Librería para análisis dinámico de programas.

Oscar Mira Sánchez
Ingeniería Técnica en Informática de Sistemas

Joaquin López Sánchez-Montañés

13 de junio de 2011

Resumen

El presente documento recoge la memoria del Proyecto de Final de Carrera (en adelante TFC) titulado: Librería para análisis dinámico de programas.

Este proyecto consiste en el diseño y desarrollo de una librería de interposición de propósito general en lenguaje C para la plataforma GNU/Linux, así como de un prototipo para demostrar la utilidad de la librería.

La idea es que dicha librería se pueda utilizar en aquellos programas que necesiten modificar el comportamiento de otros programas, interceptar las llamadas al sistema y, en un sentido amplio, obtener información sobre las actividades de otros procesos en tiempo real. Por tanto, se tratará de diseñar un marco adecuado para futuros proyectos relacionados con el análisis de software.

Se pretende también realizar un estudio que permita conocer las diferentes técnicas de depuración, diagnóstico y optimización de aplicaciones, y el software libre existente para realizar dichas tareas.

Indice de contenido

Indice de tablas y figuras

Capitulo I. Introducción

En el debate de los sistemas operativos, uno de los argumentos más utilizados en favor de la plataforma de software libre GNU/Linux es que es más seguro que sus competidores. Pero, a pesar de todo, aparecen de forma continua publicaciones que reportan fallos de seguridad en este sistema. Este proyecto se enmarca dentro del campo de la seguridad informática en la plataforma GNU/Linux.

En el contexto de una auditoría técnica de seguridad, en la fase de análisis de aplicaciones y detección de vulnerabilidades, es habitual que los profesionales se enfrenten al reto de disponer de una descripción detallada del flujo de operaciones de un determinado software. En concreto, operaciones como la apertura de ficheros y puertos de red, la gestión de memoria, y en general todos los accesos a recursos del sistema operativo por parte de una aplicación son de especial interés para el auditor durante estas pruebas técnicas. Esta información, que constituye la traza de una aplicación, es de vital importancia para poder garantizar la seguridad y el correcto funcionamiento de un sistema informático.

Normalmente, el técnico obtiene esta información durante la lectura de código fuente de la aplicación que se este auditando. También es habitual consultar la documentación del programa, o acudir a los ficheros de registro o logs en busca de respuestas. Sin embargo, muchas veces el software se distribuye en formato binario y no se dispone del código fuente, o éste es demasiado extenso. También sucede con frecuencia que la documentación, si existe, no está actualizada. Y que los ficheros de registro son incompletos o poco rigurosos.

La idea de este proyecto es estudiar las diferentes herramientas que dan solución a estas dificultades y desarrollar una librería propia capaz de registrar y analizar esta actividad en tiempo real, sin necesidad de disponer del código fuente del programa ni de la información previa sobre el mismo.

1. Experiencia previa

Trabajando como administrador de sistemas y técnico de redes en diferentes empresas he adquirido una experiencia notable en la plataforma Linux y BSD.

En la actualidad me he especializado en el análisis forense informático.

La motivación de este proyecto nace de la necesidad de diseñar una librería de interposición completa que pueda formar parte de una suite de análisis de vulnerabilidades para Linux en la que estoy trabajando.

2. Terminología

Para facilitar la comprensión de este primer capítulo se incluye a continuación una breve descripción de los términos tecnológicos más relevantes.

Una librería de enlace-dinámico consiste en un conjunto de funciones que son compiladas y enlazadas en archivos independientes, con objeto de ser compartidas por diferentes procesos simultáneamente. De esta forma de evita la duplicación de código, es decir, que las mismas funciones se copien de forma redundante en cada aplicación.

A las librerías de enlace-dinámico también se las conoce con el nombre de bibliotecas compartidas.

Al enlazar una aplicación, las referencias a objetos en bibliotecas compartidas quedan sin resolver, dejando esta tarea pendiente hasta que llega el momento de ejecutarlas.

Por tanto, la diferencia entre una librería estática y una librería compartida está en que las librerías compartidas se enlazan con las aplicaciones en tiempo de ejecución, en lugar de en tiempo de compilación. Esto se conoce como enlace dinámico, y es llevado a cabo por el cargador de la aplicación de forma automática.

El proceso por el cual, durante la llamada a una función de una biblioteca compartida, se interpone código externo entre la aplicación y la función original se conoce como interposición. El código interpuesto puede o no llamar a la función real.

Una vulnerabilidad es una debilidad en un sistema informático que puede poner en riesgo la seguridad de la información y normalmente es el resultado de un error de software o bug, susceptible de ser utilizado por un atacante.

Las auditorias técnicas de seguridad son revisiones técnicas exhaustivas de los sistema de información desde el punto de vista de la seguridad. Una de las fases del proceso es la identificación de las posibles vulnerabilidades del sistema.

Otros términos como log o tracing se desarrollarán en profundidad en el capitulo II.

3. Objetivos

Los objetivos generales del proyecto son los siguientes:

- Estudiar las herramientas y mecanismos de tracing existentes en GNU/Linux a nivel del Kernel y de la aplicación.
- Analizar la técnica de interposición y su uso en herramientas de monitorización y auditoria.
- Desarrollar una librería de interposición en C y una utilidad que demuestre su uso.
- Evaluar la funcionalidad y el rendimiento de la librería.

4. Enfoque del proyecto

Para cubrir los objetivos establecidos, la realización del proyecto se divide en dos partes.

La primera parte será una introducción teórica a las diferentes tecnologías de tracing. Primero se realizará una comparativa de las tecnologías disponibles en Linux para realizar trazas de las aplicaciones y del Kernel. Después analizaré el entorno sobre el que se pretende desarrollar la librería.

Durante el análisis de la técnica de interposición está previsto desarrollar una pequeña prueba de concepto que permita tener una visión previa de la librería, y poder así verificar su funcionamiento.

Terminado el estudio teórico, se planteará el diseño de la librería que se desea desarrollar. En este punto se definirá la estructura y la funcionalidad de la librería. La parte práctica del proyecto continuará con la implementación de los diferentes componentes en lenguaje C. Se realizaran también las correspondientes pruebas para validar su correcto funcionamiento.

Finalmente, se llevará a cabo un estudio del rendimiento y un valoración general del proyecto.

5. Requisitos de diseño

A continuación, se detallan los requisitos de diseño que debe cumplir cada uno de los componentes del proyecto.

5.1. Librería de interposición

Los requisitos previstos son:

- Capturar las llamadas a funciones, parámetros y valores de retorno, con un impacto mínimo en el rendimiento de la aplicación auditada.
- Permitir subscribirse a eventos y responder en tiempo real a las llamadas de funciones capturadas.
- Soportar múltiples procesos y threads simultáneamente.
- Utilizar mecanismos IPC del estándar POSIX.
- Funcionar en el espacio de usuario.

En el diseño de la librería se prestará especial atención a la flexibilidad de la misma, en el sentido de que pueda ser utilizada para diferentes aplicaciones.

5.2. Utilidad de tracing

En principio, el programa que demostrará el uso de la librería será una sencilla utilidad que generará por consola la traza de las aplicaciones de forma interactiva.

Otras utilidades de seguridad que se podrían desarrollar son:

- Analizador de *race conditicions* en el acceso al sistema de ficheros.
- Detector de *heap overflows*.
- *Fuzzer*.
- Sistema de Prevención de Intrusos (IPS).

Sin embargo, dado que el tiempo asignado al proyecto no es suficiente, queda excluida la posibilidad de desarrollar cualquiera de estas herramientas en el TFC.

6. Planificación

En la tabla 1 se representan las tareas del proyecto y la agenda prevista para realizarlas.

Tabla 1. Lista de tareas

#	Título de la tarea	Duración	# pred.	Comienzo
0	**TFC**			02/03/2011
1	Inicio del proyecto			02/03/2011
2	**Planificación**		1	02/03/2011
3	Lectura información	7 días		02/03/2011
4	Plan de trabajo		3	30/03/2011
5	PAC1			28/03/2011
6	**Estudio tecnologías tracing**		2	29/10/2011
7	Recopilación de información	5 días		29/03/2011
8	Comparativa herramientas existentes	5 días	7	05/04/2011
9	**Análisis de interposición dinámica en Linux**		6	12/04/2011
10	Descripción de funcionamiento	2 días		12/04/2011
11	Desarrollo prueba de concepto	4 días	10	14/04/2011
12	Test de compatibilidad entre Unices	1 día	11	20/04/2011
13	PAC2			25/04/2011
14	**Diseño y especificación**		9	21/04/2011
15	Revisión de requerimientos	1 día		21/04/2011
16	Arquitectura interna e IPC	6 días	15	22/04/2011
17	Definición estructuras y mensajes	5 días	16	02/05/2011
18	PAC3			23/05/2011
19	**Implementación**		14	23/11/2010

#	Título de la tarea	Duración	# pred.	Comienzo
20	Desarrollo código libreria	12 días		09/05/2011
21	Desarrollo código utilidad	3 días	20	25/05/2011
22	Depuración y juego de pruebas	7 días	21SS	25/05/2011
23	Empaquetado Producto	1 día	22	03/06/2011
24	**Análisis y conclusiones**		19	06/06/2011
25	Evaluación de rendimiento	2 días		06/06/2011
26	Valoración general	1 día	25	08/06/2011
27	**Documentación**		3	10/03/2011
28	Redacción Memoria		24FF	10/03/2011
29	Memoria y Producto		27	13/06/2011
30	Presentación		29	17/06/2011
31	Entrevista virtual	4 días	30	20/06/2011
32	Fin del proyecto		31	23/06/2011

La representación gráfica de las tareas de la tabla anterior, así como sus relaciones y porcentaje de progreso tras presentar el plan de trabajo (PAC1) puede verse en el siguiente diagrama.

