "Program Library HOWTO," *Free Software Foundation, Inc.*, 2003.

[31] M. J. Bach *et al.*, *The design of the UNIX operating system*. Prentice-Hall Englewood Cliffs, NJ, 1986, vol. 5.

[32] S. J. Leffler, "The design and implementation of the 4.3 BSD UNIX operating system," *Reading: Addison Wesley, 1989*, 1989.

[33] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[34] L. Presser and J. R. White, "Linkers and loaders," *ACM Computing Surveys*, vol. 4, no. 3, pp. 149–167, 1972.

[35] H. Casanova, "Linking and Loading," ICS312. University of Hawaii, Tech. Rep., 2010.

[36] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.

[37] M. Kerrisk, "ld.so(8) - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man8/ld.so.8.html

[38] ——, "dlsym(3) - Linux manual Page." [Online]. Available: http://man7.org/linux/man-pages/man3/dlsym.3.html

[39] R. Stallman *et al.*, "The GNU project," 1998.

[40] C. O'Donell, "Carlos O'Donell - The GNU C Library version 2.30 is now available." [Online]. Available: https://sourceware.org/ml/libc-announce/2019/msg00001.html

[41] L. Lu, A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and S. Lu, "A Study of Linux File System Evolution 1 Introduction," *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, 2013.

[42] A. Campbell, "SonarQube Documentation," *SonarSource SA*, vol. 11, 2015.

[43] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[44] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Citeseer, 1991, pp. 93–103.

[45] S. Zhang, B. G. Ryder, and W. Landi, "Program decomposition for pointer aliasing: A step toward practical analyses," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 81–92, 1996.

[46] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering.* ACM, 2001, pp. 54–61.

[47] M. Payer and T. R. Gross, "Protecting applications against TOCTTOU races by user-space caching of file metadata," *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pp. 215–226, 2012.

[48] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *Proceedings 2002 IEEE Symposium on Security and Privacy.* IEEE, 2002, pp. 177–187.

[49] D. Dean and A. J. Hu, "Fixing Races for Fun and Profit: How to Use access (2)," in *USENIX Security Symposium*, 2004, pp. 195–206.

[50] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, "Fixing Races for Fun and Profit: How to Abuse atime," in *USENIX Security Symposium*, 2005.

[51] X. Cai, Y. Gui, and R. Johnson, "Exploiting unix file-system races via algorithmic complexity attacks," in *2009 30th IEEE Symposium on Security and Privacy.* IEEE, 2009, pp. 27–41.

[52] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[53] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities," in *USENIX Security Symposium*, 2001, pp. 165–176.

[54] B. Goyal, S. Sitaraman, and S. Venkatesan, "A unified approach to detect binding based race condition attacks," in *Int'l Workshop on Cryptology & Network Security (CANS)*, 2003.

[55] J. Park, G. Lee, S. Lee, and D.-k. Kim, "RPS: An extension of reference monitor to prevent race-attacks," in *Pacific-Rim Conference on Multimedia*. Springer, 2004, pp. 556–563.

[56] C. Pu and J. Wei, "A Methodical Defense against TOCTTOU Attacks: The EDGI Approach," in *Proceedings of the 2006 International Symposium on Secure Software Engineering*, 2006.

[57] J. Wei and C. Pu, "Multiprocessors may reduce system dependability under file-based race condition attacks," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007, pp. 358–367.

[58] P. Uppuluri, U. Joshi, and A. Ray, "Preventing race condition attacks on file-systems," in *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, 2005, pp. 346–353.

[59] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling Transactional File Access via Lightweight Kernel Extensions," in *FAST*, vol. 9, 2009, pp. 29–42.

[60] H. Vijayakumar, J. Schiffman, and T. Jaeger, "STING: Finding Name Resolution Vulnerabilities in Programs," *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pp. 585–599, 2012.

[61] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating system transactions," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 161–176.

[62] S. Designer *et al.*, "Kernel patches from the Openwall Project," 2002.

[63] T. Garfinkel *et al.*, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," in *NDSS*, vol. 3, 2003, pp. 163–176.

[64] R. N. M. Watson, "Exploiting Concurrency Vulnerabilities in System Call Wrappers," *System*, vol. 4, no. 11, p. 2, 2007.

[65] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. IEEE, 2000, pp. 257–267.

[66] H. Chen and D. Wagner, "MOPS: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 235–244.

[67] B. V. Chess, "Improving computer security using extended static checking," in *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE, 2002, pp. 160–173.

[68] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 237–252.

[69] "GNU General Public License Version 3," *Free Software Foundation, June*, vol. 29, 2007.

[70] B. Smith, "A Quick Guide to GPLv3," *Free Software Foundation, Inc. Online: http://www. gnu. org/licenses/quick-guide-gplv3. html.*, vol. 4, p. 2008, 2007.

[71] "SPEC CINT2006 Benchmarks." [Online]. Available: https://www.spec.org/cpu2006/CINT2006/

Anexo A

Horas de trabajo y seguimiento

En el presente anexo se desglosan las horas dedicadas al desarrollo del trabajo.

A.1. Seguimiento

El seguimiento se ha llevado a cabo semanalmente mediante reuniones con el tutor del trabajo. En estas reuniones se ha comentado todo lo realizado durante la semana, las dudas surgidas, cuál es el trabajo para la siguiente semana y cómo afrontarlo.

A.2. Horas de trabajo

A lo largo del trabajo se han realizado varias actividades. Estas actividades se han agrupado en 4 tareas: Estudio del problema, desarrollo y pruebas de la librería, reuniones con el tutor y la elaboración de la memoria. Se han dedicado un total de 630 horas. En la Tabla A.1 y en la Figura A.1 se puede apreciar el desglose del tiempo invertido. Hay que tener en cuenta que las tareas no son totalmente heterogéneas pues cuando se ha desarrollado no se ha dejado de estudiar, cuando se han producido las reuniones no se ha dejado de hablar del desarrollo, etc. Cada una de las actividades se ha ordenado cronológicamente en la Figura A.2.