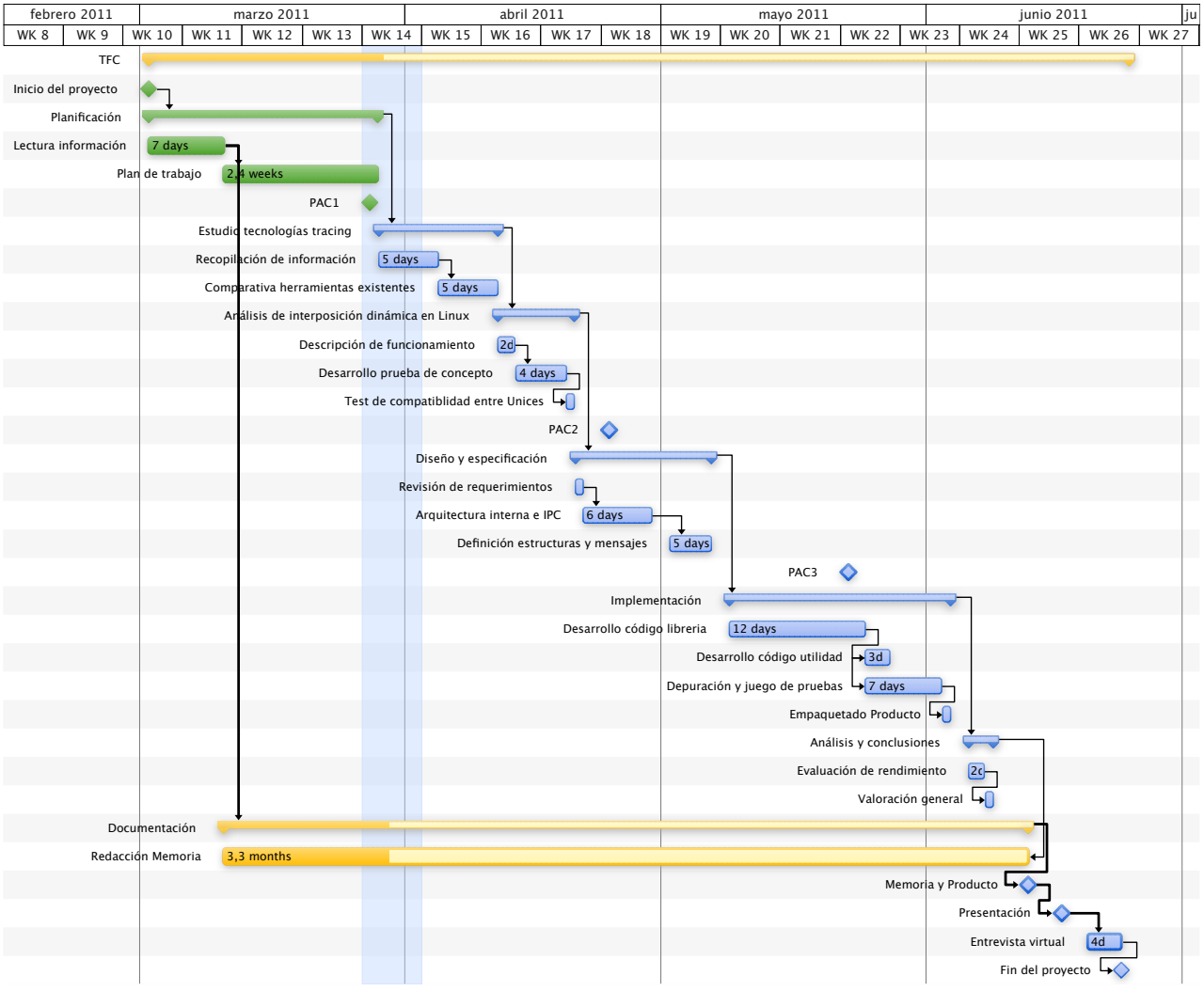

Figura 1. Diagrama de Gantt

7. Productos obtenidos

El código fuente de la librería y la utilidad de tracing se suministrarán en un archivo comprimido **gtrace-0.1.tar.gz** junto con los archivos necesarios para su puesta en marcha. Siguiendo el estándar en Linux, los programas se compilarán e instalarán con los comandos *configure* y *make*.

Se incluirá también un manual de usuario en un fichero README (Anexo 3) y una guía de instalación en un fichero INSTALL.

8. Composición de la memoria

El resto de capítulos de esta memoria está estructurado de la siguiente forma:

- **Capítulo II**. Se introducen los conceptos de tracing y profiling y se compara el software existente.
- **Capítulo III**. Se analiza el propósito de las bibliotecas compartidas, su funcionamiento, y se describe la técnica de interposición.
- **Capitulo IV**. Se detalla el desarrollo práctico de la librería de interposición.
- **Capítulo V**. Recoge las conclusiones del autor con respecto al TFC.

Capítulo II. Tracing y profiling

La traza de aplicaciones, o software tracing, es una técnica común utilizada en la depuración y diagnostico de aplicaciones. Consiste básicamente en la detección y grabación del flujo de eventos producidos durante la ejecución de un programa para su inmediato o posterior análisis. La idea es conocer que esta sucediendo dentro del programa o sistema para mejorarlo o mantenerlo monitorizado.

El termino tracing es muy similar al conocido concepto de *logging*, y de hecho, se puede considerar un subtipo de este. No es fácil hacer una clara distinción entre ambos términos, pues en muchos aspectos las tecnologías se solapan y se complementan. En general, solemos hablar de logging cuando el registro y la grabación de los eventos forma parte de la funcionalidad de la aplicación.

Por ejemplo, el registro de actividad del servidor web Apache o la infraestructura syslog[1] incluida en Linux formarían parte de la categoría de logging. En cambio, es más correcto hablar de tracing cuando nos referimos a macros insertadas en el código fuente por un desarrollador para depurar un error, o cuando utilizamos herramientas de monitorización como **LLTng** o **DTrace**.

Tabla 2. Distinción entre Logging y Software Tracing

Logging	Software Tracing
Enfocado a crear informes de uso y errores.	Enfocado al tratamiento sistematizado.
Genera poca información de “alto nivel”.	Genera mucha información de “bajo nivel”.
Fácil lectura. Omite duplicados.	Gran nivel de detalle. Ruidoso.
Eventos categorizados. Suelen seguir un estándar.	Sin limitaciones en el formato. Más caótico.
No causa impacto en el rendimiento de la aplicación.	El impacto en el rendimiento de la aplicación puede ser notable.

[1] syslog(3) es el estándar de facto para el envío de mensajes de registro. Por *syslog* se conoce tanto al servidor como al protocolo de red.

Además de la depuración de errores de software, las técnicas de tracing también se utilizan en otras áreas como:

- Monitorización de sistemas.
- Optimización de software. Lo que técnicamente se conoce como profiling.
- Seguridad y prevención de intrusiones.

Una traza es una secuencia de eventos ordenada en el tiempo. Puede contener información a nivel de sistema operativo, como eventos del planificador de tareas, entradas y salidas de *syscalls*, gestión de IRQs, o puede contener eventos de cualquier otra aplicación. Se puede generar de forma continua y grabar a petición según las necesidades, de forma periódica, o cuando se sospeche de algún problema. Se puede almacenar de forma permanente en un archivo del disco o transmitirlo por la red a otro ordenador para su procesamiento en tiempo real. Las posibilidades son variadas, pero se pueden resumir en los siguientes aspectos:

- Que mecanismos permiten determinar el comportamiento del programa, identificar los eventos relevantes y capturarlos.
- Como se transportan y se almacenan dichos datos. Que opciones tenemos para dejar estos datos a disposición del siguiente nivel de procesamiento.
- Como se analizan dichos datos. Que métricas se utilizan y como se extrae valor de los datos obtenidos.
- Y por último, como se visualizan e interpretan los resultados. Es en esta fase donde el usuario interactúa con el sistema.

Existe multitud de software para realizar tracing en sistemas operativos compatibles con POSIX como Linux. De hecho, a lo largo de los últimos años ha habido un desarrollo considerable de nuevos componentes y frameworks centrados en el Kernel de Linux, que de forma complementaria facilitan el tracing a las aplicaciones de usuario.

A continuación nos centraremos en los primeros eslabones de la cadena: la adquisición de datos y su almacenamiento. Estos serán los dos aspectos más relevantes para el desarrollo de esta memoria y nos permitirá comentar algunas de las herramientas más representativas disponibles en Linux.

1. Instrumentalización del código fuente

La instrumentalización consiste en añadir instrucciones al código fuente del programa para explícitamente calcular tiempos de ejecución o utilizar componentes que provean los mecanismos de tracing. Esto puede hacerse de manera automática o manual.

En la instrumentalización automática un **parser** se encarga de leer el código fuente de la aplicación e insertar en lugares apropiados, como son el comienzo y el final del cuerpo de las funciones, el código de instrumentalización.

En la instrumentalización manual es el programador quien realiza esta tarea, especificando los lugares donde insertar las instrucciones de trazabilidad en el código fuente.

1.1. GCC: opciones de compilación

El compilador GCC[2] incorpora varias extensiones para instrumentalizar el código fuente.

Una herramienta llamada **gcov** integrada en el compilador se puede utilizar para obtener información sobre la cobertura de ejecución de una aplicación. Esta herramienta es lo que se conoce como un *profiler*, y nos permite averiguar:

- Que lineas del código fuente se ejecutan realmente.
- Cuantas veces se ejecutan.
- Y que tiempo ha consumido cada sección del código.