Tabla A.1. Desglose de horas invertidas. (Fuente: propia)

Actividad	Horas invertidas
Estudio	90
Desarrollo y pruebas	360
Reuniones	22
Memoria	158
Total	**630**

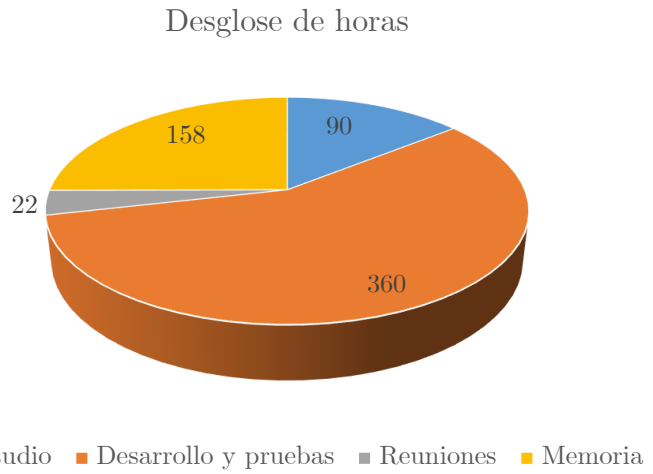

Figura A.1. Desglose de horas invertidas. (Fuente: propia)

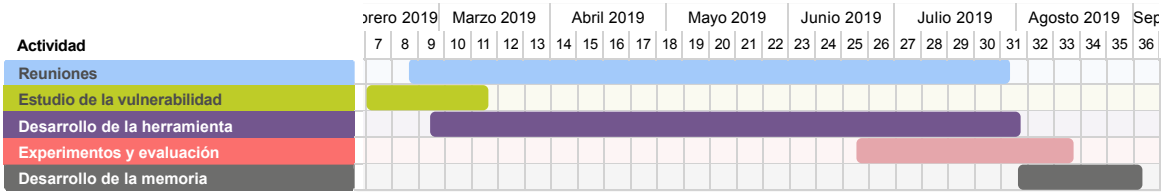

Figura A.2. Diagrama GANTT del trabajo. (Fuente: propia)

Anexo B

Instalación

En el presente anexo se detallan los requisitos de sistema de la librería, cómo instalarla y cómo interpretar su documentación. Cabe mencionar que, al ser este un proyecto de software libre, todo el código entregado es funcional y las dependencias, como la librería `Zlog`, se entregan con él. Para más información acerca de las condiciones de la licencia, los derechos que otorga y los deberes que impone, se recomienda leer el texto íntegro de la *General Public Licence* GNU GPL v3 [69, 70].

B.1. Requisitos

La librería se ha desarrollado íntegramente en C. Para su uso es necesario:

- Sistema operativo Linux, distribución de este o cualquier sistema operativo que haga uso del cargador dinámico ld.so [37]
- GCC para compilar el código.
- Librería GLIBC para implementación de las funciones estándar. (En Linux es la implementación por defecto.)
- Privilegios elevados en el sistema.

Opcionalmente se puede instalar el siguiente software:

- Git. Fácil clonado y gestión del repositorio y su código.
- Make. Compilación sencilla de la librería.

B.2. Clonado e instalación

Para instalar la librería, en primer lugar se debe clonar el código fuente. Para clonarlo se recomiendo hacer uso de Git, aunque no es estrictamente necesario.

```
git clone https://github.com/RazviOverflow/libTTThwart.git
```

Si no se dispone o no se quiere hacer uso de Git, se puede descargar directamente el código accediendo al repositorio.

Una vez clonado o descargado el repositorio, se debe compilar el código. Para ello se ha proporcionado un fichero de tipo `Makefile` que facilita las tareas. Se recomienda hacer uso de Make, aunque no es estrictamente necesario. La aplicación se puede compilar en dos modos. Los modos afectan a la cantidad de información que la aplicación escribe en los logs.

- **Defecto**. Este es el modo por defecto. En este modo la aplicación simplemente registra los intentos de TOCTOU detectados así como posibles errores que surjan a la hora de recuperar los metadatos.
- **Debug**. En el modo debug la aplicación escribe en los logs mucha más información como, por ejemplo, cada vez que se invoca a una función, cuando una entrada de la estructura de datos es actualizada, cuando se comparan los inodos y, por supuesto, cuando se detecta la vulnerabilidad.

Para compilar la aplicación en modo normal basta con utilizar el comando `make` sin ningún parámetro (desde el directorio raíz del proyecto).

```
make
```

Para compilar en modo debug, será necesario utilizar el siguiente comando:

```
make debug
```

Si no se quiere o no se dispone de Make, se puede compilar manualmente el código. Cabe recordar que el código ha sido desarrollado y compilado con GCC 6.3.0 20170516 (Debian 6.3.0-18+deb9u1). No se asegura el correcto compilado con otras versiones de GCC. Para compilar en modo por defecto será necesario lanzar el siguiente comando (desde el directorio raíz del proyecto):

```
gcc -shared -fPIC -Wall -Wextra -fstack-protector-all -O2 src/
    libTTThwart_file_objects_info.c src/libTTThwart_hooked_functions.c src
    /libTTThwart_internals.c src/libTTThwart_wrappers.c src/zlog.c -
    Iinclude -o libTTThwart.so -lpthread -ldl
```

Por otra parte, para compilar en modo debug:

```
gcc -shared -fPIC -Wall -Wextra -fstack-protector-all -O2 src/
    libTTThwart_file_objects_info.c src/libTTThwart_hooked_functions.c src
    /libTTThwart_internals.c src/libTTThwart_wrappers.c src/zlog.c -
    Iinclude -o libTTThwart.so -lpthread -ldl -D DEBUG
```

Se puede apreciar que una de las opciones de compilación es `-fpic`. Esta opción es característica de la compilación de librerías compartidas o módulos dinámicos. Se utiliza para indicarle al compilador que debe generar código independiente de la posición, del inglés *Position-Independent Code* (PIC). Esto hace que el código acceda a los símbolos mediante direcciones de memoria relativas y no absolutas. La principal ventaja de esto es que el código PIC se puede alojar en cualquier dirección de memoria y permite al sistema operativo reubicar el código en diferentes regiones de memoria virtual sin la necesidad de cambiar las instrucciones [29]. La bandera `-shared` sirve para indicar que se va a generar una librería compartida. Las banderas `-Wall` y `-Wextra` sirven para indicarle al compilador que lance todos los warnings que detecte a la hora de interpretar el código. La opción `-fstack-protector-all` le indica al compilador que debe activar la protección de canarios en la pila. `-O2` indica el nivel de optimización de código. `-lpthread` le indica al enlazador que debe enlazar la librería `pthread` usada en `Zlog`. Por último, `-ldl` le indica al enlazador que enlace la librería `libdl.so`, que es la encargada de proporcionar la interfaz para la carga dinámica.