Para activar esta funcionalidad hay que compilar el programa con las opciones *-fprofile-arcs* y *-ftest-coverage*. Al ejecutar un programa con gcov se crean automáticamente varios ficheros con extensiones .gcno y .gcda que contienen la información de cobertura. De este modo el análisis se efectúa después de ejecutar la aplicación.

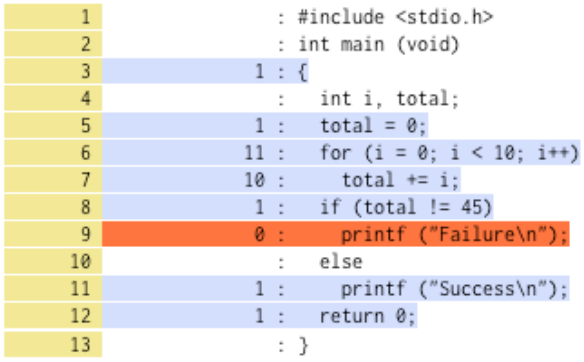

Figura 2. lcov: Interfaz gráfico de gcov

Otras opciones posibles son *-finstrument-functions* y *-fno_instrument_function*. Cada vez que se entra, y se sale de una función, se llama a las funciones de instrumentalización con la dirección de la función actual como parámetro.

```
void __cyg_profile_func_enter (void *this_fn, void *caller);
void __cyg_profile_func_exit  (void *this_fn, void *caller);
```

Figura 3. Declaración de funciones de instrumentalización

1.2. API en lenguajes de alto nivel

La mayoría de lenguajes de programación de alto nivel implementan mecanismos para tracing y profiling:

- Java, con la API JVMPI (JVM Profiling Interface).
- Python y el módulo hotshot.

[2] GNU Compiler Collection es el compilador estándar en el mundo Unix.

```
>>> import hotshot, hotshot.stats, test.pystone
>>> prof = hotshot.Profile("stones.prof")
>>> benchtime, stones = prof.runcall(test.pystone.pystones)
>>> prof.close()
>>> stats = hotshot.stats.load("stones.prof")
>>> stats.strip_dirs()
>>> stats.sort_stats('time', 'calls')
>>> stats.print_stats(20)
         850004 function calls in 10.090 CPU seconds

   Ordered by: internal time, call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    3.295    3.295   10.090   10.090 pystone.py:79(Proc0)
   150000    1.315    0.000    1.315    0.000 pystone.py:203(Proc7)
    50000    1.313    0.000    1.463    0.000 pystone.py:229(Func2)
 .
 .
 .
```

Figura 4. Benchmark del programa de prueba 'pystone' con hotshot

1.3. API de bajo nivel: LTTng

Las API de bajo nivel se centran en reducir el impacto en el rendimiento de la aplicación, reduciendo la latencia que producen las llamadas a las funciones de tracing, y mejorando la eficiencia en la gestión de los *buffers* internos donde se guardan los eventos. Esto es especialmente importante cuando se utilizan en aplicaciones como el Kernel, donde el rendimiento es un factor critico.

En el caso de LTTng, del que hablaremos más adelante, se han desarrollado algoritmos de buffer circular muy optimizados que ya han sido incluido en el Kernel, reemplazando proyectos anteriores como DTI (Driver Tracing Infrastructure).

La librería UST (Userspace Tracer) es la versión de LTTng para aplicaciones de usuario, y ofrece dos interfaces para instrumentalizar código: *makers* y *tracepoints*. Cada vez que se alcanza un marcador o tracepoint se genera un evento. El proceso **ustd** se encarga entonces de recoger los eventos y guardarlos de forma eficiente en el disco duro.

```
#include <ust/marker.h>
int main(int argc, char **argv)
{
    /* Un marcardor */
    trace_mark(main, inicio, "programa %s llamado con %d argumentos", argv[0], argc);
    return 0;
}
```

Figura 5. LLTng: Markers y tracepoints

Estos eventos se pueden analizar con el interfaz gráfico *LTTV-gui*.

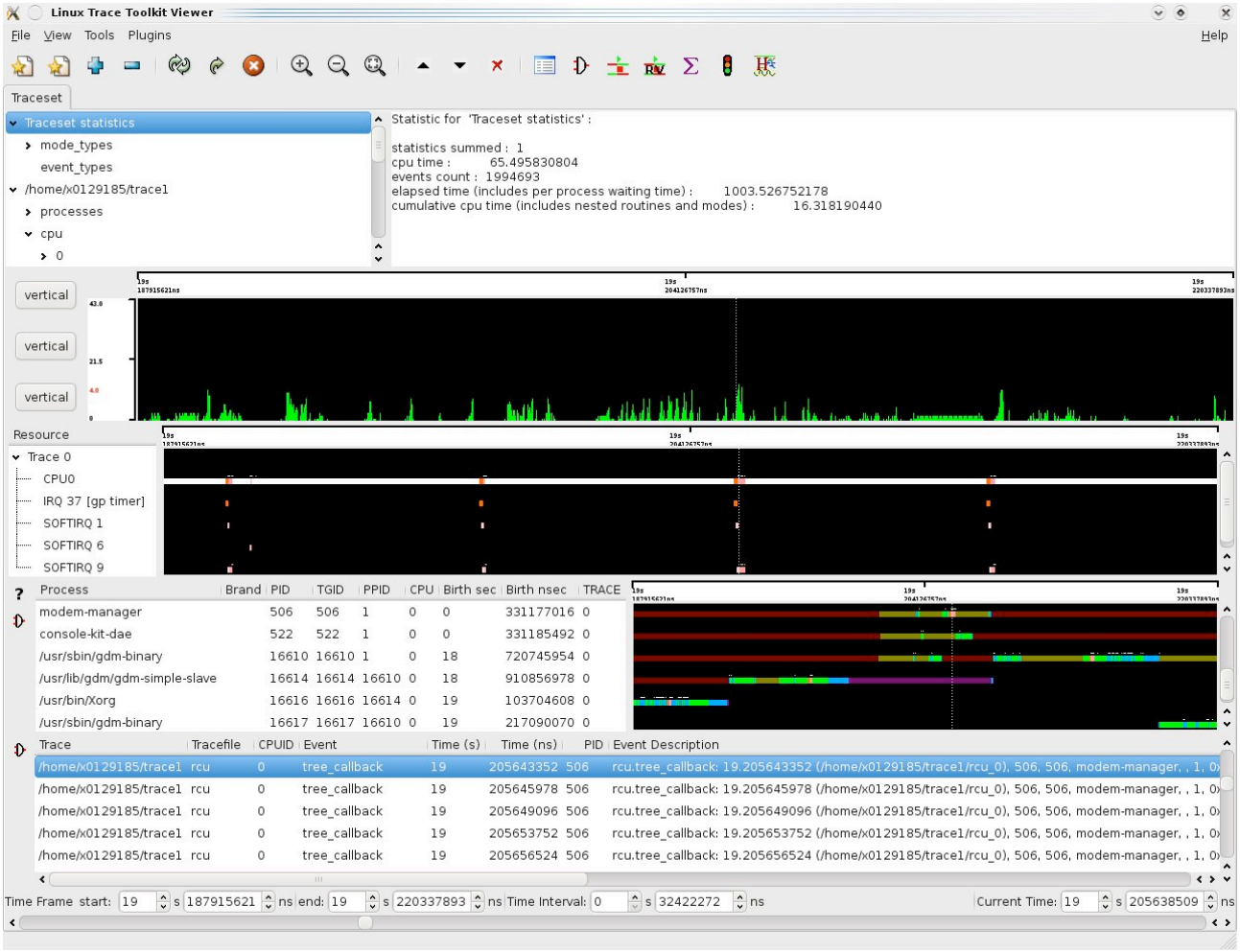

Figura 6. Visualización de trazas con LTTV

2. Emulación binaria

En un entorno virtual se puede lograr un control total sobre el sistema emulado. La ejecución de las instrucciones emuladas y los accesos a memoria son visibles y pueden ser instrumentalizados, sin tener que disponer del código fuente.

La desventaja de la emulación es el coste en términos de rendimiento, ya que un programa puede ejecutarse entre 10 o 100 veces más lento en un sistema emulado. Esto hace que la emulación no sea útil para realizar análisis de rendimiento.

En la actualidad, la mayoría de emuladores analizan las secuencias de instrucciones antes de ejecutarlas y las reemplazan por instrucciones binarias nativas[3] en el sistema que ejecuta la emulación.

Los emuladores libres más conocidos para Linux son Qemu y el depurador Valgrind.

[3] A esta técnica se la conoce como Just-in-time compilation (JIT)

3. Muestreo

Otra forma de recolectar información de un programa en ejecución es mediante un proceso de muestreo estadístico.

El muestreo consiste en monitorizar un programa, interrumpiendolo a intervalos regulares para conocer donde esta el contador de programa (registro IP), de modo que permita calcular heurísticamente el tiempo dedicado a la ejecución de una función.

Por ejemplo, la herramienta de análisis de GCC **gprof** muestra los procesos 100 veces por segundo y produce un histograma de distribución temporal del código.

```
Flat profile:

Each sample counts as 0.01 seconds.
       %   cumulative   self              self     total
      time   seconds   seconds    calls  ms/call  ms/call  name
      33.34      0.02     0.02     7208     0.00     0.00  open
      16.67      0.03     0.01      244     0.04     0.12  offtime
      16.67      0.04     0.01        8     1.25     1.25  memccpy
      16.67      0.05     0.01        7     1.43     1.43  write
      16.67      0.06     0.01                             mcount
       0.00      0.06     0.00      236     0.00     0.00  tzset
       0.00      0.06     0.00      192     0.00     0.00  tolower
       0.00      0.06     0.00       47     0.00     0.00  strlen
       0.00      0.06     0.00       45     0.00     0.00  strchr
       0.00      0.06     0.00        1     0.00    50.00  main
       0.00      0.06     0.00        1     0.00     0.00  memcpy
       0.00      0.06     0.00        1     0.00    10.11  print
       0.00      0.06     0.00        1     0.00     0.00  profil
       0.00      0.06     0.00        1     0.00    50.00  report
...
```

Figura 7. Ejemplo de muestreo

OProfile también es una herramienta de análisis para Linux basada en el muestreo que utiliza contadores por hardware.