Una vez compilado el código, en el directorio raíz aparecerá un fichero llamado `libTTThwart`. Esta es la librería compartida desarrollada. Para activarla en el sistema es tan sencillo como escribir su ruta absoluta en el fichero `/etc/ld.so.preload`. Si este fichero no existe, será necesario crearlo.

Adicionalmente, en el repositorio se han incluido los siguientes ficheros:

- `configure.sh`. Un script que crea una carpeta llamada `libTTThwart` en la variable `$HOME` del usuario y lo monta en un sistema de ficheros de tipo TMPFS.

- `vulnerableAccessFopen.c` y `vulnerableLstatOpen.c`. Dos ficheros de código fuente escrito en C vulnerables al par TOCTOU <`access`, `fopen`> y <`lstat`, `open`> respectivamente.

B.3. Documentación

Toda la librería está exhaustivamente documentada. La documentación se ha generado con `NaturalDocs`. La documentación se entrega junto al código fuente en una carpeta llamada `docs`. Para navegar e interpretar la documentación sólo se precisa de un navegador web. Para leer la documentación basta con abrir el fichero `index.html` que hay en la carpeta `docs`. A partir de él se accede al resto de la documentación.

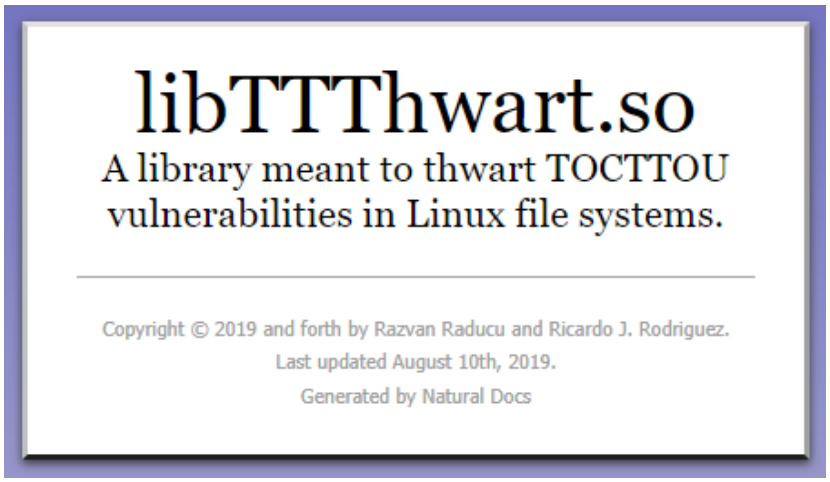

Figura B.1. Portada de la documentación. (Fuente: propia)

La organización de la documentación es muy sencilla. En el panel de la izquierda están todos los ficheros de interés, como los ficheros de cabecera y la licencia. Al hacer click en cualquiera de ellos, se abre un panel contiguo que indica todos los tipos de datos declarados, si es que los hay y, por último, la propia documentación. En todos los ficheros se indica, en primer lugar, información acerca de la licencia y los autores. La Figura B.2 muestra la organización de la documentación.

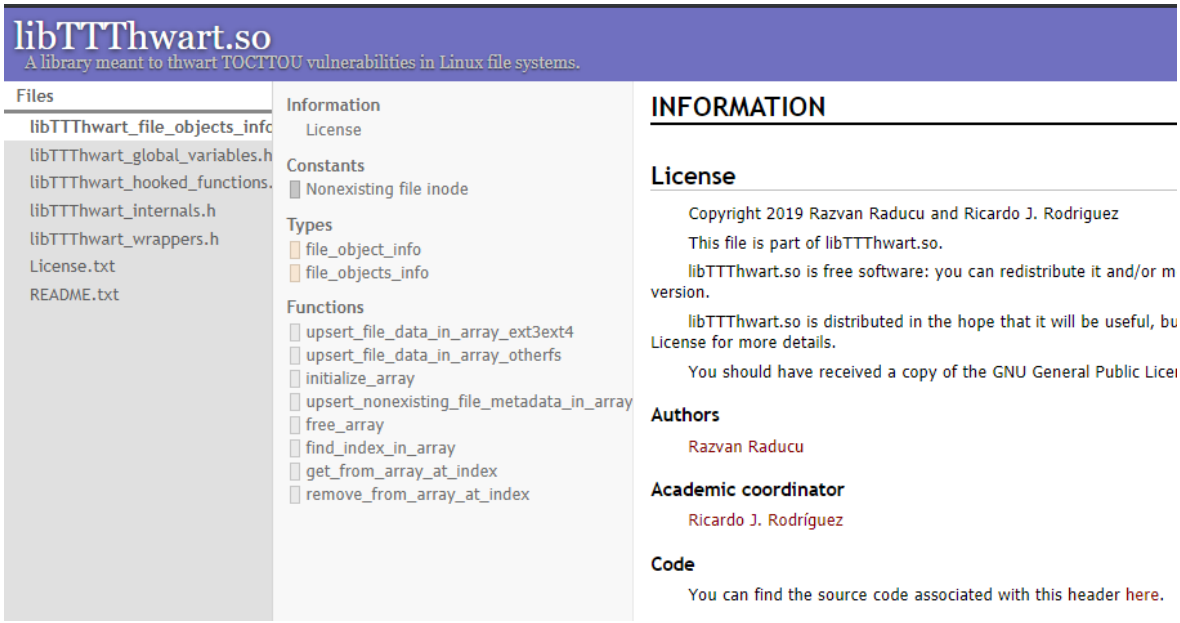

Figura B.2. Estructura de la documentación. (Fuente: propia)

Anexo C

Desglose resultados SPEC CPU2006

En el presente anexo se detallan los test realizados en el benchmark SPEC CPU2006 así como los resultados obtenidos. Como se ha mencionado a lo largo del documento, SPEC CPU2006 es una *suite* de benchmarking. Es decir, es un programa que ejecuta múltiples programas y obtiene el resultado de cada uno de ellos.