4. Instrumentalización dinámica

La instrumentalización dinámica se refiere a la técnica de introducir el código de instrumentalización durante la ejecución del programa. Las herramientas de instrumentalización dinámica cambian las instrucciones de un proceso cargado en memoria inyectando el código de instrumentalización, sobreescribiendo el código original del proceso.

Las herramientas de instrumentalización dinámica se basan en los mecanismos del Kernel para depurar procesos e inyectar código de interrupción (*traps*). De hecho, la mayoría desarrollan componentes para analizar el propio Kernel, y de paso, obtener información sobre el funcionamiento de los procesos de usuario.

Queda fuera del ámbito de esta memoria analizar en profundidad las herramientas de instrumentalización dinámica, pues el objetivo del proyecto es realizar tracing en aplicaciones de usuario. Sin embargo, sin pretender ser exhaustivos, se ha elaborado la siguiente tabla con un resumen de características de cuatro de las principales herramientas existentes en Linux.

Tabla 3. Comparativa de frameworks

	LTTng	Systemtap	DTrace	perf
Licencia	GPL	GPL	CDDL	GPL
Sistema operativo	Linux	Linux	Solaris, Mac OSX, BSD, QNX, Linux	Linux
Desarrollado por	Comunidad abierta	Comunidad abierta	Comunidad abierta	lkml
Inicio del proyecto	Enero 2005	Enero 2005	Octubre 2001	2009
Estado del proyecto	En desarrollo	En desarrollo	Estable	En desarrollo
Audiencia	Desarrolladores, usuarios, sysadmins	Desarrolladores, usuarios, sysadmins	Desarrolladores, usuarios, sysadmins	Desarrolladores, sysadmins
Uso	Depuración, tracing, profiling	Depuración, tracing, profiling	Depuración, tracing, profiling, monitorización	Tracing, profiling
Sistema operativo soportado	Linux	Linux	Solaris, Mac OSX, BSD, QNX, Linux	Linux
Desarrollado por	Comunidad abierta	Comunidad abierta	Comunidad abierta	lkml
Estilo	Lenguaje C	Scripting	Scripting	Utilidad de linea de comandos
Análisis realizado	Offline (post-mortem)	Online	Online	Offline
Impacto en el rendimiento	Bajo	Alto	Alto	Bajo
Tracing de binarios	Si	Si	Si	No
Tracing de programas Java	En desarrollo	Si	Si	No
Tracing de scripts	Si (interfaz con C)	Si	Si	No
Tracepoints en el kernel	Cientos (funciones, tracepoints, markers)	Millones (statements, tracepoints, markers)	Cientos (funciones, markers)	Millones (funciones, tracepoints, statements)
Tracepoints en user-space	Cientos (markers)	Millones (statements, funciones)	Millones (funciones, makers)	Ninguno
Tipo de ejecución	Código nativo	Código nativo	Bytecodes	Integrado en el kernel

Capítulo III. Interposición de bibliotecas compartidas

Las bibliotecas compartidas son una característica esencial de Linux. Permiten reducir el tamaño de los archivos ejecutables y el consumo de memoria, y por tanto, mejorar el rendimiento. También logran minimizar el tiempo que tarda en ponerse en marcha una aplicación cuando es ejecutada.

Parte de la funcionalidad de las aplicaciones está implementada en bibliotecas o librerías. Cuando una aplicación se enlaza con una librería estática, el código de la librería se copia en el cuerpo de la aplicación y se convierte en parte del archivo ejecutable resultante. Esto produce algunas desventajas, como archivos ejecutables muy grandes, y obliga a volver a compilar el programa para beneficiarse de las futuras mejoras en las librerías.

Una alternativa para las aplicaciones es utilizar la flexibilidad de las bibliotecas compartidas de enlace dinámico.

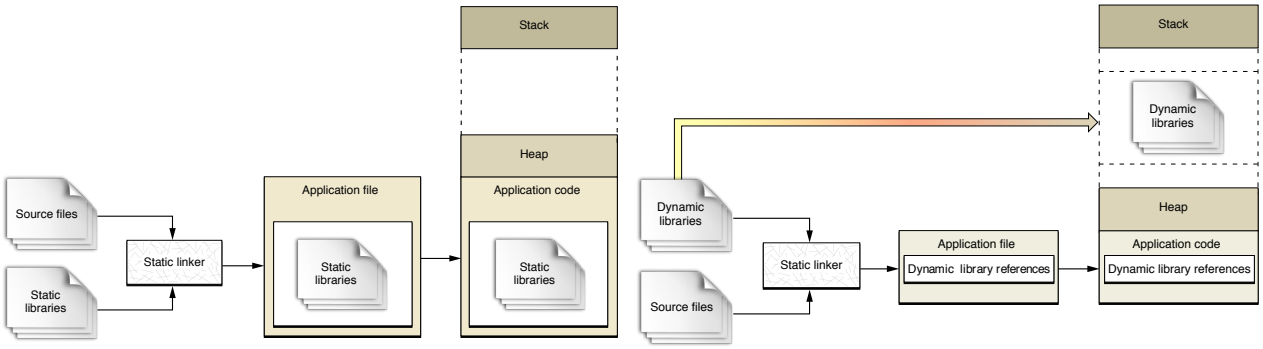

Figura 8. Librerías estáticas vs. Librerías compartidas

1. Introducción a las bibliotecas compartidas

En Linux, los programas ejecutables están contenidos en una imagen de archivo con el formato ELF[4]. Cuando se inicia un programa, el Kernel carga el código y los datos del programa en la memoria de un nuevo proceso. Una sección del ELF llamada **.interp** informa al Kernel que cargador, o *dynamic linker*, se debe utilizar. El Kernel procede entonces de copiar el cargador al proceso para pasarle el control de la ejecución.

```
[xoms@Hideo ~]$ readelf -l /bin/sh
Elf file type is EXEC (Executable file)
Entry point 0x415b90
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001c0 0x00000000000001c0  R E    8
  INTERP         0x0000000000000200 0x0000000000400200 0x0000000000400200
                 0x000000000000001a 0x000000000000001a  R      1
      [Requesting program interpreter: /lib/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000a23e4 0x00000000000a23e4  R E    200000
...
```

Figura 9. Secciones de una imagen ELF

2. El cargador de Linux

El programa **ld-linux.so** es a la vez el intérprete de archivos ELF y el enlazador dinámico. Su función es encontrar y vincular al programa las librerías compartidas que el programa necesita, es decir, las dependencias, e iniciar el programa una vez inicializado. Este proceso de vinculación o *linkage* se llama *relocation* y consiste en reemplazar las referencias a símbolos en librerías por direcciones de memoria reales [5].

```
[xoms@Hideo ~]$ ldd /bin/sh
      linux-vdso.so.1 =>  (0x00007fff3e944000)
      libreadline.so.6 => /lib/libreadline.so.6 (0x00007f19d7adf000)
      libdl.so.2 => /lib/libdl.so.2 (0x00007f19d78db000)
      libc.so.6 => /lib/libc.so.6 (0x00007f19d7584000)
      libncursesw.so.5 => /lib/libncursesw.so.5 (0x00007f19d732a000)
      /lib/ld-linux-x86-64.so.2 (0x00007f19d7d22000)
```

Figura 10. Ldd: Listado de dependencias

[4] Executable and Linking Format es el formato estándar del mundo Unix. Tiene la ventaja de que su diseño es flexible y no es dependiente de la plataforma hardware.

[5] Se entiende por direcciones reales las direcciones válidas en el espacio de memoria virtual del proceso.

Las dependencias se identifican con el nombre de archivo de la biblioteca correspondiente, y esta información se graba en la imagen del ejecutable. Para saber cuales son las dependencias de un programa se usa el comando *ldd* (figura 10).

El cargador utiliza esta información para localizar las bibliotecas a través del sistema de ficheros. Si se da el caso de que no encuentra alguna, o no es compatible, se aborta la ejecución. La variable de entorno **LD_LIBRARY_PATH** define los directorios donde se buscarán las librerías.

2.1. Dynamic Loading API

Además de este mecanismo de *linkage* automático, las aplicaciones pueden optar por cargar las librerías durante su ejecución. Por ejemplo, no hace falta cargar en memoria la funcionalidad de ayuda (F1) desde el primer momento, cuando se inicia la aplicación. Se puede cargar más tarde sólo si el usuario solicita el interfaz de ayuda con el comando correspondiente.

Para cargar librerías compartidas en tiempo de ejecución, las aplicaciones deben interactuar con el cargador, el cual siempre forma parte de la memoria del proceso. El mecanismo es el mismo que el descrito en el apartado anterior, pero ahora el *linkage* se realiza a petición de la aplicación, mediante las funciones definidas en **<dlfnc.h>**:

- **dlopen**: abre una biblioteca compartida. La aplicación llama a esta función antes de utilizar los símbolos exportados por la librería. Devuelve un manipulador, o *handle*, para referirse a la biblioteca abierta en las llamadas a dlsym y dlclose.
- **dlsym**: devuelve la dirección de memoria de un símbolo exportado. Esta función toma como parámetros el *handle* de la librería obtenido con dlopen, una constante que define el ámbito de búsqueda del símbolo, y el nombre del símbolo o función que se desea obtener.
- **dlclose**: cierra una biblioteca compartida.
- **dlerror**: devuelve una cadena que describe una condición de error durante la última llamada a dlopen, dlsym o dlclose.

3. Interposición con LD_PRELOAD

La interposición es el proceso por el cual se instala una librería dinámica diferente entre la aplicación y la biblioteca original de referencia. Se puede interceptar las llamadas a funciones en librerías compartidas, instando al *dynamic linker* a enlazar primero con la librería interpuesta antes de buscar la librería original en el sistema de ficheros.

En linux la interposición se consigue configurando la variable de entorno LD_PRELOAD para que apunte a la librería de interposición. Si, por ejemplo, la librería interpuesta contiene una definición de la función *fopen()* el código interpuesto de ejecutará siempre que la aplicación haga referencia a dicha función. Esto nos obliga a replicar la funcionalidad de la función *fopen()* en nuestra librería interpuesta, lo cual es imposible. La solución pasa por buscar la función *fopen()* original desde la librería de interposición con la API dlsym.

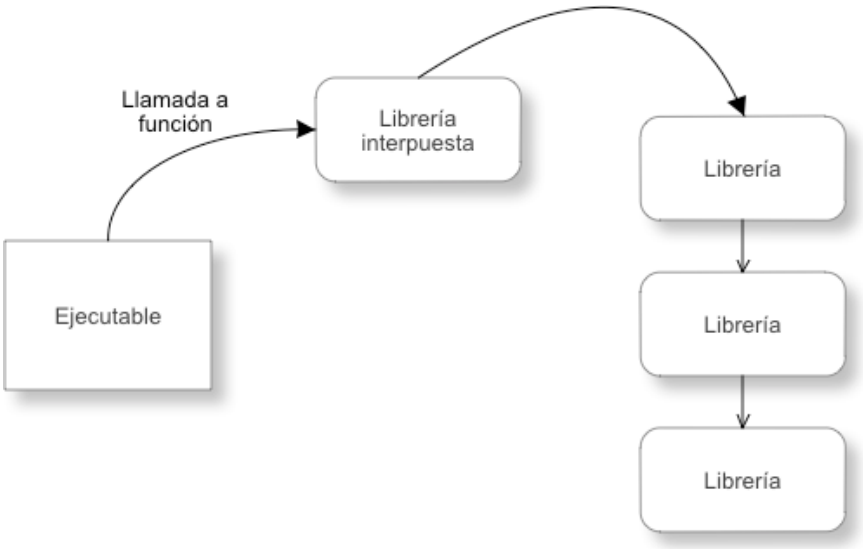

Figura 11. Llamadas a funciones con librería interpuesta

3.1. Como se interceptan llamadas a funciones

Para demostrar esta técnica vamos a construir una librería que intercepte llamadas a la función *fwrite()*. Esta función escribe una cadena de texto en el *stream* especificado. El objetivo es cambiar el color del texto que imprime la aplicación en la pantalla.

Primero escribimos el código de la función que substituirá a *fwrite()* en **rainbow.c**. El prototipo de la función interpuesta debe coincidir con la función original.

```
/* RTLD_NEXT es una extension GNU de libc */
#define _GNU_SOURCE
#include <dlfcn.h>
#include <string.h>

/* Codigos del terminal para colorear texto */
const char *colors[] = {
        "\x1b[3;31m", "\x1b[3;32m", "\x1b[3;33m", "\x1b[3;34m",
        "\x1b[3;35m", "\x1b[3;36m", "\x1b[3;37m"
};

/* Color predeterminado */
const char *clr_reset = "\x1b[0m";

/* Puntero a la función interpuesta */
size_t (*real_fwrite)(const void *, size_t, size_t, void *);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, void *stream)
{
        static int i;
        size_t ret;

        /* RTLD_NEXT significa buscar el simbolo en las siguientes librerias. */
        if (!real_fwrite)
                real_fwrite = dlsym(RTLD_NEXT, "fwrite");

        /* Escribimos el codigo para cambiar el color y luego el texto */
        real_fwrite(colors[i], strlen(colors[i]), 1, stream);
        ret = real_fwrite(ptr, size, nmemb, stream);
```

```
        real_fwrite(clr_reset, strlen(clr_reset), 1, stream);

        /* Alternar los colores */
        i = (i + 1) % 7;

        return ret;
}
```

Figura 12. Archivo rainbow.c

Compilamos la función en una librería compartida. Las opciones de gcc para producir librerías compartidas son *-fPIC* y *-shared*.

Por último, ejecutamos un programa típico que haga uso de *fwrite()*, por ejemplo el comando *date*, y comprobamos el resultado.

```
[xoms@Hideo rainbow]$ gcc -fPIC -shared -o rainbow.so rainbow.c -ldl
[xoms@Hideo rainbow]$ date
Mon May 23 00:58:10 CEST 2011
[xoms@Hideo rainbow]$ LD_PRELOAD=./rainbow.so date
Mon May 23 00:58:21 CEST 2011
[xoms@Hideo rainbow]$
```

Figura 13. Ejemplo de interposición con rainbow.so

3.2. Compatibilidad

Cada plataforma implementa sus propios cargadores o dynamic linkers de forma diferente. No todos los sistemas operativos tipo Unix comparten el mecanismo LD_PRELOAD. Linux, Solaris, y la familia BSD si lo hacen. Pero en el mundo Mac las cosas son diferentes.

3.2.1. Mac OS X

El sistema Mac OS X usa GCC, pero no utiliza el *linker* de GNU. En su lugar tenemos el cargador **dyld** y las librerías compartidas *Mach-O* **.dylib**.

Existe un equivalente a LD_PRELOAD que es la variable de entorno DYLD_PRELOAD. Pero su funcionamiento es diferentes, pues en Mac OS X la mayoría de programas están compilados con dos niveles de *namespaces.* Sin entrar en detalles, la opción DYLD_PRELOAD se debe utilizar con DYLD_FORCE_FLAT_NAMESPACE, pero en la práctica esto produce colisiones en los nombres de los objetos y las aplicaciones dejan de funcionar.

Sin embargo, desde la versión 10.4 de Mac OS X la interposición es posible gracias a unos cambios introducidos en el sistema. Primero, la librería de interposición debe contener una sección llamada “__interpose” en el segmento DATA con una lista de funciones interpuestas. Después, se fija la variable DYLD_INSERT_LIBRARIES y se obtiene el mismo efecto que con LD_PRELOAD.

En el Anexo 1 está listado el fichero de cabecera **dyld-interposing.h** que se utiliza para interponer funciones en Mac OS.

4. Ventajas de la interposición

Una parte importante de la funcionalidad de las aplicaciones esta implementada en bibliotecas compartidas. Interceptando el acceso a estos objetos se podría obtener una valiosa fuente de información de tracing en las aplicaciones de usuario.

En una librería con funciones prototipo se pueden registrar las llamadas a funciones y sus parámetros. Eventualmente también sería posible alterar el contenido de dichos parámetros antes o después de la llamada a la función real.

Al ser una librería externa a la aplicación se puede crear sin tener que recompilar la aplicación y, por tanto, no haría falta disponer del código fuente. El nivel de detalle del registro, así como el lugar donde almacenarla y en que formato, se podría controlar dinámicamente en múltiples procesos en ejecución.

Además, una librería de interposición sólo actuaría sobre una colección de funciones interesantes definidas previamente, sin afectar al resto del sistema, así que tendría un impacto mínimo en el rendimiento.

Estas ventajas también las podemos encontrar en muchas de las utilidades que hemos examinado en el capitulo II. La utilidad *ltrace* por ejemplo, que utiliza el mecanismo *ptrace* del Kernel para interceptar las llamadas al sistema, ofrece esta misma funcionalidad, aunque con un rendimiento bastante malo.

En conclusión, utilizar la técnica de interposición puede suponer una ventaja en algunos aspectos, pues aún no se ha desarrollado ninguna herramienta universal de tracing.

Capítulo IV. Librería gtrace

Una vez establecido el marco teórico podemos comenzar a abordar el objetivo principal del proyecto: el desarrollo de la librería de la interposición teniendo en cuenta los requisitos de diseño plantando en la introducción de la memoria.

Primero echaremos un vistazo general a la arquitectura prevista para después comentar algunos detalles de la implementación.