La *suite* SPEC CPU2006 define dos perfiles de benchmarking: `int` y `fp`. La primera de ellas está enfocada a la medición de la potencia de cálculo con enteros mientras que la segunda mide la potencia de cálculo con números de coma flotante. En el presente trabajo se ha ejecutado exclusivamente el perfil `int`, también conocido como `SPECint` o `CINT2006` [71]. Cuando se ejecuta este perfil, SPEC CPU2006 lanza 12 programas consecutivos, probando la capacidad de la CPU en diferentes aspectos.

Para obtener el resultado se lanzan tres ejecuciones consecutivas del perfil escogido. Es decir, del primer programa de test hasta el último, 3 veces. Se miden los segundos que tarda cada test en completarse y, en base a eso, se obtiene la ratio. La ratio se obtiene a partir de los tiempos de referencia que define la *suite* CINT2006 de la siguiente forma:

$$ratio_{benchmark} = \frac{T_{ref}(benchmark)}{T_{SUT}(benchmark)}$$

donde

$T_{ref}(benchmark)$ = tiempo de ejecución del benchmark *benchmark* de referencia

SUT = System Under Test (Sistema que se está evaluando)

Los tiempos de ejecución de referencia de los diferentes benchmarks se pueden ver en la Tabla C.1. Tras obtener el resultado de cada uno de los test en las tres ejecucio-

nes diferentes, se selecciona la ratio media (mediana). La ratio seleccionada de cada test es lo que finalmente se utiliza para calcular el resultado de todo el proceso, que representará la potencia de la CPU.

Tabla C.1. Tiempos de referencia SPEC CPU2006 (CINT2006). (Fuente: [71])

Test	Tiempo (s)
400.perlbench	9770
401.bzip2	9650
403.gcc	8050
429.mcf	9120
445.gobmk	10490
456.hmmer	9330
458.sjeng	12100
462.libquantum	20720
464.h264ref	22130
471.cmnetpp	6250
473.astar	7020
483.xalancbmk	6900

En las Tablas C.2 y C.3 se muestran las estadísticas descriptivas de la puntuación obtenida y del tiempo de ejecución con y sin libTTThwart, respectivamente.

Por último, las Figuras C.1, C.2, C.3 y C.4 muestran las gráficas con los resultados de la ejecución de CINT2006 de forma nativa. Las Figuras C.5, C.6, C.7 y C.8 muestran su ejecución con libTTThwart activada en el sistema.

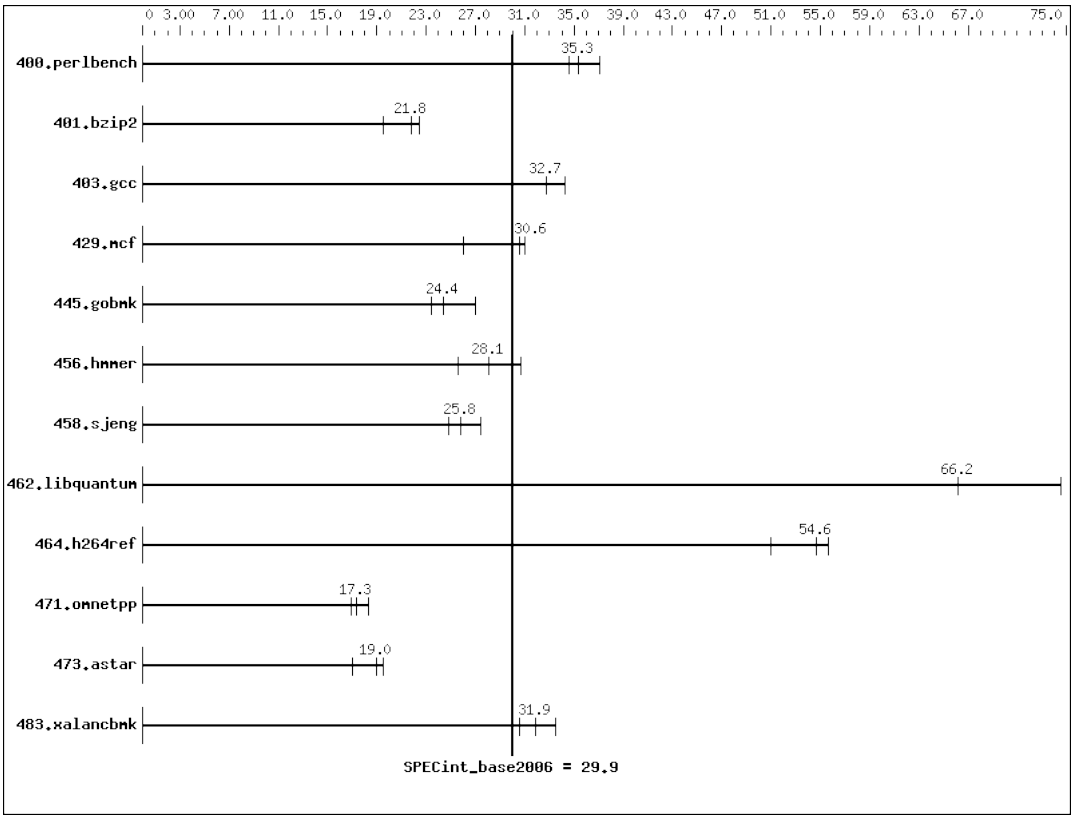

Figura C.1. SPEC CINT2006 Nativo Ejecución 1. (Fuente: propia)

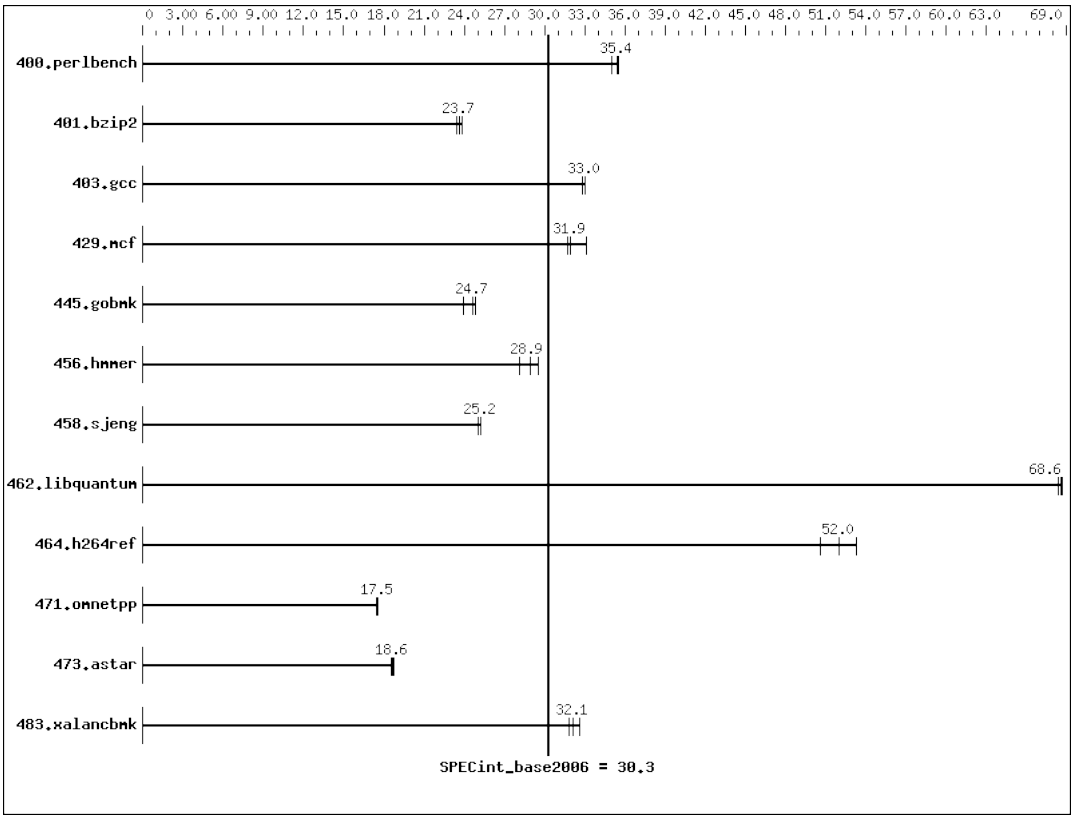

Figura C.2. SPEC CINT2006 Nativo Ejecución 2. (Fuente: propia)

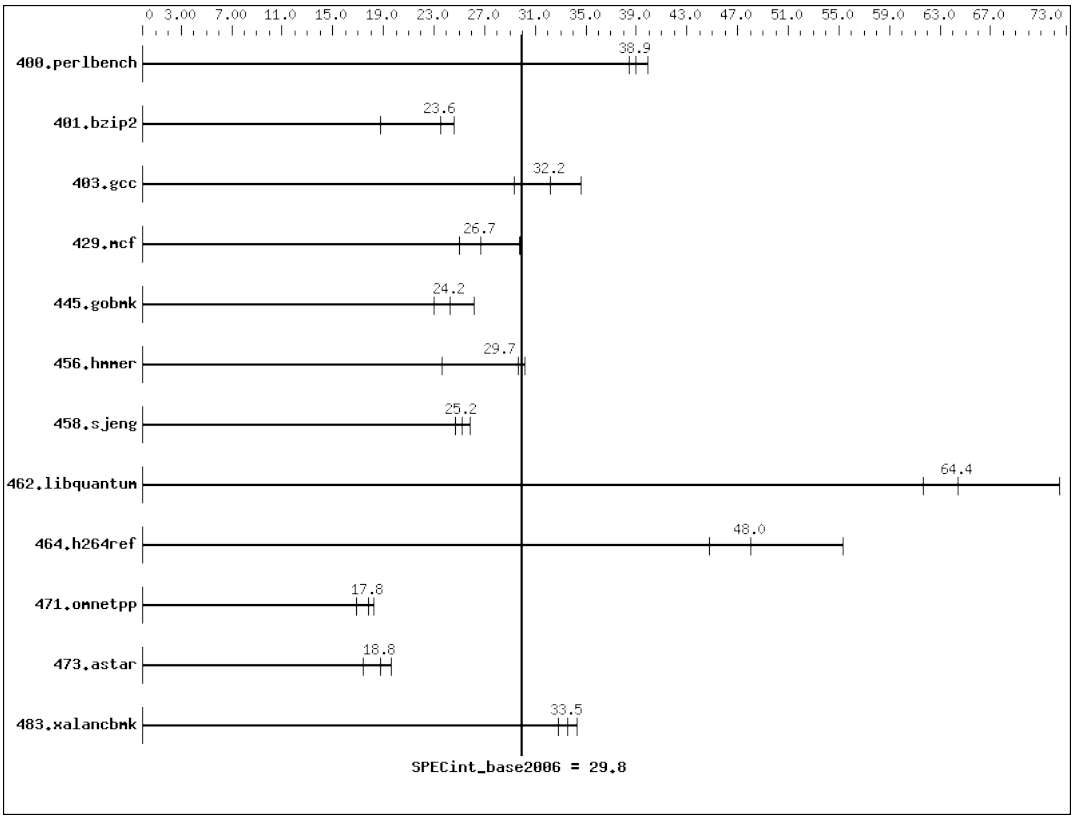

Figura C.3. SPEC CINT2006 Nativo Ejecución 3. (Fuente: propia)