1. Arquitectura

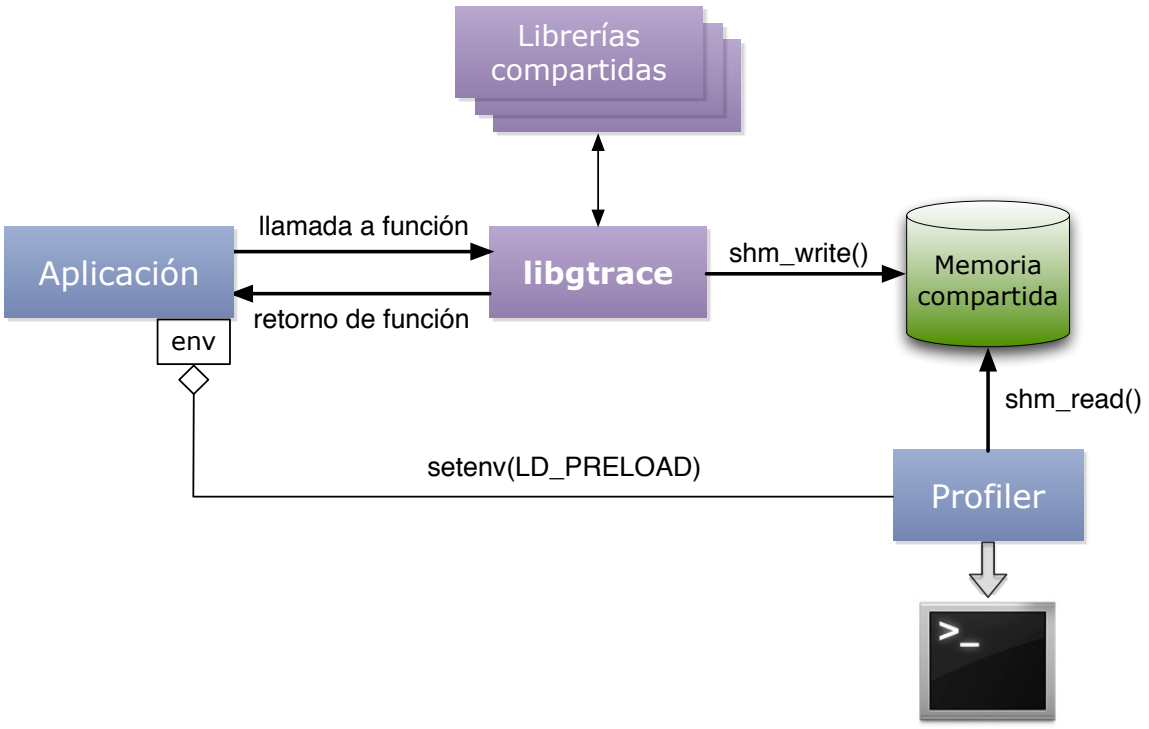

Figura 14. Visión general de los componentes de gtrace

El esquema propuesto se puede ver en la figura 14. Se distinguen dos componentes principales en el esquema: la librería *libgtrace* y el *profiler*.

Para operar con este sistema, el primer paso consiste en ejecutar el profiler, el cual configura el mecanismo de comunicación (IPC) para recibir los mensajes de la librería gtrace. El profiler puede recibir mensajes de múltiples instancias de la librería.

Después se ejecutan las aplicaciones que se desean auditar indicando al cargador que interponga la librería libgtrace. Durante la ejecución la librería recoge la información de llamadas a funciones, parámetros, etc., y la encapsula en una estructura apropiada para transmitir la información por IPC al profiler. Este último recibe los mensajes, los agrupa en eventos y los pasa finalmente a la última etapa del sistema, donde se tratan según la funcionalidad que se quiera obtener.

1.1. Mensajes

Los mensajes son registros con contienen la siguiente información:

- **gtr_pid**. Identificador del proceso (PID) que origina el mensaje.
- **gtr_time**. Marca de tiempo (*timestamp*).
- **gtr_type**. Valor que identifica el tipo de mensaje (ver tabla 4).
- **gtr_value**. Valor numérico que depende del tipo de mensaje.

- **gtr_len**: Tamaño en bytes del siguiente campo.
- **gtr_buffer**: Datos adicionales de tamaño variable.

Para hacer los mensajes más compactos los nombres de funciones se identifican con una constante de 32 bits, en lugar de con una cadena de texto. Por ejemplo, a la función *fopen* le corresponde el valor 0x55642948.

Tabla 4. Lista de mensajes gtrace

gtr_type		gtr_value	gtr_buffer
GTR_LIBCALL	Llamada a función	Identificador de función	Lista de parámetros
GTR_LIBRET	Retorno de función	Identificador de función	Código de retorno de la función Valor de *errno*
GTR_GENIO	Lectura/escritura datos	Descriptor del archivo (*FD*)	Flag lectura/escritura
GTR_PSIG	El proceso ha recibido una señal	Número de señal	(vacío)

1.2. IPC

Para la transmisión de mensajes entre la librería y el profiler se ha optado por utilizar la memoria compartida (SHM) como mecanismo de comunicación.

Linux provee otros mecanismo IPC como archivos, sockets, pipes o colas de mensajes, pero la memoria compartida parece la mejor opción teniendo en cuenta los requisitos de diseño:

- Es el método más rápido de comunicación entre procesos [6].
- Puede ser compartido por más de dos procesos de forma simultánea y sin necesidad de que exista un proceso ancestro común, como sucede con las pipes.

Para usar la memoria compartida se dispone de cuatro llamadas al sistema: *shmget*, *shmctl*, *shmat* y *shmdt*. El profiler se encarga de crearla e inicializarla con *shmget*. Después los procesos que usen la librería se conectan vía *shmat.* Ambos comparten una llave, o *key,* elegida al azar que sirve como identificador único de la memoria.

Si lanzamos el profiler se puede ver el tamaño y la clave ejecutando el comando *ipcs*.

```
[xoms@Hideo gtrace-0.1]$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x67747221 1376257    xoms       600        524312     1
...
```

Figura 15. Atributos de la memoria compartida

[6] Otra opción es mmap, que es muy similar y ofrece las misma funcionalidad que shm. Ambos son mecanismos POSIX-compatible.

1.2.1. Estructura de la memoria compartida

La memoria compartida se organiza en una buffer circular, cuya cabecera se muestra a continuación.

```
/**
 * SHM block descriptor
 */
struct shm_record {
            volatile s32   lock;            /* shared read/write spinlock    */
            volatile u32   length,          /* buffer length (set by tracer) */
                           write_count,     /* write cycle counter           */
                           write_off,       /* current write head offset     */
                           read_count,      /* read cycle counter            */
                           read_off;        /* current read head offset      */
            volatile u8    data[0];         /* gtrace record cyclic buffer   */
}  __attribute__((packed));
```

Figura 16. Estructura del buffer circular

La memoria compartida no ofrece mecanismos de sincronización por si misma. Se puede dar el caso de que un proceso, o *thread*, intente escribir en el buffer de forma simultánea a otro, sin esperar a que este haya completado la escritura. Esto provocaría perdida de datos y una corrupción en la estructura de la memoria compartida.

1.2.2. Sincronización

Para sincronizar los acceso de lectura y escritura se utiliza una variable especial llamada *lock* de la cabecera del buffer que funciona a modo de semáforo, con operaciones CAS[7].

```
#define atomic_cmpxchg        __sync_bool_compare_and_swap
#define atomic_add            __sync_add_and_fetch
#define atomic_sub            __sync_sub_and_fetch
#define atomic_inc(v)         __sync_add_and_fetch(v, 1)
#define atomic_dec(v)         __sync_sub_and_fetch(v, 1)
#define atomic_or             __sync_or_and_fetch
```

Figura 17. Primitivas atómicas disponibles en GCC

En cada acceso a la memoria se intenta adquirir el bloqueo, excepto si ya esta adquirido por otra instancia de la librería, en cuyo caso se devuelve el ciclo al planificador con **sched_yield()** y se repite la operación más tarde.

Se ha considerado que es mejor no utilizar los mecanismos de sincronización que ofrece el Kernel, para poder así reducir al máximo las llamadas al sistema, dada la cantidad de mensajes que se generan por segundo y que cada mensaje requiere un dos accesos al buffer (escritura y lectura).

[7] Compare-And-Swap es una instrucción especial que compara el valor de una variable, y si coincide, se intercambia su valor por el de otra variable. Es un tipo de instrucción indivisible (ningún proceso puede interrumpirla) que el procesador ejecuta de forma atómica.

2. Implementación de la librería

La mayor parte del código de la librería esta implementado en los archivos **gtrace.h** y **libgtrace.c**, y el código que gestiona el acceso a la estructura SHM se comparte con el profiler en el archivo de cabecera **shm.h**.

Las funciones de la librería se puede agrupar en tres grandes bloques:

- Inicialización de la librería. Es el código encargado de conectar con la memoria compartida e inicializar el estado interno. También configura la gestión de ciertas señales (SIGKILL, SIGSEGV, ...) necesarias para el buen funcionamiento de la librería (ver apartado 2.2).
- Escritura en la memoria compartida. Las funciones **gtr_signal**, **gtr_libcall** y **gtr_libret** crean el mensaje correspondiente y lo escriben en el buffer. Las tres funciones devuelven un entero que identifica inequívocamente el mensaje escrito para poder referenciarlo más tarde. Este identificador se puede utilizar para sincronizar con *shm_sync()*, de forma que se detiene la ejecución de la librería hasta que el mensaje ha sido recibido y procesado por el profiler.
- Definición de funciones interpuestas. Todas las funciones que queramos capturar tendrán que estar definidas en la librería. Por esta razón se ha creado una macro especifica que facilita esta tarea, y que además, evita la duplicidad de código.

2.1. Definición de funciones interpuestas

Para definir una función interpuesta en la librería se utiliza la macro **DEFINE_INTERPOSE**.

Esta macro declara un puntero estático a la función original (sufijo _real) que se inicializa la primera vez que el programa realiza una llamada a la función con la macro **LOAD_SYM.**

Por ejemplo, la figura 18 muestra el código en C generado para el prototipo de función *fopen()*.

```
 1: DEFINE_INTERPOSE(FILE *, fopen, const char *path, const char *mode)
 2: {
 3:     LOAD_SYM(fopen);
 4:     gtr_libcall(NAME_FOPEN, "path \"%s\" mode %s", path, mode);
 5:     FILE *ret = REAL(fopen)(path, mode);
 6:     gtr_libret(NAME_FOPEN, (ret == NULL));
 7:     return ret;
 8: }

 9: FILE * (*fopen_real)(const char *path, const char *mode);
10: FILE * fopen(const char *path, const char *mode)
11: {
12:     if (!fopen_real)
13:         fopen_real = dlsym(((void *) -1l), "fopen");
14:     gtr_libcall(0x55642948, "path \"%s\" mode %s", path, mode);
15:     FILE *ret = fopen_real(path, mode);
16:     gtr_libret(0x55642948, (ret == ((void *)0)));
17:     return ret;
18: }
```

Figura 18. DEFINE_INTERPOSE fdopen

En la linea 14 se pueden ver la constante de la función *fopen()* y como se pasan los parámetros *path* y *mode* a la función *gtr_libcall* para registrarlos en la memoria compartida.

También en la linea 15 y 16 se observa como se guarda el valor de retorno de *fopen()* para devolverlo después a la aplicación.

2.2. Bloqueo de señales asíncronas

Linux utiliza señales para informar a los procesos sobre alguna condición particular, realizar esperas entre procesos, etc. Cuando llega una señal a un proceso el sistema interrumpe la ejecución normal del proceso, o de cualquier llamada al sistema que se este realizando, y pasa el control a la rutina de gestión de la señal.

Por tanto, la función *gtr_write*, que se utiliza para escribir en la memoria compartida, también puede ser interrumpida tras adquirir el acceso en exclusiva a la memoria compartida, pero antes de liberarlo. Esta circunstancia puede causar dos condiciones de *deadlock* que detienen el funcionamiento de la librería y el profiler. En concreto:

- Si se recibe una señal de interrupción (SIGTERM, SIGQUIT) la aplicación puede finalizar sin liberar el acceso a memoria compartida. Esto es especialmente problemático si la aplicación es multi-threading.
- Si se configura un gestor de señal en la aplicación y se realiza llamadas a funciones interpuestas en el cuerpo de la función que gestiona la señal.

Para prevenir estos problemas se ha optado por proteger el área critica[8] de la función gtr_write con una variable global *defer_signals* e interceptar todas las señales problemáticas (figura 19).

```
static u32 gtr_write(struct gtr_header *gth)
{
...
            defer_signal++;

            shm_write_wait(GTH_SIZE + len);
            shm_write(gth, GTH_SIZE);
            if (len)
                        shm_write(gth->gtr_buffer, len);

            write_seq = shm_post();

            defer_signal--;
            if (!defer_signal && sig_pending) {
                        sig = sig_pending;
                        sig_pending = 0;
                        raise(sig);
            }

            return write_seq;
}
```

Figura 19. Fragmento de la función gtr_write

[8] Se denomina sección crítica a la porción de código de un programa en la cual se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un hilo en ejecución.

3. Implementación del profiler

El código del profiler contiene la funcionalidad mínima para hacer uso de la librería. En total se han definido 22 funciones interpuestas usadas en la mayoría de comandos de Linux, tales como *fork, scanf* o *strcmp*.

El programa **gtrace** es el *frontend* del profiler. Puede funcionar en segundo plano y así analizar múltiples procesos de forma simultánea. En el anexo 3 se podrá encontrar la ayuda del programa e instrucciones de uso (README).

Como muestra se adjunta una captura de la traza de la utilidad *whoami* en la figura 20.

```
[xoms@Hideo gtrace-0.1]$ ./gtrace -l .libs/libgtrace.so -- whoami
gtrace 0.1 by <xoms@uoc.edu>
11:32:08.227511 +++ trace of 'whoami' started +++
xoms
11:32:08.247322 pid(3111): fopen (path "/etc/passwd" mode re)
11:32:08.247332 pid(3111): fopen: returns 0
11:32:08.247333 pid(3111): fileno (stream 0x7220d0)
11:32:08.247335 pid(3111): fileno: returns 3
11:32:08.247351 pid(3111): fclose (fp 0x7220d0)
11:32:08.247357 pid(3111): puts (str "xoms")
11:32:08.247365 pid(3111): exit (status 0)
11:32:08.247367 pid(3111): fclose (fp 0xf98f9780)
11:32:08.247370 pid(3111): fclose (fp 0xf98f9860)
11:32:08.247479 +++ exited (code 0) +++
11:32:08.247485 +++ 10 messages processed (10 received) +++
```

Figura 20. Ejemplo del comando gtrace

4. Rendimiento

Por cada llamada al sistema que capturamos necesitamos obtener una información que implica realizar otras llamadas al sistema, lo que provoca un consumo de ciclos y cambios de contexto del procesador.

Por ejemplo, para saber el PID del proceso que realiza una llamada habría que incluir *getpid()* en el cuerpo de todas las funciones interpuestas. En este caso se ha podido evitar incluyendo una variable global **gtrace_pid** que se inicializa con cada nueva instancia de la librería y forzando la inicialización en *fork()*.

```
DEFINE_INTERPOSE(pid_t, fork, void)
{
            LOAD_SYM(fork);
            pid_t pid = REAL(fork)();
            if (!pid)
                        libgtrace_init();
            gtr_wait(gtr_libret(NAME_FORK, pid));
            return pid;
}
```

Figura 21. Inicialización del proceso hijo

Sin embargo, para obtener en el análisis un *timestamp* de suficiente precisión es inevitable saber la hora del sistema en cada llamada. En la librería se usa *gettimeofday()* de libc, que puede tener un retardo del orden de milisegundos en algunos sistema. Para futuras versiones de la librería sería interesante buscar una alternativa y programar un reloj de tiempo real como los que se usan en las aplicaciones multimedia.

5. Limitaciones

No todos los programas y librerías pueden ser analizados con gtrace. Por la documentación consultada parece ser que las librerías programadas en C++ son especialmente difícil de interponer, aunque no se han realizado pruebas en este sentido. La dificultad se presenta a la hora de obtener las definiciones de funciones por la utilización en C++ de operadores sobrecargados, constructores, etc.

Otra excepción son los programas privilegiados, es decir, aquellos que se ejecutan como *root* porque tienen activado los bits ***setuid*** o ***setgid***. En este caso el sistema ignora la variable LD_PRELOAD por razones de seguridad y no pueden ser traceados con *gtrace*.

Aun así, si se quiere analizar programas *setuid* se puede compilar un lanzador que evite esta restricción. Se ha desarrollado una versión en C de dicho lanzador que esta incluido en el **Anexo 2** de esta memoria, junto con las instrucciones de uso.

Capítulo V. Conclusiones

En este proyecto se ha realizado un estudio sobre las tecnologías de tracing en Linux, que ha servido de introducción teórica y aproximación al problema que se quería resolver. Asimismo, se ha diseñado e implementado una librería de interposición, cumpliendo cada uno de los requisitos de diseño planteados.

Dicha librería constituye una solución viable para futuro desarrollos como demuestra la funcionalidad mínima pero completa del prototipo presentado. Además se han solventado problemas inesperados de implementación, como los problemas de *deadlock* y los ficheros *autoconf*, cuyas soluciones han supuesto un autentico desafío.

En este sentido se han cumplido los objetivos del proyecto.

Hay dos aspectos, sin embargo, que en mi opinión hubieran merecido mayor atención en este proyecto: el análisis de rendimiento de la librería, que ha quedado parcialmente incompleto; y la creación de una batería de pruebas automáticas, imprescindible como buena práctica de programación.

Como experiencia personal encuentro que, en general, he tomado conciencia de la importancia que tiene los aspectos más formales del desarrollo de proyectos. Por ejemplo, ahora que el proyecto ha finalizado, me doy cuenta de lo importante que es seguir fielmente la planificación establecida al inicio del proyecto para lograr cumplir todos los objetivos propuestos sin sobresaltos.

Es evidente, entonces, que una buena organización es indispensable para que el trabajo resulte agradable y fructífero.

Bibliografía

1. Curry, Timothy W. (1994). Profiling and Tracing Dynamic Library Usage Via Interposition. *USENIX Summer 1994 Technical Conference*.
2. Kuperman, Benjamin A.; Spafford, Eugene (1999). *Generation of Application Level Audit Data via Library Interposition*. COAST Laboratory. Purdue University.
3. Garfinkel, Tal. *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools*. Computer Science Department, Stanford University.
4. Myers, Daniel S.; Bazinet, Adam L. (2004). *Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms*. Institute for Advanced Computer Studies, University of Maryland.
5. Jones, M. Tim. (2008). “Anatomy of Linux Dynamic Libraries”. *IBM developerWorks*. http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/
6. Grover, Sandeep (2002). “Linkers and Loaders”. *Linux Journal*. http://www.linuxjournal.com/article/6463
7. Nakhimovsky, Greg (2001). “Building library interposers for fun and profit”. *ITworld*. http://www.itworld.com/print/35352
8. Lu, Hongjiu (1995). *ELF: From The Programmer's Perspective*. NYNEX Science & Technology, Inc.
9. GNU Project. “GCC online documentation”. *Free Software Foundation (FSF)*.
10. “Runtime Linking Programming Interface (Linker and Libraries Guide)”. *Oracle*. http://download.oracle.com/docs/cd/E19082-01/819-0690/6n33n7f8b/index.html
11. Izik. *Reverse Engineering with LD_PRELOAD*. http://www.exploit-db.com/papers/13233/
12. Wheeler, David A. (2003). “Program Library HOWTO”. *The Linux Documentation Project*.
13. LTTng. *TracingBook.* http://lttng.org/tracingwiki/index.php/TracingBook
14. Röck, Harald (2007). *Survey of Dynamic Instrumentation of Operating Systems*. Department of Computer Sciences. University of Salzburg, Austria.
15. Fournier, Pierre-Marc; Desnoyers, Mathieu; Dagenais, Michel R. *Combined Tracing of the Kernel and Applications with LTTng*. École Polytechnique de Montréal
16. “Signal Handling”. *The GNU C Library*.
17. Wikipedia (en). “Dynamic program analysis”.