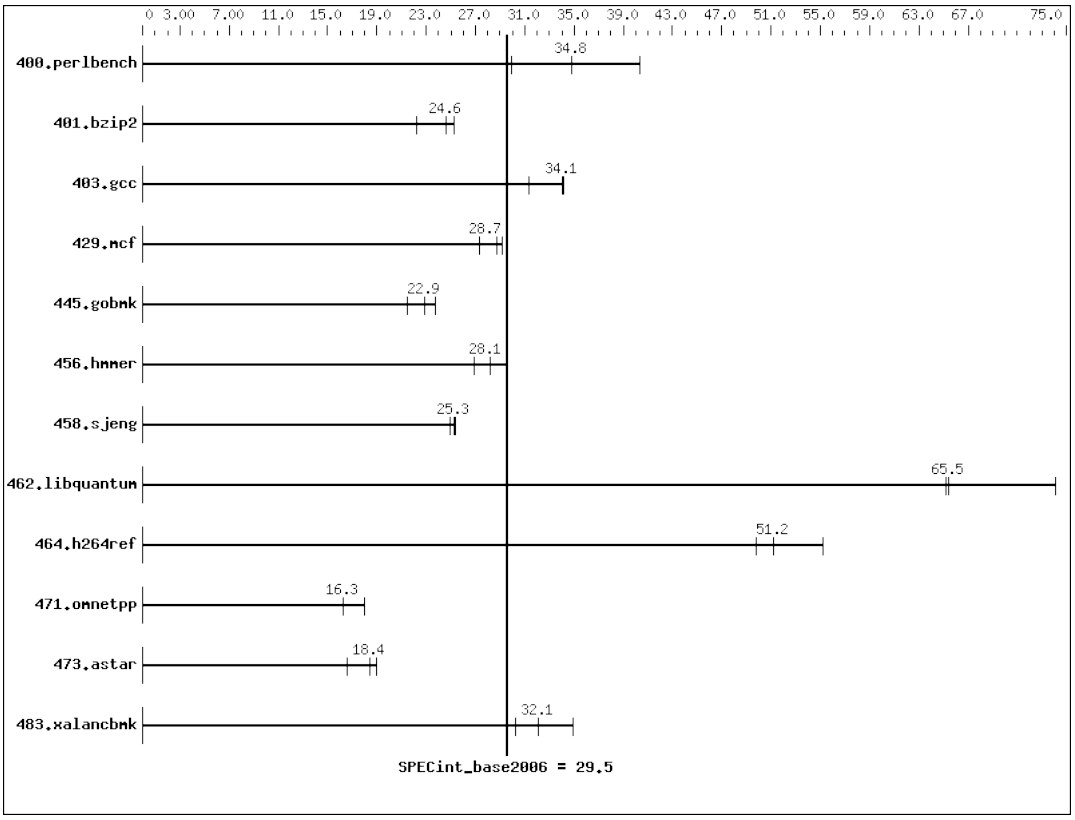

Figura C.4. SPEC CINT2006 Nativo Ejecución 4. (Fuente: propia)

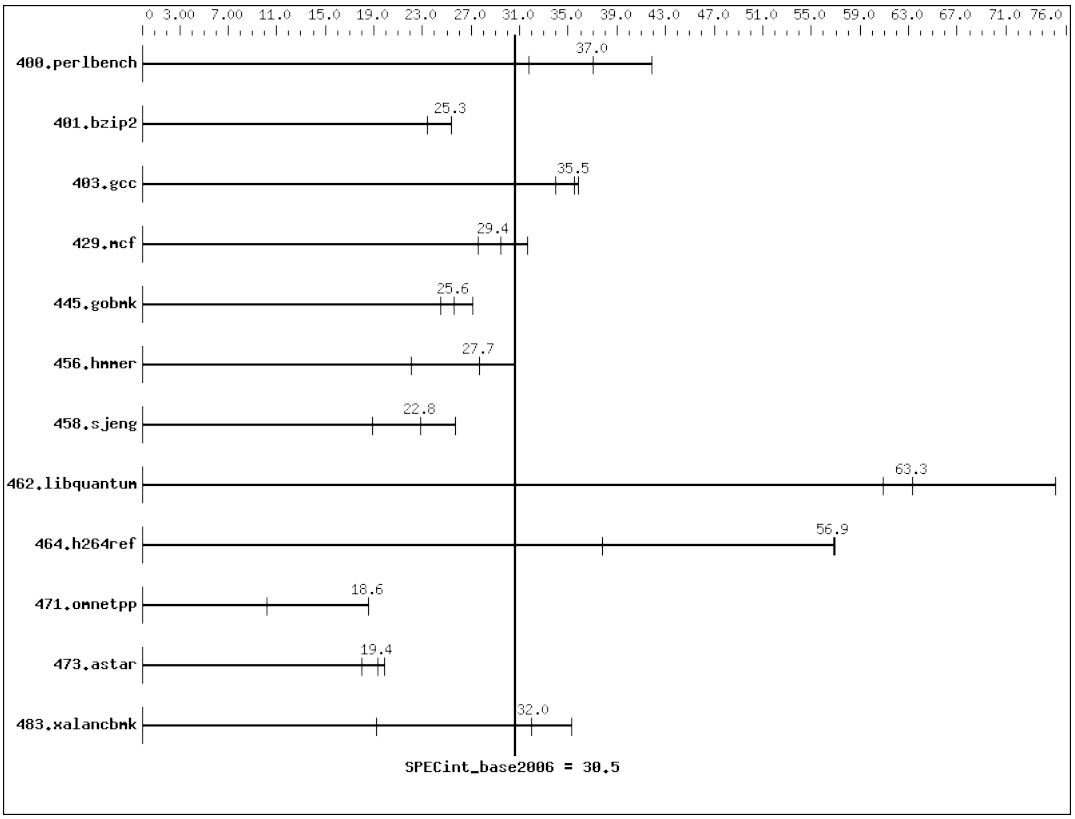

Figura C.5. SPEC CINT2006 Librería Ejecución 1. (Fuente: propia)

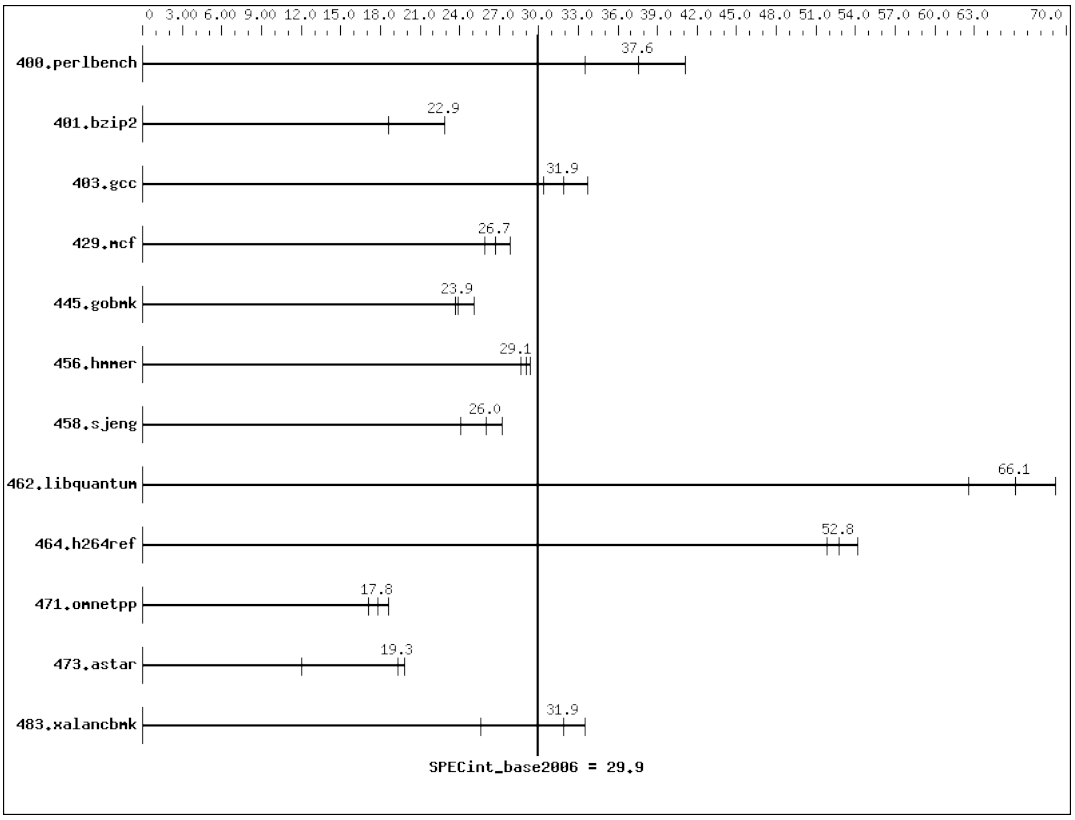

Figura C.6. SPEC CINT2006 Librería Ejecución 2. (Fuente: propia)

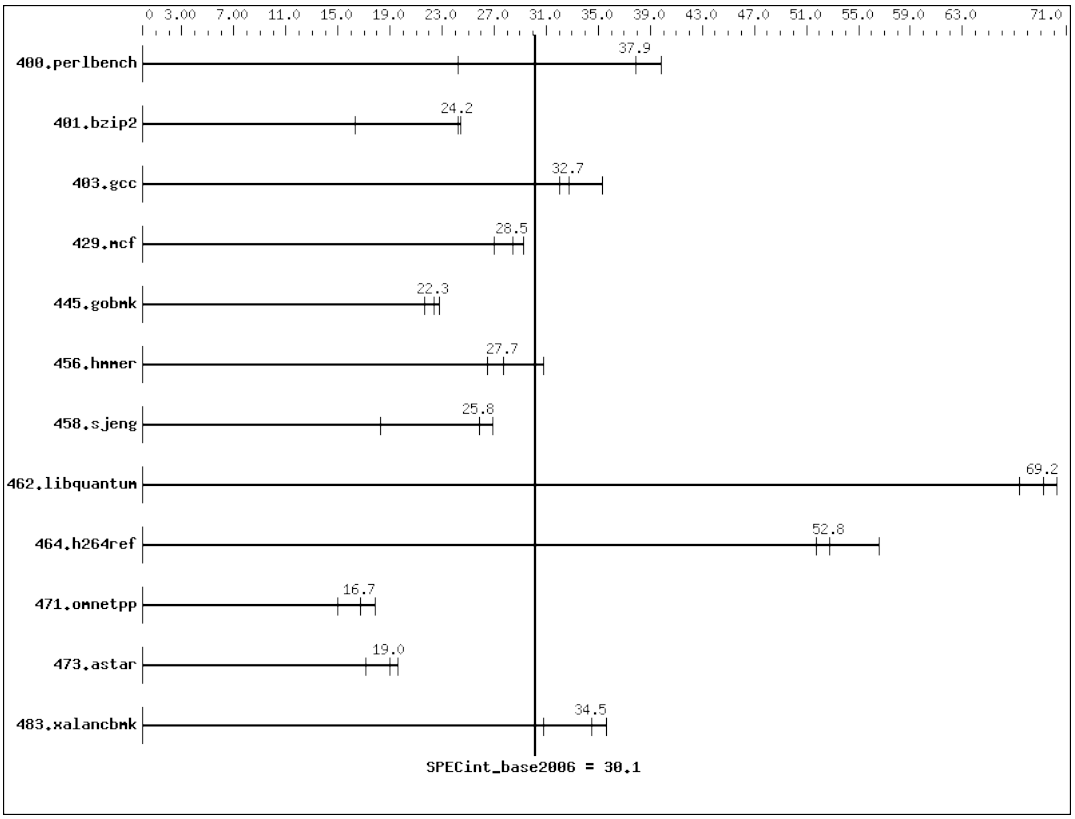

Figura C.7. SPEC CINT2006 Librería Ejecución 3. (Fuente: propia)