18. Wikipedia (en). “Semaphore (programming)”.

19. Wikipedia (en). “Profiling (computer programming)”.

20. Wikipedia (en). “Shared memory”.

21. Wikipedia (en). “Non-blocking algorithm”.

22. mij. (2005). “A tutorial for porting to autoconf & automake”. http://mij.oltrelinux.com/devel/autoconf-automake/

23. GNU Project. “Autoconf - Creating Automatic Configuration Scripts”. *Free Software Foundation (FSF).*

24. ltrace(1). “library call tracer”. *Linux manual page.*

Glosario

API
Del inglés Application Programming Interface. Serie de rutinas que ofrece cierta biblioteca o sistema operativo para ser utilizado por otro software como una capa de abstracción.

Buffer
Espacio de memoria donde se almacenan datos de forma temporal.

Compilar
Traducir un programa escrito en lenguaje de programación a otro lenguaje, normalmente código objeto.

Enlazar
Es el proceso realizado por un enlazador o linker y consiste en leer el código objeto creado por el compilador y las librerías estáticas y crear un único archivo ejecutable.

Fork
Creación de una copia de si mismo por parte de un programa en ejecución.

Kernel
Es el núcleo de los sistemas operativos. Provee los servicios más básicos del sistema y se encarga de gestionar los recursos del ordenador. También ofrece una serie de abstracciones de hardware para que los programas de usuario no tengan que acceder directamente al hardware.

Handle
Puntero abstracto y opaco que representa objetos o áreas de memoria controlados por otros sistemas.

Heap overflow
Desbordamiento de la memoria intermedia.

IPC
Del inglés Inter-process Communication. La comunicación entre procesos es una función básica de los sistemas operativos. Provee un mecanismo que permite a los procesos comunicarse y sincronizarse entre si, normalmente a través de un sistema de envío de mensajes o compartiendo espacios de memoria.

IPS
Sistema de prevención de intrusos. En seguridad informática es un programa orientado a proteger de forma proactiva el sistema informático de ataques y abusos.

IRQ
Del ingles Interrupt Request. Señal recibida por el procesador indicando que debe interrumpir el curso de ejecución actual y pasar a ejecuta código especifico para tratar esta situación. Se suelen generar en los dispositivos periféricos.

Parser
Analizador sintáctico que convierte texto de entrada en otras estructuras más útiles para ser procesadas.

Pipe
Una tubería consiste en una cadena de procesos conectados de forma que la salida de datos de cada proceso de la cadena es la entrada de datos del próximo.

POSIX
Acrónimo de Portable Operating System Interface. Son una familia de estándares definidos por el IEEE que surgieron del mundo Unix. Define una serie de servicios y funciones del sistema operativo.

Profiling
Es la tarea en la que se ejecuta un programa especializado (profiler) para realizar el análisis de rendimiento de una aplicación.

Race condition
Situación que se da cuando el resultado de la ejecución de múltiples procesos relacionados depende del orden en que se ejecuten. Si no están correctamente sincronizados se pueden producir corrupción de datos, y en última instancia, vulnerar un sistema.

Socket

Define un concepto abstracto por el cual dos programas pueden intercambiar flujos de datos de forma ordenada. Hay sockets de dominio Unix para comunicación IPC entre procesos y sockets TCP/IP para la transmisión de paquetes por Internet.

Stream

En Unix es el origen de una secuencia de bytes. Puede ser un fichero, un dispositivo de entrada como el teclado, o de salida como un terminal. Hay como mínimo tres disponibles para cualquier programa: *stdin*, *stdout* y *stderr*.

Syscall

Abreviatura de system call o llamada al sistema. Es el mecanismo por el cual una aplicación solicita un servicio al sistema operativo.

Thread

En programación un thread es un hilo de ejecución que permite a un proceso llevar a cabo varias tareas de forma concurrente. Los distintos hilos de un proceso comparten los recursos del proceso, como el espacio de memoria y los archivos abiertos.

Anexo 1. Archivo dyld-interposing.h

```
/*
 * Copyright (c) 2005-2008 Apple Computer, Inc. All rights reserved.
 */

#if !defined(_DYLD_INTERPOSING_H_)
#define _DYLD_INTERPOSING_H_

/*
 *  Example:
 *
 *  static
 *  int
 *  my_open(const char* path, int flags, mode_t mode)
 *  {
 *    int value;
 *    // do stuff before open (including changing the arguments)
 *    value = open(path, flags, mode);
 *    // do stuff after open (including changing the return value(s))
 *    return value;
 *  }
 *  DYLD_INTERPOSE(my_open, open)
 */

#define DYLD_INTERPOSE(_replacement,_replacee) \
   __attribute__((used)) static struct{ const void* replacement; const void* replacee; } 
_interpose_##_replacee \
            __attribute__ ((section ("__DATA,__interpose"))) = { (const void*)(unsigned long) 
&_replacement, (const void*)(unsigned long)&_replacee };

#endif
```

Anexo 2. Utilidad para ejecutar programas setuid con LD_PRELOAD

```
/*
 * setuid_wrapper.c -- wrap programs to run under LD_PRELOAD
 *
 *      This wrapper is designed to run privileged programs with a
 *      preloaded dynamic library.
 *
 *      The original setuid or setgid program is replaced by this
 *      wrapper. The original program is being moved to another
 *      location and its permissions restricted.
 *
 *      This wrapper set ups the environment variable LD_PRELOAD
 *      before executing the original program.
 *
 * Author:      <xoms@uoc.edu>
 *
 * DISCLAIMER:  The use of this program is at your own risk.  It is
 *              designed to combat a particular vulnerability, and may
 *              not combat other vulnerabilities, either past or future.
 *              The decision to use this program is yours, as are the
 *              consequences of its use.
 *
 *
 * Installation instructions
 * =========================
 *
 *  1.  su to root
 *
 *  2.  Determine the location of the program you wish to run.
 *
 *      For example purposes, we'll assume the program we wish to wrap is
 *      /usr/bin/foobar.
 *
 *  3.  Determine the permissions, owner, and group of foobar.  Note this
 *      information as it will be used later.  For example:
 *
 *          # ls -l /usr/bin/foobar
 *          -r-sr-xr-x  1 root  bin  20480 Jul 17 12:30 /usr/bin/foobar
 *
 *      In particular, note whether the program is setuid or setgid.
 *
 *  4.  Copy the foobar program to foobar.real, and then restrict
 *      its permissions.
 *
 *              # cd /usr/bin
 *              # cp foobar foobar.real
 *              # chmod 511 foobar.real
 *
```

```
 *  5.  Note the location of foobar.real.  This will be used as the
 *      definition of REAL_PROG when compiling this wrapper.
 *      This should be an absolute pathname.  In this example,
 *      "/usr/bin/foobar.real"
 *
 *  6.  Compile this program in a non world writable directory other than
 *      /usr/bin.
 *
 *      For example, to use /usr/local/src, first copy this file to
 *      /usr/local/src.
 *
 *              # cd /usr/local/src
 *
 *      There are two defines required to compile this program:
 *
 *      REAL_PROG:  This is the location noted in step #5.
 *
 *      For this example, REAL_PROG is "/usr/bin/prog.real"
 *
 *      PRELOAD_LIB:  The pathname of the interposition library.
 *
 *      Once you have the values of REAL_PROG and PRELOAD_LIB you can
 *      compile this program.
 *
 *              # cc \
 *                      -DREAL_PROG='"/usr/bin/foobar.real"' \
 *                      -DPRELOAD_LIB='"libgtrace.so"' \
 *                      -o foobar_wrapper setuid_wrapper.c
 *
 *      Note that when compiling the values of REAL_PROG and PRELOAD_LIB
 *      needs to be enclosed in single quotes (') as shown above.
 *
 *  7.  Copy this new wrapper program, foobar_wrapper,  into the directory
 *      originally containing foobar.  This will replace the existing
 *      foobar program.
 *
 *      Make sure this directory and its parent directories are protected so
 *      only root is able to make changes to files in the directory.
 *
 *      Use the information found in step #3 and set the same
 *      owner, group, permissions and privileges on the new foobar program.
 *
 *      For example:
 *
 *              # cp foobar_wrapper /usr/bin/foobar
 *              # cd /usr/bin
 *              # chown root foobar
 *              # chgrp bin foobar
 *              # chmod 4555 foobar
 *
```

```
 *      Check that the owner, group, permissions and privileges exactly
 *      match those noted in step #3.
 *
 *              # ls -l /usr/bin/foobar
 *
 *      Users will not be able to use the foobar program during the time
 *      when the wrapper is copied into place until the chmod command
 *      has been executed.
 *
 * 8.   Check that foobar still works!
 *
 */

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

/*
 * This wrapper will run REAL_PROG setting 'LD_PRELOAD' to PRELOAD_LIB
 */

int main(int argc, char *argv[])
{

     putenv("LD_PRELOAD=" PRELOAD_LIB); /* set up the environment */

     execve(REAL_PROG, argv);
     perror("execve failed");
     exit(1);
}
```

Anexo 3. Ayuda del comando gtrace y fichero README

```
Trace library calls of a given program.

Usage:
  gtrace [options] -- [command [args ...]]

Example:
  gtrace -- ls -al

Options:
  -l path   Interposer library path. Default is /usr/local/lib/libgtrace.so
  -f list   Trace only specified functions (comma separated list)

==================
README Instructions
==================

This README file details how to run gtrace.

For full instructions on how to compile and install gtrace,
please read the INSTALL file.

Easy installation
-----------------

1.  Uncompress the package:

        tar xzf gtrace-0.1.tar.gz
        cd gtrace-0.1

2.  Compile and build the library:

        ./configure
        make

3.  Test before installation:

        ./gtrace -l .libs/libgtrace.so -- whoami

      -- OR --

        ./gtrace &
        LD_PRELOAD=.libs/libgtrace.so whoami
        kill %1

4.  Install the software:

        make install
```