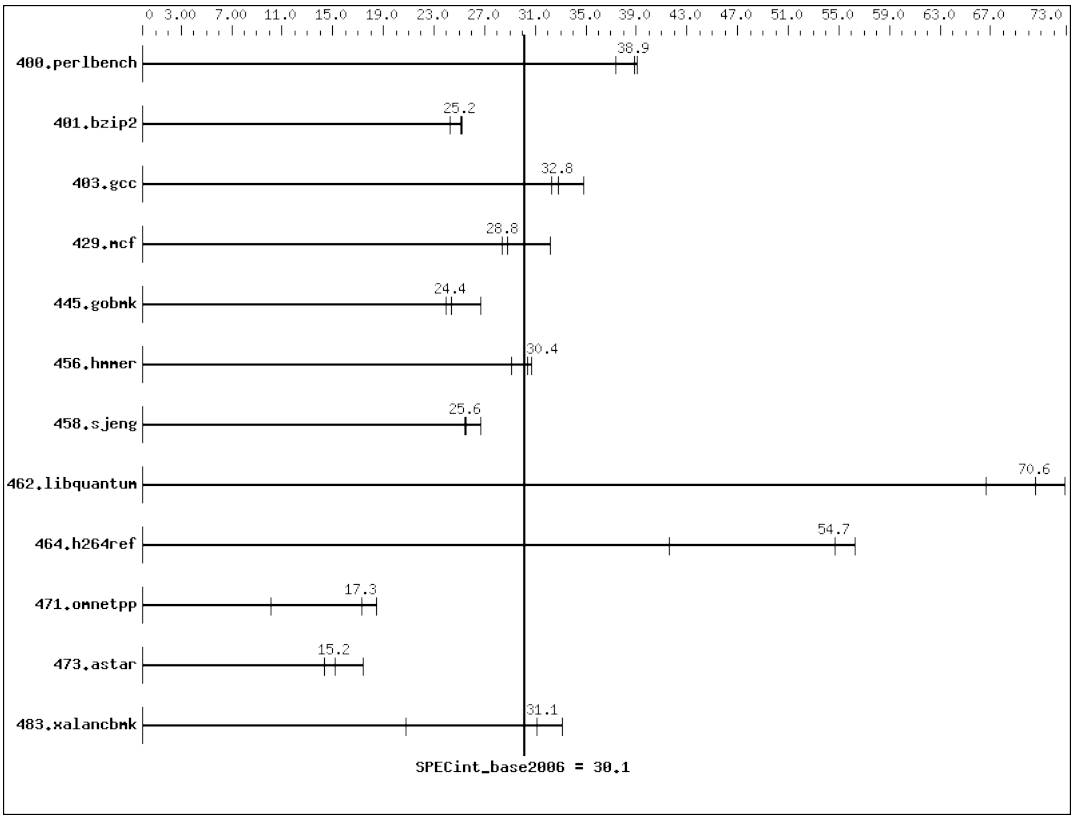

Figura C.8. SPEC CINT2006 Librería Ejecución 4. (Fuente: propia)

Tabla C.2. Estadísticas descriptivas acerca de la puntuación CINT2006. (Fuente: propia)

	Nativo	**Librería**
Media	29.875	30.15
Error estándar	0.17	0.13
Mediana	29.85	30.1
Desviación estándar	0.33	0.25
Varianza muestral	0.10	0.06
Curtosis	1.17	2.23
Asimetría	0.44	1.13
Rango	0.8	0.6
Mínimo	29.5	29.9
Máximo	30.3	30.5

Tabla C.3. Estadísticas descriptivas acerca de los tiempos de ejecución CINT2006. (Fuente: propia)

	Nativo	**Librería**
Media	13628.25	14068.5
Error estándar	68.92	84.13
Mediana	13655.5	14060
Desviación estándar	137.83	168.26
Varianza muestral	18997.58	28313
Curtosis	-2.09	0.52
Asimetría	-0.67	0.28
Rango	294	404
Mínimo	13454	13875
Máximo	13748	14279

Anexo D

Códigos microbenchmark

En el presente anexo se muestra el código fuente utilizado para realizar las pruebas de microbenchmarking. El código utilizado para realizar las pruebas de microbenchmarking, los scripts para lanzarlos y un script para monitorizar la actividad de la CPU se encuentran públicamente disponibles en: `https://github.com/RazviOverflow/cmicrobenchmark`.

El Código D.1 muestra el código ejecutado en los microbenchmarks *access*, *open/close* y *largo*. El Código D.2 muestra el código del microbenchmark *muy largo*.

```
/* Access */
float start_time = (float)clock()/CLOCKS_PER_SEC;
int rnt = access("temp.file", (R_OK|W_OK));
float end_time = (float)clock()/CLOCKS_PER_SEC;
float time_elapsed = end_time - start_time;

/* Open/close */
float start_time = (float)clock()/CLOCKS_PER_SEC;
int descriptor = open("temp.file", 0);
close(descriptor);
float end_time = (float)clock()/CLOCKS_PER_SEC;
float time_elapsed = end_time - start_time;

/* Largo */
float start_time = (float)clock()/CLOCKS_PER_SEC;
if(access("temp.file", (R_OK|W_OK)) == 0){
  int descriptor_1 = creat("test.file", 660);
```

```
18   if(descriptor_1 == -1){
19     fprintf(stderr, "Creat error %s\n", strerror(errno));
20   }
21   int descriptor_2 = open("temp.file", 0);
22   if(descriptor_2 == -1){
23     fprintf(stderr, "Open error %s\n", strerror(errno));
24   }
25   close(descriptor_1);
26   close(descriptor_2);
27 }
28   float end_time = (float)clock()/CLOCKS_PER_SEC;
```

Código D.1. Microbenchmarks access, open/close y largo. (Fuente: [47])

```
1  float start_time = (float)clock()/CLOCKS_PER_SEC;
2  if(access("temp.file", (R_OK|W_OK)) == 0){
3    int descriptor_1 = creat("test.file", 660);
4    if(descriptor_1 == -1){
5      fprintf(stderr, "Creat error %s\n", strerror(errno));
6    }
7    remove("test.file");
8    mkdir("temporal.dir", 0777);
9    rmdir("temporal.dir");
10   descriptor_1 = creat("test.file", 660);
11   if(descriptor_1 == -1){
12     fprintf(stderr, "Creat error %s\n", strerror(errno));
13   }
14   struct stat file_info;
15   stat("test.file", &file_info);
16   lstat("test.file", &file_info);
17   remove("test.file");
18   int descriptor_2 = open("temp.file", 0);
19   if(descriptor_2 == -1){
20     fprintf(stderr, "Open error %s\n", strerror(errno));
21   }
22   close(descriptor_1);
23   close(descriptor_2);
24 }
```

```
25 float end_time = (float)clock()/CLOCKS_PER_SEC;
```

Código D.2. Microbenchmark muy largo. (Fuente: propia)